KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Electrical Engineering (ESAT)

Analysis and Design of Cryptographic Hash Functions

Sebastiaan INDESTEEGE

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

May 2010

Analysis and Design of Cryptographic Hash Functions

Sebastiaan INDESTEEGE

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

U.D.C. 681.3*D46

May 2010

There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable.

There is another theory which states that this has already happened.

Douglas Adams

D/2010/7515/52
ISBN 978-94-6018-213-6

Preface

A PhD dissertation does not suddenly appear out of thin air. On the contrary, it is the result of a long process, involving much more people than just those mentioned on the cover. This one is no different. Many have contributed to it, in one way or another. I'd like to take this opportunity to thank (at least some of) the people who have helped and encouraged me during the past years.

First of all, I'd like to thank my promotor, prof. Bart Preneel. He has created a pleasant environment for me to do research in, granting me a great deal of freedom to plot my own course, but still giving me guidance when I required it. He also offered me plenty of opportunities (and funding) to make trips to conferences and other events abroad, where I met several distinguished people in the cryptologic community. I would like to express my gratitude to the members of my jury — dr. Orr Dunkelman, prof. Lars Knudsen, prof. Bart Preneel, prof. Vincent Rijmen, prof. Joos Vandewalle, and prof. Luc Van Eycken — for reviewing this manuscript and for their valuable feedback, and to prof. Adhemar Bultheel for chairing the jury. I would also like to acknowledge the financial support of the Fund for Scientific Research, Flanders (FWO-Vlaanderen) that made this research possible.

When I joined COSIC in 2006, I had the privilege of sharing an office with Christophe De Cannière, and later also Orr Dunkelman, both of which taught me various tricks of the trade. I've really learnt a tremendous amount from their experience; not only on purely technical topics, but also on life as a researcher. I'd like to thank my other colleagues in COSIC as well, including the numerous COSIC visitors I met, and especially my office mates and fellow 'Alma addicts'. Sorry I don't mention all of you here, but the group has grown too large for that. We had plenty of nice discussions on a multitude of topics, ranging from serious to, well, not quite so serious.

Péla Noë deserves special mention. Without her help, I'd probably still be lost somewhere in the labyrinth of administration and paperwork. She was — and still is — always ready to help out. For every administrative issue I encountered, she managed to quickly find a solution, and did so with great enthusiasm. The financial aspects were handled impeccably by Elvira Wouters and Elsy Vermoesen.

Throughout the course of my PhD, I've had the pleasure to work together with several very bright researchers. I've gained a lot from these fruitful collaborations and I'd like to acknowledge my co-authors: Elena Andreeva, Jean-Philippe

Aumasson, Eli Biham, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Nathan Keller, Florian Mendel, Svetla Nikova, Bart Preneel, Christian Rechberger, Martin Schläffer, and Elmar Tischhauser.

In March 2008, I visited TU Graz, Austria. I am very grateful to the people of the Krypto group at IAIK who hosted me. Thank you for the many interesting discussions we had in Graz, Mario Lamberger, Florian Mendel, Tomislav Nad, Norbert Pramstaller, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. In the summer of 2009, I visited Sony corp. in Tokyo, Japan as a research intern. It was a very enriching and unforgettable experience for me to discover Japan, and to experience the Japanese way of living. Thank you very much to everyone at Sony who made this possible, and in particular to Shiho Moriai, Kyoji Shibutani, and Taizo Shirai.

During the past years, I've also supervised the Master's thesis of several students. Even though you may not have noticed, I have learnt from each one of you. Thank you, Dries Cuypers, Jens Hoffmann, Igor Jacques, Nicky Mouha, Hans Narinx, Wim Ramakers, Dirk Rijkx, Joris Thielen, and Markus Ullrich.

Many thanks to my friends, who kept on trying to convince me that there is life outside of research. Although you may not always have been very successful at persuading me, thank you for being persistent nevertheless. Thanks to my fellow 'bosses' at Ulyssis for making me feel not quite so abnormal for being passionate about computers and UNIX-like systems.

And last, but definitely not least, I would like to thank my parents and my brothers, Dries and Mathijs. I know it must not always have been easy to have a PhD student as a son or a brother — they are rather peculiar creatures indeed — but thank you very much for your continuous support and encouragement.

Sebastiaan Indesteege
Heverlee, April 2010

Abstract

Cryptographic hash functions play an important role in the security of many applications such as digital signatures, the protection of passwords, the derivation of cryptographic keys, tamper detection, and countless others. This versatility has earned them the nickname 'Swiss army knives of cryptography'.

Most of the widespread and popular hash functions, such as MD5, SHA-1 and SHA-2, share a common design philosophy. Recent cryptanalytic advances have raised serious concerns regarding the long-term security of these hash functions. Some of them, e.g., MD4 and MD5, were broken in practice, and for others, e.g., SHA-1, severe theoretical weaknesses were shown. Even though the SHA-2 family is not (yet) really threatened by any attack, it receives little confidence because it is based on the same design principles. Hence, there is a clear need for new, secure cryptographic hash functions. The United States 'National Institute of Standards and Technology' (NIST) has started an international competition to develop the next generation cryptographic hash function standard, which will be called SHA-3. This competition started in 2007, and is still ongoing with 14 candidates left in the second round of evaluations.

Most of the research presented in this dissertation is closely related to this competition. We have designed a candidate cryptographic hash function called Lane. The primary objectives of Lane are to be secure, easy to understand, elegant and flexible in implementation. It was entered into the SHA-3 competition, but did not advance to the second round of evaluations.

Furthermore, we have actively contributed to the evaluation of several SHA-3 candidates. For a number of first round candidates, we have demonstrated attacks that contradict the security claims made by their designers. In particular, we have shown practical collision attacks on the candidates Dynamic SHA, EnRUPT and SHAMATA, as well as a theoretical collision attack on Dynamic SHA2 and a practical preimage attack on Maraca. Beside the SHA-3 competition, we have contributed to the cryptanalysis of the hash functions RC4-Hash, SHA-2, and Tiger, and the block cipher KeeLoq that is used among others in vehicle anti-theft systems.

Samenvatting

Cryptografische hashfuncties spelen een belangrijke rol in de veiligheid van een groot aantal toepassingen, zoals digitale handtekeningen, het beveiligen van wachtwoorden, het afleiden van cryptografische sleutels, het ontdekken van het ongeoorloofd wijzigen van gegevens, en talloze andere. Dankzij deze veelzijdigheid hebben ze de bijnaam ‘het Zwitsers zakmes van de cryptografie’ gekregen.

Het merendeel van de vaak gebruikte en populaire hashfuncties, zoals MD5, SHA-1 en SHA-2, is gebaseerd op dezelfde ontwerpfilosofie. Recente cryptanalytische doorbraken hebben ernstige zorgen gebaard over de veiligheid van deze hashfuncties op lange termijn. Sommigen onder hen, zoals MD4 en MD5, werden in de praktijk gebroken. Voor anderen, zoals SHA-1, werden er ernstige theoretische zwakheden aangetoond. Hoewel de SHA-2 familie (voorlopig) nog niet bedreigd wordt door om het even welke aanval, is er weinig vertrouwen in de veiligheid ervan omdat er op dezelfde ontwerpprincipes gesteund wordt. Er is dus een duidelijke behoefte aan nieuwe, veilige cryptografische hashfuncties. Het ‘National Institute of Standards and Technology’ (NIST) uit de Verenigde Staten organiseert een internationale competitie die moet leiden tot het ontwikkelen van de nieuwe standaard cryptografische hashfunctie, die SHA-3 zal heten. Deze competitie is van start gegaan in 2007 en loopt nog, met 14 kandidaten in de tweede evaluatieronde.

Het grootste deel van het onderzoek in deze verhandeling is nauw verbonden met deze competitie. We hebben de cryptografische hashfunctie LANE ontworpen. De voornaamste doelstellingen van LANE zijn veiligheid, verstaanbaarheid, elegantie en flexibiliteit in implementatie. LANE heeft deelgenomen aan de SHA-3 competitie, maar werd niet geselecteerd voor de tweede evaluatieronde.

Verder hebben we actief bijgedragen tot de evaluatie van verschillende SHA-3 kandidaten. Voor een aantal kandidaten uit de eerste ronde hebben we aanvallen getoond die de veiligheidsbeweringen van de ontwerpers tegenspreken. In het bijzonder hebben we praktische botsingsaanvallen ontwikkeld voor de kandidaten Dynamic SHA, EnRUPT en SHAMATA, alsook een theoretische botsingsaanval voor Dynamic SHA2. Verder hebben we in de praktijk aangetoond dat Maraca de vereiste éénwegseigenschap niet bezit. Naast de SHA-3 competitie hebben we ook bijgedragen tot de cryptanalyse van de hashfuncties RC4-Hash, SHA-2 en Tiger, en het blokcijfer KeeLoq, dat onder meer gebruikt wordt in anti-diefstalsystemen voor auto’s.

Contents

List of Figures

List of Tables

Part I

Analysis and Design of Cryptographic Hash Functions

Chapter 1

Introduction

1.1 Cryptology

Cryptology has a long history, during which it has evolved from an arcane art into a science. For most of its history, cryptology was the domain of diplomats, secret services and the military. The methods and devices used were kept strictly secret, lest they fall into the hands of the enemy. Kahn [46] gives a comprehensive account of this early history of cryptology. In the 1970's, cryptology gradually escaped from the shroud of secrecy and mystery that had surrounded it for centuries, and found its way into open research. Some of the catalysts for this sweeping change were the publication of the Data Encryption Standard (DES) [59] in 1977 and the invention of public key cryptography [21].

Cryptology has been — and still is — an important factor in the evolution towards the information society. More and more transactions that used to be carried out in person are being replaced by their digital counterparts, taking place over worldwide networks like the internet. When information is being transmitted over a potentially insecure medium, messages may be intercepted, modified, rerouted, and so forth. One can no longer trust that communications are not being eavesdropped on, or that the information one receives is genuine. Even the identity of the other party in a communication becomes uncertain, as the other party may very well be an impostor. Thus, the digital world creates several new security challenges. For example, a handwritten signature on a contract is difficult to copy accurately — or at least this is assumed to be difficult. In the digital world, on the other hand, a signature is just a series of bits. It is very simple to make a perfect copy of such a series of bits. Cryptology provides us with the necessary technology to regain (some) of the guarantees that could be taken for granted in classical face-to-face transactions. For a comprehensive overview of cryptology in general, we refer to the 'Handbook of Applied Cryptography' by Menezes, van Oorschot and Vanstone [57].

1.2 Confidentiality and Authenticity

When discussing cryptology, most people consider only the problem of protecting the confidentiality or secrecy of information. It is thus not surprising that also historically, this has been the focus of cryptology for a long time. The goal is to ensure that only certain authorised parties have the ability to learn the content of the protected information, while others can not.

To achieve secrecy, the sensitive information or plaintext can be encrypted into a ciphertext. This ciphertext is illegible to anyone, except to those who know how to decrypt it. Following Kerckhoffs' principle [49] from the 19th century, it is assumed that the system or algorithm used to encrypt and decrypt is known to the adversary, except for a secret key. Hence, the ciphertext should be illegible to anyone who does not know this decryption key. In symmetric encryption, the encryption and decryption keys are the same. Asymmetric encryption uses a different key for the two operations. The encryption key is made known to anyone, hence it also called the public key, while the decryption key or private key is kept secret. Anyone can encrypt messages using the public key, but only the intended recipient knows the private key and is thus the only one able to decrypt.

Often, however, the authenticity of a message is more important than its secrecy. Consider for instance financial transactions. While secrecy of such transactions is important for reasons of privacy, their authenticity is paramount. If information on financial transactions leaks to the public, this may cause an embarrassment to the affected bank, but this is not nearly as disastrous as undetected fraudulent transactions. It is thus important to ensure that all transactions that are carried out have originated from a properly authenticated party, and have not been tampered with. Note that two types of authentication are required: authentication of the sender (entity authentication) as well as the message itself (data authentication). For a long time, it was believed that the problem of authenticity was automatically solved by encryption. The idea was that anyone capable of generating a ciphertext that decrypts to a meaningful plaintext surely must know the encryption key. This is not true, however. A trivial counterexample is the Vernam cipher or one-time pad. While this encryption scheme is proven to be unconditionally secure, an attacker only has to change one bit of the ciphertext to flip the corresponding plaintext bit.

Cryptographic hash functions play an important role in the protection of the authenticity of information. Simply put, the hash result they generate can be considered as a fingerprint of the message, and hence they reduce the problem of protecting the authenticity of a long message to protecting the authenticity of a short hash value. For a secure cryptographic hash function, it is not feasible to find a collision, i.e., two distinct messages with the same hash result, or to find a preimage, i.e., a message corresponding to a given hash result. Because of these properties, one can assume a one-to-one correspondence between messages and hash results.

1.3 Cryptanalysis

Cryptology consists of two closely related subdisciplines: cryptography and cryptanalysis. Cryptography concerns the research on designing new algorithms, and devising new applications. Cryptanalysis attempts to break these algorithms, i.e., construct attacks that subvert their security. There is a continuing reciprocity between cryptography and cryptanalysis, as new designs need to be analysed, and novel analyses give ideas for new designs to resist them.

Some cryptographic schemes and protocols can be formally proven to be unconditionally secure. This implies that they can not be broken, not even if the adversary would have an unlimited computational power at his or her disposal. For other schemes, it can be proven that their security reduces to some hard mathematical problem. Such proofs show that any attack on the scheme can be transformed into an algorithm to solve the underlying hard mathematical problem. Hence, breaking the scheme cannot be easier than solving the underlying problem, which is considered to be hard.

However, often, the luxury of such strong formal proofs of security is not available. Especially in symmetric cryptology, proofs of security are the exception rather than the rule. In such cases, we must resort to practice oriented methods and attempt to develop attacks on cryptographic algorithms. There exist generic attacks that apply to all cryptographic algorithms of a certain type, e.g., all block ciphers, or all hash functions. The computing power required for such attacks can be estimated, and the parameters of the algorithms are chosen such that these attacks are well beyond what is feasible in practice. If a cryptanalyst succeeds to develop an efficient shortcut attack, that considerably outperforms any generic attack, this is a clear indication that the required security goals are not being met, and the algorithm is considered broken. Cryptanalytic attacks yield valuable insights that can be used to avoid similar attacks in the design of subsequent algorithms. For certain types of cryptanalytic attacks, design strategies have been developed for which it can be proven that the attack methodology does not apply. For instance, the 'Wide Trail' design strategy, which was used to design the Advanced Encryption Standard (AES) [16], offers provable resistance against (plain) linear and differential cryptanalysis.

Note however that the absence of attacks does not guarantee that a scheme is secure. Perhaps no attacks have been found because no one has attempted to find any. In order to build confidence in the security of a cryptographic algorithm, it needs to be evaluated for a sufficiently long time. This evaluation should be performed, not only by the designers of the algorithm, but also by independent cryptanalysts. It is common practice to also consider reduced or weakened variants of cryptographic primitives for the purpose of cryptanalysis. If the full primitive can not be broken, but some reduced variant can, this gives an indication of the security margin offered by the full primitive, and can help to establish confidence in its security.

1.4 About this Dissertation

This dissertation is based on publications, and consists of two parts. The first part gives a general introduction to the field cryptology and cryptographic hash functions, and introduces relevant concepts. It also provides a brief outline of our contributions to the design and analysis of cryptographic hash functions. The second part consist of a selection of our publications, reproduced as they were originally published. For a detailed list of publications, see p. 53.

This first part consists of five chapters. This chapter, Chapter 1, introduced the field of cryptology in a nutshell. Chapter 2 focuses on cryptographic hash functions. Cryptographic hash functions and their relevant properties are defined, and certain important known results related to hash functions are discussed. However, it is not intended to be a comprehensive overview of the relevant literature. Chapters 3 and 4 give a summary of our contributions to the design and analysis of cryptographic hash functions. The main purpose of these chapters is to illustrate how our publications fit together, and how they relate to other work in the field. Finally, Chapter 5 concludes and discusses possible directions for future work.

Chapter 2

Cryptographic Hash Functions

2.1 Introduction

This chapter aims to give a brief introduction to cryptographic hash functions and their most important properties and applications. For an in-depth discussion of hash functions, we refer to the treatment of Preneel in [69].

A hash function is an efficient, deterministic algorithm that maps an input of arbitrary length into an output of fixed length, see Fig. 2.1. Often, the term message is used to denote the input of a hash function, and the output is called the digest, the fingerprint, or the hash value. A hash function thus associates any message with a certain fingerprint, which can be used to represent the message. For this purpose, it is desirable that no two messages have the same fingerprint, i.e., that so-called collisions do not occur.

However, it is impossible to avoid collisions, as there is only a limited number of possible hash values. Even if the input is limited to some (large) maximum length, which is often done for practical reasons, the number of possible inputs is still far greater than the number of possible outputs, hence collisions must exist. Consider for instance the hash function SHA-256 [62], which generates digests of 256 bits and accepts input messages with a length from 0 bits up to $2^{64} - 1$ bits. There are 'only' 2^{256} possible hash values, but the number of possible inputs is $\sum_{i=0}^{2^{64}-1} 2^i \approx 2^{2^{64}}$, which is far greater. Thus, while collisions unavoidably exist in any hash function, for a cryptographic hash function, it should not be feasible to find them.

Note that there are also plain, non-cryptographic hash functions. These do not satisfy such stringent requirements. For instance, with respect to collisions, for a non-cryptographic hash function it is sufficient if collision among random messages are rare enough to not be of practical concern. These non-cryptographic hash functions are among others used in data structures such as hash tables [51]. In this work, only cryptographic hash functions are considered.

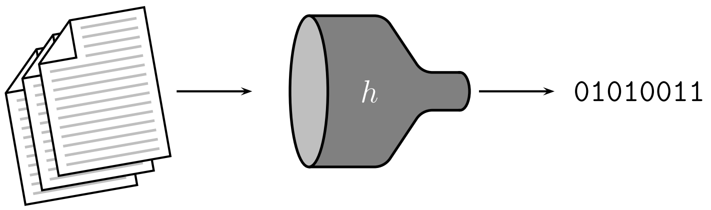


Figure 2.1 – A hash function compresses an arbitrary-length input message into a short, fixed length output.

The security requirements that separate cryptographic hash functions from their non-cryptographic counterparts form the topic of Sect. 2.2.

An attack on a cryptographic hash function, or a cryptographic algorithm in general, shows that a certain security property is not achieved. For instance, a collision attack on a hash function consists of an efficient algorithm to find a collision. The existence of such an efficient algorithm clearly shows that it is not infeasible to find collisions for this particular hash function. Thus, it is not collision resistant, and considered broken.

2.2 Security Requirements

Cryptographic hash functions are required to satisfy a number of security properties. Which of these properties is important, depends on the application the hash function is used in. It is generally accepted, though, that a cryptographic hash function has to satisfy the following three main security requirements:

- Preimage resistance,
- Second preimage resistance, and
- Collision resistance.

There are other security requirements, e.g., a cryptographic hash function should destroy any algebraic structure in its input. Ideally, a cryptographic hash function should behave like a 'random oracle' [10]. A random oracle is an idealised component that outputs a random string for every new input. If the same input is repeated, the same output value is returned. This idealised concept is used in security proofs for protocols and applications built on hash functions. It is proven that the system is secure, provided that the hash function behaves like a random oracle. Since a true random oracle does not exist, the best that can be achieved is that it is not feasible to distinguish the hash function from a random oracle.

For simplicity, we focus only on the three basic security requirements for a cryptographic hash function mentioned above. For a theoretical treatment of security notions for hash functions, we refer to [79, 84].

2.2.1 Preimage Resistance

A hash function is preimage resistant or one-way if it is ‘hard’ to invert, i.e., given an output y it is ‘hard’ to find an input x such that $h(x) = y$.

To avoid degenerate cases, it is required that y is a valid output, i.e., $\exists x : h(x) = y$. The following example shows such a degenerate case. Consider the hash function $h(x) \equiv 0$, i.e., a hash function that always outputs 0. Let $y \neq 0$. It is clearly impossible to find a preimage for y, as there is no x such that $h(x) = y \neq 0$.

There are several variations of the notion of preimage resistance that capture a number of subtleties in the definition of preimage resistance. For instance, what if a hash function is hard to invert, except for a certain weak output $y^\star$? Rogaway and Shrimpton [79] formalised the definitions of three types of preimage resistance: ‘Pre’, ‘aPre’ and ‘ePre’ and studied how they relate to each other, and to different security notions.

2.2.2 Second Preimage Resistance

A hash function is second preimage resistant if it is ‘hard’ to, given an input x, find a second input $x' \neq x$ such that $h(x) = h(x')$.

Note that there is only a subtle difference between preimage resistance and second preimage resistance. From a practical point of view, this difference can be very significant. Many hash functions have an output transformation, or some padding rule that restricts the freedom of an adversary in the final part of the hash computation, making it more difficult to invert. A (first) preimage attack needs to overcome these difficulties, whereas a second preimage attack can simply copy these parts from the challenge message.

Rogaway and Shrimpton [79] also studied formal definitions for second preimage resistance and, similar to preimage resistance, formally define three distinct notions: ‘Sec’, ‘aSec’ and ‘eSec’. They show several implications between the various notions, e.g., the default second preimage notion ‘Sec’ implies the default (first) preimage notion ‘Pre’, but not vice versa. This theoretical result agrees with the observation mentioned above that a hash function can be (first) preimage resistant, while being vulnerable to a second preimage attack.

2.2.3 Collision Resistance

For a collision resistant hash function, it must be ‘hard’ to find two distinct inputs $x \neq x'$ such that $h(x) = h(x')$. Note that this is not the same as second preimage resistance as here, both x and x' can be chosen by the adversary.

According to some, e.g., Rogaway [78], there is a problem with this definition. If a collision pair is known, a trivial adversary can be constructed that just outputs this pair. Moreover, as collisions always exist, it exists in theory for any hash function. Thus, if ‘hard’ is interpreted as ‘there exists no efficient adversary’ — as is often done in theoretical works — there is indeed a conceptual problem. The typical way around this, is to introduce a key to the hash function. The term ‘key’ is perhaps an unfortunate choice, as these keys are not in any way secret. Rather they are indices that designate a particular hash function as a member of a large family. The notion of collision resistance then becomes: given a randomly chosen key, find a collision for the member of the hash function family indexed by this key. Rogaway [78] proposed an another way to sidestep the issue, without the need for keys, based on formalising the concept of human ignorance. By requiring explicit reductions, the existence of such an adversary is no longer a problem, as long as it can not be written down. This is closer to the practical interpretation of ‘hard’.

2.3 Applications

Hash functions are used in such a wide variety of applications that they have been dubbed the ‘Swiss army knives of cryptography’. This section briefly introduces a number of applications of cryptographic hash functions.

Probably the most well known example is the optimisation of digital signatures. For instance, a hash functions is an essential part of the ‘Digital Signature Algorithm’ (DSA) [64]. Digital signatures schemes are based on public key cryptography, e.g., RSA [77]. A signature on a message is generated using the signer’s private key and signatures can be verified using the public key of the signer. The algorithms used are typically several orders of magnitude slower than symmetric algorithms, making straightforward digital signatures prohibitively slow for long messages. This issue can be solved through the use of a hash function. A message to be signed is first hashed — which is a fast operation even for long messages — and then the hash value is signed. Since hash values are short, this construction also results in compact signatures. An additional advantage is that the hash function usually destroys the algebraic structure of the message space, and may thus prevent attacks on the asymmetric algorithm. Of course, the security of the hash function is critical to the security of the scheme. If an adversary can generate a collision, and manages to convince the victim to sign one of the messages, the signature will also be valid for the other, colliding message. Similarly, forgeries can be constructed if a second preimage can be found for any signed message.

Another common application is password hashing. It is not desirable to store the passwords of users in a password file, as this file may be exposed. Instead of storing the user’s passwords, a hash value of each password is stored. When a user wishes to authenticate, the entered password is hashed, and its digest is compared to the correct digest from the password file. But reconstructing a password from the password file requires finding a preimage. Hash chains or Lamport chains [53] are an extension of this idea that can be used for one-time passwords.

Many cryptographic protocols use hash functions as a building block. Hash functions can be used to derive short lived keys from long term secrets (key derivation), to generate pseudorandom information from a key, or to confirm knowledge of a shared secret, e.g., a session key, without revealing this secret.

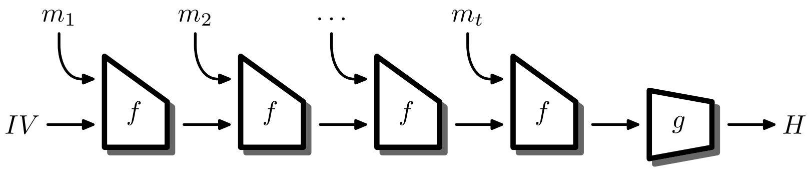


Figure 2.2 – An iterated hash function.

They can also be used to commit to some information that is to be revealed only at a later time.

Another application of hash functions is to detect tampering of digital files. Any modification results in an easily detectable change in the hash digest, except if the adversary succeeds to construct a second preimage. Using the HMAC construction [8], a 'Message Authentication Code' (MAC) — a function using a secret key to authenticate messages — can be built from a hash function.

2.4 Iterated Hash Functions

Hash functions map a variable length input into a fixed length output. A natural way to achieve this, is to construct a hash function h by iterating a compression function f with a fixed length input. Early hash functions, such as Rabin's hash [71] were iterated, as are the vast majority of hash functions that have been proposed since. Iterated hash functions have the advantage that the message can be processed in a single pass, and need not be kept in memory in its entirety.

In general, an iterated hash function consists of three parts: a padding routine, a compression function f and an (optional) output transformation g. The padding routine takes the input message M of variable length and extends it to a multiple of the block length. The padded message is then split into t blocks of equal size, m_1 through m_t:

$$(m_1, \cdots, m_t) \quad \leftarrow \quad \mathrm{pad}\,(M) \ . \tag{2.1}$$

Then, the message is processed by applying the compression function f iteratively for each padded message block, starting from a fixed IV or initial value. Finally, the hash result is generated from the last chaining value h_t using the (optional) output transformation g (see Fig. 2.2):

$$\begin{cases} h_0 &= IV \ , \\ h_i &= f(h_{i-1}, m_i) \qquad \text{for} \quad 1 \le i \le t \ , \\ h(m) &= g(h_t) \ . \end{cases} \tag{2.2}$$

In many designs, the output transformation g is omitted, or consists of a simple truncation to the desired output length. The use of an output transformation can be useful in preventing (first) preimage attacks and length extension attacks. A length extension attack aims to compute $h(x \,||\, y)$ when given $h(x)$, but not x.

In the absence of an output transformation, a trivial length extension attack is possible, as the last chaining value is equal to $h(x)$ and thus known. An adversary can simply append arbitrary message blocks and resume the hash computation to find $h(x \,||\, y)$ without requiring the knowledge of x.

2.4.1 The Merkle-Damgård Construction

In 1989, Damgård [17] and Merkle [58] independently showed an iterated construction that can be proven to be collision resistant if the compression function is collision resistant. In other words, the Merkle-Damgård construction transfers the collision resistance of the compression function to the iterated hash function.

The key difference between the Merkle-Damgård construction and a plain iterated hash function, which was described above, lies in the message padding. The Merkle-Damgård construction includes the message length in the last padded block, which makes the padding routine suffix free, i.e., a padded message can not be a suffix of another, distinct padded message. This practice is called 'Merkle-Damgård strengthening' since Lai and Massey [52]. Many hash functions are constructed using this principle. For instance, (most of) the MD4-family hash functions are padded by appending a single '1' bit, a number of '0' bits, and the message length as a 64-bit unsigned integer. The number of '0' bits is the minimum required for the padded message to consist of a whole number of 512-bit blocks.

Theorem 2.1 (Merkle-Damgård)**.** *Let h be an iterated hash function constructed from the compression function f using the Merkle-Damgård design principle. If the compression f is collision resistant, then also the iterated hash function h is collision resistant.*

Proof. Assume that h is not collision resistant. This implies that there exists an efficient adversary that can find a collision pair (M, M'), i.e., $M \neq M'$ and $h(M) = h(M')$. We distinguish two cases, depending on whether the length of both messages is equal or not.

Case 1: $|M| \neq |M'|$**.** The last compression function call yields a collision as its output is the same for both messages, but its input differs as it includes the message length $|M|$ resp. $|M'|$.

Case 2: $|M| = |M'|$**.** Since the messages collide under h and have the same length, there must be (at least) one compression function call for which a compression function collision is found.

Hence, if h is not collision resistant, f is also not collision resistant. From this, the theorem follows. $\square$

Note that it is assumed here that no output transformation is used. Otherwise, the proof needs to be adapted to take into account the possibility for collisions in the output transformation. For a more detailed discussion of this general case, we refer to Sect. 3.3.4.

2.4.2 Other Constructions

Several improved iteration modes have been proposed, such as EMD by Bellare and Ristenpart [9], HAIFA by Biham and Dunkelman [14], the Sponge construction of Bertoni et al. [12, 13], and ROX by Andreeva et al. [4]. These iteration modes offer more provable properties or resistance against (some of) the generic attacks on plain Merkle-Damgård iterations described in Sect. 2.5.4.

2.5 Generic Attacks

There exists a class of attacks that apply to any cryptographic hash function, regardless of its design. Such attacks are called generic attacks. Due to their generic nature, it is not possible to prevent them. However, the effort required to mount such a generic attack is (only) dependent on the parameters of the hash function. Thus, to protect against generic attacks, the parameters of a hash function are chosen such that the computational effort required by generic attacks is large enough — by some considerable margin — to make them infeasible in practice.

In the definition of the security properties of cryptographic hash function in Sect. 2.2, it was said that it should be 'hard' to find preimages, collisions, etc., but this leaves the question what 'hard' means precisely. A natural definition of 'hard' would be infeasible in practice. However, this is a very vague, and rapidly changing definition. Another option is to use generic attacks as a benchmark. For instance, finding a preimage for a certain hash function is considered to be 'hard' if the best (known) method is a generic preimage attack.

A dedicated attack needs to require significantly less effort than a corresponding generic attack to be of any interest, even if it is still highly impractical. For a secure hash function, the best possible attacks should be these generic attacks, i.e., no shortcut attack should exist. If there are shortcut attacks that outperform generic attacks by a considerable margin, the hash function is considered to be broken.

It is not always very straightforward to compare attacks. If an attack is practical, it can be implemented and run on real computing hardware. Since actual designs are scaled such that generic attacks are not feasible in practice, any attack that is practical surely outperforms the generic attacks. In the case of non-practical or theoretical attacks, however, it is not always straightforward to determine whether or not they are faster than a generic attack.

A crude way to compare attacks is to estimate the number of computations required for the attack to complete with some non-negligible probability. This is called the time complexity, and is typically expressed in units of hash function computations, or compression function evaluations. However, this is not the only important measure when comparing attacks. The memory requirements need to be considered, as well as the memory access patterns. Often, there exists a trade-off between time and memory, but it is not always clear which point on the trade-off curve is optimal. The importance of the memory access pattern is rooted in the fact that large memories can only be accessed with a substantial latency. An attack that has a lower time complexity, but requires random access to a large memory, may be orders of magnitude slower in reality. Finally, the potential for parallelisation is an important factor to consider as well. Some attacks can be parallelised perfectly: running them on m machines in parallel reduces the running time with a factor m. Other attacks are inherently serial, and can not benefit at all from the availability of multiple machines.

2.5.1 Exhaustive Search

A straightforward approach to find (second) preimages for a hash function is a simple exhaustive search. Consider a hash function with an n-bit result. In a preimage attack, the adversary is given the desired n-bit output y, and tries to find a message x, such that $h(x) = y$. As the output is n bits long, a random message hashes to y with a probability of 2^{-n}, under the assumption that each output is equally likely. For all real cryptographic hash functions, this is a very reasonable assumption. After about 2^n random trials, it is expected to find a preimage. Such an attack thus requires about 2^n hash function evaluations and negligible memory. It is trivial to parallelise, as each trial is fully independent, so the performance of the attack scales linearly with the number of processors.

In the case of multiple targets, the attack can be improved. Consider the scenario where a preimage is sought for any of 2^t target digests. The probability that a random trial matches one of the targets is 2^{-n+t}, hence after about 2^{n-t} trials, it is expected that a preimage is found. Because of this simple fact, the preimage security of a hash function degrades in the case where there are multiple targets. In applications like password hashing, the use of salts (or keys) can effectively avoid the undesirable effects of multiple targets.

The same attack can be used to find second preimages. The only precaution that needs to be taken is to ensure that the challenge message x is not chosen as one of the random trials, as that would yield a false positive. Note that this attack can also be used to find collisions by starting from a random message and then finding a second preimage. However, a much more efficient generic collision attack exists, based on the birthday paradox. This is explained in Sect. 2.5.3.

2.5.2 Time-Memory Trade-Offs

Generic time-memory trade-off attacks may be used to quickly construct preimages for hash functions at the expense of an extensive precomputation and a large memory. This type of attack was pioneered by Hellman [29] in 1980. It was later improved by Rivest, as described in [20], and a variant was proposed by Oechslin [67]. The basic idea is to precompute long hash chains, and store the starting and ending points of each chain in memory. To find a preimage in the online phase, the same iteration is performed until a known ending point is found.

Since the corresponding starting point is also stored, the entire chain can be reconstructed. Note that it is not guaranteed that the reconstructed chain will contain the desired preimage, as chains may merge due to collisions.

Time-memory trade-offs can be used to attack password hashing schemes, as they allow an adversary to quickly invert many password hashes after a one-time precomputation. To protect against these attacks, password hashing schemes should use a technique called ‘salting’. For each password, a unique salt or key is randomly generated and stored in the password file. This salt is incorporated into the hash computation, which has the effect that for each password, a different hash function is used. Hence, the precomputation of a time-memory trade-off attack would have to be repeated for each possible salt, which is not feasible.

2.5.3 The Birthday Attack

The birthday paradox states that in a group of at least 23 randomly chosen people, there is more than 50% chance that (at least) two of them have the same birthday. Intuitively, people expect that a significantly larger group of people is required to observe such birthday collisions with a non-negligible probability. Assuming that people’s birthdays are uniformly distributed and ignoring leap years, the probability that N people have a different birthday (with $N \leq 365$) is

$$p'(N) = \prod_{i=0}^{N-1} \left(1 - \frac{i}{365}\right) = \frac{365!}{365^N (365-N)!} \ . \tag{2.3}$$

At least one birthday collision occurs with the complementary probability $p(N) = 1 - p'(N)$. For $N = 23$, the probability surpasses 50%, and as the number of people N increases, it rapidly approaches 100%. This can be seen clearly in Table 2.1, which contains the collision probability for several values of the number N up to 100.

This can be applied to the problem of finding collisions for a hash function. Consider an n-bit hash function, i.e., the hash outputs are n bits long. The probability that two random inputs collide is 2^{-n}, assuming that the output distribution of the hash function is uniform. The probability that there are no collisions in a set of N randomly chosen inputs is

$$p'(N) = \prod_{i=0}^{N-1} \left(1 - \frac{i}{2^n}\right) \approx \prod_{i=0}^{N-1} \exp\left(-\frac{i}{2^n}\right) = \exp\left(-\frac{1}{2^n}\sum_{i=0}^{N-1} i\right) \ . \tag{2.4}$$

Here, the approximation $\exp(-x) \approx 1 - x$ (for small x) was used. Note that this is a good approximation when $N \ll 2^n$. The probability that a collision occurs is then given by

$$p(N) = 1 - p'(N) \approx 1 - \exp\left(-\frac{1}{2^n} \cdot \frac{N(N-1)}{2}\right) \approx 1 - \exp\left(-\frac{N^2}{2^{n+1}}\right) \ . \tag{2.5}$$

Table 2.1 – The birthday paradox.

Number of people	Collision probability
10	11.69 %
20	41.14 %
23	50.73 %
30	70.63 %
40	89.12 %
50	97.04 %
60	99.41 %
70	99.92 %
80	99.991 %
90	99.9994 %
100	99.99997 %

The expected number of randomly chosen messages that need to be hashed before a collision is found, is 'only' $\sqrt{\pi/2} \cdot 2^{n/2}$ [25, 85]. Thus, the effort required to find a collision using the birthday attack is, ignoring the constant, only the square root of a naive exhaustive search.

The simplest variant of the birthday attack thus proceeds as follows. Choose about $\sqrt{\pi/2} \cdot 2^{n/2}$ random messages, compute their hash value, and store them in a list. Given the size of the list, it is expected to contain a collision. Finding the collision can be done efficiently by sorting the list. Alternatively, an explicit sorting step can be avoided through the use of an appropriate data structure for the list, for instance a hash table. The time complexity of this attack is $\Theta(2^{n/2})$ hash function evaluations. The memory requirements are given by the size of the list, and thus also $\Theta(2^{n/2})$ memory elements.

Meaningful Collisions

Note that this attack offers a great deal of freedom to the adversary, as the messages can be chosen freely. Instead of finding just two random colliding messages, an adversary can try to find a collision between variations of a 'good' message and variations of a 'bad' message. After such a collision is found, the adversary convinces the victim to sign the good message. This signature will then also be valid for the bad message, as both messages have the same hash value. The only difference to the collision finding algorithm is that a collision is only useful with probability 1/2. However, as this can be seen as a halving of the collision probability, it can be offset by increasing the number of messages by a factor $\sqrt{2}$.

Shortcut attacks are typically much more restrictive in the types of messages they allow, and typically require at least parts of the message to be seemingly random gibberish. Still, there are several examples in the literature of how to construct meaningful colliding messages based on shortcut attacks allowing only very limited control over the message. For instance, Stevens et al. succeeded to create X.509 certificates that collide under MD5 [82], and even constructed a rogue 'Certification Authority' (CA) certificate that is trusted by web browsers, based on an advanced collision attack on MD5 [83].

Memoryless Birthday Attack

A more advanced variant of the birthday attack aims to eliminate the large memory requirements. Consider an n-bit hash function $h(x)$, and let $f(x) = h\left(g\left(x\right)\right)$. Here, the function $g(x)$ maps an n-bit string into a suitable message. For instance, $g(x)$ could output a variation of a meaningful message in natural language, where the input bits introduce small changes, such as typing errors or substituting certain words with synonyms. It is also possible to search for collisions between variations of a 'good' and a 'bad' message using an appropriate definition for $g(x)$. But, for simplicity, $g(x)$ can be chosen to be the identity function, so $f(x) = h(x)$.

Consider the pseudorandom walk through the set of n-bit strings generated by iterating $f(x)$ starting from some random starting point $x_0 \in \{0,1\}^n$. Since the set is finite, the sequence must repeat eventually. Hence, the graph of such a walk will look like the Greek letter ρ. At the entry of the cycle, we find a collision for $f(x)$, as there are two different points $x_i \neq x_j$, which map to the first point of the cycle: $f(x_i) = f(x_j)$.

Using Floyd's cycle finding algorithm [50], collisions can be found using only negligible memory. The algorithm is based on the same pseudorandom walk, but considers two sequences starting from the same point x_0. For the first sequence, $f(x)$ is applied once per step, as before. The second sequence advances twice as fast, i.e., $f(x)$ is applied twice per step. When both sequences meet in some point $x_v = x_{2v}$, the second sequence has gone through the same steps as the first, plus an additional number of cycles. Hence, the cycle length μ divides the difference in steps, which is v. Now, start iterating again from x_0 and x_v, using a single application of $f(x)$ per step for both sequences. As the sequences are an integer number of cycle lengths apart, they will always be in the same point, as soon as the first sequence enters the cycle. Thus, they will first meet at the entrance point of the cycle, $x_\lambda = x_{\lambda+v}$, where λ is the length of the 'tail' of the ρ-shape.

The expected values of the tail length λ and the cycle length μ are both $\sqrt{\pi/8} \cdot 2^{n/2}$ [25, 85]. In the best case, $v = \mu$ and Floyd's algorithm requires $3\mu + 2\lambda$ computations of the function $f(x)$. Thus, in the best case, its expected time complexity is $\frac{5}{2}\sqrt{\pi/2} \cdot 2^{n/2}$, which is 2.5 times larger than the standard birthday attack. But, the memory requirements are negligible. Nivasch's algorithm [66] requires less time than Floyd's algorithm, at the expense of logarithmic memory.

Parallel Birthday Attack

Running Floyd's algorithm in parallel on m processors only gives a factor $\sqrt{m}$ improvement in runtime [85], which is rather inefficient. An efficient parallel variant of the birthday attack, which gains a factor m, was described by van Oorschot and Wiener [85].

Each processor independently selects a random starting point, and iterates the function $f(x)$ until a distinguished point is encountered. A distinguished point is a point that has some property that can be checked easily, e.g., a certain number of leading zero bits. When a distinguished point is found, the processor transmits it to a central database, together with the starting point and the trail length. As the fraction of distinguished points is small, the amount of information that needs to be kept centrally is limited. When the same distinguished point is found a second time, this implies that two trails have been found that end in the same point. If the starting points differ, the trails must merge at some point. At this merging point, a collision for $f(x)$ is found. As the lengths and starting points of both trails are known, the collision can be recovered easily.

A potential problem is that false positives may occur. When one trail contains the starting point of another trail, they will both end in the same distinguished point, but no collision can be found from them. However, when the fraction of distinguished points is small, the trails are long and false positives are unlikely. Another problem is that a trail could enter a cycle that does not contain a distinguished point. Such endless loops can be prevented by limiting the maximum length of a trail to a suitable value.

2.5.4 Generic Attacks on Iterated Constructions

Several attacks have been proposed that are generic in the sense that they only rely on certain limitations of common iterated hash function constructions. Joux [45] showed how to efficiently find multicollisions: k colliding messages can be found with an effort of only $\Theta(\lceil \log_2(k) \rceil 2^{n/2})$. The long message second preimage attacks of Dean [19] and Kelsey and Schneier [48] demonstrate that generic second preimage attacks become easier as the challenge message is longer. A second preimage attack on a message of 2^k blocks has a complexity of only $\Theta(2^{n-k})$ instead of $\Theta(2^n)$. The herding attack by Kelsey and Kohno [47] allows an adversary to commit to a hash value of a yet unknown message after performing a precomputation step. Later, the adversary can, for any message, compute a suffix such that it matches the commitment with an effort significantly below the $\Theta(2^n)$ operations one would expect. Andreeva et al. [2,3] proposed several further extensions and generalisations of these ideas.

2.6 Conclusion

This chapter introduced cryptographic hash functions, their properties and applications. Considerable attention was given to generic attacks on the various security requirements of cryptographic hash functions. As generic attacks apply to any hash function, regardless of its design, they provide a useful benchmark to compare dedicated attacks to.

Chapter 3

Design of Cryptographic Hash Functions

3.1 Introduction

This chapter describes our contribution to the design of cryptographic hash functions: the hash function Lane. First, a short sketch of the history of hash function designs is given in Sect. 3.1.1, leading to the NIST SHA-3 competition for cryptographic hash function designs, see Sect. 3.1.2. Then, Sect. 3.2 introduces Lane, our submission to the SHA-3 competition. Finally, Sect. 3.3 discusses the relevance of pseudo-collisions, addressing questions that were raised by independent cryptanalysis of Lane.

3.1.1 History and State of the Art

Historically, most cryptographic hash function designs were based on block ciphers, e.g., the early DES-based hash function proposed by Rabin [71] in 1978. Preneel et al. [70] studied in a systematic way how a block cipher can be used to construct a hash function whose output size corresponds to the block size of the cipher. In 1990, Rivest designed the dedicated hash function MD4 [72]. The most widely used hash functions today are descendants of MD4. Because of their similarity in design, they are often called the MD4 family. Actually, one could still think of MD4 as a block cipher based hash function, except that it is not based on a pre-existing block cipher. Rather, MD4 contains a block cipher that was designed specifically for the MD4 hash function. Following a number of attacks on MD4, Rivest proposed MD5 [74], a strengthened version of MD4, in 1991. Both MD4 and MD5 were adopted as standards by the Internet Engineering Task Force (IETF) [73,74], and were consequently used in numerous applications. The current state of the security of MD4 and MD5 is not very good, as highly practical collision attacks were shown, e.g. [82, 83, 87, 89].

The National Institute of Standards and Technology (NIST) published the first version of its 'Secure Hash Standard' (FIPS 180) [62] in 1993. It contained one algorithm: the 'Secure Hash Algorithm' or SHA. First made public in 1992, the design of SHA was clearly inspired by MD4 and MD5. In 1995, SHA was replaced by SHA-1. The difference between SHA-1 and the original SHA, which is often referred to as SHA-0, is very small. At that time, it was not known what the motivation for this change was, but later, cryptanalytic results have shown that SHA-0 is significantly weaker than SHA-1. Other hash functions of the MD4 family include RIPEMD, and its successors RIPEMD-128 and RIPEMD-160 [22], and HAVAL [93]. The most recent addition to the MD4 family are the SHA-2 hash functions [62]: SHA-224, SHA-256, SHA-384 and SHA-512.

The ISO/IEC standard on dedicated hash functions, ISO/IEC 10118-3:2004 [43], contains seven hash functions. In addition to the already mentioned algorithms RIPEMD-128, RIPEMD-160, SHA-1, SHA-256, SHA-384 and SHA-512, the standard includes the hash function Whirlpool which, unlike the other six, does not follow the MD4 design strategy. Instead, Whirlpool is a block cipher based construction, built on a conservative, special purpose block cipher called W. This block cipher W was designed following the 'Wide Trail' strategy [16].

3.1.2 The NIST SHA-3 Competition

The National Institute of Standards and Technology's (NIST) 'Secure Hash Standard' [62] contains the cryptographic hash function SHA-1 and the four members of the SHA-2 family: SHA-224, SHA-256, SHA-384 and SHA-512. Currently, SHA-1 is in bad shape, and has been since at least 2005 [18, 88]. Even though almost five years later, still no collision examples for SHA-1 have been shown, there is hardly any trust left in the security of the SHA-1 hash function. This also reflects on the SHA-2 family. Even though the most advanced collision attacks on SHA-256 known at this time target a mere 24 out of 64 steps [33, 80], there is very little confidence in the long-term security of SHA-2. This can be explained by the similarities in design and structure between the SHA-2 family of hash functions and its predecessors, such as MD5 and SHA-1.

These considerations have led NIST to initiate the development of a new hash function standard, that will be called SHA-3, to serve as a backup solution in case the security of SHA-2 would fail in the future. In November 2007, NIST issued a call for candidate algorithms to participate in an international competition for cryptographic hash functions [60]. This call for contributions was the official start of the SHA-3 competition [61], which is expected to continue until 2012. The idea of organising such an international competition to develop a new cryptographic algorithm is not new. A very similar process led to the selection of the Advanced Encryption Standard (AES) in 2001 [16].

Table 3.1 – The second round SHA-3 candidates.

Name	Principal Submitter
BLAKE	Jean-Philippe Aumasson
Blue Midnight Wish	Svein Johan Knapskog
CubeHash	Daniel J. Bernstein
ECHO	Henri Gilbert
Fugue	Charanjit S. Jutla
Grøstl	Lars R. Knudsen
Hamsi	Özgül Küçük
JH	Hongjun Wu
Keccak	The Keccak Team
Luffa	Dai Watanabe
Shabal	Jean-François Misarsky
SHAvite-3	Orr Dunkelman
SIMD	Gaëtan Leurent
Skein	Bruce Schneier

On October 31st 2008, the deadline for the submission of candidate algorithms, NIST had received a total of 64 proposals from teams across the entire world. Of these 64 candidates, 51 were accepted to the first round of evaluations. We have designed one of these first-round candidates: the hash function Lane [31]. On 24th July, 2009 NIST announced the 14 candidates that were accepted to the second round of evaluations [63], see Table 3.1. Our candidate, Lane, was not accepted to the second round of the SHA-3 competition.

3.2 The Lane Hash Function

Lane is an iterated cryptographic hash function supporting digest size ranging from 224 bits to 512 bits. The aims of Lane are to be secure, easy to understand, elegant and flexible in implementation. Lane can take advantage of the parallelism offered by modern high-performance CPUs, but also scales down to embedded systems. Another advantage of Lane is the fact that each element of its design is supported by a clear design rationale. Furthermore, Lane has undergone a extensive security analysis. For a comprehensive treatment of Lane, we refer to [31] (see p. 153).

The iteration mode of Lane is a simple and straightforward iteration mode, similar to Merkle-Damgård, but with a number of modern enhancements, such as the use of a counter as in HAIFA [14], and an optional salt to support randomised hashing [28]. The chaining values are of the same length as the digest, i.e., Lane is a narrow-pipe design.

The compression function of Lane has a novel structure based on a light message expansion and parallel permutations. The permutations are based on components from the AES block cipher [16]. More precisely, entire rounds of AES are used as building blocks. This has the advantage that experience from the AES can be reused in the analysis and implementation of Lane. The message expansion is kept simple and lightweight on purpose. Thanks to a coding theoretic bound, it can be proven that the message expansion of Lane precludes a type of meet-in-the-middle attack. It also provides a useful bound with respect to differential cryptanalysis. For a discussion of the rationale behind the design of the message expansion of Lane, we refer to [37] (see p. 143).

Lane offers a multitude of options to the implementer, from fast massively parallel implementations to small implementations fitting in resource constrained environments. The fact that Lane uses AES rounds as building blocks enables dedicated AES hardware units to be employed in implementations of Lane. For instance, Intel has included a new instruction set called AES-NI [41] in their latest microprocessors to perform hardware accelerated AES encryption. It is possible to make a fast implementation of Lane taking advantage of these instructions. As processors supporting AES-NI were not yet available commercially when Lane was designed, we had to resort to a rough estimate of the performance benefit. We estimated that, for Lane-256, a performance of about 5 clock cycles per message byte could be achieved using the AES-NI instructions [31]. Benadjila et al. [11] published a more accurate estimation of the performance potential of various AES-based SHA-3 candidates on processors with AES-NI support. Their estimations are based on experiments with different, existing instructions that have a performance profile that matches the new AES-NI instructions as closely as possible. They estimate the performance of Lane-256 at 5.5 cycles per byte, which is remarkably close to our rough estimate, and would make Lane one of the fastest SHA-3 candidates on these processors.

3.2.1 Independent Cryptanalysis of Lane

Two independent analyses of the security of Lane have been published. Both are based on the 'rebound attack', which is a new cryptanalytic technique aimed mainly at byte-oriented symmetric cryptographic primitives. The rebound attack was introduced by Mendel et al. [56] at FSE 2009, several months after the submission deadline of the SHA-3 competition. The initial hash functions targeted by the rebound attack were weakened variants of Whirlpool and round-reduced versions of the SHA-3 candidate Grøstl [26]. The basic idea behind the rebound attack is to use an efficient match-in-the-middle technique called the 'inbound phase' to satisfy the low probability part in the middle of a truncated differential path. It is in this inbound phase that the degrees of freedom available to the adversary are used primarily. The inbound phase is then followed by a purely probabilistic 'outbound phase'.

Wu et al. [90] were the first to apply the techniques of the rebound attack to reduced versions of Lane. They give a semi-free start collision attack on reduced Lane-256, where the number of rounds in the permutations is halved from 6 to 3. A semi-free start collision attack is a weaker variant of a collision attack, in which the attacker can choose the initial chaining value that is used. Wu et al. claim a time complexity of 2^{62} compression function evaluations and require a memory of 2^{69}. For reduced Lane-512, they show a semi-free start collision attack (time 2^{62} and memory 2^{69}) and a collision attack (time 2^{94} and memory 2^{133}) for 3 out of 8 rounds, and a semi-free start collision attack (time 2^{254} and memory 2^{261}) for 4 out of 8 rounds. While the time complexity of these attacks is, except for the last attack, significantly below the birthday bound of 2^{128} resp. 2^{256}, the memory requirements are very high.

Matusiewicz et al. [54] present attacks on the full Lane compression function. Their attacks are also based on the basic technique of the rebound attack, but it is augmented with several novel ideas. For Lane-256, they show how to construct semi-free start collisions with a time complexity of 2^{96} and a very large memory requirement of 2^{88}. For Lane-512, semi-free start collisions can be constructed in time 2^{224} with an even higher memory requirement of 2^{128}.

These attacks represent a great cryptanalytic advance in the development of the rebound attack technique. Due to their very large memory requirements, however, it is questionable whether they are more efficient than generic attacks. Furthermore, the memory access pattern is not considered in [54]. The attack on Lane-256, as it is described in [54], has a random memory access pattern, resulting in a tremendous slowdown due to memory latency, as discussed in Sect. 2.5. This can be avoided, but then the memory complexity increases from 2^{88} to 2^{96}. Also note that a generic parallel birthday attack can achieve the same time complexity of 2^{96} with 'only' 2^{32} independent parallel processors. Although $2^{32} \approx 4,29 \times 10^9$ is a large number, it is certainly more reasonable than $2^{88} \approx 3,09 \times 10^{26}$ memory.

Furthermore, a semi-free start collision attack targets only the compression function. It can not, in general, be extended to a collision attack on the hash function. Since the compression function is never intended to be used in isolation, only attacks on the entire hash function have any relevance to the security of applications. While compression function attacks are useful for our theoretical understanding of security, the do not necessarily threaten the security of the hash function. Matusiewicz et al. seem to agree with this, as they conclude their paper as follows [54]:

> "Although these collisions on the compression function do not imply an attack on the hash functions, they violate the reduction proofs of Merkle and Damgård, or Andreeva in the case of Lane. However, due to the limited degrees of freedom, a collision attack on the hash function seems to be difficult for full round Lane."

It is indeed true that the conditions for the Merkle-Damgård proof, see Theorem 2.1, are violated by a compression function collision attack. In Sect. 3.3

we explore the impact of this on the security of LANE, and show that the proof can be adapted such that the compression function is no longer required to be collision resistant, but that a weaker notion is sufficient.

NIST did not provide any motivation for their decision to not select LANE for the second round of the SHA-3 competition. But it is likely that LANE was not selected because of the compression function attacks of Matusiewicz et al. [54].

3.3 On Pseudo-Collisions

The classical Merkle-Damgård proof [17, 58] shows that an iterated cryptographic hash function following the Merkle-Damgård design paradigm is collision resistant if its compression function is collision resistant; see Theorem 2.1. Collision resistance of the compression function is thus a sufficient condition, but is it also a necessary condition?

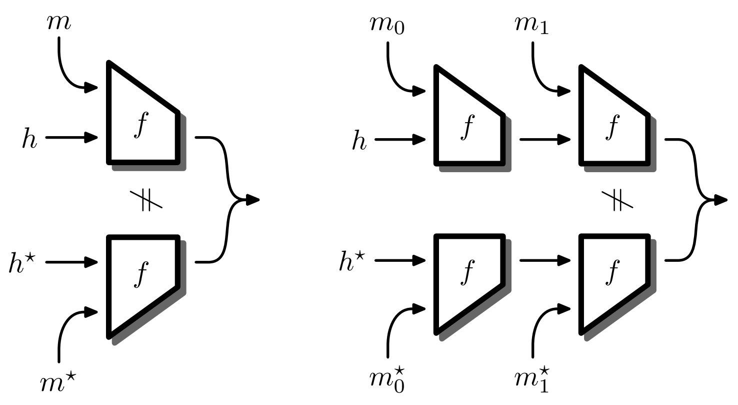
3.3.1 Pseudo-Collision Attacks

A pseudo-collision attack on an iterated hash function is a variant of a collision attack where the attacker is given the additional freedom to choose the initial value for both messages. Lai and Massey [52] further distinguish two types of pseudo-collision attacks: semi-free start and free start collision attacks. In a semi-free start collision attack, the attacker is allowed to choose the initial chaining value, but the same value should be used for both messages. In a free-start collision attack, a (small) difference may appear in the initial chaining value. Often, this terminology is also used to denote compression function attacks where direct control over the chaining input is required for the attack. Note that a (pseudo-)collision attack on the compression function directly leads to a pseudo-collision attack on the iterated hash function by using the compression function attack in the first message block.

Clearly, an efficient pseudo-collision attack of either type violates the collision resistance of the compression function, and thus the Merkle-Damgård proof (see Theorem 2.1) no longer applies. But there is no known method to construct a collision attack on the entire hash function based only on a pseudo-collision attack. More so, it seems that this is not possible in general. This raises the question if collision resistance of a compression function is really required, or that a different, weaker notion may suffice.

3.3.2 Towards Two-Step Compression Function Collisions

The core of the issue is that a pseudo-collision attack assumes that the adversary has full and direct control over an intermediate chaining value. While a hash function adversary can indeed affect intermediate chaining values by varying earlier message blocks, this is only indirect control. To capture this, we introduce the notion of a ‘two-step compression function collision’.



(a) (b)

Figure 3.1 – A compression function collision (a), and a two-step compression function collision (b).

Definition 3.1. A collision for a compression function $f(h, m)$, see Fig. 3.1 (a), consists of a pair of chaining inputs $\langle h, h^\star \rangle$ and a pair of message blocks $\langle m, m^\star \rangle$ such that

- a *collision* is reached: $f(h, m) = f(h^\star, m^\star)$, and
- the compression function is *active*, i.e., its inputs are not equal: $h \,||\, m \neq h^\star \,||\, m^\star$.

A two-step compression function collision is an extension of this definition that captures the fact that there is only indirect control over the chaining input by requiring it to be the output of another compression function call.

Definition 3.2. A two-step collision for a compression function $f(h, m)$, see Fig. 3.1 (b), consists of a pair chaining inputs $\langle h, h^\star \rangle$ and two pairs of message blocks $\langle m_0, m_0^\star \rangle$ and $\langle m_1, m_1^\star \rangle$ such that

- a *collision* is reached: $f\left(f\left(h, m_0\right), m_1\right) = f\left(f\left(h^\star, m_0^\star\right), m_1^\star\right)$, and
- the second compression function is *active*: $f(h, m_0) \,||\, m_1 \neq f(h^\star, m_0^\star) \,||\, m_1^\star$

In Sect. 3.3.3 we show that, for the iteration mode used in Lane, resistance to two-step compression collisions is a sufficient condition for collision resistance of the iteration. Section 3.3.4 generalises this to a generic Merkle-Damgård iteration with an output transformation. The reason for first considering the Lane iteration, is that it results in a simpler, more concise proof.

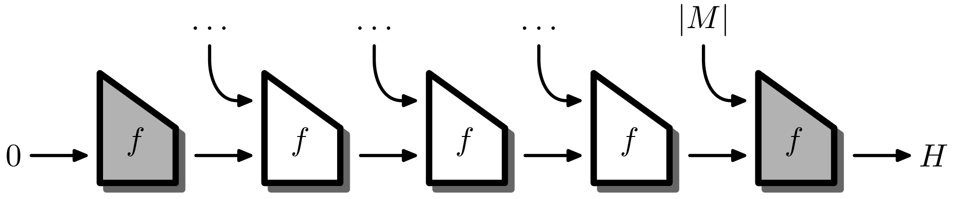


Figure 3.2 – The Lane iteration mode.

3.3.3 The Lane Iteration Mode

The iteration mode of Lane is a simple and straightforward iteration mode, see Fig. 3.2. For a detailed specification, we refer to [31] (see p. 153). The first and the last compression function calls are special, and do not consume any message data. The purpose of the first compression function call is to derive the initial value for the actual iteration. It takes a constant input, and can thus be precomputed. But embedded implementations can opt to recompute it and avoid having to store the initial value. The last compression function call serves as an output transformation. It takes the message length as an input, so Lane performs Merkle-Damgård strengthening in the output transformation. Through the counter input it is ensured that, although the compression function is exactly the same, all compression function calls are distinct and not interchangeable.

We can now prove the following theorem in a way that is completely analogous to the proof of Theorem 2.1 for the Merkle-Damgård construction.

Theorem 3.1. *Let h be an iterated hash function using the* Lane *iteration with compression function f. If the compression f is two-step collision resistant as in Definition 3.2, then the iterated hash function h is collision resistant.*

Proof. Assume that h is not collision resistant. This implies that there exists an efficient adversary that can find a collision pair (M, M'), i.e., $M \neq M'$ and $h(M) = h(M')$. We distinguish two cases, depending on whether the length of both messages is equal or not.

Case 1: $|M| \neq |M'|$. The last two compression function calls, i.e., the output transformation and the preceding compression function call, yield a two-step compression function collision. Indeed, the final output collides, and the last compression function call is active as its input includes the message length $|M| \neq |M'|$. This case is depicted in Fig. 3.3 (a).

Note that there is always at least one compression function call before the output transformation, even if one of the messages has zero length. This is shown in Fig. 3.3 (b).

Case 2: $|M| = |M'|$. Since the messages collide under h, have the same length and start from the same initial value, there must be (at least) one

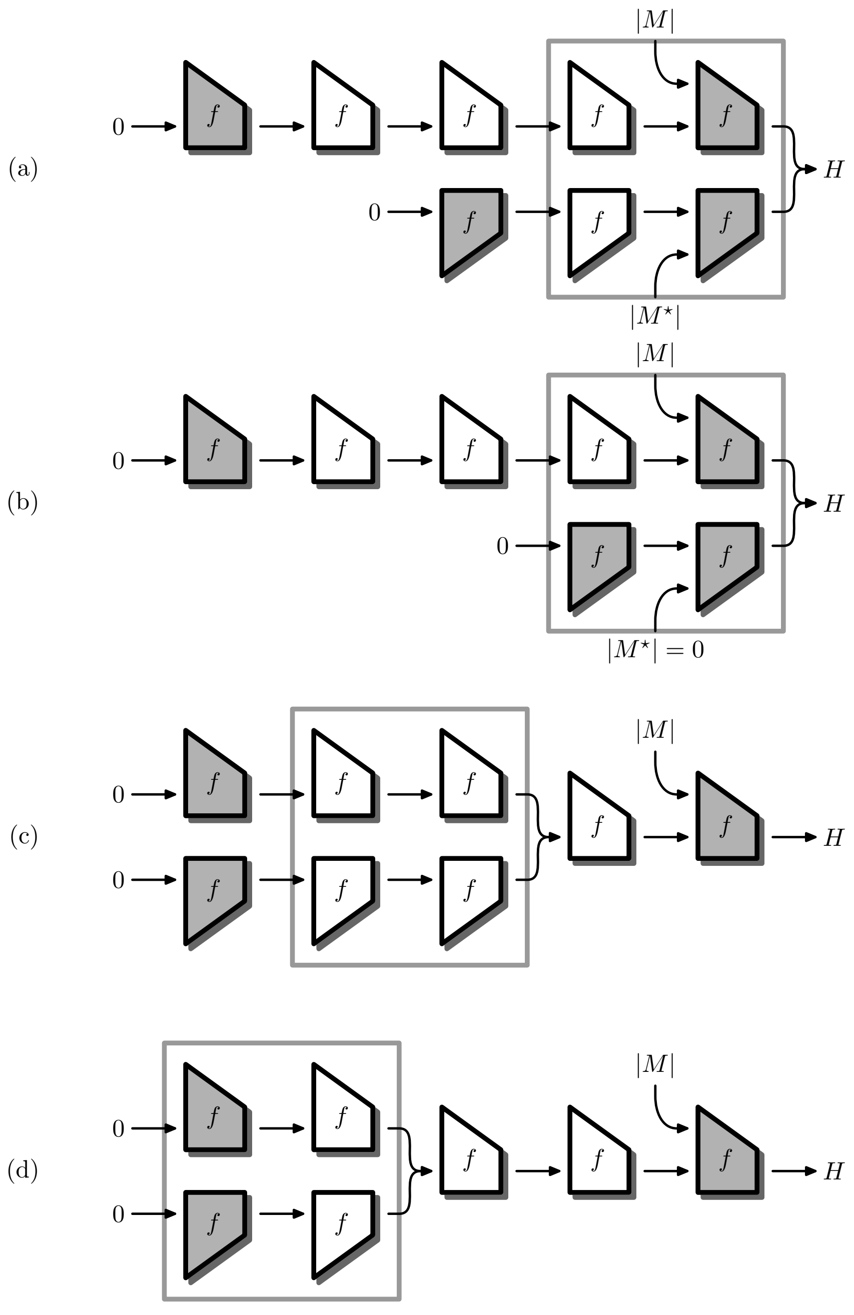


Figure 3.3 – Two-step compression function collisions and the LANE iteration mode.

compression function call for which a compression function collision is found. Then, this compression function call and the one preceding it form a two-step compression function collision. Indeed, the outputs of the second compression function call collide, and it is active. Figure 3.3 (c) shows this case. Note that the compression function call in which the collision occurs can be the output transformation, even when the message lengths are equal.

Also note that there always is at least one compression function call before the one exhibiting the collision. Indeed, the earliest place where a compression function collision can occur is in the first compression function call that processes message data, which is preceded by the compression function call that derives the initial value. This case is shown in Fig. 3.3 (d).

Hence, if h is not collision resistant, f is not resistant to two-step compression function collisions. From this, the theorem follows. $\square$

We can also trivially prove the following theorem, indicating that the requirement of two-step compression function collision resistance is a weaker condition than plain compression function resistance.

Theorem 3.2. *A collision resistant compression function is also two-step collision resistant.*

Proof. Assume that the compression function f is collision resistant, but not two-step collision resistant. This implies that there exists an efficient adversary that can find a two-step compression function collision, i.e., a tuple $\langle h, m_0, m_1, h^\star, m_0^\star, m_1^\star \rangle$ such that $f\left(f\left(h, m_0\right), m_1\right) = f\left(f\left(h^\star, m_0^\star\right), m_1^\star\right)$, and $f(h, m_0) \,||\, m_1 \neq f(h^\star, m_0^\star) \,||\, m_1^\star$.

Now, let $h_1 = f(h, m_0)$ and $h_1^\star = f(h^\star, m_0^\star)$. Then it is clear that $f(h_1, m_1) = f(h_1^\star, m_1^\star)$ and $h_1 \,||\, m_1 \neq h_1^\star \,||\, m_1^\star$. In other words, the second compression function call yields a compression function collision. This violates the assumption that f is compression function resistant, proving the theorem. $\square$

3.3.4 A Generic Merkle-Damgård Iteration

This result can be generalised in a trivial way to a generic Merkle-Damgård iteration mode with output transformation, as shown in Fig. 3.4. The only difference is that we can no longer treat all cases uniformly and need to consider a number of corner cases involving the initial value or the output transformation separately. Note that the original Merkle-Damgård proof (Theorem 2.1) also does not consider an output transformation and thus needs to be extended in any case. This is often overlooked, and does have an impact, especially for hash functions with a weak output transformation. For instance, the ‘output transformation’ of SHA-224 and SHA-384 consists of a simple truncation [62]. Because of this, a hash function collision is no longer guaranteed to contain a compression function collision, and thus the original Merkle-Damgård proof no longer applies.

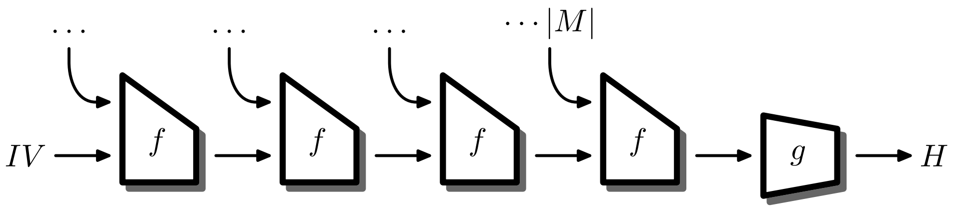
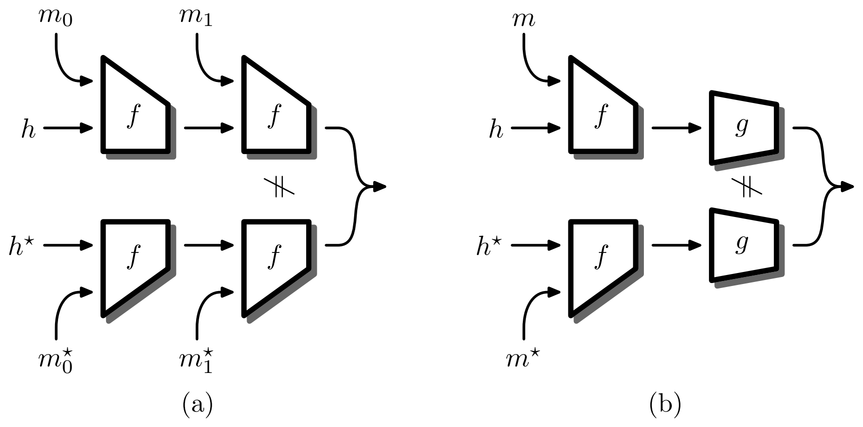


Figure 3.4 – A generic Merkle-Damgård iteration with output transformation.



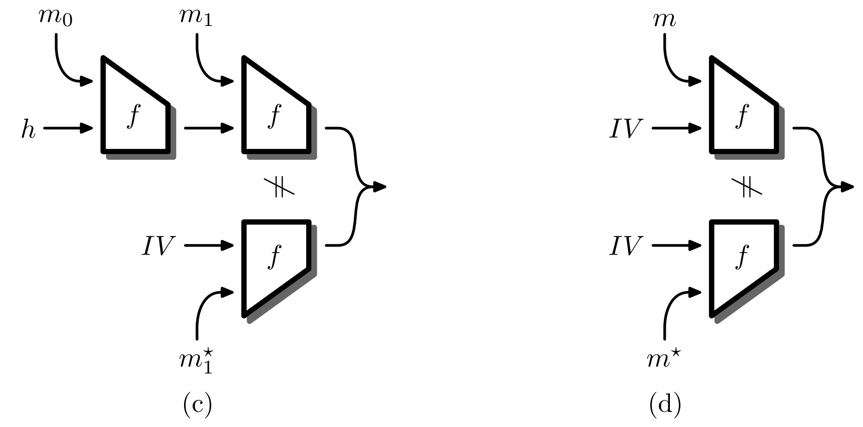


Figure 3.5 – Four cases of two-step compression function collisions for a Merkle-Damgård iteration with output transformation.

Instead of requiring just resistance against finding two-step compression function collisions, we now require that it is hard to construct any of the following four cases, shown graphically in Fig. 3.5:

(a) Find $\langle h, m_0, m_1, h^\star, m_0^\star, m_1^\star \rangle$ such that

- $f\left(f\left(h, m_0\right), m_1\right) = f\left(f\left(h^\star, m_0^\star\right), m_1^\star\right)$, and
- $f(h, m_0) \,||\, m_1 \neq f(h^\star, m_0^\star) \,||\, m_1^\star$.

(b) Find $\langle h, m, h^\star, m^\star \rangle$ such that

- $g\left(f\left(h, m\right)\right) = g\left(f\left(h^\star, m^\star\right)\right)$, and
- $f\left(h, m\right) \neq f\left(h^\star, m^\star\right)$.

(c) Find $\langle h, m_0, m_1, m_1^\star \rangle$ such that

- $f\left(f\left(h, m_0\right), m_1\right) = f\left(IV, m_1^\star\right)$, and
- $f(h, m_0) \,||\, m_1 \neq IV \,||\, m_1^\star$.

(d) Find $\langle m, m^\star \rangle$ such that

- $f\left(IV, m\right) = f\left(IV, m^\star\right)$, and
- $m \neq m^\star$.

Case (a) is a two-step compression function collision, as in Definition 3.2. Note that for the LANE iteration, all cases reduce to this single case. Indeed, the output transformation in LANE is the same as the compression function, so case (b) is identical to case (a). Also, the initial value IV is derived using the compression function in LANE. Hence, cases (c) and (d) can be reduced to case (a) through the use of the IV derivation.

3.4 Conclusion

This chapter introduced our contribution to the design of cryptographic hash functions: the hash function LANE. LANE was submitted as a candidate to the NIST SHA-3 competition, but did not advance beyond the first round of the competition. Independent cryptanalysis of LANE has raised questions on the relevance of pseudo-collision attacks, which were also addressed in this chapter.

Chapter 4

Analysis of Cryptographic Hash Functions

4.1 Introduction

This chapter presents a survey of our contributions related to the cryptanalysis of cryptographic hash functions. We have successfully analysed several hash functions that were candidates in the SHA-3 competition: Dynamic SHA, Dynamic SHA2, EnRUPT, Maraca and SHAMATA. Outside of the SHA-3 competition, we have shown collision attacks on reduced variants of the SHA-2 hash functions, up to 24 rounds. At the time of writing, this is still the best known result for collision attacks on SHA-2. We also investigated the security of RC4-Hash and reduced versions of the Tiger hash function. Finally, although not a hash function, our analysis of the KeeLoq remote keyless entry system is also included here.

Many of our attacks are practical and were implemented and verified to work. For instance, we show collision examples for Dynamic SHA, EnRUPT, RC4-Hash, 24-step reduced SHA-256, 23-step reduced SHA-512, and SHAMATA-256. For Maraca, our attack can find preimages for any digest in mere seconds. Our attack on the block cipher KeeLoq was implemented and tested with data extracted from an actual KeeLoq chip.

The remainder of this chapter gives a short summary of each of these analysis results, in alphabetical order. For technical details on each of the attacks, we refer to the publications included in part II of this dissertation.

4.2 Dynamic SHA and Dynamic SHA2

The hash functions Dynamic SHA [91] and Dynamic SHA2 [92] were proposed by Xu as candidates in the NIST SHA-3 competition. The principal idea in the design of both functions is the use of data-dependent rotations, i.e., bit-rotations where the rotation amount is not fixed, but dependent on the processed data. The use of this building block is not new. It had been used before in certain block ciphers, e.g., RC5 and RC6 [75, 76].

The designer's motivation to use data-dependent rotations in a hash function is (an attempt) to thwart differential attacks. The reasoning of Xu is that, if the rotation amounts are variable depending on the data, it is not possible to trace differences through the hash function. However, we note that a hash function adversary not only knows exactly what happens inside the function, but even has a certain amount of control. In our analysis of Dynamic SHA and Dynamic SHA2, we make extensive use of this fact to control the rotation amounts.

In a joint work with Jean-Philippe Aumasson, Orr Dunkelman and Bart Preneel [7] (see p. 263), we present practical collision attacks on Dynamic SHA, and a close to practical collision attack on Dynamic SHA2. We give a collision example for Dynamic SHA-256, thereby proving that our attack works. Also, by forcing all rotation amounts to be zero, it is possible to construct a simple theoretical preimage attack on Dynamic SHA that is faster than exhaustive search. Neither Dynamic SHA nor Dynamic SHA2 were accepted into the second round of the SHA-3 competition, likely because of their clear lack of security, as pointed out by our cryptanalytic results.

4.3 EnRUPT

The EnRUPT family of hash functions was submitted to the SHA-3 competition by O'Neil, Nohl and Henzen [68]. It consists of seven EnRUPT variants with digest lengths ranging from 128 bits to 512 bits. The core operation of EnRUPT is a simple multiplication of 32- or 64-bit words with the constant 9.

Our attack [38, 40] (see p. 221) on EnRUPT is based on linearisation of the hash function. All operations in EnRUPT are linear (over GF(2)), except for the multiplication with 9. However, this operation can be approximated well by a bitwise shift and an XOR, both of which are linear. Based on techniques from coding theory, differential paths with a relatively high probability can be constructed. However, these probabilities are still prohibitively small if the entire path needs to be satisfied at once.

But due to the design of EnRUPT, it is straightforward to search for a conforming pair, i.e., messages satisfying the differential path, in a step-by-step fashion. This greatly reduces the time required to find such a pair, as the complexities associated with each step of the differential can be added rather than multiplied. Furthermore, this strategy can be taken into account when searching for good differential paths, leading to an even better overall complexity.

We show practical collision attacks on all seven EnRUPT variants, with complexities ranging from 2^{36} to 2^{40} round computations, depending on the EnRUPT variant. The memory requirements of our attack are negligible and parallelisation is possible. For all EnRUPT variants, our attack thus significantly outperforms a generic birthday attack. To demonstrate the practicality of our attack, we give an actual collision example for EnRUPT-256. Even though EnRUPT was patched in response to our analysis, it was not selected for the second round of the SHA-3 competition.

4.4 KeeLoq

Unlike the other primitives in this chapter, KeeLoq is not a hash function. KeeLoq is a lightweight block cipher, or alternatively, a suite of authentication protocols built on this block cipher. KeeLoq technology is (or was) allegedly used by various car manufacturers in anti-theft mechanisms. It was designed in the 1980's and sold to Microchip Inc. in 1995. For a long period of time, the definition of the cipher was a well-kept secret. The first cryptanalysis results on KeeLoq were published only in 2007, not long after a confidential document containing a specification of KeeLoq leaked on the internet.

Our attack on KeeLoq is a practical key recovery attack requiring only 2^{16} known plaintext-ciphertext pairs. While this may seem unrealistic, one of the protocols built on KeeLoq, the 'KeeLoq Identify Friend or Foe' (IFF) protocol, allows to trivially extract the ciphertexts corresponding to chosen plaintexts from a device. It is a simple challenge-response protocol in which the transponder chip responds with the KeeLoq encryption of any challenge it is sent, without any user interaction whatsoever. In little more than an hour within communication range of a transponder, the required 2^{16} plaintext-ciphertext pairs can be collected. Then, our attack requires the time equivalent to about $2^{44.5}$ KeeLoq encryptions, compared to the 2^{63} encryptions required for a brute force key search of the 64-bit secret key. As our attack can be parallelised efficiently, a modestly-sized compute cluster of 100 CPU cores can find the secret key in only two days.

For a detailed discussion of our attack on KeeLoq, we refer to [32] (see p. 81). Apart from side-channel attacks [23], which exploit weaknesses in the implementation rather than the cryptographic algorithm, this is still the best known attack on KeeLoq.

4.5 Maraca

The cryptographic hash function Maraca was submitted to the NIST SHA-3 competition by Jenkins [44], but it was not accepted to the first round of the competition, likely due to shortcomings of the submission package. Our analysis of Maraca predates the announcement of the first-round candidates.

Maraca is based on an 8×8 bit S-box with rather remarkable properties. Typically, an S-box is designed in such a way that all (non-zero) differential transitions occur with a similar, low probability. The reason for avoiding high-probability differentials is to resist differential cryptanalysis. Not so in Maraca, where many transitions are impossible, and others occur with probabilities as high as 75%. There are also many very good linear approximations for the S-box, making it vulnerable to linear cryptanalysis as well.

In our analysis [39] (see p. 251) of Maraca, we take advantage of these weaknesses in the S-box to construct a practical preimage attack on the Maraca hash function. The starting point of our analysis is the observation that the S-box can be approximated by a linear function with probability 1, provided that the input is restricted to an affine subspace of a non-trivial dimension. More so, there are multiple distinct approximations for which this is possible. By chaining such approximations together, we managed to construct a similar property for the entire Maraca hash function. When the input in restricted to a very carefully chosen affine space, Maraca becomes a simple affine function. Such a function can be inverted easily, for instance using Gaussian elimination.

This results in a practical preimage attack on Maraca, which can find a message corresponding to any digest in mere seconds. Practical preimage attacks on hash functions are relatively rare, especially attacks as powerful as our attack on Maraca. This clearly shows that Maraca is a very weak hash function.

4.6 RC4-Hash

RC4-Hash [15] is a cryptographic hash function proposal inspired by the RC4 stream cipher. It is an iterated hash function and its chaining value consists of an array of 256 elements representing a permutation of the integers 0 through 255, and a pointer to an array element, much like the internal state of the RC4 stream cipher. The compression function of RC4-Hash processes message blocks of 64 bytes. Each message byte is used four times, in a predefined order, to update the permutation array and pointer in a similar way as the key scheduling algorithm of RC4 uses the bytes of the secret key.

In [36] (see p. 125), we show two methods to construct fixed points for RC4-Hash, and demonstrate how these can be used to construct collisions and multicollisions following Joux' technique [45]. The first construction method is very similar to the so-called 'Finney states' [24] in the RC4 stream cipher. In RC4, the initialisation makes it impossible to ever reach a Finney state. But in RC4-Hash, the additional message freedom makes this possible, resulting in fixed points. The second method can be seen as a generalisation of Finney states that are made possible due to peculiarities in the design of RC4-Hash.

The computational effort required for our attacks is very low at only about 2^9 compression function evaluations for a single collision, or $2^7 + k \cdot 2^8$ for a 2^k-way multicollision. This means that our attack is practical, and we show collision examples.

4.7 SHA-2

The SHA-2 hash functions [62] are the most recent, and most sophisticated members of the MD4 family of hash functions. Four SHA-2 hash functions are specified in [62], with digest lengths ranging from 224 bits up to 512 bits: SHA-224, SHA-256, SHA-384 and SHA-512. Compared to SHA-1, the SHA-2 functions include several additional hurdles for the cryptanalyst. For instance, while the message expansion of SHA-1 is linear, it contains non-linear operations in SHA-2. Also the state update transformation is more complex in SHA-2 compared to SHA-1. The building blocks are still the same — modular additions, bitwise shifts and rotations, XOR and bitwise Boolean functions — but SHA-2 performs more operations per step, and updates two state variables instead of just one.

SHA-1 updates only a single state variable per step, and it is straightforward to re-write SHA-1 using only this single variable in each step. This can be a useful representation for the cryptanalyst, e.g., see [18]. At first sight, this is no longer possible in SHA-2. However, in [30] (see p. 71) we are the first to show that also for SHA-2, such a representation is possible. As an example, [30] uses this alternative representation of SHA-2 to construct a trivial and practical collision attack on SHA-256-XOR-24. This is a simplified and reduced variant of SHA-256, where the number of rounds is reduced from 64 to 24, and all modular additions in the message expansion are replaced by XOR operations. This makes the message expansion linear again, similar to SHA-1.

In [33] (see p. 103) we improve upon the analysis of SHA-256 by Nikolić and Biryukov [65]. We extend the Nikolić-Biryukov attack from collisions for 21-step SHA-256 to 24-step collisions. An important building block of our improved attack is the alternative SHA-2 representation that we introduced in [30], which enables us to convert a 24-step semi-free start collision attack into a real collision attack on SHA-256 reduced to 24 out 64 steps. Our attack can also be applied to SHA-512 up to 24 (out of 80) steps.

We demonstrate that our attacks are practical by giving example collision pairs for 23-step and 24-step SHA-256 as well as for 23-step SHA-512. Our attack on 24-step SHA-512 is on the border of practicality, with an estimated time complexity of $2^{53.0}$ operations. Additionally, we show a number of weaker attacks, demonstrating non-random behaviour of the SHA-256 compression function for up to 31 steps.

Later, Sanadhya and Sarkar [81] improved upon our results. They claim to achieve lower time complexities at the expense of a (very) large memory requirement for a large look-up table. It is not clear if this really yields any improvement in reality. Another modification they propose, is the omission of a step of the attack procedure that was only included in [33] for the sake of clarity, and does not impact the attack complexity. By using our techniques with a different local collision, they do achieve a significant improvement for SHA-512, enabling them to exhibit the first collision example for 24-step SHA-512.

4.8 SHAMATA

The SHA-3 submission SHAMATA [6] is a stream cipher like hash function design with components from the AES block cipher. It is one of the fastest hash functions submitted to the SHA-3 competition.

In our attack [34] (see p. 287), we show that SHAMATA is not collision resistant. SHAMATA is built using operations on blocks of 128 bits. By considering only a single, special 128-bit difference, that interacts favourably with the various components of SHAMATA, the search for differential paths can be simplified tremendously. Furthermore, many conditions imposed by the differential path can be satisfied directly using a simple, but powerful message modification technique, similar to the technique introduced by Wang [89] in the context of MD5 and other hash functions. With the help of techniques from coding theory, we found a collision differential path for which this message modification technique can be used to satisfy all conditions, except for a single 128-bit condition in the first step.

Using an efficient guess-and-determine technique, this last condition can also be satisfied for SHAMATA-256, resulting in a practical collision attack with a time complexity of 2^{40} operations. We show a collision example for SHAMATA-256 constructed using our attack. For SHAMATA-512, the last condition can only be satisfied using an exhaustive search phase with complexity 2^{128} operations. While no longer practical, this still yields a theoretical collision attack on SHAMATA-512 that is significantly faster than a generic attack.

4.9 Tiger

Tiger [1] is a 192-bit hash function that was designed by Anderson and Biham in 1995. It has been able to withstand cryptanalysis attempts remarkably well. The most notable aspect of the design of the hash function Tiger, is that it was built to make efficient use of 64-bit processors. Note that when Tiger was designed, 64-bit processors were still far from common.

While collision attacks on reduced variants of Tiger had appeared earlier, our analysis [35] (see p. 57) made a first attempt towards preimage attacks on Tiger. We show straightforward guess-and-determine preimage attacks on the compression function of Tiger, reduced to 12 or 13 out of 24 rounds. This requires a computation of $2^{63.5}$ or $2^{127.5}$ compression function evaluations, respectively. Moreover, we show how to extend this to first and second preimage attacks on similarly reduced variants of the Tiger hash function.

Since the publication of [35] in 2007, several more powerful preimage attacks on Tiger have appeared [42, 55, 86], improving the number of rounds that can be attacked up to 23 out of 24 rounds. And very recently, Guo et al. [27] claimed the first preimage attack on full Tiger.

Table 4.1 – Our cryptanalysis contributions.

Name	Variant	SHA-3 candidate	Attack type	Practical
Dynamic SHA	all	✓	collision	✓
	all	✓	preimage	
Dynamic SHA2	all	✓	collision	
EnRUPT	all	✓	collision	✓
KeeLoq			key recovery	✓
Maraca	all	✓	preimage	✓
RC4-Hash	all		collision	✓
SHA-256	≤24 steps		collision	✓
SHA-512	≤23 steps		collision	✓
	24 steps		collision	
SHAMATA	256 bits	✓	collision	✓
	512 bits	✓	collision	
Tiger	≤13 steps		preimage	

4.10 Conclusion

Table 4.1 gives a summary of our contributions related to cryptanalysis, that were discussed in this chapter. Several first round SHA-3 candidates were analysed successfully. As a direct consequence of our analysis, these candidates were not accepted to the second round of the SHA-3 competition. The majority of the attacks we presented are efficient enough to be practical, and were fully implemented and verified.

Chapter 5

Conclusion

Cryptographic hash functions are of critical importance to the security of a large number of applications. Recent cryptanalytic advances have raised serious concerns regarding the security of several popular hash functions such as MD5 and SHA-1. Their successors, the SHA-2 hash functions, are not (yet) threatened by any attack, but they enjoy very little confidence due to their similar design philosophy. These concerns have caused the National Institute of Standards and Technology (NIST) to organise the SHA-3 competition, aiming to develop the next generation cryptographic hash function standard.

There is a strong connection between our research, presented in this dissertation, and the SHA-3 competition. Firstly, we have designed the cryptographic hash function LANE, which aims to be secure, easy to understand, elegant and flexible in implementation. We have submitted LANE as a candidate to the SHA-3 competition, but it did not advance to the second round of evaluations.

Secondly, we have contributed actively to the evaluation of SHA-3 candidates through the cryptanalysis of the candidates Dynamic SHA, Dynamic SHA2, EnRUPT, Maraca and SHAMATA. Apart from the SHA-3 competition, our cryptanalysis contributions include a practical attack on the block cipher KeeLoq and attacks on the hash functions RC4-Hash and (reduced) Tiger. Finally, we have shown practical collision attacks on up to 24 steps of SHA-2, which is still the highest number of steps ever achieved in a collision attack on SHA-2.

5.1 Directions for Future Research

Research on cryptographic hash functions has certainly gained much momentum over the past years, but many questions still remain to be answered. We thus conclude with some directions for future research.

- Find actual collisions for full, unreduced SHA-1. After the announcement of an attack on full SHA-1 by Wang et al. [88] in 2005, it was generally believed that an actual collision pair would soon follow. Now, five years later, this still has not happened. Finding actual collisions in practice may be the only way to convince industry to migrate away from SHA-1. The SHA-3 competition seems to have diverted the attention of cryptanalysts away from SHA-1, but perhaps it is time to have another look at it.

- Analysis of the SHA-2 family of hash functions. Currently, the best collision attacks target only 24 out of 64 (or 80) steps [33, 80] and the best preimage attack is up to 43 steps [5]. As SHA-2 is being recommended as a replacement for MD5 and SHA-1, it is important to analyse it thoroughly.
- Select a good hash function as SHA-3. Even though the selection will ultimately be made by NIST, the cryptologic community has an important responsibility here to provide NIST with analysis results and detailed, comprehensive advice, enabling them to make a good selection.
- Several SHA-3 candidates are so-called ARX (Addition, Rotation, XOR) designs, i.e., they consist (primarily) of these three types of operations. The second-round candidates that (more or less) fit in this category are BLAKE, Blue Midnight Wish, CubeHash, Shabal, SIMD and Skein. In general, it is difficult to analyse ARX designs, and even more challenging to prove bounds against certain types of attacks.
- Since the beginning of the SHA-3 competition, new attack strategies for cryptographic hash functions have emerged. The most notable example is the 'rebound attack' [56]. Besides further extending and refining the methodology, it is an interesting problem to devise ways to defend against it. Perhaps it is possible to find constructions that can be proven to resist rebound attacks, much like this is possible for standard differential cryptanalysis.
- Provable security, the 'holy grail' of cryptology. In an ideal world, all security mechanisms would be based on strong mathematical foundations that allow to prove their properties. Especially in symmetric cryptology, reality is still very far away from this utopia.
- Lightweight cryptography is becoming more and more important. Emerging applications, such as systems using RFID tags, require security. But standard cryptographic primitives do not fit within the resource constraints. There is a need for good lightweight primitives, as well as analysis of such proposals. In the absence of these, industry may resort to ad hoc designs that are likely to be dreadfully insecure.
- `f7c135672b778d22c762859060dd0a47575c89e49797f1166ebfbbfd5a9eb08c`*

*This is the LANE-256 hash of the most ingenious idea in symmetric cryptology ever, or perhaps not.

Bibliography

[1] R. J. Anderson and E. Biham. Tiger: A fast new hash function. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop — FSE '96*, volume 1039 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 1996.

[2] E. Andreeva, C. Bouillaguet, O. Dunkelman, and J. Kelsey. Herding, second preimage and trojan message attacks beyond merkle-damgård. In M. J. Jacobson, Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 393–414. Springer, 2009.

[3] E. Andreeva, C. Bouillaguet, P.-A. Fouque, J. J. Hoch, J. Kelsey, A. Shamir, and S. Zimmer. Second preimage attacks on dithered hash functions. In N. P. Smart, editor, *Advances in Cryptology — EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[4] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton. Seven-property-preserving iterated hashing: ROX. In K. Kurosawa, editor, *Advances in Cryptology — ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2007.

[5] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang. Preimages for step-reduced SHA-2. In M. Matsui, editor, *Advances in Cryptology — ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 578–597. Springer, 2009.

[6] A. Atalay, O. Kara, F. Karakoç, and C. Manap. SHAMATA hash function algorithm specifications. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[7] J.-P. Aumasson, O. Dunkelman, S. Indesteege, and B. Preneel. Cryptanalysis of Dynamic SHA(2). In M. J. Jacobson, Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2009.

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[9] M. Bellare and T. Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In X. Lai and K. Chen, editors, *Advances in Cryptology — ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2006.

[10] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[11] R. Benadjila, O. Billet, S. Gueron, and M. J. B. Robshaw. The Intel AES instructions set and the SHA-3 candidates. In M. Matsui, editor, *Advances in Cryptology — ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 162–178. Springer, 2009.

[12] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. In *ECRYPT Hash Workshop*. European Network of Excellence in Cryptology ECRYPT, May 2007.

[13] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the indifferentiability of the sponge construction. In N. P. Smart, editor, *Advances in Cryptology — EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[14] E. Biham and O. Dunkelman. A framework for iterative hash functions — HAIFA. Second NIST Hash Workshop, 2006.

[15] D. Chang, K. C. Gupta, and M. Nandi. RC4-Hash: A new hash function based on RC4. In R. Barua and T. Lange, editors, *Progress in Cryptology — INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2006.

[16] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer, 2002.

[17] I. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[18] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. In X. Lai and K. Chen, editors, *Advances in Cryptology — ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[19] R. D. Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Princeton University, Jan. 1999.

[20] D. E. Denning. *Cryptography and Data Security*, page 100. Addison-Wesley, 1982.

[21] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[22] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop — FSE '96*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1996.

[23] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani. On the power of power analysis in the real world: A complete break of the KeeLoq code hopping scheme. In D. Wagner, editor, *Advances in Cryptology — CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2008.

[24] H. Finney. An RC4 cycle that can't happen. Newsgroup post in `sci.crypt`, Sept. 1994.

[25] P. Flajolet and A. M. Odlyzko. Random mapping statistics. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology — EUROCRYPT '89*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354. Springer, 1990.

[26] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl — a SHA-3 candidate. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://www.groestl.info/`.

[27] J. Guo, S. Ling, C. Rechberger, and H. Wang. Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. Cryptology ePrint Archive, Report 2010/016, 2010. `http://eprint.iacr.org/2010/016/`.

[28] S. Halevi and H. Krawczyk. Strengthening digital signatures via randomized hashing. In C. Dwork, editor, *Advances in Cryptology — CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2006.

[29] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[30] S. Indesteege. Trivial collisions for simplified and reduced SHA-2. Technical report, COSIC, Jan. 2008.

[31] S. Indesteege, E. Andreeva, C. De Cannière, O. Dunkelman, E. Käsper, S. Nikova, B. Preneel, and E. Tischhauser. The LANE hash function. Submission to the NIST SHA-3 competition, Oct. 2008.

[32] S. Indesteege, N. Keller, O. Dunkelman, E. Biham, and B. Preneel. A practical attack on KeeLoq. In N. P. Smart, editor, *Advances in Cryptology — EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[33] S. Indesteege, F. Mendel, B. Preneel, and C. Rechberger. Collisions and other non-random properties for step-reduced SHA-256. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography — SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 2009.

[34] S. Indesteege, F. Mendel, B. Preneel, and M. Schläffer. Practical collisions for SHAMATA-256. In M. J. Jacobson, Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009.

[35] S. Indesteege and B. Preneel. Preimages for reduced-round Tiger. In S. Lucks, A.-R. Sadeghi, and C. Wolf, editors, *Research in Cryptology, Second Western European Workshop — WEWoRC 2007*, volume 4945 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2007.

[36] S. Indesteege and B. Preneel. Collisions for RC4-Hash. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *Information Security 11th International Conference — ISC 2008*, volume 5222 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2008. Best Student Paper Award.

[37] S. Indesteege and B. Preneel. Coding theory and hash function design. In B. Preneel, S. Dodunekov, V. Rijmen, and S. Nikova, editors, *Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes*, volume 23 of *NATO Science for Peace and Security Series D — Information and Communication Security*, pages 63–68. IOS Press, 2009.

[38] S. Indesteege and B. Preneel. Practical collisions for EnRUPT. In O. Dunkelman, editor, *Fast Software Encryption, 16th International Workshop — FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 246–259. Springer, 2009.

[39] S. Indesteege and B. Preneel. Practical preimages for Maraca. In T. Tjalkens and F. Willems, editors, *Proceedings of the 30th Symposium on Information Theory in the Benelux*, pages 119–126. Werkgemeenschap voor Informatie- en Communicatietheorie, 2009.

[40] S. Indesteege and B. Preneel. Practical collisions for EnRUPT. *Journal of Cryptology*, 2010. To appear in print. Published online at `http://dx.doi.org/10.1007/s00145-010-9058-x`.

[41] Intel Corporation. Advanced encryption standard (AES) instructions set. White paper, July 2008. Available online at `http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set_WP.pdf`.

[42] T. Isobe and K. Shibutani. Preimage attacks on reduced Tiger and SHA-2. In O. Dunkelman, editor, *Fast Software Encryption, 16th International Workshop — FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 139–155. Springer, 2009.

[43] ISO/IEC. *ISO/IEC 10118-3:2004: Information technology — Security techniques — Hash-functions — Part 3: Dedicated hash-functions.* ISO/IEC, 2004.

[44] R. J. Jenkins Jr. Maraca: Algorithm specification. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://burtleburtle.net/bob/crypto/maraca/nist/`.

[45] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In M. K. Franklin, editor, *Advances in Cryptology — CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[46] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, revised edition, 1996.

[47] J. Kelsey and T. Kohno. Herding hash functions and the Nostradamus attack. In S. Vaudenay, editor, *Advances in Cryptology — EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[48] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[49] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX:5–38, Jan. 1883.

[50] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, third edition, 1997.

[51] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, second edition, 1998.

[52] X. Lai and J. L. Massey. Hash function based on block ciphers. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 1993.

[53] L. Lamport. Password authentication with insecure communication. *Communcations of the ACM*, 24(11):770–772, 1981.

[54] K. Matusiewicz, M. Naya-Plasencia, I. Nikolić, Y. Sasaki, and M. Schläffer. Rebound attack on the full Lane compression function. In M. Matsui, editor, *Advances in Cryptology — ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 106–125. Springer, 2009.

[55] F. Mendel. Two passes of Tiger are not one-way. In B. Preneel, editor, *Progress in Cryptology — AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2009.

[56] F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. The Rebound attack: Cryptanalysis of reduced Whirlpool and Grøstl. In O. Dunkelman, editor, *Fast Software Encryption, 16th International Workshop — FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2009.

[57] A. J. Menezes, S. A. Vanstone, and P. C. van Oorschot. *Handbook of Applied Cryptography.* CRC Press, 1996. `http://www.cacr.math.uwaterloo.ca/hac/`.

[58] R. C. Merkle. One way hash functions and DES. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[59] National Bureau of Standards, U.S. Deparment of Commerce. Data Encryption Standard. Federal Information Processing Standards Publication 46, 1977.

[60] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, Nov. 2007.

[61] National Institute of Standards and Technology. Cryptographic hash algorithm competition, 2007. `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[62] National Institute of Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-3, Oct. 2008.

[63] National Institute of Standards and Technology. Announcement of second round SHA-3 candidates. E-mail posted to the NIST SHA-3 competition mailing list, July 2009. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/`.

[64] National Institute of Standards and Technology. Digital signature standard (DSS). Federal Information Processing Standards Publication 186-3, 2009.

[65] I. Nikolić and A. Biryukov. Collisions for step-reduced SHA-256. In K. Nyberg, editor, *Fast Software Encryption, 15th International Workshop — FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[66] G. Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90:135–140, 2004.

[67] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In D. Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[68] S. O'Neil, K. Nohl, and L. Henzen. EnRUPT hash function specification. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[69] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, 1993.

[70] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. In D. R. Stinson, editor, *Advances in Cryptology — CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1994.

[71] M. O. Rabin. Digitalized signatures. In R. Lipton and R. DeMillo, editors, *Foundations of Secure Computations*, pages 155–168. Academic Press, 1978.

[72] R. L. Rivest. The MD4 message digest algorithm. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology — CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1991.

[73] R. L. Rivest. The MD4 message-digest algorithm. Internet Engineering Task Force (IETF) Request for Comments (RFC) 1320, Apr. 1992.

[74] R. L. Rivest. The MD5 message-digest algorithm. Internet Engineering Task Force (IETF) Request for Comments (RFC) 1321, Apr. 1992.

[75] R. L. Rivest. The RC5 encryption algorithm. In B. Preneel, editor, *Fast Software Encryption, Second International Workshop — FSE '94*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 1995.

[76] R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin. RC6 as the AES. In *Third AES Candidate Conference*, pages 337–342. National Institute of Standards and Technology, 2000.

[77] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communcations of the ACM*, 21(2):120–126, 1978.

[78] P. Rogaway. Formalizing human ignorance. In P. Q. Nguyen, editor, *Progress in Cryptology — VIETCRYPT 2006*, volume 4341 of *Lecture Notes in Computer Science*, pages 211–228. Springer, 2006.

[79] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In B. K. Roy and W. Meier, editors, *Fast Software Encryption, 11th International Workshop — FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[80] S. K. Sanadhya and P. Sarkar. New local collisions for the SHA-2 hash family. In K.-H. Nam and G. Rhee, editors, *Information Security and Cryptology — ICISC 2007*, volume 4817 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2007.

[81] S. K. Sanadhya and P. Sarkar. New collision attacks against up to 24-step SHA-2. In D. R. Chowdhury, V. Rijmen, and A. Das, editors, *Progress in Cryptology — INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 91–103. Springer, 2008.

[82] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In M. Naor, editor, *Advances in Cryptology — EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.

[83] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In S. Halevi, editor, *Advances in Cryptology — CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2009.

[84] D. R. Stinson. Some observations on the theory of cryptographic hash functions. *Des. Codes Cryptography*, 38(2):259–277, 2006.

[85] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[86] L. Wang and Y. Sasaki. Finding preimages of Tiger up to 23 steps. In S. Hong and T. Iwata, editors, *Fast Software Encryption, 17th International Workshop — FSE 2010*, Lecture Notes in Computer Science. Springer, 2010. To appear.

[87] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[88] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In V. Shoup, editor, *Advances in Cryptology — CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[89] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[90] S. Wu, D. Feng, and W. Wu. Cryptanalysis of the LANE hash function. In M. J. Jacobson, Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2009.

[91] Z. Xu. Dynamic SHA. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[92] Z. Xu. Dynamic SHA2. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[93] Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL — a one-way hashing algorithm with variable length of output. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology — AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 83–104. Springer, 1993.

Part II

Publications

List of Publications

International Journals

1. Sebastiaan Indesteege and Bart Preneel. Practical collisions for EnRUPT. *Journal of Cryptology*, 2010. To appear in print. Published online at `http://dx.doi.org/10.1007/s00145-010-9058-x`.

   – See p. 221.

Lecture Notes in Computer Science

1. Sebastiaan Indesteege and Bart Preneel. Preimages for reduced-round Tiger. In Stefan Lucks, Ahmad-Reza Sadeghi, and Christopher Wolf, editors, *Research in Cryptology, Second Western European Workshop — WEWoRC 2007*, volume 4945 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2007.

   – See p. 57.

2. Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A practical attack on KeeLoq. In Nigel P. Smart, editor, *Advances in Cryptology — EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

   – See p. 81.

3. Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and other non-random properties for step-reduced SHA-256. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography — SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 2009.

   – See p. 103.

4. Sebastiaan Indesteege and Bart Preneel. Collisions for RC4-Hash. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security 11th International Conference — ISC 2008*, volume 5222 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2008. Best Student Paper Award.

   – See p. 125.

5. Orr Dunkelman, Sebastiaan Indesteege, and Nathan Keller. A differential-linear attack on 12-round Serpent. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology — INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 308–321. Springer, 2008.

6. Sebastiaan Indesteege and Bart Preneel. Practical collisions for EnRUPT. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop — FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 246–259. Springer, 2009.

   – See p. 221 for the extended journal version of this article.

7. Jean-Philippe Aumasson, Orr Dunkelman, Sebastiaan Indesteege, and Bart Preneel. Cryptanalysis of Dynamic SHA(2). In Michael J. Jacobson, Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2009.

   – See p. 263.

8. Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Martin Schläffer. Practical collisions for SHAMATA-256. In Michael J. Jacobson, Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009.

   – See p. 287.

9. Nicky Mouha, Christophe De Cannière, Sebastiaan Indesteege, and Bart Preneel. Finding collisions for a 45-step simplified HAS-V. In Heung Youl Youm and Moti Yung, editors, *Information Security Applications, 10th International Workshop — WISA 2009*, volume 5932 of *Lecture Notes in Computer Science*, pages 206–225. Springer, 2009.

National Conferences

1. Sebastiaan Indesteege and Bart Preneel. Practical preimages for Maraca. In Tjalling Tjalkens and Frans Willems, editors, *Proceedings of the 30th Symposium on Information Theory in the Benelux*, pages 119–126. Werkgemeenschap voor Informatie- en Communicatietheorie, 2009.

   – See p. 251.

Book Chapters

1. Sebastiaan Indesteege and Bart Preneel. Coding theory and hash function design. In Bart Preneel, Stefan Dodunekov, Vincent Rijmen, and Svetla Nikova, editors, *Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes*, volume 23 of *NATO Science for Peace and Security Series D — Information and Communication Security*, pages 63–68. IOS Press, 2009.

   – See p. 143.

2. Sebastiaan Indesteege. ARIA, Keeloq and Tiger. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*. Springer, second edition, 2010. To appear.

Technical Reports

1. Sebastiaan Indesteege. Trivial collisions for simplified and reduced SHA-2. Technical report, COSIC, January 2008.

   – See p. 71.

2. Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser. The LANE hash function. Submission to the NIST SHA-3 competition, October 2008.

   – See p. 153.

3. Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser. The LANE hash function — extended abstract. Technical report, COSIC, 2008.

   – See p. 153 for the full version of this technical report.

Publication

Preimages for Reduced-Round Tiger

Publication Data

Sebastiaan Indesteege and Bart Preneel. Preimages for reduced-round Tiger. In Stefan Lucks, Ahmad-Reza Sadeghi, and Christopher Wolf, editors, *Research in Cryptology, Second Western European Workshop — WEWoRC 2007*, volume 4945 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2007.

Contributions

- Principal author.

Preimages for Reduced-Round Tiger

Sebastiaan Indesteege* and Bart Preneel

Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven.
Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
{sebastiaan.indesteege, bart.preneel}@esat.kuleuven.be

Abstract. The cryptanalysis of the cryptographic hash function Tiger has, until now, focussed on finding collisions. In this paper we describe a preimage attack on the compression function of Tiger-12, i.e., Tiger reduced to 12 rounds out of 24, with a complexity of $2^{63.5}$ compression function evaluations. We show how this can be used to construct second preimages with complexity $2^{63.5}$ and first preimages with complexity $2^{64.5}$ for Tiger-12. These attacks can also be extended to Tiger-13 at the expense of an additional factor of 2^{64} in complexity.

Key words: Tiger, hash functions, preimages

1 Introduction

A cryptographic hash function is expected to possess three properties: collision resistance, preimage resistance and second preimage resistance. While other properties exist, the above three are the most well known.

Collision resistance: It is difficult to find two distinct messages $m \neq m'$ that hash to the same result, i.e., $h(m) = h(m')$.

Preimage resistance: When given a hash result y (for which it holds that $\exists x : h(x) = y$), it is difficult to find a message m which hashes to y, i.e., $h(m) = y$.

Second preimage resistance: When given a message m, it is difficult to find a message $m' \neq m$ that hashes to the same result as the given message, i.e., $h(m) = h(m')$.

There are generic attacks that apply to any hash function and whose time complexity only depends on the size of the hash result. Collisions for a hash function with an n-bit result can be found in time $2^{n/2}$ using a birthday attack [6,7], and preimages can be found by brute force in time 2^n. Weaker attacks may aim at finding pseudo-collisions, where slight differences in the hash results are allowed,

*F.W.O. Research Assistant, Fund for Scientific Research – Flanders (Belgium).

or pseudo-near-collisions, where differences may also appear in the initial chaining values.

All attacks on the cryptographic hash function Tiger [1] have so far been collision attacks. Kelsey and Lucks [3] showed a collision attack on Tiger reduced to 16 rounds with a complexity of 2^{44} compression function evaluations. Mendel et al. [4] extended this to a collision attack on 19 rounds of Tiger with a complexity of 2^{62} compression function evaluations. In both papers some weaker attacks (e.g. pseudo-collisions) for a larger number of rounds were also shown. These results were further improved by Mendel et al. [5] towards a pseudo-near-collision for the full hash function and a pseudo-collision for 23 rounds of Tiger.

We focus on finding preimages for reduced variants of Tiger instead. More specifically, we describe a method to find first and second preimages for 12 and 13 rounds reduced Tiger. This method is conceptually similar to Dobbertin's preimage attack on reduced MD4 [2]. Our attack finds first and second preimages for Tiger-12 with a complexity of $2^{64.5}$ and $2^{63.5}$ compression function evaluations, respectively. It can be extended to Tiger-13, where the complexities become $2^{128.5}$ and $2^{127.5}$, respectively. As Tiger has a digest size of 192 bits, the theoretical complexity for finding first or second preimages is 2^{192} compression function evaluations. To the best of our knowledge, this is the first result concerning preimages for reduced round Tiger.

The structure of the paper is as follows. In Sect. 2, the Tiger hash function is described, along with the notation that will be used throughout the paper. Section 3 describes a preimage attack on three rounds of Tiger. The three round preimage attack is then used as a building block to construct preimages for the compression function of Tiger-12 and Tiger-13 in Sect. 4. Then, in Sect. 5 it is shown how first and second preimages for these reduced variants of the Tiger hash function can be constructed. Finally, Sect. 6 presents our conclusions.

2 Description of Tiger

Tiger [1] is an iterative cryptographic hash function, designed by Anderson and Biham in 1996. It has an output size of 192 bits. Truncated variants with a digest size of 160 and 128 bits were also defined. It was designed for 64-bit architectures and hence all words are 64 bits wide and arithmetic is performed modulo 2^{64}. Tiger uses the little-endian byte ordering.

First, the message to be hashed is padded by appending a single "1"-bit and as many "0"-bits as necessary to make the message length 64 bits less than the next multiple of 512 bits. Then the message length (in bits) is appended as a 64-bit unsigned integer. After this procedure, the padded message consists of an integer number of 512-bit blocks. Then, Tiger's compression function is applied iteratively to each 512-bit block of the padded message.

Tiger's compression function operates on a 192-bit chaining value and a 512-bit message block. The message block is split into eight 64-bit words X_i. The 192-bit

Table 1 – Notations.

$X + Y$	Addition of X and Y modulo 2^{64}
$X - Y$	Subtraction of X and Y modulo 2^{64}
$X \times Y$	Multiplication of X and Y modulo 2^{64}
$X \oplus Y$	Bit-wise exclusive or of X and Y
$\overline{X}$	Bit-wise complement of X
$X \ll n$	Logical left bit shift of X by n positions
$X \gg n$	Logical right bit shift of X by n positions
$X\|\|Y$	The concatenation of X and Y
X_i	The i-th expanded message word
Y_i	The i-th intermediate value of the key schedule algorithm
A_i, B_i, C_i	State variables at the output of round i, $0 \leq i < 24$
K_i	The round constant used in round i, $0 \leq i < 24$
K_i^{-1}	Multiplicative inverse of K_i modulo 2^{64}
$T_1,\ldots,T_4$	The four 8-to-64-bit S-boxes used in Tiger

chaining value is split into three 64-bit words which are used as the initial state variables A_{-1}, B_{-1} and C_{-1}. The compression function consists of three passes of 8 rounds of a state update transformation (24 rounds in total), each using one X_i to update the three state variables A_i, B_i and C_i. Table 1 summarises the notations used in this paper.

The i-th round of Tiger ($0 \leq i < 24$) is depicted in Fig. 1. Equivalently, the state update transformation can be described by the following equations:

$$\begin{aligned} A_i &= K_i \times (B_{i-1} + \mathrm{odd}\,(C_{i-1} \oplus X_i)) \ , \\ B_i &= C_{i-1} \oplus X_i \ , \\ C_i &= A_{i-1} - \mathrm{even}\,(C_{i-1} \oplus X_i) \ . \end{aligned} \tag{1}$$

In every round, a round constant K_i is used. These constants are given by:

$$K_i = \begin{cases} 5 & \text{if } 0 \leq i < 8 \ , \\ 7 & \text{if } 8 \leq i < 16 \ , \\ 9 & \text{if } 16 \leq i < 24 \ . \end{cases} \tag{2}$$

The non-linear functions odd$(\cdot)$ and even$(\cdot)$ are defined as follows.

$$\begin{aligned} \mathrm{odd}(c_7\|\ldots\|c_0) &= T_4[c_1] \oplus T_3[c_3] \oplus T_2[c_5] \oplus T_1[c_7] \ , \\ \mathrm{even}(c_7\|\ldots\|c_0) &= T_1[c_0] \oplus T_2[c_2] \oplus T_3[c_4] \oplus T_4[c_6] \ . \end{aligned} \tag{3}$$

Here, c_i denotes the i-th byte of a 64-bit word, using the little-endian byte ordering, i.e., c_0 is the least significant byte.[1] Both functions use four 8-to-64-bit S-boxes,

[1]Note that there was a misinterpretation of the byte order in [3, 4]. The attacks described there can however be modified to overcome this problem. [5]

T_1 through T_4. Note that both functions only use four out of eight input bytes, and thus map 32 bits to 64 bits. They are called odd$(\cdot)$ and even$(\cdot)$ because they operate on the odd, respectively even bytes of the input word.

The first eight message words X_i, $0 \leq i < 8$, are taken directly from the message block. The message words $X_8,\ldots,X_{15}$ are derived from $X_0,\ldots,X_7$ using an algorithm which the designers of Tiger refer to as the key schedule algorithm [1]. Then, using the same algorithm, $X_{16},\ldots,X_{23}$ are determined from $X_8,\ldots,X_{15}$. This key schedule algorithm consists of two passes, given by the following equations:

$$\begin{array}{rclrcl}
Y_0 & = & X_0 - (X_7 \oplus \texttt{A5}\ldots\texttt{A5}_x)\ , & X_8 & = & Y_0 + Y_7\ , \\
Y_1 & = & X_1 \oplus Y_0\ , & X_9 & = & Y_1 - \big(X_8 \oplus (\overline{Y_7} \ll 19)\big)\ , \\
Y_2 & = & X_2 + Y_1\ , & X_{10} & = & Y_2 \oplus X_9\ , \\
Y_3 & = & X_3 - \big(Y_2 \oplus (\overline{Y_1} \ll 19)\big)\ , & X_{11} & = & Y_3 + X_{10}\ , \\
Y_4 & = & X_4 \oplus Y_3\ , & X_{12} & = & Y_4 - \big(X_{11} \oplus (\overline{X_{10}} \gg 23)\big)\ , \\
Y_5 & = & X_5 + Y_4\ , & X_{13} & = & Y_5 \oplus X_{12}\ , \\
Y_6 & = & X_6 - \big(Y_5 \oplus (\overline{Y_4} \gg 23)\big)\ , & X_{14} & = & Y_6 + X_{13}\ , \\
Y_7 & = & X_7 \oplus Y_6\ . & X_{15} & = & Y_7 - (X_{14} \oplus \texttt{01}\ldots\texttt{EF}_x)\ .
\end{array} \tag{4}$$

Finally, after 24 rounds, the initial state variables are fed forward, using a combination of exclusive or, subtraction and addition.

$$\begin{array}{rcl}
A^\star & = & A_{-1} \oplus A_{23}\ , \\
B^\star & = & B_{-1} - B_{23}\ , \\
C^\star & = & C_{-1} + C_{23}\ .
\end{array} \tag{5}$$

The 192-bit output of the compression function is $A^\star||B^\star||C^\star$, i.e., the concatenation of $A^\star$, $B^\star$ and $C^\star$.

3 Preimages for Three Rounds of Tiger

In this section we describe a solution due to Mendel et al. [4] to the problem of finding preimages for three rounds of the state update transformation of Tiger. There is always exactly one solution, which can be found in constant time. Although rather straightforward, it will prove to be a useful building block in preimage attacks on a larger number of Tiger rounds.

More in detail, we are given A_{-1}, B_{-1}, C_{-1}, A_2, B_2 and C_2 and want to determine the three message words X_0, X_1 and X_2 such that the constraints originating from the state update transformation are satisfied. Note that, without knowing any of the message words, all the state variables in these three rounds

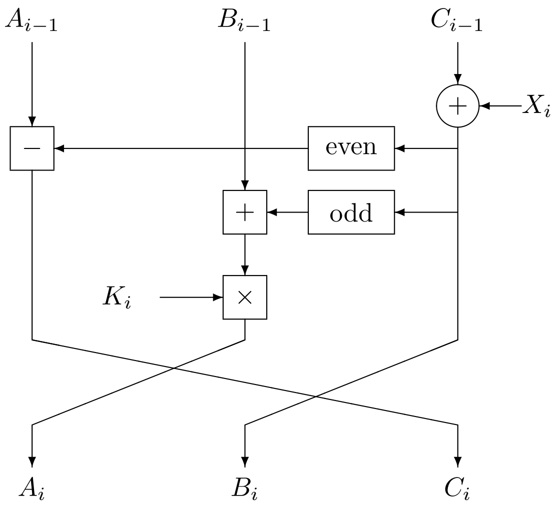


Figure 1 – The state update transformation of Tiger.

can already be determined. Indeed, from (1) it follows that

$$\begin{array}{rcl} A_1 & = & C_2 + \text{even}\,(B_2) \enspace , \\ B_1 & = & \left(A_2 \times K_2^{-1}\right) - \text{odd}\,(B_2) \enspace , \\ B_0 & = & \left(A_1 \times K_1^{-1}\right) - \text{odd}\,(B_1) \enspace , \\ A_0 & = & K_0 \times (B_{-1} + \text{odd}\,(B_0)) \enspace , \\ C_0 & = & A_{-1} - \text{even}\,(B_0) \enspace , \\ C_1 & = & A_0 - \text{even}\,(B_1) \enspace . \end{array} \tag{6}$$

Note that each K_i as given in (2) is coprime with 2^{64} so its multiplicative inverse modulo 2^{64} exists and can be computed easily. Knowing the state variables, it is trivial to determine X_0, X_1 and X_2.

$$\begin{array}{rcl} X_0 & = & C_{-1} \oplus B_0 \enspace , \\ X_1 & = & C_0 \oplus B_1 \enspace , \\ X_2 & = & C_1 \oplus B_2 \enspace . \end{array} \tag{7}$$

This procedure is fully deterministic and always gives exactly one solution. The time complexity of this procedure is equivalent to three rounds of Tiger.

Of course this can equally be applied to any three consecutive rounds of Tiger, as part of a larger attack. To conclude, control over three consecutive expanded message words yields complete control over the intermediate state of Tiger.

4 Preimages for the Compression Function of Tiger-12

In this section, we first describe a method to find preimages for the compression function of Tiger, reduced to 12 rounds. Then we extend this to Tiger-13, i.e., Tiger reduced to 13 rounds.

Given the algorithm from Sect. 3, one can easily find sets of expanded message words X_i which ensure that the output of the compression function of Tiger (or a round-reduced version thereof) is equal to some desired value. However, if the number of attacked rounds is greater than eight there is no guarantee that these expanded message words satisfy the constraints from the key schedule algorithm. For eight or less rounds of Tiger, the message expansion becomes trivial, as each of the first eight expanded message words is under direct control of an adversary. Hence also finding preimages for these variants of Tiger is trivial by making arbitrary choices and using the algorithm from Sect. 3 for the last three rounds.

The circular dependency can be broken by guessing some intermediate variable(s) and later verifying if the guess was correct. If the guess was wrong, the attack is simply repeated. Hence the time complexity of the attack is highly dependent on the probability that the correct guess was made. Since we assume that every value for the guessed variables is equally likely, this probability is equal to 2^{-n} where n is the total number of guessed bits.

Conceptually, this approach is very similar to the work of Dobbertin [2] on finding preimages for a reduced variant of MD4. Of course the similarity only exists on a very high level, due to the fact that MD4 and Tiger are very different hash functions.

4.1 Algorithm

In this section, a detailed description of the algorithm for finding preimages for the compression function of Tiger-12 is given. As we are given the desired input and output chaining values, the feed-forward given in (5) can easily be removed. Therefore, the state variables A_{-1}, B_{-1}, C_{-1}, A_{11}, B_{11} and C_{11} are known at the beginning of the attack.

1. Make arbitrary choices for the message words used in the four last rounds, (i.e. X_8, X_9, X_{10} and X_{11}). The state update transformation can be used in the backwards direction to determine A_7, B_7 and C_7, as follows:

$$\begin{cases} A_{i-1} &= C_i + \operatorname{even}(B_i) \ , \\ B_{i-1} &= \left(A_i \times K_i^{-1}\right) - \operatorname{odd}(B_i) \ , \\ C_{i-1} &= B_i \oplus X_i \ . \end{cases} \qquad \text{for } i = 11, \ldots, 8 \tag{8}$$

2. Guess Y_7, an intermediate value of the key schedule algorithm. This 64-bit guess is the only guess that will be made in the attack. It will be verified in the final step of the attack.

3. The message words X_8 through X_{11} are normally computed from the key schedule. These equations can easily be inverted to find the intermediate values Y_0, Y_1, Y_2 and Y_3 for which the values chosen in step 1 will appear:

$$\begin{aligned} Y_0 &= X_8 - Y_7 \ , \\ Y_1 &= X_9 + \left(X_8 \oplus \left(\overline{Y_7} \ll 19\right)\right) \ , \\ Y_2 &= X_{10} \oplus X_9 \ , \\ Y_3 &= X_{11} - X_{10} \ . \end{aligned} \tag{9}$$

This step is deterministic and always leads to a single solution. Looking further at the key schedule, the message words X_1 through X_3 can also be determined uniquely:

$$\begin{aligned} X_1 &= Y_1 \oplus Y_0 \ , \\ X_2 &= Y_2 - Y_1 \ , \\ X_3 &= Y_3 + \left(Y_2 \oplus \left(\overline{Y_1} \ll 19\right)\right) \ . \end{aligned} \tag{10}$$

4. Choose X_7 (there are 2^{64} choices) and compute X_0 using the key schedule:

$$X_0 = Y_0 + (X_7 \oplus \texttt{A5A5A5A5A5A5A5A5}_x) \tag{11}$$

5. Now, the first four expanded message words (i.e. X_0 through X_3) are known. The state update transformation can thus be used in the forward direction to calculate A_3, B_3 and C_3.

$$\begin{cases} A_i &= K_i \times (B_{i-1} + \text{odd}\,(C_{i-1} \oplus X_i)) \ , \\ B_i &= C_{i-1} \oplus X_i \ , \\ C_i &= A_{i-1} - \text{even}\,(C_{i-1} \oplus X_i) \ . \end{cases} \quad \text{for } i = 0, \ldots, 3 \qquad (12)$$

Similarly, as X_7 is known, the state update transformation can be applied in the backwards direction to calculate A_6, B_6 and C_6.

$$\begin{aligned} A_6 &= C_7 + \text{even}\,(B_7) \ , \\ B_6 &= \left(A_7 \times K_7^{-1}\right) - \text{odd}\,(B_7) \ , \\ C_6 &= B_7 \oplus X_7 \ . \end{aligned} \qquad (13)$$

6. Note that, because A_3, B_3, C_3, A_6, B_6 and C_6 are now known, the algorithm from Sect. 3 can be applied to determine the unique solution for X_4, X_5 and X_6.

$$\begin{aligned} A_5 &= C_6 + \text{even}\,(B_6) \ , \\ B_5 &= \left(A_6 \times K_6^{-1}\right) - \text{odd}\,(B_6) \ , \\ B_4 &= \left(A_5 \times K_5^{-1}\right) - \text{odd}\,(B_5) \ , \\ A_4 &= K_4 \times (B_3 + \text{odd}\,(B_4)) \ , \\ C_4 &= A_3 - \text{even}\,(B_4) \ , \\ C_5 &= A_4 - \text{even}\,(B_5) \ , \\ X_4 &= C_3 \oplus B_4 \ , \\ X_5 &= C_4 \oplus B_5 \ , \\ X_6 &= C_5 \oplus B_6 \ . \end{aligned} \qquad (14)$$

7. Finally, apply the key schedule, which is given in (4), to compute the correct value for Y_7 from the message words X_0 through X_7, all of which have now been determined. Verify if the guess for Y_7 made in step 2 is correct. If it is, a preimage has been found. If not, restart from step 4 with a different choice for X_7.

The probability that the guess for Y_7 is correct is 2^{-64} so we expect to find a preimage after 2^{64} tries. Note that one attempt requires just 8 rounds of the state update transformation and 5 equations of the key schedule algorithm, which is only about 2/3 of the computations of a compression function evaluation. For simplicity, we assume that every equation of the key schedule algorithm takes an equivalent amount of work. Hence, the overall complexity of the attack is equivalent to slightly less than $2^{63.5}$ evaluations of the compression function.

4.2 Extension to Tiger-13

The attack can be extended to 13 rounds, by additionally guessing the value of X_{12} before the attack and verifying if the guess was correct afterwards. This again

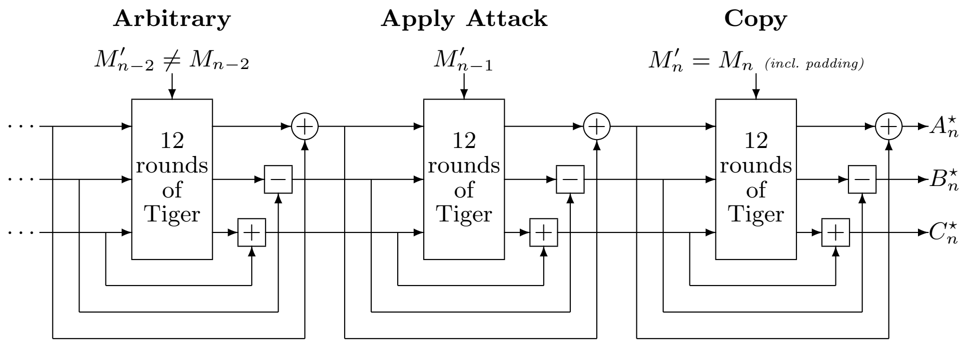


Figure 2 – Constructing second preimages for Tiger-12.

happens with a probability of 2^{-64}, yielding a total complexity of $2^{127.5}$. While it is theoretically possible to make an extension towards 14 rounds of Tiger, this hardly has an advantage over a simple exhaustive search.

5 First and Second Preimages for Tiger-12

The technique that has been developed in the previous section will now be applied to construct first and second preimages for Tiger-12. An extension of this construction to Tiger-13 is also possible.

5.1 Second Preimages for Tiger-12

Figure 2 shows how second preimages for Tiger-12 can be constructed, for (padded) messages with at least two message blocks and no padding bits in the first message block. This is equivalent to the requirement that the given message is at least 512 bits long.

In order to circumvent any issues that arise from the padding (which includes the message length) we choose the length of the preimage to be equal to that of the given message. We can hence reuse the last message block from the given message. All message blocks from the beginning up to the second to last message block can be chosen arbitrarily. This leaves us with exactly one message block, the central block in Fig. 2. Because the chaining values are known before and after this block, the attack from Sect. 4 can be applied. Of course a trivial generalisation where more than one message block is copied from the given message exists. In this case, the attack is applied to an earlier message block instead.

This procedure to find second preimages adds negligible overhead to the attack as described in Sect. 4. Hence, the time complexity remains at $2^{63.5}$ evaluations of the Tiger-12 compression function.

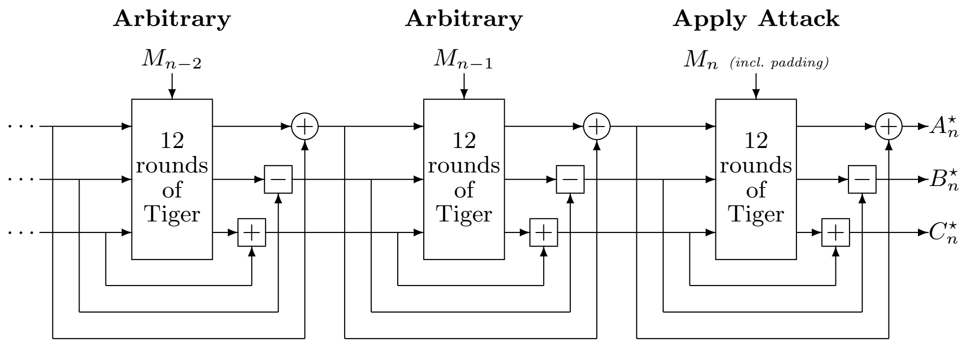


Figure 3 – Constructing first preimages for Tiger-12.

5.2 First Preimages for Tiger-12

Finding first preimages is a bit more involved due to the fact that there is no given message which can be used to easily circumvent issues originating from the padding. To construct first preimages for Tiger-12, we proceed as follows.

First we choose the message length such that only a single bit of padding will be placed in X_6 of the last message block. This is equivalent to choosing a message length of $k \cdot 512 + 447$ bits, where k is a positive integer. Next, as shown in Fig. 3, all message blocks besides the last one are chosen arbitrarily and the attack is applied to this last block.

By choosing the message length in this way, X_7 of the last message block contains the message length as a 64-bit integer, which is fixed. Hence we can no longer choose X_7 freely during step 4 of the attack. By using the freedom in the choice of Y_7 in step 2 instead, the attack still works. Because step 3 is now also repeated, a larger part of the key schedule has to be redone on every attempt. The complexity figure of $2^{63.5}$ compression function evaluations can however be maintained because even with the larger part of the key schedule, the work of a single attempt does not exceed 70% — a fraction $2^{-0.5}$ — of a compression function evaluation. But additionally, we have to verify if the last bit of X_6 is a "1", as dictated by the padding rule. This happens with probability 2^{-1}, resulting in an overall complexity of $2^{64.5}$ compression function evaluations.

Note that the first preimages constructed in this way do not contain an integer number of bytes, which may not be acceptable. This problem can be solved by choosing the message length equal to $k \cdot 512 + 440$ bits instead. The only difference is that X_6 of the last message block now contains an entire byte of padding. The probability that this byte turns out to be correct after executing the attack is only 2^{-8}, and hence the overall complexity increases to $2^{71.5}$ compression function evaluations.

5.3 Extension to Tiger-13

Both attacks can be extended to Tiger-13, as explained in Sect. 4.2. The complexities become $2^{127.5}$ for second preimages, $2^{128.5}$ for first preimages and $2^{135.5}$ for first preimages of an integer number of bytes. A similar extension to Tiger-14 could be made, but as previously explained it does not give any advantage over an exhaustive search.

6 Conclusion

In this paper we have shown preimage attacks on reduced variants of the Tiger hash function. A method to find preimages for the compression function of Tiger-12 and Tiger-13 with a complexity of $2^{63.5}$ and $2^{127.5}$, respectively, was described. It was shown how to construct first and second preimages for these variants of Tiger based on this method. To the best of our knowledge, this is the first result with respect to preimages of the Tiger hash function.

Acknowledgements

We would like to thank Florian Mendel, Christian Rechberger, Hirotaka Yoshida for interesting discussions, and the anonymous reviewers for their helpful comments.

This work was supported in part by the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

[1] R. J. Anderson and E. Biham. Tiger: A fast new hash function. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop — FSE '96*, volume 1039 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 1996.

[2] H. Dobbertin. The first two rounds of MD4 are not one-way. In S. Vaudenay, editor, *Fast Software Encryption, 5th International Workshop — FSE '98*, volume 1372 of *Lecture Notes in Computer Science*, pages 284–292. Springer, 1998.

[3] J. Kelsey and S. Lucks. Collisions and near-collisions for reduced-round Tiger. In M. J. B. Robshaw, editor, *Fast Software Encryption, 13th International*

Workshop — FSE 2006, volume 4047 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2006.

[4] F. Mendel, B. Preneel, V. Rijmen, H. Yoshida, and D. Watanabe. Update on Tiger. In R. Barua and T. Lange, editors, *Progress in Cryptology — INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 2006.

[5] F. Mendel and V. Rijmen. Cryptanalysis of the Tiger hash function. In K. Kurosawa, editor, *Advances in Cryptology — ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 536–550. Springer, 2007.

[6] B. Preneel. *Analysis and Design of Cryptographic Hash Functions.* PhD thesis, Katholieke Universiteit Leuven, 1993.

[7] B. Preneel. Cryptographic primitives for information authentication — state of the art. In B. Preneel and V. Rijmen, editors, *State of the Art in Applied Cryptography*, volume 1528 of *Lecture Notes in Computer Science*, pages 49–104. Springer, 1998.

Publication

Trivial Collisions for Simplified and Reduced SHA-2

Publication Data

Sebastiaan Indesteege. Trivial collisions for simplified and reduced SHA-2. Technical report, COSIC, January 2008.

Contributions

- Principal author.

Trivial Collisions for Simplified and Reduced SHA-2

Sebastiaan Indesteege*

Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven.
Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
`sebastiaan.indesteege@esat.kuleuven.be`

Abstract. In this report we describe a trivial method to find collisions for strongly simplified variants of the SHA-2 family of hash functions. The simplifications include the linearization of the message expansion by replacing modular addition with XOR, and the reduction of the number of steps to 24 out of 64 (or 80). These simplifications allow to find collision pairs for any digest length with an expected time complexity of just 2^8 compression function evaluations. The same method can be applied to 30 steps of SHA-1, where the expected workload is about 2^5 compression function evaluations.

Key words: hash functions, collisions, SHA-2

1 Introduction

This report describes a collision finding attack on simplified variants of the SHA-2 family of hash functions.

The SHA-2 Family. The SHA-2 family consists of several iterated cryptographic hash functions with different digest sizes built on similar compression functions, e.g., SHA-256, SHA-384 and SHA-512. For a complete specification of the SHA-2 hash functions, we refer to [4]. Because of the similarity between the members of the SHA-2 family, we will focus on SHA-256 here. All the results in this paper can be equally applied to the other members of the SHA-2 family.

Brief Summary of Related Work. Gilbert and Handschuh [1] showed a 9 step local collision for SHA-256 with probability 2^{-66}. This was improved by Hawkes et al. [2] to 2^{-39}. In [6], a variant of SHA-256 where all modular additions are replaced by XOR was studied, resulting in pseudo-collisions for 34 steps of this variant. Mendel et al. [3] reported collision producing characteristics for step-reduced, but otherwise unmodified SHA-256, for up to 18 steps.

*F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

2 Description of SHA-256

The compression function of SHA-256 consists of a message expansion, which transforms a 512-bit message block into 64 expanded message words W_i of 32 bits each, and a state update transformation. The latter updates eight 32-bit state variables $A, \ldots, H$ in 64 identical steps, each using one expanded message word. The message expansion can be defined recursively as follows.

$$W_i = \begin{cases} M_i & \text{for } 0 \leq i < 16 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} & \text{for } 16 \leq i < 64 \end{cases} \quad . \tag{1}$$

The functions $\sigma_0(x)$ and $\sigma_1(x)$ are given by

$$\begin{aligned} \sigma_0(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3) \ , \\ \sigma_1(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10) \ . \end{aligned} \tag{2}$$

The state update transformation updates two of the state variables in every step. It uses the bitwise Boolean functions f_{IF} and f_{MAJ} as well as the GF(2)-linear functions

$$\begin{aligned} \Sigma_0(x) &= (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22) \ , \\ \Sigma_1(x) &= (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25) \ . \end{aligned} \tag{3}$$

The following equations describe the state update transformation, where K_i is a step constant.

$$\begin{aligned} T_1 &= H_i + \Sigma_1(E_i) + f_{\mathrm{IF}}(E_i, F_i, G_i) + K_i + W_i \ , \\ T_2 &= \Sigma_0(A_i) + f_{\mathrm{MAJ}}(A_i, B_i, C_i) \ , \\ A_{i+1} &= T_1 + T_2 \ , \ B_{i+1} = A_i \ , \ C_{i+1} = B_i \ , \ D_{i+1} = C_i \ , \\ E_{i+1} &= D_i + T_1 \ , \ F_{i+1} = E_i \ , \ G_{i+1} = F_i \ , \ H_{i+1} = G_i \ . \end{aligned} \tag{4}$$

After 64 rounds, the initial state variables are fed forward using word-wise addition modulo 2^{32}.

2.1 A Simplified Variant of SHA-256

The simplified variant studied in this article differs from the real SHA-256 in two ways. First, all additions modulo 2^{32} in the message expansion are replaced by XORs, making it GF(2)-linear. Second, the number of steps is reduced to 24. Hence, the simplified message expansion becomes

$$W_i = \begin{cases} M_i & \text{for } 0 \leq i < 16 \\ \sigma_1(W_{i-2}) \oplus W_{i-7} \oplus \sigma_0(W_{i-15}) \oplus W_{i-16} & \text{for } 16 \leq i < 24 \end{cases} \quad . \tag{5}$$

We will refer to this variant as SHA-256-XOR-24. The same simplifications can be applied to the other members of the SHA-2 family.

3 Finding Collisions

In this section we describe how collisions for SHA-256-XOR-24 (and other simplified SHA-2 members like SHA-384-XOR-24 and SHA-512-XOR-24) can be found using a technique that is very similar to single-message modification. Single-message modification was first introduced by Wang [5] in collision attacks on MD5, SHA-0 and others.

3.1 Alternate Description of SHA-256

In SHA-0 and SHA-1, only a single state variable is updated in every step. This naturally leads to a description where only the first state variable is considered. Something similar can be done with the SHA-2 hash functions, even though two state variables are updated in every step.

To accomplish this, we derive from the state update equations (4) a series of equations which express the inputs of the i-th state update transformation, $A_i, \ldots, H_i$, as a function of only A_i through A_{i-7}.

$$\begin{array}{rcl}
B_i &=& A_{i-1} \ , \\
C_i &=& A_{i-2} \ , \\
D_i &=& A_{i-3} \ , \\
E_i &=& A_{i-4} + A_i - \Sigma_0(A_{i-1}) - f_{\mathrm{MAJ}}(A_{i-1}, A_{i-2}, A_{i-3}) \ , \\
F_i &=& A_{i-5} + A_{i-1} - \Sigma_0(A_{i-2}) - f_{\mathrm{MAJ}}(A_{i-2}, A_{i-3}, A_{i-4}) \ , \\
G_i &=& A_{i-6} + A_{i-2} - \Sigma_0(A_{i-3}) - f_{\mathrm{MAJ}}(A_{i-3}, A_{i-4}, A_{i-5}) \ , \\
H_i &=& A_{i-7} + A_{i-3} - \Sigma_0(A_{i-4}) - f_{\mathrm{MAJ}}(A_{i-4}, A_{i-5}, A_{i-6}) \ .
\end{array} \tag{6}$$

Substituting these into the state update transformation of SHA-256 (4) yields an alternative description requiring only a single state variable. This description can be written as

$$A_{i+1} = F(A_i, A_{i-1}, A_{i-2}, A_{i-3}, A_{i-4}, A_{i-5}, A_{i-6}, A_{i-7}) + W_i \ . \tag{7}$$

The function $F(\cdot)$ encapsulates (4) and (6) except for the addition of the expanded message word W_i.

Ignoring the message expansion, it is easy to see that control over eight consecutive expanded message words allows for any difference in the state variables to be eliminated. This is very similar to the idea of single message modification [5]. Note that this description of SHA-256 is only interesting for analysis purposes, not for implementation.

3.2 Inserting Odd Additive Differences.

Consider a pair of 32-bit words $\langle X, X' \rangle$ having an XOR difference of `0xffffffff`, i.e., there is a difference in every bit. What are the possible additive differences that can be achieved by such a pair?

Table 1 – The SHA-256-XOR-24 expanded message difference.

$\Delta W_0 =$ `0x56c4b38b`	$\Delta W_8 =$ `0xffffffff`	$\Delta W_{16} =$ `0x00000000`
$\Delta W_1 =$ `0x1e01bbe2`	$\Delta W_9 =$ `0xffffffff`	$\Delta W_{17} =$ `0x00000000`
$\Delta W_2 =$ `0x71721c34`	$\Delta W_{10} =$ `0xffffffff`	$\Delta W_{18} =$ `0x00000000`
$\Delta W_3 =$ `0x037ab391`	$\Delta W_{11} =$ `0xffffffff`	$\Delta W_{19} =$ `0x00000000`
$\Delta W_4 =$ `0x0c28f460`	$\Delta W_{12} =$ `0xffffffff`	$\Delta W_{20} =$ `0x00000000`
$\Delta W_5 =$ `0xefbc47ff`	$\Delta W_{13} =$ `0xffffffff`	$\Delta W_{21} =$ `0x00000000`
$\Delta W_6 =$ `0xfdc03800`	$\Delta W_{14} =$ `0xffffffff`	$\Delta W_{22} =$ `0x00000000`
$\Delta W_7 =$ `0x1fffffff`	$\Delta W_{15} =$ `0xffffffff`	$\Delta W_{23} =$ `0x00000000`

From two's complement arithmetic, we know that the additive inverse of a word X can be found as $-X = \overline{X} + 1$, where $\overline{X}$ is the one's complement of X. The additive difference of the pair $\langle X, X' \rangle$ is

$$\delta X = X' - X = \overline{X} - X = \overline{X} + (\overline{X} + 1) = 2\overline{X} + 1 \ . \qquad (8)$$

Hence, any *odd* additive difference can be generated by an appropriate choice of the word X. There are exactly two possible choices for X for each odd additive difference as the most significant bit of X does not influence the difference δX.

3.3 The Message Difference

For SHA-256-XOR-24, there exists a message difference that will produce a difference of `0xffffffff` in the eight expanded message words W_8 through W_{15} and a zero difference in W_{16} through W_{23}. Indeed, since the message expansion of this simplified SHA-256 variant is GF(2)-linear one can simply use (5) to determine the differences in the first eight expanded message words. Table 1 shows the expanded message difference.

3.4 The Collision Search

Putting the pieces together yields a simple collision finding algorithm for SHA256-XOR-24. The algorithm proceeds as follows:

1. Choose the message words W_0 through W_7 arbitrarily. Imposing the message difference from Tbl. 1 allows to compute W'_0 through W'_7.

2. For each i, $8 \le i < 16$,

   (a) Determine the additive difference that needs to be introduced in step i to ensure a zero difference in A_{i+1}, i.e.,

$$\begin{aligned} \delta W_i &= W'_i - W_i \\ &= F(A_i, A_{i-1}, A_{i-2}, A_{i-3}, A_{i-4}, A_{i-5}, A_{i-6}, A_{i-7}) \\ &\quad -F(A'_i, A'_{i-1}, A'_{i-2}, A'_{i-3}, A'_{i-4}, A'_{i-5}, A'_{i-6}, A'_{i-7}) \ . \end{aligned} \qquad (9)$$

   (b) With probability 1/2 the difference δW_i is odd. In this case, there are two suitable choices for W_i. In case δW_i is even, we start over with a different choice of W_0 through W_7.

3. After 16 steps, a zero difference in reached in the internal state. Because the message difference is zero in the next eight rounds, this zero difference is maintained with certainty for an additional eight rounds.

Under reasonable independence assumptions, the overall success probability is 2^{-8}, hence we expect to find a collision pair after about 2^8 attempts. An early abort strategy and not fully backtracking slightly improves this. An example collision pair is given in Tbl. 2.

Note that a very similar approach can be applied to 30 steps of SHA-1.

4 Conclusion

We described a trivial method to find collisions for strongly simplified variants of the SHA-2 family of hash functions. Linearizing the message expansion by replacing modular addition with XOR and reducing the number of steps to 24 allows to find collision pairs with an expected effort of just 2^8 compression function evaluations, for any digest length.

Acknowledgements

This work was supported in part by the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

[1] H. Gilbert and H. Handschuh. Security analysis of SHA-256 and sisters. In M. Matsui and R. J. Zuccherato, editors, *Selected Areas in Cryptography — SAC 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 175–193. Springer, 2004.

[2] P. Hawkes, M. Paddon, and G. G. Rose. On corrective patterns for the SHA-2 family. Cryptology ePrint Archive, Report 2004/207, 2004. `http://eprint.iacr.org/`.

[3] F. Mendel, N. Pramstaller, C. Rechberger, and V. Rijmen. Analysis of step-reduced SHA-256. In M. J. B. Robshaw, editor, *Fast Software Encryption,*

Table 2 – Example collision pair for SHA-256-XOR-24.

	A_i	B_i	C_i	D_i	E_i	F_i	G_i	H_i	W_i
0	6a09e667	bb67ae85	3c6ef372	a54ff53a	510e527f	9b05688c	1f83d9ab	5be0cd19	6667938f
1	62701bdc	6a09e667	bb67ae85	3c6ef372	ff2f7631	510e527f	9b05688c	1f83d9ab	7bd6934a
2	648a60cf	62701bdc	6a09e667	bb67ae85	3087407e	ff2f7631	510e527f	9b05688c	07dc201c
3	d738ee5a	648a60cf	62701bdc	6a09e667	39bdb81e	3087407e	ff2f7631	510e527f	2b2fc2a7
4	8d535940	d738ee5a	648a60cf	62701bdc	6c82ebbc	39bdb81e	3087407e	ff2f7631	2bf4b111
5	cd9aff09	8d535940	d738ee5a	648a60cf	c6baf39c	6c82ebbc	39bdb81e	3087407e	7292120a
6	867e3675	cd9aff09	8d535940	d738ee5a	3d18a3d9	c6baf39c	6c82ebbc	39bdb81e	0b748afe
7	67d3c637	867e3675	cd9aff09	8d535940	86c7ccdb	3d18a3d9	c6baf39c	6c82ebbc	72b15c2c
8	d4490d64	67d3c637	867e3675	cd9aff09	f26a5de3	86c7ccdb	3d18a3d9	c6baf39c	1da9165d
9	92b431e6	d4490d64	67d3c637	867e3675	1ff2b83a	f26a5de3	86c7ccdb	3d18a3d9	3a5cf5ca
10	04ef8437	92b431e6	d4490d64	67d3c637	b93eb1b4	1ff2b83a	f26a5de3	86c7ccdb	0202394b
11	7719af6b	04ef8437	92b431e6	d4490d64	eb6d55da	b93eb1b4	1ff2b83a	f26a5de3	10c5f64b
12	5333e55b	7719af6b	04ef8437	92b431e6	509ece8d	eb6d55da	b93eb1b4	1ff2b83a	3d868320
13	edb1efd1	5333e55b	7719af6b	04ef8437	f7a76eb7	509ece8d	eb6d55da	b93eb1b4	5a030974
14	011440c7	edb1efd1	5333e55b	7719af6b	cc7583d6	f7a76eb7	509ece8d	eb6d55da	61aa831e
15	b17ccc81	011440c7	edb1efd1	5333e55b	4fe69182	cc7583d6	f7a76eb7	509ece8d	06e69332
16	7c5b7561	b17ccc81	011440c7	edb1efd1	32f9cff8	4fe69182	cc7583d6	f7a76eb7	729011bf
17	4c06af0f	7c5b7561	b17ccc81	011440c7	6e939fad	32f9cff8	4fe69182	cc7583d6	d2d96dbe
18	eb01f745	4c06af0f	7c5b7561	b17ccc81	bdef5445	6e939fad	32f9cff8	4fe69182	a6774853
19	d1d36374	eb01f745	4c06af0f	7c5b7561	501a063e	bdef5445	6e939fad	32f9cff8	86987156
20	7b176155	d1d36374	eb01f745	4c06af0f	cc0f6a70	501a063e	bdef5445	6e939fad	a2c87883
21	40ca974d	7b176155	d1d36374	eb01f745	87d28e05	cc0f6a70	501a063e	bdef5445	fa5f5e00
22	288eedac	40ca974d	7b176155	d1d36374	01b7eb18	87d28e05	cc0f6a70	501a063e	3f535b7e
23	582f3054	288eedac	40ca974d	7b176155	647f5ebd	01b7eb18	87d28e05	cc0f6a70	b8c68fad
24	0e108a8a	582f3054	288eedac	40ca974d	1830da87	647f5ebd	01b7eb18	87d28e05	
H	781a70f1	1396ded9	64fde11e	e61a8c87	693f2d06	ff84c749	213bc4c3	e3b35b1e	

	A'_i	B'_i	C'_i	D'_i	E'_i	F'_i	G'_i	H'_i	W'_i
0	6a09e667	bb67ae85	3c6ef372	a54ff53a	510e527f	9b05688c	1f83d9ab	5be0cd19	30a32004
1	2caba851	6a09e667	bb67ae85	3c6ef372	c96b02a6	510e527f	9b05688c	1f83d9ab	65d728a8
2	954cfd60	2caba851	6a09e667	bb67ae85	008d7692	c96b02a6	510e527f	9b05688c	76ae3c28
3	1f2514b1	954cfd60	2caba851	6a09e667	b0e0bf6b	008d7692	c96b02a6	510e527f	28557136
4	73d035fb	1f2514b1	954cfd60	2caba851	4a98e779	b0e0bf6b	008d7692	c96b02a6	27dc4571
5	bcd102a2	73d035fb	1f2514b1	954cfd60	9e3ff6d2	4a98e779	b0e0bf6b	008d7692	9d2e55f5
6	f05260e1	bcd102a2	73d035fb	1f2514b1	47a31cbb	9e3ff6d2	4a98e779	b0e0bf6b	f6b4b2fe
7	1771d0de	f05260e1	bcd102a2	73d035fb	132e2742	47a31cbb	9e3ff6d2	4a98e779	6d4ea3d3
8	e21633c9	1771d0de	f05260e1	bcd102a2	dd2570fe	132e2742	47a31cbb	9e3ff6d2	e256e9a2
9	92b431e6	e21633c9	1771d0de	f05260e1	9f2d07f4	dd2570fe	132e2742	47a31cbb	c5a30a35
10	04ef8437	92b431e6	e21633c9	1771d0de	67afafb8	9f2d07f4	dd2570fe	132e2742	fdfdc6b4
11	7719af6b	04ef8437	92b431e6	e21633c9	ad423400	67afafb8	9f2d07f4	dd2570fe	ef3a09b4
12	5333e55b	7719af6b	04ef8437	92b431e6	5e6bf4f2	ad423400	67afafb8	9f2d07f4	c2797cdf
13	edb1efd1	5333e55b	7719af6b	04ef8437	f7a76eb7	5e6bf4f2	ad423400	67afafb8	a5fcf68b
14	011440c7	edb1efd1	5333e55b	7719af6b	cc7583d6	f7a76eb7	5e6bf4f2	ad423400	9e557ce1
15	b17ccc81	011440c7	edb1efd1	5333e55b	4fe69182	cc7583d6	f7a76eb7	5e6bf4f2	f9196ccd
16	7c5b7561	b17ccc81	011440c7	edb1efd1	32f9cff8	4fe69182	cc7583d6	f7a76eb7	729011bf
17	4c06af0f	7c5b7561	b17ccc81	011440c7	6e939fad	32f9cff8	4fe69182	cc7583d6	d2d96dbe
18	eb01f745	4c06af0f	7c5b7561	b17ccc81	bdef5445	6e939fad	32f9cff8	4fe69182	a6774853
19	d1d36374	eb01f745	4c06af0f	7c5b7561	501a063e	bdef5445	6e939fad	32f9cff8	86987156
20	7b176155	d1d36374	eb01f745	4c06af0f	cc0f6a70	501a063e	bdef5445	6e939fad	a2c87883
21	40ca974d	7b176155	d1d36374	eb01f745	87d28e05	cc0f6a70	501a063e	bdef5445	fa5f5e00
22	288eedac	40ca974d	7b176155	d1d36374	01b7eb18	87d28e05	cc0f6a70	501a063e	3f535b7e
23	582f3054	288eedac	40ca974d	7b176155	647f5ebd	01b7eb18	87d28e05	cc0f6a70	b8c68fad
24	0e108a8a	582f3054	288eedac	40ca974d	1830da87	647f5ebd	01b7eb18	87d28e05	
H'	781a70f1	1396ded9	64fde11e	e61a8c87	693f2d06	ff84c749	213bc4c3	e3b35b1e	

13th International Workshop — FSE 2006, volume 4047 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2006.

[4] National Institute of Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-3, Oct. 2008.

[5] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[6] H. Yoshida and A. Biryukov. Analysis of a SHA-256 variant. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography — SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2006.

Publication

A Practical Attack on KeeLoq

Publication Data

Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A practical attack on KeeLoq. In Nigel P. Smart, editor, *Advances in Cryptology — EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

Contributions

- Principal author, except for:
  - Appendix A (Related-Key Attacks on KeeLoq).

A Practical Attack on KeeLoq

Sebastiaan Indesteege[1,*], Nathan Keller[2,†], Orr Dunkelman[1], Eli Biham[3], and Bart Preneel[1]

[1] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
{sebastiaan.indesteege,orr.dunkelman,bart.preneel}@esat.kuleuven.be

[2] Einstein Institute of Mathematics, Hebrew University. Jerusalem 91904, Israel.
nkeller@math.huji.ac.il

[3] Computer Science Department, Technion. Haifa 32000, Israel.
biham@cs.technion.ac.il

Abstract. KeeLoq is a lightweight block cipher with a 32-bit block size and a 64-bit key. Despite its short key size, it is widely used in remote keyless entry systems and other wireless authentication applications. For example, authentication protocols based on KeeLoq are supposedly used by various car manufacturers in anti-theft mechanisms. This paper presents a practical key recovery attack against KeeLoq that requires 2^{16} known plaintexts and has a time complexity of $2^{44.5}$ KeeLoq encryptions. It is based on the slide attack and a novel approach to meet-in-the-middle attacks. The fully implemented attack requires 65 minutes to obtain the required data and 7.8 days of calculations on 64 CPU cores. A variant which requires 2^{16} chosen plaintexts needs only 3.4 days on 64 CPU cores. Using only 10 000 euro, an attacker can purchase a cluster of 50 dual core computers that will find the secret key in about two days. We investigated the way KeeLoq is intended to be used in practice and conclude that our attack can be used to subvert the security of real systems. An attacker can acquire chosen plaintexts in practice, and one of the two suggested key derivation schemes for KeeLoq allows to recover the master secret from a single key.

Key words: KeeLoq, cryptanalysis, block ciphers, slide attacks, meet-in-the-middle attacks.

[*] F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

[†] This author is supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

1 Introduction

The KeeLoq technology [13] by Microchip Technology Inc. includes the KeeLoq block cipher and several authentication protocols built on top of it. The KeeLoq block cipher allows for very low cost and power efficient hardware implementations. This property has undoubtedly contributed to the popularity of the cipher in various wireless authentication applications. For example, multiple car manufacturers supposedly use, or have used KeeLoq to protect their cars against theft [5–7, 9, 17].[4]

Despite its design in the 80's, the first cryptanalysis of KeeLoq was only published by Bogdanov [6] in February 2007. This attack is based on the slide technique and a linear approximation of the non-linear Boolean function used in KeeLoq. The attack has a time complexity of 2^{52} KeeLoq encryptions and requires 16 GB of storage. It also requires the entire codebook, i.e., 2^{32} known plaintexts.

Courtois et al. apply algebraic techniques to cryptanalyse KeeLoq [7, 9]. Although a direct algebraic attack fails for the full cipher, they reported various successful slide-algebraic attacks. For example, they claim that an algebraic attack can recover the key when given a slid pair in 2.9 seconds on average. As there is no way to ensure or identify the existence of a slid pair in the data sample, the attack is simply repeated 2^{32} times, once for each pair generated from 2^{16} known plaintexts. They also described attacks requiring the entire codebook, which exploit certain assumptions with respect to fixed points of the internal state. The fastest of these requires 2^{27} KeeLoq encryptions and has an estimated success probability of 44% [7].

In [5], Bogdanov published an updated version of his attack. A refined complexity analysis yields a slightly smaller time complexity, i.e., $2^{50.6}$ KeeLoq encryptions while still requiring the entire codebook. This paper also includes an improvement using the work of Courtois et al. [9] on the cycle structure of the cipher. We note that the time complexity of the attack using the cycle structure given in [5] is based on an assumption from an earlier version of [9], that a random word can be read from 16 GB of memory with a latency of only 1 clock cycle. This is very unrealistic in a real machine, so the actual time complexity is probably much higher. In a later version of [9], this assumption on the memory latency was changed to be 16 clock cycles.

Our practical attack is based on the slide attack as well. However, unlike other attacks, we combine it with a novel meet-in-the-middle attack. The optimised version of the attack uses 2^{16} known plaintexts and has a time complexity of $2^{44.5}$ KeeLoq encryptions. We have implemented our attack and the total running time is roughly 500 days. As the attack is fully parallelizable, given x CPU cores, the total running time is only $500/x$ days. A variant which requires 2^{16} chosen plaintexts needs only $218/x$ days on x CPU cores. For example, for 10 000 euro, one can obtain 50 dual core computers, which will take about two days to find

[4]We verified these claims to the best of our ability, however, no car manufacturer seems eager to publically disclose which algorithms are used.

the key. Another, probably even cheaper, though illegal option would be to rent a botnet to carry out the computations.

KeeLoq is used in two protocols, the "Code Hopping" and the "Identify Friend or Foe (IFF)" protocol. In practice, the latter protocol, a simple challenge response protocol, is the most interesting target to acquire the data that is necessary to mount the attack. Because the challenges are not authenticated in any way, an attacker can obtain as many chosen plaintext/ciphertext pairs as needed from a transponder (e.g., a car key) implementing this protocol. Depending on the transponder, it takes 65 or 98 minutes to gather 2^{16} plaintext/ciphertext pairs.

Finally, as was previously noted by Bogdanov [5], we show that one of the two suggested key derivation algorithms is blatantly flawed, as it allows an attacker to reconstruct many secret keys once a single secret key has been exposed.

Given that KeeLoq is a cipher that is widely used in practice, side-channel analysis may also be a viable option for attacking chips that implement KeeLoq. However, we do not consider this type of attack in this paper. One could also attack the "Identify Friend or Foe (IFF)" protocol itself. For instance, as the responses are only 32 bits long, one could mount the birthday attack using 2^{16} known challenge/response pairs. This would not recover the secret key, thus posing less of a threat to the overall security of the system.

Table 1 presents an overview of the known attacks on KeeLoq, including ours. In order to make comparisons possible, we have converted all time complexities to the number of KeeLoq encryptions needed for the attack.[5]

The structure of this paper is as follows. In Sect. 2, we describe the KeeLoq block cipher and how it is intended to be used in practice. Our attacks are described in Sect. 3. In Sect. 4 we discuss our experimental results and in Sect. 5 we show the relevance of our attacks in practice. Finally, in Sect. 6 we conclude. In Appendix A, we explore some related key attacks on KeeLoq that are more of theoretical interest.

2 Description and Usage of KeeLoq

2.1 The KeeLoq Block Cipher

The KeeLoq block cipher has a 32-bit block size and a 64-bit key. It consists of 528 identical rounds each using one bit of the key. A round is equivalent to an iteration of a non-linear feedback shift register (NLFSR), as shown in Fig. 1.

More specifically, let $Y^{(i)} = (y_{31}^{(i)}, \ldots, y_0^{(i)}) \in \{0,1\}^{32}$ be the input to round i ($0 \leq i < 528$) and let $K = (k_{63}, \ldots, k_0) \in \{0,1\}^{64}$ be the key. The input to round 0 is the plaintext: $Y^{(0)} = P$. The ciphertext is the output after 528 rounds:

[5]We list slightly better complexities for the attacks from [7,9] because we used a more realistic conversion factor from CPU clocks to KeeLoq rounds (i.e., 12 rather than 4 CPU cycles per KeeLoq round).

Table 1 – An overview of the known attacks on KeeLoq.

Attack Type	Complexity			Reference
	Data	Time	Memory	
Time-Memory Trade-Off	2 CP	$2^{42.7}$	$\approx 100\,\mathrm{TB}$	[11]
Slide/Algebraic	2^{16} KP	$2^{65.4}$	?	[7,9]
Slide/Algebraic	2^{16} KP	$2^{51.4}$	?	[7,9]
Slide/Guess-and-Determine	2^{32} KP	2^{52}	16 GB	[6]
Slide/Guess-and-Determine	2^{32} KP	$2^{50.6}$	16 GB	[5]
Slide/Cycle Structure	2^{32} KP	$2^{39.4}$	16.5 GB	[9]
Slide/Cycle/Guess-and-Det.[a]	2^{32} KP	(2^{37})	16.5 GB	[5]
Slide/Fixed Points	2^{32} KP	2^{27}	$> 16\,\mathrm{GB}$	[7]
Slide/Meet-in-the-Middle	2^{16} KP	$2^{45.0}$	$\approx 2\,\mathrm{MB}$	Sect. 3.3
Slide/Meet-in-the-Middle	2^{16} KP	$2^{44.5}$	$\approx 3\,\mathrm{MB}$	Sect. 3.4
Slide/Meet-in-the-Middle	2^{16} CP	$2^{44.5}$	$\approx 2\,\mathrm{MB}$	Sect. 3.5
Time-Memory-Data Trade-Off	68 CP, 34 RK	$2^{39.3}$	$\approx 10\,\mathrm{TB}$	[2]
Related Key	66 CP, 34 RK$^{\ggg}$	negligible	negligible	Sect. A.1
Related Key	512 CP, 2 RK$^{\ggg}$	2^{32}	negligible	Sect. A.1
Related Key/Slide/MitM	2^{17} CP, 2 RK$^{\oplus}$	$2^{41.9}$	$\approx 16\,\mathrm{MB}$	Sect. A.2

Time complexities are expressed in full KeeLoq encryptions (528 rounds).
KP: known plaintexts; CP: chosen plaintexts
RK$^{\ggg}$: related keys (by rotation); RK$^{\oplus}$: related keys (flip LSB)

[a]The time complexity for this attack is based on very unrealistic memory latency assumptions and hence will be much higher in practice.

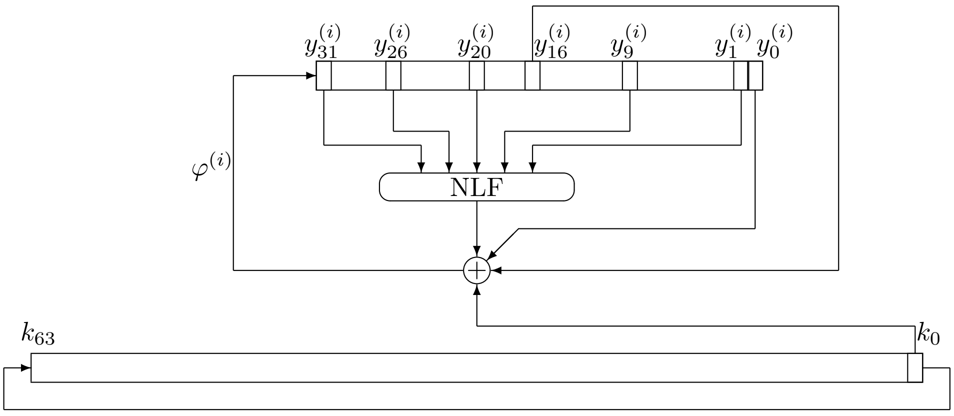


Figure 1 – The i-th KeeLoq encryption cycle.

$C = Y^{(528)}$. The round function can be described as follows (see Fig. 1):

$$\begin{aligned} \varphi^{(i)} &= \mathrm{NLF}\left(y_{31}^{(i)}, y_{26}^{(i)}, y_{20}^{(i)}, y_{9}^{(i)}, y_{1}^{(i)}\right) \oplus y_{16}^{(i)} \oplus y_{0}^{(i)} \oplus k_{i \bmod 64} \ , \\ Y^{(i+1)} &= (\varphi^{(i)}, y_{31}^{(i)}, \ldots, y_{1}^{(i)}) \ . \end{aligned} \tag{1}$$

The non-linear function NLF is a Boolean function of 5 variables with output vector $3A5C742E_x$ — i.e., NLF(i) is the i-th bit of this hexadecimal constant, where bit 0 is the least significant bit. We can also represent the non-linear function in its algebraic normal form (ANF):

$$\begin{aligned} \mathrm{NLF}(x_4, x_3, x_2, x_1, x_0) = \ & x_4x_3x_2 \oplus x_4x_3x_1 \oplus x_4x_2x_0 \oplus x_4x_1x_0 \oplus \\ & x_4x_2 \oplus x_4x_0 \oplus x_3x_2 \oplus x_3x_0 \oplus x_2x_1 \oplus x_1x_0 \oplus \\ & x_1 \oplus x_0 \ . \end{aligned} \tag{2}$$

Decryption uses the inverse round function, where i now ranges from 528 down to 1.

$$\begin{aligned} \theta^{(i)} &= \mathrm{NLF}\left(y_{30}^{(i)}, y_{25}^{(i)}, y_{19}^{(i)}, y_{8}^{(i)}, y_{0}^{(i)}\right) \oplus y_{15}^{(i)} \oplus y_{31}^{(i)} \oplus k_{i-1 \bmod 64} \ , \\ Y^{(i-1)} &= (y_{30}^{(i)}, \ldots, y_{0}^{(i)}, \theta^{(i)}) \ . \end{aligned} \tag{3}$$

There used to be some ambiguity about the correct position of the taps. Our description agrees with the “official” documentation [5–7, 15]. Additionally, we have used test vectors generated by an actual HSC410 chip [16], manufactured by Microchip Inc., to verify that our description and implementation of KeeLoq are indeed correct. Finally, we note that our attacks are unaffected by this difference.

2.2 Protocols Built on KeeLoq

A device like the HCS410 by Microchip Technology Inc. [16] supports two authentication protocols based on KeeLoq: “KeeLoq Hopping Codes” and “KeeLoq Identify Friend or Foe (IFF)”. The former uses a 16-bit secret counter, synchronised between both parties. In order to authenticate, the encoder (e.g., a car key) increments the counter and sends the encrypted counter value to the decoder (e.g., the car), which verifies if the received ciphertext is correct. In practice, this protocol would be initiated by a button press of the car owner.

The second protocol, “KeeLoq Identify Friend or Foe (IFF)” [16], is a simple challenge response protocol. The decoder (e.g., the car) sends a 32-bit challenge. The transponder (e.g., the car key) uses the challenge as a plaintext, encrypts it with the KeeLoq block cipher[6] under the shared secret key, and replies with the ciphertext. This protocol is executed without any user interaction whenever the transponder receives power and an activation signal via inductive coupling from

[6]This corresponds to what is called the “HOP algorithm” in [16]. The other option, the so-called “IFF algorithm”, uses a reduced version of KeeLoq with 272 rounds instead of 528. Our attacks are also applicable to this variant, without any change.

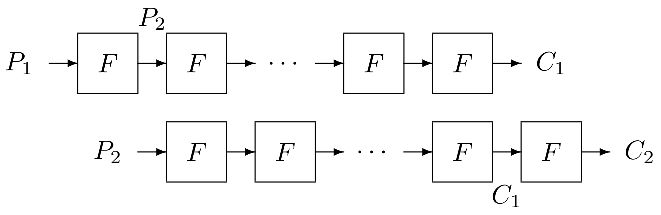


Figure 2 – A typical slide attack.

a nearby decoder. Hence, no battery or button presses are required. It could for instance be used in vehicle immobilisers by placing the decoder near the ignition. Inserting the car key in the ignition would place the transponder within range of the decoder. The latter would then activate the transponder and execute the protocol, all completely transparent to the user. The car would then either disarm the immobiliser or activate the alarm, depending on whether the authentication was successful.

Of course both protocols can be used together in a single device, thereby saving costs. For example, the HCS410 chip [16] supports this combined mode of operation, possibly using the same secret key for both protocols, depending on the configuration options used.

3 Our Attacks on KeeLoq

This section describes our attacks on KeeLoq. We combine a slide attack with a novel meet-in-the-middle approach to recover the key from a slid pair. First we explain some preliminaries that are used in the attacks. Then we proceed to the description of the attack scenario using known plaintexts and a generalisation thereof. Finally, we show how chosen plaintexts can be used to improve the attack.

3.1 The Slide Property

Slide attacks were introduced by Biryukov and Wagner [3] in 1999. The typical candidate for a slide attack is a block cipher consisting of a potentially very large number of iterations of an identical key dependent permutation F. In other words, the subkeys are repeated and therefore the susceptible cipher can be written as

$$C = \underbrace{F\left(F\left(\dots F(P)\right)\right)}_{r} = F^r(P) \ . \tag{4}$$

This permutation does not necessarily have to coincide with the rounds of the cipher, i.e., F might combine several rounds of the cipher.

A slide attack aims at exploiting such a self-similar structure to reduce the strength of the entire cipher to the strength of F. Thus, it is independent of the number of rounds of the cipher. To accomplish this, a so-called *slid pair* is needed. This is a pair of plaintexts that satisfies the slide property

$$P_2 = F(P_1) \ . \tag{5}$$

We depict such a slid pair in Fig. 2. For a slid pair, the corresponding ciphertexts also satisfy the slide property, i.e., $C_2 = F(C_1)$. By repeatedly encrypting this slid pair, we can generate as many slid pairs as needed [4, 10]. As each slid pair gives us a pair of corresponding inputs and outputs of the key dependent permutation F, it can be used to mount an attack against F.

KeeLoq has 528 identical rounds, each using one bit of the 64-bit key. After 64 rounds the key is repeated. So in the case of KeeLoq, we combine 64 rounds into F. However, because the number of rounds in the cipher is not an integer multiple of 64, a straightforward slid attack is not possible. A solution to this problem is to guess the 16 least significant bits of the key and use this to strip off the final 16 rounds. Then, a slide attack can be applied to the remaining 512 rounds [6, 7, 9].

In order to get a slid pair, 2^{16} known plaintexts are used. As the block size of KeeLoq is 32 bits, we expect that a random set of 2^{16} plaintexts contains a slid pair due to the birthday paradox.[7] Determining which pair is a slid pair is done by the attack itself. Simply put, the attack is attempted with every pair. If it succeeds, the pair is a slid pair, otherwise it is not.

3.2 Determining Key Bits

If two intermediate states of the KeeLoq cipher, separated by 32 rounds (or less) are known, all the key bits used in these rounds can easily be recovered. This was first described by Bogdanov [6], who refers to it as the "linear step" of his attack.

Let $Y^{(i)} = (y_{31}^{(i)}, \ldots, y_0^{(i)})$ and $Y^{(i+t)} = (y_{31}^{(i+t)}, \ldots, y_0^{(i+t)})$ be the two known states; $t \leq 32$. If we encrypt $Y^{(i)}$ by one round, the newly generated bit is

$$\varphi^{(i)} = \mathrm{NLF}\left(y_{31}^{(i)}, y_{26}^{(i)}, y_{20}^{(i)}, y_{9}^{(i)}, y_{1}^{(i)}\right) \oplus y_{16}^{(i)} \oplus y_{0}^{(i)} \oplus k_{i \bmod 64} \ . \tag{6}$$

Because of the non-linear feedback shift register structure of the round function and since $t \leq 32$, the bit $\varphi^{(i)}$ is equal to $y_{32-t}^{(i+t)}$, which is one of the bits of $Y^{(i+t)}$ and thus known. Hence

$$k_{i \bmod 64} = \mathrm{NLF}\left(y_{31}^{(i)}, y_{26}^{(i)}, y_{20}^{(i)}, y_{9}^{(i)}, y_{1}^{(i)}\right) \oplus y_{16}^{(i)} \oplus y_{0}^{(i)} \oplus y_{32-t}^{(i+t)} \ . \tag{7}$$

By repeating this t times, all t key bits can be recovered. The amount of computations that need to be carried out is equivalent to t rounds of KeeLoq. This simple step will prove to be very useful in our attack.

[7] The probability that a set of 2^{16} random plaintexts contains at least one slid pair is $1 - \left(1 - 2^{-32}\right)^{2^{32}} \approx 0.63$. Hence, the attack has a success probability of about 63%. With not much higher data complexity, higher success rates can be achieved.

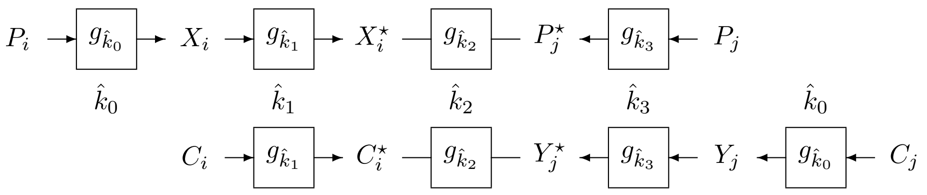


Figure 3 – The notation used in the attack.

3.3 Basic Attack Scenario

We now describe the basic attack scenario, which uses 2^{16} known plaintexts. For clarity, the notation used is shown in Fig. 3 and a pseudocode overview is given in Fig. 4. We denote 16 rounds of KeeLoq by $g_{\hat{k}}$, where $\hat{k}$ denotes the 16 key bits used in these rounds. The 64-bit key k is split into four equal parts: $k = (\hat{k}_3, \hat{k}_2, \hat{k}_1, \hat{k}_0)$, where $\hat{k}_0$ contains the 16 least significant key bits.

As already mentioned in Sect. 3.1, the first step of the attack is to guess $\hat{k}_0$ — the 16 least significant bits of the key. This enables us to partially encrypt each of the 2^{16} plaintexts by 16 rounds (P_i to X_i) and partially decrypt each of the 2^{16} ciphertexts by 16 rounds (C_j to Y_j).

Encrypting X_i by 16 more rounds yields $X_i^\star$. Similarly, decrypting P_j by 16 rounds yields $P_j^\star$ (see Fig. 3). We denote the 16 most significant bits of $X_i^\star$ by $\overline{X_i^\star}$, and the 16 least significant bits of $P_j^\star$ by $\underline{P_j^\star}$. Note that, because $X_i^\star$ and $P_j^\star$ are separated by 16 rounds, it holds that $\overline{X_i^\star} = \underline{P_j^\star}$, provided that P_i and P_j form a slid pair. This is due to the structure of the cipher.

The next step in the attack is to apply a meet-in-the-middle approach. We guess the 16-bit value $\underline{P_j^\star}$. For each plaintext P_j we can then determine $\hat{k}_3$ using the algorithm described in Sect. 3.2. Indeed, as the other bits of $P_j^\star$ are determined by P_j, we know all of $P_j^\star$ when given the plaintext. There is always exactly one solution per plaintext. Using this part of the key, we can now partially decrypt Y_j to $Y_j^\star$. This result is saved in a hash table indexed by the 16-bit value $\underline{Y_j^\star}$. Each record in the hash table holds a tuple consisting of $P_j^\star$, $Y_j^\star$ and the 16 key bits $\hat{k}_3$.

Now we do something similar from the other side. For each plaintext we use the algorithm from Sect. 3.2 to determine $\hat{k}_1$. Again this can be done because we know all of $X_i^\star$, and there is exactly one solution per plaintext. Knowing $\hat{k}_1$, we partially encrypt C_i to $C_i^\star$.

Note that if P_i and P_j are indeed a slid pair their partial encryptions and decryptions (under the correct key) must "meet in the middle". More specifically, it must hold that $\overline{C_i^\star} = \underline{Y_j^\star}$. So, we look for a record in the hash table for which such a collision occurs. Because the hash table is indexed by $\underline{Y_j^\star}$ this can be done very efficiently. A slid pair produces a collision, provided the guesses for $\hat{k}_0$ and $\underline{P_j^\star}$ are correct. Therefore, we are guaranteed that all slid pairs are found at some

```
for all $\hat{k}_0 \in \{0,1\}^{16}$ do
    for all plaintexts $P_i, 0 \leq i < 2^{16}$ do
        Partially encrypt $P_i$ to $X_i$.
        Partially decrypt $C_i$ to $Y_i$.
    end for
    for all $\underline{P_j^\star} \in \{0,1\}^{16}$ do
        for all plaintexts $P_j, 0 \leq j < 2^{16}$ do
            Determine the key bits $\hat{k}_3$.
            Partially decrypt $Y_j$ to $Y_j^\star$.
            Save the tuple $\left\langle P_j^\star, Y_i^\star, \hat{k}_3 \right\rangle$ in a table.
        end for
        for all plaintexts $P_i, 0 \leq i < 2^{16}$ do
            Determine the key bits $\hat{k}_1$.
            Partially encrypt $C_i$ to $C_i^\star$.
            for all collisions $\overline{C_i^\star} = \underline{Y_j^\star}$ in the table do
                Determine the key bits $\hat{k}_2$ from $X_i^\star$ and $P_j^\star$.
                Determine the key bits $\hat{k}_2'$ from $C_i^\star$ and $Y_j^\star$.
                if $\hat{k}_2 = \hat{k}_2'$ then
                    Encrypt 2 known plaintexts with the key $k = (\hat{k}_3, \hat{k}_2, \hat{k}_1, \hat{k}_0)$.
                    if the correct ciphertexts are found then
                        return success (the key is $k$)
                    end if
                end if
            end for
        end for
    end for
end for
return failure (i.e., there was no slid pair)
```

Figure 4 – The attack algorithm.

point. Of course, a collision does not guarantee that the pair is actually a slid pair.

Finally, we check each candidate slid pair found. We determine the remaining key bits $\hat{k}_2$ from $X_i^\star$ and $P_j^\star$ and similarly $\hat{k}_2'$ from $C_i^\star$ and $Y_j^\star$. If $\hat{k}_2$ and $\hat{k}_2'$ are not equal, the candidate pair is not a slid pair. Note that we can determine the key bits one by one and stop as soon as there is a disagreement. This slightly reduces the complexity of the attack.

If $\hat{k}_2 = \hat{k}_2'$, we have found a pair of plaintexts and a key with the property that encrypting P_i by 64 rounds gives P_j and encrypting C_i by 64 rounds gives C_j. This is what is expected from a slid pair. It is however possible that the recovered key is not the correct key, so we can verify it by a trial encryption of one of the known plaintexts. Even if a wrong key is suggested during the attack, and discarded by the trial encryption, we are still guaranteed to find the correct key eventually, provided there is at least one slid pair among the given plaintexts.

Complexity Analysis

Using one round of KeeLoq as a unit, the time complexity of the attack can be expressed as

$$2^{16}\left(32 \cdot 2^{16} + 2^{16}\left(32 \cdot 2^{16} + 2^{16}\left(32 + N_{\text{coll}} \cdot V\right)\right)\right) \quad , \tag{8}$$

when N_{coll} denotes the expected number of collisions for a single guess of $\hat{k}_0$, $\underline{P_j^\star}$ and a given plaintext P_i, and V denotes the average cost of verifying one collision, i.e., checking if it leads to a candidate key and if this key is correct. This follows directly from the description of the attack. As the hash table has 2^{16} entries and a collision is equivalent to a 16-bit condition, $N_{\text{coll}} = 1$. In the verification step, we can determine one bit at a time and stop as soon as there is a disagreement, which happens with probability 1/2. Only when there is no disagreement after 16 key bits, we do two full trial encryptions to check the recovered key. Of course the second trial encryption is only useful if the first one gave the expected result. Hence, due to this early abort technique, the average cost of verifying one collision is

$$V = 2 \cdot \sum_{i=0}^{15} 2^{-i} + 2^{-16} \cdot \left(528 + 528 \cdot 2^{-32}\right) \approx 4 \quad . \tag{9}$$

Thus the overall complexity of the attack is $2^{54.0}$ KeeLoq rounds, which amounts to $2^{45.0}$ full KeeLoq encryptions.

As mentioned before, the data complexity of the attack is 2^{16} known plaintexts. The storage requirements are very modest. The attack stores the plaintext/ciphertext pairs, 2^{16} values for X_i and Y_i, and a hash table with 2^{16} records of 80 bits each. This amounts to a bit over 2 MB of RAM.

3.4 A Generalisation of the Attack

The attack presented in the previous section can be generalised by varying the number of rounds to partially encrypt/decrypt in each step of the attack. We denote by t_p the number of rounds to partially encrypt from the plaintext side (left on Fig. 3) and by t_c the number of rounds to partially decrypt from the ciphertext side (right on Fig. 3). More specifically, encrypting X_i by t_p rounds yields $X_i^\star$, encrypting C_i by t_p rounds yields $C_i^\star$. On the ciphertext side, $P_j^\star$ is obtained by decrypting P_j by t_c rounds and $Y_j^\star$ by decrypting Y_j by t_c rounds. Also, the partial keys $\hat{k}_0$ through $\hat{k}_3$ are adapted accordingly to contain the appropriate key bits.

Let t_o denote the number of bits that, provided P_i and P_j form a slid pair, overlap between $X_i^\star$ and $P_j^\star$. As $X_i^\star$ and $P_j^\star$ are separated by $48 - t_p - t_c$ rounds, it holds that $t_o = 32 - (48 - t_p - t_c) = t_p + t_c - 16$. The t_o least significant bits of $P_j^\star$ are denoted by $\underline{P_j^\star}$ and the t_o most significant bits of $X_i^\star$ are denoted by $\overline{X_i^\star}$.

Depending on the choices for the parameters t_p and t_c, the attack scenario has to be modified slightly. If $t_c < t_o$, not all plaintexts necessarily yield a solution for a given $\underline{P_j^\star}$ when determining $\hat{k}_3 = (k_{63}, \ldots, k_{64-t_c})$ because $t_o - t_c$ of the guessed bits overlap with plaintext bits. Similarly, if $t_c > t_o$, each plaintext is expected to offer multiple solutions because $t_c - t_o$ extra bits have to be guessed before all of $P_j^\star$ is known. From the other side, similar observations can be made.

In Sect. 3.3, the parameters were $t_p = t_c = 16$ which results in $t_o = 16$. It is clear that the choice of these parameters influences both the time and memory complexity of the attack.

Complexity Analysis

The generalisation leads to a slightly more complex formula for expressing the time complexity of the attack. Because of the duality between guessing extra bits and filtering because of overlapping bits, all cases can be expressed in a single formula, which is a generalisation of (8) (i.e., with $t_p = t_c = 16$, it reduces to (8)):

$$2^{16}\left(32 \cdot 2^{16} + 2^{t_o}\left(2t_c \cdot 2^{16+t_c-t_o} + 2^{16+t_p-t_o}\left(2t_p + N_{\mathrm{coll}} \cdot V\right)\right)\right) \ . \tag{10}$$

In the generalised case, finding a collision is equivalent to finding an entry in a table of $2^{16+t_p-t_o}$ elements that satisfies a t_o bit condition, so $N_{\mathrm{coll}} = 2^{16+t_c-t_o}/2^{t_o}$. Verifying a collision now requires an average effort of

$$V = 2 \cdot \sum_{i=0}^{47-t_p-t_c} 2^{-i} + 2^{t_p+t_c-48} \cdot \left(528 + 528 \cdot 2^{-32}\right) \tag{11}$$

KeeLoq rounds. Simplification yields that the total complexity is equal to

$$32 \cdot 2^{32} + 2t_c \cdot 2^{32+t_c} + 2t_p \cdot 2^{32+t_p} + 4 \cdot 2^{80-t_p-t_c} + 528 \cdot 2^{32} \ . \tag{12}$$

The optimum is found when $t_p = t_c = 15$ and thus $t_o = 14$, where the complexity reduces to $2^{53.524}$ KeeLoq rounds or $2^{44.5}$ full KeeLoq encryptions.

The memory requirements in the generalised case can also easily be evaluated. As before, 2^{16} plaintext/ciphertext pairs and 2^{16} values for X_i and Y_i are stored. The hash table now has $2^{16+t_p-t_o}$ entries of $64+t_p$ bits each. For $t_p = t_c = 15$, the required memory is still less than 3 MB.

3.5 A Chosen Plaintext Attack

Using chosen plaintexts instead of known plaintexts, the attack can be improved. Consider the generalised attack from Sect. 3.4 in the case where $t_c < t_o$ (which is equivalent to $t_p > 16$). In this case, the $t_o - t_c$ least significant bits of the plaintext P_j are bits $(t_o, \ldots, t_c+1)$ of $P_j^\star$. Hence, choosing the 2^{16} plaintexts in such a way that these $t_o - t_c$ least significant bits are equal to some constant, only 2^{t_c} guesses for $\underline{P_j^\star}$ have to be made at the beginning of the meet-in-the-middle step, instead of $\overline{2^{t_o}}$.

Complexity Analysis

As chosen plaintexts are only useful for the attack when $t_c < t_o$, we will only consider this case. The time complexity of the attack, in KeeLoq rounds, can be expressed as

$$2^{16}\left(32 \cdot 2^{16} + 2^{t_c}\left(2t_c \cdot 2^{16} + 2^{16+t_p-t_o}\left(2t_p + N_{\text{coll}} \cdot V\right)\right)\right) \ . \tag{13}$$

The expected number of collisions is $N_{\text{coll}} = 2^{16}/2^{t_o}$. The verification cost, V, is given by (11). Simplification yields

$$32 \cdot 2^{32} + 2t_c \cdot 2^{32+t_c} + 2t_p \cdot 2^{48} + 4 \cdot 2^{80-t_p-t_c} + 528 \cdot 2^{32} \ . \tag{14}$$

The optimum is found when $t_p = 20$, $t_c = 13$ and thus $t_o = 17$, where the attack has a time complexity of $2^{53.500}$ KeeLoq rounds or $2^{44.5}$ full KeeLoq encryptions. It is clear that the (theoretical) advantage over the known plaintext attack from Sect. 3.4 is not significant. However, as is discussed in the next section, the chosen plaintext variant can provide a significant gain in our practical implementation, because the verification cost V turns out to be higher there.

The memory complexity is about 2 MB as in Sect. 3.3 because the size of the hash table is the same. The data complexity remains at 2^{16} plaintext/ciphertext pairs, but note that we now require chosen plaintexts instead of known plaintexts.

4 Experimental Results

We have fully implemented and tested the attacks, using both simulated data and real data acquired from a HCS410 chip [16]. We made extensive use of bit slicing to do many encryptions in parallel throughout the implementation. However, because this parallelisation is not useful while verifying a collision, this verification step becomes more expensive in comparison. Hence, the optimal parameters for our implementation differ slightly from the theoretical ones. For the known plaintext attack from Sect. 3.4, the optimal parameters for our implementation were found to be $t_p = t_c = 16$. This means that, at least in our implementation, the best attack is the basic attack from Sect. 3.3. For the chosen plaintext attack, the optimal parameters are $t_p = 22$ and $t_c = 13$.

If we give the correct values for the 16 least significant key bits, the known plaintext attack completes in 10.97 minutes on average.[8] The chosen plaintext attack needs just 4.79 minutes to complete the same task.[9] This large difference can be explained by considering the impact of V, the cost of the verification step, on the time complexity of the attack. If V increases, and t_p and t_c are adapted as needed because their optimal values may change, the time complexity of the known plaintext attack increases much faster than the time complexity of the chosen plaintext attack does. Hence, even though their theoretical time complexities are the same, the chosen plaintext attack performs much better in our practical implementation because V is higher than the theoretical value.

We did not stop either of the attacks once a slid pair and the correct key were found, so we essentially tested the worst-case behaviour of the attack. This also explains the very small standard deviations of the measured running times. The machine used is an AMD Athlon 64 X2 4200+ with 1 GB of RAM (only one of the two CPU cores was used) running Linux 2.6.17. The attack was implemented in C and compiled with gcc version 4.1.2 (using the `-O3` optimiser flag). Critical parts of the code are written in assembly. Because the memory access pattern is random, but predictable to some extent, prefetching helped us to make maximum use of the cache memory.

The known plaintext attack performs over 288 times faster than the fastest attack with the same data complexity from [7, 9], although the actual increase in speed is probably slightly smaller due to the difference in the machines used. Courtois et al. used (a single core of) a 1.66 GHz Intel Centrino Duo microprocessor [8]. The chosen plaintext attack performs more than 661 times faster, but this comparison is not very fair because chosen plaintexts are used. We note that the practicality of our results should also be compared with exhaustive key search due to the small key size. For the price of about 10 000 euro, one can obtain a COPACOBANA machine [12] with 120 FPGAs which is estimated to take about 1000 days to find a single 64-bit KeeLoq key.[10] Using our attack and 50 dual core computers (which can be obtained for roughly the same price), a KeeLoq key can be found in only two days.

[8]We performed 500 experiments. The average running time was 658.15 s and the standard deviation was 1.69 s.

[9]We performed 500 experiments. The average running time was 287.17 s and the standard deviation was 0.55 s.

[10]The estimate was done by adapting the 17 days (worst case) required for finding a 56-bit DES key, taking into consideration the longer key size, the fact that more KeeLoq implementations fit on each FPGA, but in exchange take more clocks to test a key.

5 Practical Applicability of the Attacks

5.1 Gathering Data

One might wonder if it is possible to gather 2^{16} known, or even chosen plaintexts from a practical KeeLoq authentication system. As mentioned in Sect. 2.2, a device like the HCS410 by Microchip Technology Inc. [16] supports two authentication protocols based on KeeLoq: "KeeLoq Hopping Codes" and "KeeLoq Identify Friend or Foe (IFF)". As the initial value of the counter used in "KeeLoq Hopping Codes" is not known, it is not easy to acquire known plaintexts from this protocol apart from trying all possible initial counter values. Also, since only 2^{16} plaintexts are ever used, knowing this sequence of 2^{16} ciphertexts suffices to break the system as this sequence is simply repeated.

The second protocol, "KeeLoq Identify Friend or Foe (IFF)" [16], is more appropriate for our attack. It is executed without any user interaction as soon as the transponder comes within the range of a decoder and is sent an activation signal. The challenges sent by the decoder are not authenticated in any way. Because of this, an adversary can build a rogue decoder which can be used to gather as many plaintext/ciphertext pairs as needed. The plaintexts can be fully chosen by the adversary, so acquiring chosen plaintexts is no more difficult than just known plaintexts. The only requirement is that the rogue decoder can be placed within the range of the victim's transponder for a certain amount of time. From the timings given in [16], we can conclude that one authentication completes within 60 ms or 90 ms, depending on the baud rate used. This translates into a required time of 65 or 98 minutes to gather the 2^{16} plaintext/ciphertext pairs. As these numbers are based on the maximum delay allowed by the specification [16], a real chip may respond faster, as our experiments confirm. No data is given with respect to the operational range in [16], because this depends on the circuit built around the HCS410 chip. However, one can expect the range to be short.

5.2 Key Derivation

The impact of the attack becomes even larger when considering the method used to establish the secret keys, as was previously noted by Bogdanov [5]. To simplify key management, the shared secret keys are derived from a 64-bit master secret (the manufacturer's code), a serial number and optionally a seed value [5, 14, 15]. The manufacturer's code is supposed to be constant for a large number of products (e.g., an entire series from a certain manufacturer) and the serial number of a transponder chip is public, i.e., it can easily be read out from the chip. The seed value is only used in the case of so-called "Secure Learning", and can also be obtained from a chip with relative ease [5, 14, 15]. The other option, "Normal Learning", does not use a seed value.

In both types of key derivation mechanisms, a 64-bit identifier is constructed, which contains the serial number, the (optional) seed and some fixed padding.

Then, the secret key is derived from this identifier and the master secret using one of two possible methods. The first method simply uses XOR to combine the identifier and the master key. The consequence of this is that once a single key is known, together with the corresponding serial number and (optional) seed value, the master secret can be found very easily.

The second method is based on decryption with the KeeLoq block cipher. The identifier is split into two 32-bit halves which are decrypted using the KeeLoq block cipher, and concatenated again to form the 64-bit secret key. The master secret is used as the decryption key. Although much stronger than the first method, the master secret can still be found using a brute force search. Evidently, once the master secret is known, all keys that were derived from it are also compromised, and the security of the entire system falls to its knees. Thus, it is a much more interesting target than a single secret key. This may convince an adversary to legitimately obtain a car key, for the sole purpose of recovering the master key from its secret key.

6 Conclusion

In this paper we have presented a slide and meet-in-the middle attack on the KeeLoq block cipher which requires 2^{16} known plaintexts and has a time complexity of $2^{44.5}$ KeeLoq encryptions, and a variant using 2^{16} chosen plaintexts with the same theoretical time complexity.

We have fully implemented and tested both attacks. When given 16 key bits, the known plaintext attack completes successfully in 10.97 minutes. Due to implementation details, the chosen plaintext attack requires only 4.79 minutes when given 16 key bits. To the best of our knowledge, this is the fastest known attack on the KeeLoq block cipher.

Finally, we have shown that our attack can be used to attack real systems using KeeLoq due to the way it is intended to be used in practice. Moreover, one of the two suggested ways to derive individual Keeloq keys from a master secret is extremely weak, with potentially serious consequences for the overall security of systems built using the Keeloq algorithm.

Acknowledgements

We would like to thank Wim Aerts and Elke De Mulder for their help with the experiments. Also, we would like to thank the reviewers of EUROCRYPT 2008 for their helpful comments.

This work was supported in part by the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

[1] E. Biham. New types of cryptanalytic attacks using related keys. *Journal of Cryptology*, 7(4):229–246, 1994.

[2] A. Biryukov, S. Mukhopadhyay, and P. Sarkar. Improved time-memory trade-offs with multiple data. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography — SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2006.

[3] A. Biryukov and D. Wagner. Slide attacks. In L. R. Knudsen, editor, *Fast Software Encryption, 6th International Workshop — FSE '99*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999.

[4] A. Biryukov and D. Wagner. Advanced slide attacks. In B. Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 589–606. Springer, 2000.

[5] A. Bogdanov. Attacks on the KeeLoq block cipher and authentication systems. 3rd Conference on RFID Security 2007, 2007.

[6] A. Bogdanov. Cryptanalysis of the KeeLoq block cipher. Cryptology ePrint Archive, Report 2007/055, 2007. `http://eprint.iacr.org/`.

[7] N. Courtois, G. V. Bard, and D. Wagner. Algebraic and slide attacks on KeeLoq. In K. Nyberg, editor, *Fast Software Encryption, 15th International Workshop — FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[8] N. T. Courtois. personal communication, May 2007.

[9] N. T. Courtois, G. V. Bard, and D. Wagner. Algebraic and slide attacks on KeeLoq. Cryptology ePrint Archive, Report 2007/062, 2007. `http://eprint.iacr.org/`.

[10] S. Furuya. Slide attacks with a known-plaintext cryptanalysis. In K. Kim, editor, *Information Security and Cryptology — ICISC 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2002.

[11] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[12] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with COPACOBANA — a cost-optimized parallel code breaker. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems — CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2006.

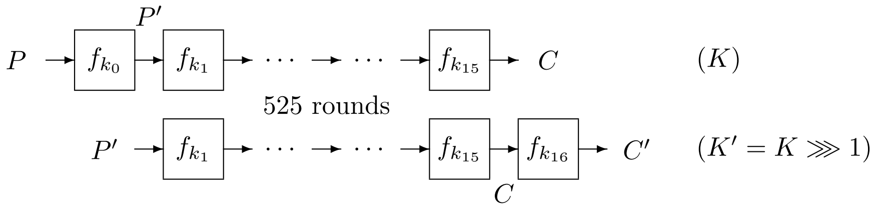


Figure 5 – A related-key attack using keys related by rotation.

[13] Microchip Technology Inc. KeeLoq® authentication products. `http://www.microchip.com/keeloq/`.

[14] Microchip Technology Inc. TB001: Secure learning RKE systems using KeeLoq encoders, 1996. Available online at `http://ww1.microchip.com/downloads/en/AppNotes/91000a.pdf`.

[15] Microchip Technology Inc. AN642: Code hopping decoder using a PIC16C56, 1998. Available online at `http://www.keeloq.boom.ru/decryption.pdf`.

[16] Microchip Technology Inc. HCS410 KeeLoq® code hopping encoder and transponder data sheet, 2001. Available online at `http://ww1.microchip.com/downloads/en/DeviceDoc/40158e.pdf`.

[17] Wikipedia. KeeLoq. `http://en.wikipedia.org/wiki/KeeLoq`, Aug. 2007.

A Related-Key Attacks on KeeLoq

Related-key attacks [1] exploit the relations between the encryption processes under different but related keys.

In this appendix we present two related-key attacks on KeeLoq. The first attack is a very efficient attack using pairs of keys related by rotation. The second attack is an improvement of the attack presented in Sect. 3.3 using pairs of keys related by flipping the least significant bit of the key.

A.1 A Related-Key Attack Using Keys Related by Rotation

The first attack exploits the extremely simple way in which the key is mixed into the state during encryption.

Denote a full encryption of a plaintext P by KeeLoq with the key K by $E_K(P)$, and encryption through a single round with the subkey bit k by $f_k(P)$. Consider a pair (K, K') of related-keys, such that $K' = (K \ggg 1)$. If for a pair (P, P') of plaintexts we have $P' = f_{k_0}(P)$, where k_0 is the LSB of K, then $E_{K'}(P') = f_{k_{16}}(E_K(P))$. Indeed, in this case the encryption of P' under the key K' is equal to the encryption of P under K shifted by one round (see Fig. 5). This property, which is clearly easy to check, can be used to retrieve two bits of the secret key K.

Consider a plaintext P. We note that there are only two possible values of $f_{k_0}(P)$, i.e., $1||(P \gg 1)$ and $0||(P \gg 1)$. Hence, we ask for the encryption of P under the key K and for the encryption of the two plaintexts $P'_0 = 0||(P \gg 1)$ and $P'_1 = 1||(P \gg 1)$ under the related-key K', and check whether the ciphertexts satisfy the relation $E_{K'}(P') = f_{k_{16}}(E_K(P))$. This check is immediate, since $E_K(P)$ and $f_{k_{16}}(E_K(P))$ have 31 bits in common. Exactly one of the candidates (P'_0 or P'_1) is expected to satisfy the relation. This pair satisfies also the relation $P' = f_{k_0}(P)$.

At this stage, since P' and P are known, we can infer the value of k_0 immediately from the update rule of KeeLoq, using the relation $P' = f_{k_0}(P)$. Similarly, we can retrieve the value of k_{16} from the relation $E_{K'}(P') = f_{k_{16}}(E_K(P))$. Hence, using only three chosen plaintexts encrypted under two related-keys, we can retrieve two key bits with a negligible time complexity.

In order to retrieve additional key bits, we repeat the procedure described above with the pair of related-keys $(K', K'' = (K' \ggg 1))$ and one of the plaintexts P'_0 or P'_1 examined in the first stage. As a result, we require the encryption of two additional chosen plaintexts (under the key K''), and get two additional key bits: k'_0 and k'_{16}, which are equal to k_1 and k_{17}.

We can repeat this procedure 16 times to get bits $k_0, \ldots, k_{31}$ of the secret key. Then, the procedure can be repeated with the 16 related keys of the form $(K \ggg 32), (K \ggg 33), \ldots, (K \ggg 47)$ to retrieve the remaining 32 key bits. The attack then requires 66 plaintexts encrypted under 34 related keys (two plaintexts under each of 32 keys, and a single plaintext under the two remaining keys), and a negligible time complexity.

An option to reduce the required amount of plaintexts and related keys in exchange for a higher time complexity, is to switch to an exhaustive key search after a suitable number of key bits has been determined. For example, if 32 key bits remain to be found, a brute force search can be conducted in several hours on a PC, or even much less on FPGAs.

Another variant of the attack, requiring less related-keys, is the following. Denote the encryption of a plaintext P through r rounds of KeeLoq with the key $k = (k_0, \ldots, k_{r-1})$ by $f_k^r(P)$. Consider a pair of related-keys of the form $(K, K' = K \ggg r)$. If a pair of plaintexts (P, P') satisfies $P' = f_k^r(P)$, then the corresponding ciphertexts satisfy $E_{K'}(P') = f_{k'}^r(E_K(P))$, where $k' = (k_{16}, \ldots, k_{16+r-1})$. Since $E_K(P)$ and $f_{k'}^r(E_K(P))$ have $32-r$ bits in common, this property is easy to check.

However, when $r > 1$, the task of detecting P' such that $P' = f_k^r(P)$ is not so easy. Actually, there are 2^r candidates for P', and hence during the attack we have to check 2^r candidate pairs. On the other hand, we can reduce the data complexity of this stage of the attack to $2^{1+r/2}$ by using structures: The first structure S_1 consists of $2^{r/2}$ plaintexts, such that the $32 - r$ least significant bits are equal to some constant C in all the plaintexts of the structure, and the other bits are arbitrary. The second structure S_2 also consists of $2^{r/2}$ plaintexts, such that the $32 - r$ most significant bits are equal to the same constant C in all the plaintexts of the structure, and the other bits are arbitrary. By birthday paradox arguments on the 2^r possible pairs (P, P') such that $P \in S_1$ and $P' \in S_2$ we expect one pair for which $P' = f_k^r(P)$, and this pair can be used for the attack.

In the attack, we go over the 2^r possible pairs and check whether the colliding bits of the relation $E_{K'}(P') = f_{k'}^r(E_K(P))$ are satisfied. If $r \leq 16$, this check discards immediately most of the wrong pairs. After finding the right pair, $2r$ bits of the key can be found using the algorithm presented in Sect. 3.2.

By choosing different values of r, we can get several variants of the attack:

1. Using $r = 16$, we can recover 32 key bits, and then the rest of the key can be recovered using exhaustive key search. The data complexity of the attack is 512 chosen plaintexts encrypted under two related-keys (256 plaintexts under each key), and the time complexity is 2^{32} KeeLoq encryptions.

2. Using $r = 8$ twice (for the pairs $(K, K \ggg 8)$, and $(K \ggg 8, K \ggg 16)$) we retrieve 32 key bits, and exhaustively search the remaining bits. The data complexity of the attack is 64 chosen plaintexts encrypted under three related-keys (16 plaintexts under two keys, and 32 plaintexts under the third key), and the time complexity is 2^{32} KeeLoq encryptions.

3. Using $r = 8$ four times (for the pairs $(K, K \ggg 8)$, $(K \ggg 8, K \ggg 16)$, $(K \ggg 32, K \ggg 40)$, and $(K \ggg 40, K \ggg 48)$) we can retrieve the full key. The data complexity of the attack is 128 chosen plaintexts encrypted under six related-keys (16 plaintexts under four keys, and 32 plaintexts under two keys), and the time complexity is negligible.

Other variants are also possible, and provide a trade-off between the number of chosen plaintexts and the number of related-keys.

A.2 Improved Slide/Meet-in-the-Middle Attack Using Related Keys

Using a related-key approach, we can improve the attack presented in Sect. 3.3. Denote the encryption of a plaintext P through 64 rounds of KeeLoq under the key K by $g_K(P)$. Denote by e_0 the least significant bit of a word. We observe that if two related-keys (K, K') satisfy $K' = K \oplus e_0$, i.e., they differ in the least significant bit, and two plaintexts (P, P') satisfy $P' = P \oplus e_0$, then we have $g_K(P) = g_{K'}(P')$. Indeed, in the first round of encryption the key difference and the data difference cancel each other. As a result, after the first round the intermediate values in both encryptions are equal, and the key difference is not mixed into the data until the 65-th round. Thus, the intermediate values after 64 rounds are equal in both encryptions.

Now, recall that in Sect. 3.1, the pair (P_i, P_j) is called a slid pair if it satisfies $P_j = g_K(P_i)$. The attack searches among 2^{32} candidates for a slid pair, and then the key can be easily retrieved. Note that by the observation above, if (P_i, P_j) is a slid pair with respect to K, then the pair $(P_i \oplus e_0, P_j)$ is a slid pair with respect to $K' = K \oplus e_0$, and thus $E_{K'}(P_j) = g_{(K' \ggg 16)}(E_{K'}(P_i \oplus e_0))$. This additional slid pair can be used to improve the check of candidate slid pairs, and thus to reduce the time complexity of the attack.

More in detail, (10) can be rewritten as

$$2^{16} \left(48 \cdot 2^{16} + 2^{t_o} \left(3t_c \cdot 2^{16+t_c-t_o} + 2^{16+t_p-t_o} \left(3t_p + N_{\text{coll}} \cdot V\right)\right)\right) \ . \qquad (15)$$

The expected number of collisions becomes $N_{\text{coll}} = 2^{16+t_c-t_o}/2^{2t_o}$. Verifying a collision now costs on average V KeeLoq rounds, where

$$V = \sum_{i=0}^{47-t_p-t_c} \left(2 \cdot 2^{-2i} + 2^{-2i-1}\right) + 2^{2t_p+2t_c-96} \cdot \left(528 + 528 \cdot 2^{-32}\right) \ . \qquad (16)$$

Simplification yields:

$$48 \cdot 2^{32} + 3t_c \cdot 2^{32+t_c} + 3t_p \cdot 2^{32+t_p} + 3.33 \cdot 2^{96-2t_p-2t_c} + 528 \cdot 2^{32} \ . \qquad (17)$$

The optimum is situated at $t_p = t_c = 12$ where the time complexity of the attack is $2^{50.9}$ KeeLoq rounds, or $2^{41.9}$ full KeeLoq encryptions.

Summarising the attack, the data complexity is 2^{17} chosen plaintexts encrypted under two related-keys (2^{16} plaintexts under each key), and the time complexity is $2^{41.9}$ KeeLoq encryptions. The memory complexity is about 16 MB.

Publication

Collisions and Other Non-Random Properties for Step-Reduced SHA-256

Publication Data

Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and other non-random properties for step-reduced SHA-256. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography — SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 2009.

Contributions

- Principal author.

Collisions and Other Non-Random Properties for Step-Reduced SHA-256

Sebastiaan Indesteege[1,2,*], Florian Mendel[3], Bart Preneel[1,2], and Christian Rechberger[3]

[1] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
`sebastiaan.indesteege@esat.kuleuven.be`
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.
[3] Institute for Applied Information Processing and Communications Inffeldgasse 16a, A-8010 Graz, Austria.

Abstract. We study the security of step-reduced but otherwise unmodified SHA-256. We show the first collision attacks on SHA-256 reduced to 23 and 24 steps with complexities 2^{18} and $2^{28.5}$, respectively. We give example colliding message pairs for 23-step and 24-step SHA-256. The best previous, recently obtained result was a collision attack for up to 22 steps. We extend our attacks to 23 and 24-step reduced SHA-512 with respective complexities of $2^{43.9}$ and $2^{53.0}$. Additionally, we show non-random behaviour of the SHA-256 compression function in the form of free-start near-collisions for up to 31 steps, which is 6 more steps than the recently obtained non-random behaviour in the form of a semi-free-start near-collision. Even though this represents a step forwards in terms of cryptanalytic techniques, the results do not threaten the security of applications using SHA-256.

Key words: SHA-256, SHA-512, hash functions, collisions, semi-free-start collisions, free-start collisions, free-start near-collisions.

1 Introduction

In the light of previous break-through results on hash functions such as MD5 and SHA-1, the security of their successors, SHA-256 and sisters, against all kinds of cryptanalytic attacks deserves special attention. This is even more important as many products and services that used to rely on SHA-1 are now migrating to SHA-256.

[*]F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

1.1 Previous Work on Members of the SHA-2 Family

Below, we briefly discuss existing work. Results on older variants of the larger MD4 related hash function family, including SHA-1, suggest that the concept of local collisions might also be important for the SHA-2 family. The first published analysis on members of the SHA-2 family, by Gilbert and Handschuh [2], goes in this direction. They show that there exists a 9-step local collision with probability 2^{-66}. Later on, the result was improved by Hawkes et al. [3]. By considering modular differences, they increased the probability to 2^{-39}. Using XOR differences, local collisions with probability as high as 2^{-38} where used by Hölbl et al. [4]. Local collisions with lower probability but with other properties were studied by Sanadhya and Sarkar in [13].

Now we turn our attention to the analysis of simplified variants of SHA-256. In [17], Yoshida and Biryukov replace all modular additions by XOR. For this variant, a search for pseudo-collisions is described, which is faster than brute force search for up to 34 steps. Matusiewicz et al. [8] analysed a variant of SHA-256 where all Σ- and σ-functions are removed. The conclusion is that for this variant, collisions can be found much faster than by brute force search. The work shows that the approach used by Chabaud and Joux [1] in their analysis of SHA-0 is extensible to that particular variant of SHA-256. The message expansion as a building block on its own was studied by Matusiewicz et al. [8] and Pramstaller et al. [12].

Finally, we discuss previous work that focuses on step-reduced but otherwise unmodified SHA-256. The first study was done by Mendel et al. [9]. The results obtained are a practical 18-step collision and a differential characteristic for 19-step SHA-224 collision. Also, an example of a pseudo-near-collision for 22-step SHA-256 is given. Similar techniques have been studied by Matusiewicz et al. [8] and recently also by Sanadhya and Sarkar [15]. Using a different technique, Nikolić and Biryukov [11] obtained collisions for up to 21 steps and non-random behaviour in the form of semi-free-start near-collisions for up to 25 steps. Very recently, Sanadhya and Sarkar [16] extended this, and showed a collision example for 22 steps of SHA-256 in [14].

1.2 Our Contribution

We extend the work of Nikolić and Biryukov [11] to collisions for 23- and 24-step SHA-256 with respective time complexities of 2^{18} and $2^{28.5}$ reduced SHA-256 compression function evaluations. These 23- and 24-step attacks are also applied to SHA-512, with complexities of $2^{43.9}$ and $2^{53.0}$ for 23-step SHA-512 and 24-step SHA-512, respectively. Example collision pairs for 23-step SHA-256 and SHA-512, and for 24-step SHA-256 are given. The collision attacks presented in this work do not extend beyond 24 steps, but we investigate several weaker collision style attacks on a larger number of rounds. Our results are summarised in Table 1.

Table 1 – Comparison of our results with the known results in the literature. Effort is expressed in (equivalent) calls to the respective reduced compression functions.

function	steps	type	effort	source	example
SHA-256	18	collision	2^0	[9]	yes
SHA-256	20	collision	$2^{1.58}$	[11]	no
SHA-256	21	collision	2^{15}	[11]	yes
SHA-256	22	collision	2^9	[14]	yes
SHA-256	**23**	collision	2^{18}	this work	yes
SHA-256	**24**	collision	$2^{28.5}$	this work	yes
SHA-512	**23**	collision	$2^{43.9}$	this work	yes
SHA-512	**24**	collision	$2^{53.0}$	this work	no
SHA-256	23	semi-free-start collision	2^{17}	[11]	yes
SHA-256	**24**	semi-free-start collision	2^{17}	this work	no
SHA-224	**25**	free-start collision	2^{17}	this work	no
SHA-256	22	free-start near-collision	2^0	[9]	yes
SHA-256	25	semi-free-start near-collision	2^{34}	[11]	yes
SHA-256	**31**	free-start near-collision	2^{32}, Table 6	this work	no

We use the terminology introduced by Lai and Massey [5] for different types of attacks on (iterated) hash functions. A collision attack aims to find two distinct messages that hash to the same result. In a semi-free-start collision attack, the attacker is additionally allowed to choose the initial chaining value, but the same value should be used for both messages. In a free-start collision attack, a (small) difference may appear in the initial chaining value. Near-collision attacks relax the requirement that the hash results should be equal and allow for small differences.

The structure of this paper is as follows. We give a short description of SHA-256 in Sect. 2. Section 3 gives an alternative description of the semi-free-start collision attack by Nikolić and Biryukov [11], which will make the subsequent description of the new attacks easier to understand. We then discuss our collision attacks on 23- and 24-step SHA-256 in Sect. 4. In Sect. 5, we apply our results to step-reduced SHA-512. Finally, Sect. 6 concludes.

2 Description of SHA-256

This section gives a short description of the SHA-256 hash function, using the notation from Table 2. For a detailed specification, we refer to [10].

The compression function of SHA-256 consists of a message expansion, which transforms a 512-bit message block into 64 expanded message words W_i of 32 bits each, and a state update transformation. The latter updates eight 32-bit state variables $A, \ldots, H$ in 64 identical steps, each using one expanded message word.

Table 2 – The notation used in this paper.

$X \ggg s$	X rotated over s bits to the right
$X \gg s$	X shifted over s bits to the right
$\overline{X}$	One's complement of X
$X \oplus Y$	Bitwise exclusive OR of X and Y
$X + Y$	Addition of X and Y modulo 2^{32}
$X - Y$	Subtraction of X and Y modulo 2^{32}
$A_i, \cdots, H_i$	State variables at step i, for the first message
$A'_i, \cdots, H'_i$	Idem, for the second message
W_i	i-th expanded message word of the first message
W'_i	Idem, for the second message
δX	Additive difference in X, i.e., $X' - X$
$\delta\sigma_0(X)$	Additive difference in $\sigma_0(X)$, i.e., $\sigma_0(X') - \sigma_0(X)$

The message expansion can be defined recursively as follows.

$$W_i = \begin{cases} M_i & 0 \leq i < 16 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} & 16 \leq i < 64 \end{cases} . \tag{1}$$

The functions $\sigma_0(X)$ and $\sigma_1(X)$ are given by

$$\begin{aligned} \sigma_0(X) &= (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3) \ , \\ \sigma_1(X) &= (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10) \ . \end{aligned} \tag{2}$$

The state update transformation updates two of the state variables in every step. It uses the bitwise Boolean functions f_{ch} and f_{maj} as well as the GF(2)-linear functions Σ_0 and Σ_1.

$$\begin{aligned} f_{\text{ch}}(X,Y,Z) &= XY \oplus \overline{X}Z \ , \\ f_{\text{maj}}(X,Y,Z) &= XY \oplus YZ \oplus XZ \ , \\ \Sigma_0(X) &= (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22) \ , \\ \Sigma_1(X) &= (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25) \ . \end{aligned} \tag{3}$$

Figure 1 describes the state update transformation, where K_i is a step constant. Equivalently, it is described by the following equations.

$$\begin{aligned} T_1 &= H_i + \Sigma_1(E_i) + f_{\text{ch}}(E_i, F_i, G_i) + K_i + W_i \ , \\ T_2 &= \Sigma_0(A_i) + f_{\text{maj}}(A_i, B_i, C_i) \ , \\ A_{i+1} &= T_1 + T_2 \ , \ B_{i+1} = A_i \ , \ C_{i+1} = B_i \ , \ D_{i+1} = C_i \ , \\ E_{i+1} &= D_i + T_1 \ , \ F_{i+1} = E_i \ , \ G_{i+1} = F_i \ , \ H_{i+1} = G_i \ . \end{aligned} \tag{4}$$

After 64 steps, the initial state variables are fed forward using word-wise addition modulo 2^{32}.

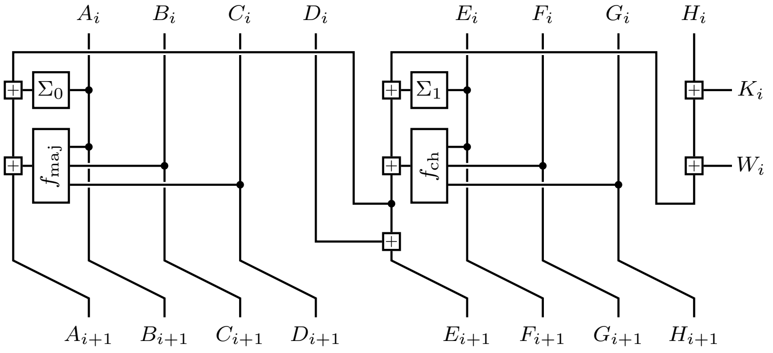


Figure 1 – The state update transformation of SHA-256.

3 Review of the Nikolić-Biryukov Semi-Free-Start Collision Attack

In this section, we review the 23-step semi-free-start collision attack by Nikolić and Biryukov [11]. The new results presented in this paper are extensions of this attack. The notations we use are given in Table 2.

The attack uses a nine step differential, which is presented in Table 3. All additive differences are fixed, as well as the actual values of some of the internal state variables. Fixing these values ensures that the differential is followed, as will be explained later. The constants α, β, γ and ϵ are determined by the attack. The first difference is inserted via the message word W_9. There are no differences in expanded message words other than those indicated in Table 3, i.e., only W_9, W_{10}, W_{11}, W_{12}, W_{16} and W_{17} can have a difference.

The attack algorithm consists of two phases. The first phase finds suitable values for the constants α, β, γ and ϵ as well as two expanded message words, W_{16} and W_{17}. A detailed description of this phase of the attack will be given in Sect. 3.2, as it is more instructive to describe the second phase first.

3.1 The Second Phase of the Attack

The second phase of the attack finds, when given suitable values for α, β, γ, ϵ, W_{16} and W_{17}, a pair of messages and a set of initial values that lead to a semi-free-start collision for 23 steps of SHA-256. It works by carefully fixing the internal state at step 11 as indicated in Table 3, and then computing forward and backward. At each step, the expanded message word W_i is computed such that the differential from Table 3 is followed. During this, four extra conditions appear, involving only

Table 3 – A 9-step differential, using additive differences (left) and conditions on the value (right). Blanks denote zero differences resp. unconstrained values.

step	δA	δB	δC	δD	δE	δF	δG	δH	δW	A	B	C	D	E	F	G	H
8										α				γ			
9									1	α	α			$\gamma+1$	γ		
10	1				1				-1	-1	α	α		-1	$\gamma+1$	γ	
11		1			-1	1			δ_1	α	-1	α	α	ϵ	-1	$\gamma+1$	γ
12			1			-1	1		δ_2	α	α	-1	α	β	ϵ	-1	$\gamma+1$
13				1			-1	1		α	α	α	-1	β	β	ϵ	-1
14					1			-1			α	α	α	-1	β	β	ϵ
15						1						α	α	0	-1	β	β
16							1		1				α	-2	0	-1	β
17								1	-1						-2	0	-1
18																-2	0

the constants determined by the first phase of the attack.

$$\sigma_1(W_{16}+1) - \sigma_1(W_{16}) - \Sigma_1(\epsilon-1) + \Sigma_1(\epsilon) - f_{\text{ch}}(\epsilon-1, 0, \gamma+1) + f_{\text{ch}}(\epsilon, -1, \gamma+1) = 0 \ . \tag{5}$$

$$\sigma_1(W_{17}-1) - \sigma_1(W_{17}) - f_{\text{ch}}(\beta, \epsilon-1, 0) + f_{\text{ch}}(\beta, \epsilon, -1) = 0 \ . \tag{6}$$

$$\beta = \alpha - \Sigma_0(\alpha) \ . \tag{7}$$

$$f_{\text{ch}}(\beta, \beta, \epsilon-1) - f_{\text{ch}}(\beta, \beta, \epsilon) = -1 \ . \tag{8}$$

The first phase guarantees that the constants are such that these conditions are satisfied. The second phase of the attack has a negligible complexity and is guaranteed to succeed. Since there is still a lot of freedom left, many 23-step semi-free-start collisions can be found, with only a negligible additional effort, by repeating this second phase several times. A detailed description of this phase, including the origins of (5)–(8), is given in Appendix A.

3.2 The First Phase of the Attack

The goal of the first phase of the attack is to determine suitable values for the constants α, β, γ and ϵ, as well as two expanded message words, W_{16} and W_{17}. Suitable values imply that the four conditions (5)–(8) are satisfied. Nikolić and Biryukov [11] do not give much detail on this procedure, hence we clarify it below.

1. Make a random choice for γ and ϵ and search for a value of W_{16} such that condition (5) is satisfied. This condition is of the form $\sigma_1(x+1) - \sigma_1(x) = \delta$. There exists a simple, generic method to solve equations of this form, which is described in Appendix B. We note however that for this particular case, a faster method exists. An exhaustive search over every possible value of x resulted in the observation that only 6 181 additive differences δ can ever be

achieved. These can be stored in a lookup table, together with one or more solutions for each difference. Hence, solving an equation of this form can be done with a simple table lookup.

If no solution exists, simply retry with different choices for γ and/or ϵ. If the right hand side difference δ is selected uniformly at random, the probability that the equation has a solution is $2^{-19.5}$, so we expect to have to repeat this step about $2^{19.5}$ times.

2. Make a random choice for α, and compute β using (7). Now check condition (8). As described in [11], this equation is satisfied if the bits of β are zero in the positions where the bits of $\epsilon - 1$ and ϵ differ. This occurs with a probability of approximately 1/3, so this condition is fairly easy to satisfy.
3. The last condition, (6), is of the same form as the first condition, so it can be solved in exactly the same way. The expected probability that a solution exists is again $2^{-19.5}$.

Note that, because not all conditions depend on all of the constants determined in this phase of the attack, the first condition can be treated independently of the last three. Thus, the first and last step of this phase of the attack are executed about $2^{19.5}$ times and the second step about 2^{21} times. One of these steps requires much less work than an evaluation of the compression function of (reduced) SHA-256 — a bit less than one step. Hence, the overall time complexity of the entire attack, when expressed in SHA-256 compression function evaluations, is below 2^{17}.

4 Our Collision Attacks on Step-Reduced SHA-256

In this section we describe a novel, practical collision attack on SHA-256, reduced to 23 steps. It has a time complexity of about 2^{18} evaluations of the reduced SHA-256 compression function. We also extend this to 24 steps of SHA-256, with an expected time complexity of $2^{28.5}$ compression function evaluations.

4.1 23-Step Collision

Our collision attack for SHA-256, reduced to 23 steps, consists of two parts. First, we construct a semi-free-start collision for 23 steps, based on the attack from Sect. 3. Then we transform this semi-free-start collision into a real collision.

Finding "Good" Constants. Finding a 23 step semi-free-start collision is done using the same attack as described in Sect. 3, with a slight change to the first phase. In Sect. 3.2, it was described how to find constants α, β, γ and ϵ such that there exist values for W_{16} and W_{17} ensuring that the conditions (5) and (6) are satisfied. There are still some degrees of freedom left in this process. Indeed, it is possible to determine the constants α, β, γ and ϵ such that there are *many* values for W_{16} and W_{17} satisfying (5) and (6).

We performed an exhaustive search for such good constants. Condition (5) depends only on ϵ and γ. An exhaustive search for this condition can be performed with approximately 2^{37} evaluations of (5), because for each value of ϵ, only some of the bits in γ can have an influence. We found several values for ϵ and γ for which more than 2^{29} choices for W_{16} ensure that (5) is satisfied, for instance

$$\gamma = \texttt{0000017c}_x \ , \quad \epsilon = \texttt{7f5f7200}_x \ . \tag{9}$$

Conditions (6) and (8) depend on ϵ and β, which in turn depends on α through (7). An interesting property is that condition (6) becomes independent of ϵ if we assume that condition (8) is satisfied. Indeed, since this assumption implies that the bits of β are zero where ϵ and $\epsilon - 1$ differ, (6) reduces to

$$\sigma_1\left(W'_{17} + 1\right) - \sigma_1\left(W'_{17}\right) = \overline{\beta} \ . \tag{10}$$

Because of this, an exhaustive search for good values of α and β is feasible. There are many of the optimal values for α and β which are consistent with (several of) the optimal values for ϵ, thus yielding a global optimum. For instance, with γ and ϵ as in (9), the following values for α and β are one of many optimal choices:

$$\alpha = \texttt{00b321e3}_x \ , \quad \beta = \texttt{fcffe000}_x \ . \tag{11}$$

There are 2^{16} possible choices for W_{17} which satisfy (6) with these constants. Thus, these values for α, β, γ and ϵ give us an additional freedom of 2^{45} in the choice of W_{16} and W_{17}. This phase can be considered a precomputation, or alternatively, one can reduce the effort spent in this phase by only searching a smaller part of the available search space, which likely leads to less optimal results. It may however be a worthwhile trade-off in practice.

Transforming into a Collision. Note that only 7 expanded message words, W_{11} until W_{17}, are actually fixed to a certain value when constructing a semi-free-start collision, ignoring the freedom left in W_{16} and W_{17} for now. The others are chosen arbitrarily or computed from the message expansion when necessary. Using this freedom, it is possible to construct many semi-free-start collisions with only a negligible additional effort. But it is also possible to use this freedom in a controlled manner to transform the semi-free-start collision into a real collision.

To this end, we first introduce an alternative description of SHA-256. In older variants of the same design strategy, like MD5 or SHA-1, only a single state variable is updated in every step. This naturally leads to a description where only the first state variable is considered. Something similar can be done with the SHA-2 hash functions, even though in the standard description, two state variables are updated in every step.

From the state update equations (4), we derive a series of equations expressing the inputs of the i-th state update transformation, $A_i, \dots, H_i$, as a function of only A_i through A_{i-7}.

$$\begin{array}{lcl} A_i = A_i \ , \ B_i = A_{i-1} \ , \ C_i = A_{i-2} \ , \ D_i = A_{i-3} \ , & & \\ E_i & = & A_{i-4} + A_i \ - \Sigma_0(A_{i-1}) - f_{\text{maj}}(A_{i-1}, A_{i-2}, A_{i-3}) \ , \\ F_i & = & A_{i-5} + A_{i-1} - \Sigma_0(A_{i-2}) - f_{\text{maj}}(A_{i-2}, A_{i-3}, A_{i-4}) \ , \\ G_i & = & A_{i-6} + A_{i-2} - \Sigma_0(A_{i-3}) - f_{\text{maj}}(A_{i-3}, A_{i-4}, A_{i-5}) \ , \\ H_i & = & A_{i-7} + A_{i-3} - \Sigma_0(A_{i-4}) - f_{\text{maj}}(A_{i-4}, A_{i-5}, A_{i-6}) \ . \end{array} \tag{12}$$

Substituting these into (4) yields an alternative description requiring only a single state variable. This description can be written concisely as

$$A_{i+1} = F\left(A_i, A_{i-1}, A_{i-2}, A_{i-3}, A_{i-4}, A_{i-5}, A_{i-6}\right) + A_{i-7} + W_i \ . \tag{13}$$

The function $F(\cdot)$ encapsulates (4) and (12), except for the addition of the expanded message word W_i and the state variable A_{i-7}. From (12), it is clear that one can easily transform an internal state in the standard description, $\langle A_i, \cdots, H_i \rangle$, to the corresponding internal state in the alternative description, $\langle A_i, \cdots, A_{i-7} \rangle$, and vice versa. Analogous to what is done for MD5 and SHA-1, the initial values can be redefined as $A_{-7}, \cdots, A_0$.

This alternative description of SHA-256 can be used to transform a 23 step semi-free-start collision for SHA-256 into a real collision. Since control over one expanded message word W_i gives full control over one state variable A_{i+1}, control over eight consecutive expanded message words gives full control over the entire internal state.

1. Start from a 23-step semi-free-start collision pair. Set $\langle A_0, \cdots, A_{-7} \rangle$ to the SHA-256 initial values, in the alternative description. Make arbitrary choices for W_0, W_1 and W_2, and recompute the first three steps.

2. The eight message words W_3 until W_{10} are now modified such that A_4 until A_{11} remain unchanged. This implies that the internal state at step 11, $\langle A_{11}, \cdots, H_{11} \rangle$ does not change, and thus we connect to the rest of the semi-free-start collision. More specifically, for every step i, $3 \le i \le 10$, the new value of the i-th message word is computed as

$$W_i = A_{i+1} - F\left(A_i, A_{i-1}, A_{i-2}, A_{i-3}, A_{i-4}, A_{i-5}, A_{i-6}\right) - A_{i-7} \ . \tag{14}$$

   In the message words W_9 and W_{10} there is an additive difference of 1 and -1, respectively. This does not pose a problem since the construction of the semi-free-start condition guarantees that these will have the intended effect, regardless of the values of W_9 and W_{10}, see Appendix A.

3. Now we need to verify again if conditions (5) and (6) are still satisfied, since they depend on W_{16} and W_{17}, which may have changed. If the conditions are

Table 4 – Example colliding message pair for 23-step reduced SHA-256.

M	29f1ebfb	4468041a	1e6565b6	4cc17e75	4ea4f993	33a77104	864a828d	1dcec3d2
	d33d7b02	bcd4a2d7	3b10201d	39953548	8e127f2b	0304fc01	e7118577	43b12ca7
M'	29f1ebfb	4468041a	1e6565b6	4cc17e75	4ea4f993	33a77104	864a828d	1dcec3d2
	d33d7b02	bcd4a2d8	3b10201c	3995d548	91129f2a	0304fc01	e7118577	43b12ca7
H	c77405ea	8bfe2016	ff0531b6	a89b81f6	e98cf052	491a6c62	fd009a40	3969dc83

not satisfied, simply restart and make different choices for W_0, W_1 and/or W_2.

Recall however that we have spent extra effort in the first phase of the attack to choose the constants α, β, γ and ϵ such that there are *many* values for W_{16} and W_{17} that satisfy the conditions. For the constants given in (9) and (11), there are 2^{45} allowed values for these two expanded message words. This translates into a probability of 2^{-19} that the conditions (5) and (6) are indeed still satisfied. We hence expect to have to repeat this procedure about 2^{19} times. Every trial requires an effort equivalent to about 10 steps of SHA-256.

4. After a successful modification of the first message words, the expanded message words W_{18} until W_{22} need to be recomputed, and also the corresponding steps need to be redone. The construction of the semi-free-start collision still guarantees that no differences will be introduced.

If we consider the first phase to be a precomputation, the overall attack complexity is about 2^{18} evaluations of the compression function of SHA-256 reduced to 23 steps. An example collision pair for 23-step reduced SHA-256 is given in Table 4.

4.2 24-Step Collision

The same approach can be extended to 24 steps of SHA-256, using the 24-step semi-free-start collision attack given in detail in Sect. 4.3. Simply put, the 23-step attack is simply shifted down by a single step, and no difference is introduced into W_0 by the message expansion in the backward direction.

When turning the semi-free-start collision into a collision, however, the value of the expanded message word W_{16} (which was the non-expanded message word W_{15} in the 23-step attack) should not change. In a straightforward extension of the 23-step collision attack to 24 steps, this extra condition would only be satisfied with a probability of 2^{-32}. Using the available freedom in a better way, this can be improved substantially.

1. Start from a 24-step semi-free-start collision pair. Set $\langle A_0, \cdots, A_{-7}\rangle$ to the SHA-256 initial values. Make an arbitrary choice for W_0 and recompute the

Table 5 – Example colliding message pair for 24-step reduced SHA-256.

M	`0187e08e`	`865cedaf`	`5b69e21a`	`e0f7485e`	`50b98993`	`217e4650`	`51e3cf65`	`c2997c68`
	`2c267e16`	`82ffa4e9`	`37b5af09`	`5b28721d`	`1be35597`	`7ff22aa1`	`e807a758`	`c1519aaa`
M'	`0187e08e`	`865cedaf`	`5b69e21a`	`e0f7485e`	`50b98993`	`217e4650`	`51e3cf65`	`c2997c68`
	`2c267e16`	`82ffa4e9`	`37b5af0a`	`5b28721c`	`1be3f597`	`82f24aa0`	`e807a758`	`c1519aaa`
H	`1584074c`	`8b810a94`	`01ea31b1`	`81bffd02`	`d29c817d`	`e4e04b51`	`b9f5ac4f`	`6b34d1f8`

first step. Now, it follows from (4) that $(A_2 - W_1)$ is a constant:

$$c_1 = A_2 - W_1 \ . \tag{15}$$

2. The new value of W_9 is determined from (14), i.e., it depends on A_2 through A_{10}. The state variables A_5 through A_{10} have already been fixed in the semi-free-start collision. If we additionally fix A_4 and A_3 to arbitrary values, it is possible to compute the sum of W_9 and A_2,

$$c_2 = W_9 + A_2 = A_{10} - F\left(A_9, \cdots, A_3\right) \ . \tag{16}$$

3. Combining (1) and (15)–(16), results in

$$W_{16} - \sigma_1(W_{14}) - c_2 + c_1 - W_0 = \sigma_0(W_1) - W_1 \ . \tag{17}$$

It is easy to find a suitable value for W_1 that ensures that W_{16} has the proper value, if it exists. It suffices to guess the 15 least significant bits of W_1 to compute all 32 bits of W_1, satisfying the above condition with probability 2^{-14}. A conservative estimate is that each trial requires an effort equivalent to one step update of SHA-256.

4. Now all the internal state variables have been fixed. The corresponding message words can be found from (14) and the message expansion. Just as in the 23-step collision attack, however, there are still some conditions left. As explained in Sect. 4.1, these are satisfied with a probability of 2^{-19}.

Hence, the overall expected time complexity is equivalent to about $2^{19} \cdot (2^{14} + 10)$ SHA-256 step computations, or about $2^{28.5}$ evaluations of the SHA-256 compression function reduced to 24 steps. An example collision pair for 24-step reduced SHA-256 is given in Table 5. An extension of this attack method beyond 24 steps fails, because then a difference in the first or in the last message word becomes unavoidable. In [14], another differential than the one shown in Table 3 is used to find 22-step collisions for SHA-256. We tried to use this differential in our extended attacks, but even for 23 steps, using this differential fails.

4.3 Further Extensions

This section discusses further extensions using weaker attack models. The starting point is the 23-step semi-free-start collision attack of Nikolić and Biryukov [11], which was described in Sect. 3.

Semi-Free-Start Collisions for 24 Steps of SHA-256. We keep the entire attack algorithm from Sect. 3 unchanged, but shift everything down by a single step. Because of this, one more message word, W_0, needs to be computed from the message expansion in the reverse direction. From (1), it follows that the additive difference in this word is

$$\delta W_0 = \delta W_{16} - \delta\sigma_1(W_{14}) - \delta W_9 - \delta\sigma_0(W_1) \ . \tag{18}$$

None of these expanded message words has a difference, so also $\delta W_0 = 0$. This yields 24-step semi-free-start collisions of SHA-256 with the same complexity of 2^{17} compression function evaluations.

Free-Start Collisions for 25 Steps of SHA-224. SHA-224 differs from SHA-256 in two ways. First, it has different initial values, and second, the output is truncated to the leftmost 224 bits. We can thus extend the 24-step semi-free-start collision of SHA-256 to a 25-step free-start collision of SHA-224 by simply shifting the same attack down one more step. Now a difference will inevitably appear in W_0, which propagates to the initial value H_0. The other initial values, A_0 through G_0 still have a zero difference. Because the word H is truncated away in SHA-224, this results in free-start collisions for 25 steps of SHA-224, with the same complexity. Note that this attack would not apply if a different method of truncation would have been chosen in the design of SHA-224.

Free-Start Near-Collisions of SHA-256. Extending the attack to more steps is possible, provided that some differences are allowed both in the initial value and in the hash result, i.e., when considering free-start near-collisions. The starting point is again the 23-step semi-free-start collision attack from Sect. 3. It is extended by adding a number of extra forward and backward steps.

As explained above, no difference is introduced in the first backward step. Note that, in general, the diffusion of differences is slower in the backward direction than in the forward direction. A difference introduced in an expanded message word W_i affects both A_{i+1} and E_{i+1} in the forward direction, as opposed to only H_i in the backward direction. Thus, in the forward direction, all state words can be affected by a single difference in an expanded message word after only four rounds. In the backward direction, this takes eight rounds.

We have done several experiments, each equivalent to an effort of 2^{32} reduced SHA-256 compression function evaluations. The results of our experiments are summarised in Table 6. The first three columns give the total number of steps,

Table 6 – Experimental results of the free-start near-collision attack on SHA-256. For each number of steps, only the combination of forward/backward steps that gave the best results is shown. For comparison, the expected numbers of solutions for a generic birthday attack with an equal effort are also given.

steps	fwd.	bwd.	$k_{\min}$	2-logarithm of the number of solutions with k							
				≤ 8	≤ 16	≤ 24	≤ 32	≤ 40	≤ 48	≤ 56	≤ 64
25	1	1	2	31.95	32.00	32.00	32.00	32.00	32.00	32.00	32.00
26	1	2	8	24.17	31.55	31.99	32.00	32.00	32.00	32.00	32.00
27	1	3	11	$-\infty$	15.41	26.20	30.65	31.89	32.00	32.00	32.00
28	1	4	18	$-\infty$	$-\infty$	8.77	20.41	27.24	30.63	31.80	31.99
29	1	5	32	$-\infty$	$-\infty$	$-\infty$	1.58	14.31	22.86	28.19	30.93
30	1	6	43	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	10.73	19.58	25.68
31	2	6	53	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	6.34	15.50
Birthday Attack			57	-143.41	-108.84	-80.49	-56.36	-35.51	-17.37	-1.57	12.14

the number of extra forward and extra backward steps, respectively. The fourth column gives $k_{\min}$, the smallest Hamming distance found. The last eight columns contain the 2-logarithm of the number of solutions with a Hamming distance k of at most 8, 16, ..., 64 bits.

For comparison, also the expected values for a generic birthday attack with an equal effort of 2^{32} is given. For a generic (free-start) near-collision attack on an ideal n-bit hash function, using the birthday paradox with an effort of 2^w compression function evaluations, the lowest expected Hamming distance is the lowest k for which

$$2^{2w} \cdot \sum_{i=0}^{k} 2^{-n} \binom{n}{i} \geq 1 \ . \tag{19}$$

For instance, with $w = 32$ and for SHA-256 (i.e., $n = 256$), this gives $k = 57$ bits. Our attack performs significantly better for up to 30 steps of SHA-256. For 31 steps, we still found 208 free-start near-collisions with a Hamming distance of at most 57 bits, whereas a birthday attack is only expected to find one with the same effort.

5 Collision Attacks on Step-Reduced SHA-512

SHA-512 is a 512-bit hash function from the SHA-2 family. Its structure is very similar to SHA-256. The sizes of all words are increased to 64 bits and the number of rounds is increased to 80. It uses a different initial chaining value, and different step constants. Finally, the GF(2)-linear functions are redefined. Refer to [10] for details on SHA-512. In this section, we extend the collision attacks on SHA-256 that were described in Sect. 4.1 and 4.2 to SHA-512. The first phase of the attacks needs to be adapted, since an exhaustive search as in Sect 3.2 is no longer feasible.

Finding "Good" Constants for SHA-512. Recall from Sect. 3.2 that the goal of the first phase of the attack is to find values for the constants α, β, γ, ϵ such that the conditions (5)–(8) are satisfied for many values of the expanded message words W_{16}, W_{17}. Since an exhaustive search for good constants is infeasible, we suggest the following approach.

1. First, make a list L of additive differences δ for which the equation

$$\sigma_1(x+1) - \sigma_1(x) = \delta \tag{20}$$

   has many solutions x. This can be accomplished by picking several values for x at random and computing the corresponding δ's. This procedure is likely to quickly find the "good" values for δ, since the more x's correspond to a δ, the more likely we are to find it. Using Appendix B, the number of solutions x for a given δ can be counted efficiently.

2. Since all conditions (5)–(8) will need to be satisfied, we can use (10) instead of (6). Hence, $\overline{\beta}$ should preferably be one of the "good" δ's from the list L. Knowing the value of β, we need to invert (7) to find α. This can, for instance, be done by guessing the 36 most significant bits of α and determining the other bits using (7). A guess succeeds with a probability of about 2^{-36}. Note that (7) cannot necessarily be inverted for all β's.

3. Now we make an arbitrary choice for ϵ which satisfies (8). Denote by l_β the length of the run of least significant "0"-bits in β. Then, (8) is satisfied if and only if the least significant "1"-bit of ϵ lies within the l_β least significant bits. Unfortunately, for SHA-512, this condition eliminates the best values for β.

4. If we choose a "good" value for $\sigma_1(W_{16}+1) - \sigma_1(W_{16})$ from the list L, and since ϵ has already been chosen, (5) can be rewritten as

$$C - f_{\mathrm{ch}}(\epsilon - 1, 0, \gamma + 1) + f_{\mathrm{ch}}(\epsilon, -1, \gamma + 1) = 0 \ , \tag{21}$$

   where C is a known constant. The bits in which ϵ and $(\epsilon - 1)$ differ can be corrected by a proper choice of γ. Hence it is advantageous to choose ϵ with a long run of least significant "0"-bits. This again constrains β, as explained above. If no choice for γ can satisfy (21), retry with a different choice for ϵ and/or β.

Unlike the exhaustive search in Sect. 3.2, this procedure does not guarantee finding the optimal solution. However, experiments show that we can quickly find many good solutions. We found many values for the constants α, β, γ and ϵ for which the conditions (5) and (8) are satisfied for $2^{49.1}$ and 2^{34} values for W_{16} and W_{17}, respectively. Example values are

$$\begin{array}{rclcrcl} \alpha & = & \texttt{3891fd20b54a8eb9}_x \ , & & \beta & = & \texttt{0001200000000000}_x \ , \\ \gamma & = & \texttt{00000fff7f7fff46}_x \ , & & \epsilon & = & \texttt{0000100000000000}_x \ . \end{array} \tag{22}$$

Table 7 – Example colliding message pair for 23-step reduced SHA-512.

M	0000000017daf2ec	000000004b7adc8e	000000000d01f49d	54cce0ac731eb4c9
	5caf52c6f3e941cd	0224e6b804216305	95bbdc5df5b491c8	9f7f1453e39ee6c0
	3e345efecc818058	93dfcee7a268ce69	90561054da994c54	7262751c31b5bdd0
	54b1d56610b9e802	7f201dfcfce968c0	2b90cc3824ee5f13	05cfd16a7b4c4ab1
M'	0000000017daf2ec	000000004b7adc8e	000000000d01f49d	54cce0ac731eb4c9
	5caf52c6f3e941cd	0224e6b804216305	95bbdc5df5b491c8	9f7f1453e39ee6c0
	3e345efecc818058	93dfcee7a268ce6a	90561054da994c53	7266551c31b5bd18
	54b0b56610b9e801	7f201dfcfce968c0	2b90cc3824ee5f13	05cfd16a7b4c4ab1
H	dd44d89f178803f5	136802b223c880ba	bbb80917dda6a3e7	be1f118889bd5415
	98adc37a0f32d151	83d35099922ee2c6	670ac37463f224da	e0835506fb66503d

23-step Collision. The second phase of the 23-step attack from Sect. 4.1 can directly be applied to SHA-512. With the constants from (22), a single attempt to turn a 23-step semi-free-start collision into a 23-step collision will succeed with an expected probability of $2^{-44.9}$ and costs about half of a reduced SHA-512 compression function evaluation. Hence, this results in a collision attack on 23-step SHA-512 with an expected time complexity of $2^{43.9}$ reduced compression function evaluations. An example collision pair for 23-step reduced SHA-512 is given in Table 7.

24-Step Collision. Also the second phase of the 24-step attack from Sect. 4.2 can be applied to SHA-512. One slight modification is required when determining a suitable value for W_0, due to the redefinition of the σ_0-function in SHA-512. Guessing the 8 least significant bits of W_0 allows to compute all of W_0, satisfying (17) with probability 2^{-8}. This results in a collision attack on 24-step SHA-512 with an expected time complexity of $2^{53.0}$ reduced compression function evaluations.

Further Extensions. The attacks on SHA-512 can also be extended, much like the extensions described for the SHA-256 attacks in Sect. 4.3. Adding more rounds trivially leads to several (semi-) free-start (near-) collision attacks. One noteworthy case is a free-start collision attack on 26 steps of SHA-384. It is analogous to the 25-step free-start collision attack on SHA-224 from Sect. 4.3, but as two words are truncated away in the case of SHA-384, the attack extends to 26 steps.

6 Conclusion

Our results push the limit for cryptanalysis of step reduced but otherwise unmodified SHA-256; we found practical collisions for up to 24 steps. For almost half of the steps (31 out of 64) non-random properties of the compression function are detectable in practice. The results also apply to SHA-512, albeit with higher time complexities.

Acknowledgements

The authors would like to thank Vincent Rijmen for his advice. We acknowledge the use of the VIC computer cluster of K.U.Leuven, which was used to obtain most of the experimental results presented in this paper. Finally, thanks to Ralph Wernsdorf for bringing [14] to our attention.

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT, and in part by the Austrian Science Fund (FWF), project P19863. This work was done during a visit of the first author to the Graz University of Technology.

References

[1] F. Chabaud and A. Joux. Differential collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[2] H. Gilbert and H. Handschuh. Security analysis of SHA-256 and sisters. In M. Matsui and R. J. Zuccherato, editors, *Selected Areas in Cryptography — SAC 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 175–193. Springer, 2004.

[3] P. Hawkes, M. Paddon, and G. G. Rose. On corrective patterns for the SHA-2 family. Cryptology ePrint Archive, Report 2004/207, 2004. `http://eprint.iacr.org/`.

[4] M. Hölbl, C. Rechberger, and T. Welzer. Searching for messages conforming to arbitrary sets of conditions in SHA-256. In S. Lucks, A.-R. Sadeghi, and C. Wolf, editors, *Research in Cryptology, Second Western European Workshop — WEWoRC 2007*, volume 4945 of *Lecture Notes in Computer Science*, pages 28–38. Springer, 2008.

[5] X. Lai and J. L. Massey. Hash function based on block ciphers. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 1993.

[6] H. Lipmaa and S. Moriai. Efficient algorithms for computing differential properties of addition. In M. Matsui, editor, *Fast Software Encryption, 8th International Workshop — FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2001.

[7] H. Lipmaa, J. Wallén, and P. Dumas. On the additive differential probability of exclusive-or. In B. K. Roy and W. Meier, editors, *Fast Software Encryption, 11th International Workshop — FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2004.

[8] K. Matusiewicz, J. Pieprzyk, N. Pramstaller, C. Rechberger, and V. Rijmen. Analysis of simplified variants of SHA-256. In C. Wolf, S. Lucks, and P.-W. Yau, editors, *Research in Cryptology, Western European Workshop — WEWoRC 2005*, volume 74 of *Lecture Notes in Informatics*, pages 123–134. GI-Edition, 2005.

[9] F. Mendel, N. Pramstaller, C. Rechberger, and V. Rijmen. Analysis of step-reduced SHA-256. In M. J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop — FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2006.

[10] National Institute of Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-3, Oct. 2008.

[11] I. Nikolić and A. Biryukov. Collisions for step-reduced SHA-256. In K. Nyberg, editor, *Fast Software Encryption, 15th International Workshop — FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[12] N. Pramstaller, C. Rechberger, and V. Rijmen. Preliminary analysis of the SHA-256 message expansion. First NIST Hash Workshop, Oct. 2005.

[13] S. K. Sanadhya and P. Sarkar. New local collisions for the SHA-2 hash family. In K.-H. Nam and G. Rhee, editors, *Information Security and Cryptology — ICISC 2007*, volume 4817 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2007.

[14] S. K. Sanadhya and P. Sarkar. 22-step collisions for SHA-2. arXiv e-print archive, arXiv:0803.1220v1, Mar. 2008. `http://www.arxiv.org/`.

[15] S. K. Sanadhya and P. Sarkar. Attacking reduced round SHA-256. In S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security — ACNS 2008*, volume 5037 of *Lecture Notes in Computer Science*, pages 130–143, 2008.

[16] S. K. Sanadhya and P. Sarkar. Non-linear reduced round attacks against SHA-2 hash family. In Y. Mu, W. Susilo, and J. Seberry, editors, *Information Security and Privacy — ACISP 2008*, volume 5107 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 2008.

[17] H. Yoshida and A. Biryukov. Analysis of a SHA-256 variant. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography — SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2006.

A Detailed Description of the Second Phase of the Nikolić-Biryukov Attack

This appendix gives a detailed description of the second phase of the Nikolić-Biryukov attack [11]. When given suitable values for α, β, γ, ϵ, W_{16} and W_{17} by the first phase, as described in Sect. 3.2, it constructs a pair of messages and a set of initial values that lead to a semi-free-start collision for 23 steps of SHA-256.

1. Start at step 11 by fixing the state variables in this step, $A_{11}, \cdots, H_{11}$ as indicated in Table 3. The constants α, β, γ and ϵ are given by the first phase of the attack.

2. Calculate W_{11} such that $A_{12} = \alpha$ and W'_{11} such that $A'_{12} = \alpha$. Now $E_{12} = \beta$ only depends on α, and we find condition (7) from Sect. 3.1.

$$E_{12} = \alpha - \Sigma_0(\alpha) = \beta \ . \tag{23}$$

3. In a similar way, calculate W_{12} such that $E_{13} = \beta$ and W'_{12} such that $E'_{13} = \beta$. This also guarantees that $A_{13} = A'_{13}$ because the majority function absorbs the difference in C_{12}.

4. Calculate W_{13} such that $E_{14} = -1$ and set $W'_{13} = W_{13}$. Now, see Table 3, δE_{14} should be equal to 1. This yields the condition

$$\delta E_{14} = f_{\mathrm{ch}}(\beta, \beta, \epsilon - 1) - f_{\mathrm{ch}}(\beta, \beta, \epsilon) + 2 = 1 \ . \tag{24}$$

   It was given before as (8), and is satisfied by the first phase of the attack. Note that this also ensures that $\delta A_{14} = 0$.

5. Calculate W_{14} such that $E_{15} = 0$ and set $W'_{14} = W_{14}$. Since the values of E_{14} and E'_{14} were chosen in the previous step to be fixed points of the function Σ_1, $\delta\Sigma_1(E_{14}) = \delta E_{14} = 1$ cancels with $\delta H_{14} = -1$. Also, f_{ch} absorbs the difference in E_{14}, so no new differences are introduced.

6. Calculate W_{15} such that $E_{16} = -2$ and set $W'_{15} = W_{15}$. The difference in F_{15} is absorbed by f_{ch}.

7. The value for W_{16} is computed in phase one of the attack. The difference $\delta W_{16} = 1$ is cancelled by the output of f_{ch}. Indeed, since the binary representation of $E_{16} = -2$ is $111 \cdots 10_b$, the f_{ch} function passes only the difference in the least significant bit.

8. Also the value for W_{17} is computed in phase one of the attack. The difference $\delta W_{17} = -1$ cancels with $\delta H_{17} = 1$, thereby eliminating the final difference in the state variables. Thus, a collision is reached.

9. Now, go back to step 11 and proceed in the backward direction. Make an arbitrary choice for W_{10}. The differential from Table 3 is followed because of the careful choice of the state variables in step 11.

10. Make an arbitrary choice for W_9, and proceed one step backward. The difference $\delta W_9 = 1$ cancels with δA_{10} and with δE_{10} such that there is a zero difference in the state variables A_9 through H_9. Now randomly choose W_8 down to W_2 and calculate backward. Because no new differences appear in these expanded message words, there is also a zero difference in the state variables A_2 through H_2.

11. It is not possible to freely choose W_0 or W_1 as 16 expanded message words have already been chosen, i.e., W_2 until W_{17}. Hence, these are computed using the message expansion in the backward direction. Although some of the message words used to compute W_0 and W_1 have differences, these differences always cancel out.

12. Continuing forward from step 18 again, note that the collision is preserved as long as no new differences are introduced via the expanded message words. From the message expansion, it follows that

$$\begin{aligned}\delta W_{18} = \sigma_1(W_{16}+1) - \sigma_1(W_{16}) - \Sigma_1(\epsilon-1) + \Sigma_1(\epsilon) \\ - f_{\mathrm{ch}}(\epsilon-1,0,\gamma+1) + f_{\mathrm{ch}}(\epsilon,-1,\gamma+1) = 0 \ . \end{aligned} \tag{25}$$

This is condition (5), which is satisfied by the first phase of the attack.

13. Similarly, in step 19, we require that $\delta W_{19} = 0$, which results in

$$\sigma_1(W_{17}-1) - \sigma_1(W_{17}) - f_{\mathrm{ch}}(\beta,\epsilon-1,0) + f_{\mathrm{ch}}(\beta,\epsilon,-1) = 0 \ . \tag{26}$$

This condition was given in (6), and is also satisfied by the first phase of the attack.

14. In steps 20–22, the message expansion guarantees that no new differences are introduced. In step 23, however, a difference of 1 is impossible to avoid, hence the attack stops after 23 steps.

Every step in this procedure is guaranteed to succeed, provided that the first phase of the attack supplied suitable constants. Thus, the complexity of the second phase of the attack is negligible. Since there is still a lot of freedom left, many 23-step semi-free-start collisions can be found, with only a negligible additional effort, by repeating this second phase several times.

B Solving $\mathcal{L}(x+\delta)=\mathcal{L}(x)+\delta'$

This appendix describes a generic method to solve equations of the form $\mathcal{L}(x+\delta) = \mathcal{L}(x) + \delta'$ where δ and δ' are given n-bit additive differences, and $\mathcal{L}$ is an n-bit to n-bit GF(2)-linear transformation. This is similar to the problems studied by Lipmaa and Moriai [6] and Lipmaa et al. [7].

Consider the modular addition $x + \delta$ and let $\Delta = (x + \delta) \oplus x$. This addition is described by the following equations, where x_i is the i-th bit of x and the c_i's are the carry bits:

$$\begin{array}{rcl} (x+\delta)_i & = & x_i \oplus \delta_i \oplus c_i \\ c_{i+1} & = & f_{\mathrm{maj}}(x_i, \delta_i, c_i) \\ c_0 & = & 0 \end{array} \quad \Leftrightarrow \quad \begin{array}{rcl} c_i & = & \delta_i \oplus \Delta_i \\ c_{i+1} & = & f_{\mathrm{maj}}(x_i, \delta_i, \delta_i \oplus \Delta_i) \\ c_0 & = & 0 \end{array} \quad . \tag{27}$$

Hence, once we fix both the additive difference δ and the XOR difference Δ, all the carries c_i are fixed. Some of the x_i's are also fixed: when $\Delta_i = 1$ and $i < n-1$, it must hold that $x_i = c_{i+1} = \delta_{i+1} \oplus \Delta_{i+1}$. The other x_i's can be chosen arbitrarily. Thus, the allowed values for x lie in an affine space. Note that not all additive differences are consistent with all XOR differences, i.e., the following conditions must be satisfied

$$\left\{ \begin{array}{rcl} c_0 & = & \delta_0 \oplus \Delta_0 = 0 \\ \delta_i & = & \delta_{i+1} \oplus \Delta_{i+1} \quad \text{when } \Delta_i = 0 \text{ and } i < n-1 \end{array} \right. . \tag{28}$$

Solving an equation of the form $\mathcal{L}(x + \delta) = \mathcal{L}(x) + \delta'$ can be done as follows. Let $\Delta' = (\mathcal{L}(x) + \delta') \oplus \mathcal{L}(x)$, i.e., the XOR-difference associated with the modular addition $\mathcal{L}(x) + \delta'$. Since $\mathcal{L}(x+\delta) = \mathcal{L}(x) + \delta'$ and $\mathcal{L}$ is GF(2)-linear, it follows that $\Delta' = \mathcal{L}(\Delta)$. We can thus simply enumerate all the XOR-differences Δ consistent with the given additive difference δ, compute $\Delta' = \mathcal{L}(\Delta)$ and check if this is consistent with the other additive difference δ'. If it is, both additions restrict x to a (different) affine space. The intersection of these spaces, which can be computed by solving a system of linear equations over GF(2), gives the solutions x for the chosen XOR-difference Δ. Note that this intersection may be empty. If no solutions are found for any value of the XOR-difference Δ, the equation $\mathcal{L}(x + \delta) = \mathcal{L}(x) + \delta'$ has no solutions. Note that the number of solutions of the equation can be counted efficiently using this method, as the number of solutions of a linear system over GF(2) is straightforward to compute.

The time complexity of this method is proportional to the minimum of the number of XOR differences consistent with the given additive differences δ or δ'. This follows from the fact that one can easily modify the method to choose Δ' instead of Δ.

Publication

Collisions for RC4-Hash

Publication Data

Sebastiaan Indesteege and Bart Preneel. Collisions for RC4-Hash. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security 11th International Conference — ISC 2008*, volume 5222 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2008. Best Student Paper Award.

Contributions

- Principal author.

Collisions for RC4-Hash

Sebastiaan Indesteege[1,2,*] and Bart Preneel[1,2]

[1] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
`{sebastiaan.indesteege,bart.preneel}@esat.kuleuven.be`
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.

Abstract. RC4-Hash is a variable digest length cryptographic hash function based on the design of the RC4 stream cipher. In this paper, we show that RC4-Hash is not collision resistant. Collisions for any digest length can be found with an expected effort of less than 2^9 compression function evaluations. This is extended to multicollisions for RC4-Hash. Finding a set of 2^k colliding messages has an expected cost of $2^7 + k \cdot 2^8$ compression function evaluations.

Key words: RC4-Hash, hash functions, collisions, multicollisions.

1 Introduction

Cryptographic hash functions have been receiving much attention from the cryptologic community recently, as several of the widely used hash functions like MD5, SHA-0 and SHA-1, have been broken, or at least shown to be weaker than expected [3, 9–11]. This is a motivation for the design of new hash functions, based on different design principles. One such proposal is RC4-Hash, which was introduced by Chang, Gupta and Nandi [1] in 2006. The design is inspired by the RC4 stream cipher. The latter was designed by Ron Rivest in 1987, but remained a trade secret until it leaked out in 1994 [8]. The motivation for basing a hash function design on RC4, which is well studied, is to be able to use existing results on RC4 in the security analysis of RC4-Hash [1]. Concerning the performance of RC4-Hash, the designers claim that SHA-1 is roughly 1, 5 times faster than RC4-Hash [1].

We focus on the collision resistance of RC4-Hash. Informally, collision resistance means that it should be hard to find two distinct messages $m \neq m'$ that hash to the same value, i.e., $h(m) = h(m')$. We show that RC4-Hash is not collision resistant, and give a method to find colliding message pairs with an expected time complexity of less than 2^9 compression function evaluations. We also extend this to multicollisions.

This paper is organised as follows. In Sect. 2, a short description of the RC4-Hash family of cryptographic hash functions is given. Section 3 introduces

[*] F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

two distinct methods to construct fixed points of the internal state of RC4-Hash. This is then used in Sect. 4 to construct colliding message pairs for RC4-Hash. In Sect. 5, extensions of the attack, as well as ways to mitigate it, are discussed. Section 6 concludes.

2 Description of RC4-Hash

RC4-Hash follows the "wide pipe" hash function design principle proposed by Lucks [7], which implies that the intermediate state size is (much) larger than the digest size. More specifically, RC4-Hash consists of a compression function $\mathcal{C} : \{0,1\}^w \times \{0,1\}^m \mapsto \{0,1\}^w$, and an output transformation $g_n : \{0,1\}^w \mapsto \{0,1\}^n$. The intermediate state size w is (much) larger than the digest length n. The compression function $\mathcal{C}$ is applied iteratively for every (padded) message block of length m, starting from an initial value. Then, the output transformation g compresses the large internal state down to the required digest length n.

In RC4-Hash, the intermediate state consists of an array S of 256 bytes and a pointer into this array, denoted by j. The array S always represents a permutation of the numbers 0 to 255. The size of the internal state is thus $\log_2(2^8!) + 8 \approx$ 1692 bits. The digest length is variable from 16 bytes to 64 bytes, which is much shorter than the internal state size. The length of the message blocks is fixed to 64 bytes.

Padding Rule. A message M is padded in the following way. The 8-bit binary representation of the digest length n (in bytes), $\text{bin}_8(n)$, is prepended to the message. A single "1" bit, v "0" bits and the 64-bit binary representation of the original message length (in bits), $\text{bin}_{64}(|M|)$, are appended to the message. The number v is the least non-negative integer such that $|M| + 73 + v \equiv 0 \bmod 512$. This ensures that the padded message length is an integer multiple of 512 bits, the message block length. Hence, the padded message can be split into t blocks of 512 bits each, denoted by M_1 through M_t.

$$\text{pad}(M) = \text{bin}_8(n) \,||\, M \,||\, 1 \,||\, 0^v \,||\, \text{bin}_{64}(|M|) = M_1||M_2||\cdots||M_t \ . \qquad (1)$$

Compression Function. The compression function of RC4-Hash, which is denoted by $\mathcal{C}\big(\langle S, j\rangle, X\big)$, is described in Fig. 1. It updates the internal state $\langle S, j\rangle$ in 256 steps. In every step, the pointer j is updated using one byte of the message block X. Then, two elements of the array S are swapped. Each of the 64 bytes of the message block is used in four steps. The order in which they are used is given by the message reordering $r(\cdot)$, see Table 5. This compression function is applied iteratively for every message block M_1 through M_t, starting from the initial state $\big\langle S^{\text{IV}}, 0\big\rangle$. The initial value permutation S^{IV} is given in Table 6.

```
Input: Internal state $\langle S, j \rangle$, 64-byte message block $X$.
Output: The updated internal state $\langle S, j \rangle$.
1: for $i = 0$ to 255 do
2:     $j \leftarrow j + S[i] + X[r(i)]$
3:     swap$(S[i], S[j])$
4: end for
5: return $\langle S, j \rangle$
```

Figure 1 – The compression function of RC4-Hash, $\mathcal{C}\big(\langle S, j \rangle, X\big)$. All arithmetic is done modulo 256.

Output Transformation. After every block of the padded message has been processed, an output transformation $g_n\big(\langle S, j \rangle\big)$ is applied. This transformation generates the message digest of the required length n from the internal state. First, the permutation S is composed with the initial value permutation S^{IV}. The resulting permutation is saved as T_1. Then, two blank iterations of the compression function $\mathcal{C}$, i.e., using a zero message block, are applied, resulting in T_2. Finally, S is replaced by a composition of the two saved permutations, $T_1 \circ T_2 \circ T_1$, and the message digest is generated using an algorithm similar to RC4's pseudo-random byte generation.

Figure 2 shows the definition of the entire output transformation. In the original description of RC4-Hash [1], the output transformation was further partitioned into the algorithms OWT ("one way transformation") and HBG ("hash byte generation"). These correspond to lines 2–9 and 10–15 of the algorithm in Fig. 2, respectively.

3 Fixed Points of the Compression Function $\mathcal{C}$

In this section, we describe how to construct two distinct types of fixed points for a certain number of iterations of the RC4-Hash compression function $\mathcal{C}$. Each of these constructions is based on one of two types of "partial state rotations", which are introduced in two lemmata, Lemma 1 and Lemma 3.

3.1 Fixed Points of Type I

Lemma 1 (Partial state rotations of type I)**.** Consider an internal state $\langle S, 0 \rangle$ of RC4-Hash with $S = \{s_0, s_1, \ldots, s_{255}\}$. Denote by $\langle S', j' \rangle$ the internal state reached after applying the compression function $\mathcal{C}$ using the message block $X = \{x, x, \ldots, x\}$ with $x = 1 - s_0 \bmod 256$:

$$\langle S', j' \rangle = \mathcal{C}\big(\langle S, j \rangle, X\big) \ . \tag{2}$$

```
**Input:** Internal state $\langle S, j\rangle$ after processing the entire padded
    message.
**Output:** The message digest $H$.
 1: $S \leftarrow S^{\mathrm{IV}} \circ S$
 2: // OWT (one way transformation)
 3: $T_1 \leftarrow S$
 4: for $i = 0$ to $511$ do
 5:     $j \leftarrow j + S[i]$
 6:     swap$(S[i], S[j])$
 7: end for
 8: $T_2 \leftarrow S$
 9: $S \leftarrow T_1 \circ T_2 \circ T_1$
10: // HBG (hash byte generation)
11: for $i = 0$ to $n$ do
12:     $j \leftarrow j + S[i]$
13:     swap$(S[i], S[j])$
14:     $H[i] \leftarrow S[S[i] + S[j]]$
15: end for
16: return $H$
```

Figure 2 – The output transformation of RC4-Hash, $g_n\big(\langle S, j\rangle\big)$. All arithmetic is done modulo 256.

Now, it holds that

$$j' = 0 \qquad \text{and} \qquad S'[i] = \begin{cases} s_0 & i = 0 \\ s_{i+1} & 1 \le i < 255 \\ s_1 & i = 255 \end{cases} . \tag{3}$$

Proof. Denote by $\langle S^{(i)}, j^{(i)} \rangle$ the internal state of RC4-Hash after the i-th step of the compression function $\mathcal{C}$. First, we prove by induction that for every $i < 256$ it holds that

$$\begin{cases} j^{(i)} = i + 1 \bmod 256 \ , \quad \text{and} \\ S^{(i)}[i+1 \bmod 256] = s_0 \ . \end{cases} \tag{4}$$

It is clear that this holds before the first step, i.e., for $i = -1$, since $j^{(-1)} = 0$ and $S^{(-1)}[0] = S[0] = s_0$. Assume that the condition holds after step i ($i < 255$). Then, the update of the pointer j in the $(i+1)$-th step is

$$\begin{aligned} j^{(i+1)} &= j^{(i)} + S^{(i)}[i+1] + X[r(i+1)] \bmod 256 \\ &= (i+1) + s_0 + (1 - s_0) \bmod 256 \\ &= i + 2 \bmod 256 \ . \end{aligned} \tag{5}$$

Thus, $S^{(i+1)}$ is found by swapping the $(i+1)$-th and $(i+2)$-th element of $S^{(i)}$. Hence, $S^{(i+1)}[i+2 \bmod 256] = S^{(i)}[i+1 \bmod 256] = s_0$, i.e., the condition also holds after step $i+1$.

After 255 steps, all the elements of S have been circularly shifted over one position, i.e., $S^{(254)} = \{s_1, s_2, \ldots, s_{255}, s_0\}$. In the final step, the first and the last element of $S^{(254)}$ are swapped since $j^{(255)} = 0$, resulting in

$$S^{(255)} = S' = \{s_0, s_2, s_3, \ldots, s_{254}, s_{255}, s_1\} \ . \tag{6}$$

From this, the lemma follows. □

Table 1 gives a detailed illustration of Lemma 1. The first column of this table gives the step number i, the second column gives the new value of the pointer j, computed in this step. The last column contains the array S after the step, where the elements that were just swapped are encircled.

Based on this first type of partial state rotations, it is straightforward to construct fixed points for 255 iterations of the compression function $\mathcal{C}$ as is shown in the next theorem.

Theorem 1 (Fixed points of type I)**.** Consider an internal state $\langle S, 0 \rangle$ of RC4-Hash with $S = \{s_0, s_1, \ldots, s_{255}\}$. After 255 iterations of the compression function $\mathcal{C}$, each using the same message block $X = \{x, x, \ldots, x\}$ with $x = 1 - s_0 \bmod 256$, the same state is reached:

$$\langle S, 0 \rangle = \mathcal{C}^{255}\big(\langle S, 0 \rangle , X \big) \ . \tag{7}$$

Table 1 – Partial state rotations of type I.

step i	$j^{(i)}$	$S^{(i)}$								
	0	s_0	s_1	s_2	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
0	1	$\boxed{s_1}$	$\boxed{s_0}$	s_2	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
1	2	s_1	$\boxed{s_2}$	$\boxed{s_0}$	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
2	3	s_1	s_2	$\boxed{s_3}$	$\boxed{s_0}$	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
3	4	s_1	s_2	s_3	$\boxed{s_4}$	$\boxed{s_0}$	$\cdots$	s_{253}	s_{254}	s_{255}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
254	255	s_1	s_2	s_3	s_4	s_5	$\cdots$	s_{254}	$\boxed{s_{255}}$	$\boxed{s_0}$
255	0	$\boxed{s_0}$	s_2	s_3	s_4	s_5	$\cdots$	s_{254}	s_{255}	$\boxed{s_1}$

Proof. The repeated application of Lemma 1 proves the theorem. □

Note that the only requirement for the construction of a fixed point of type I is that the pointer j has to be zero in the starting state. There are no conditions on the contents of the array S. Also, when given a suitable starting state, constructing a fixed point requires only a negligible amount of work, i.e., one subtraction modulo 256 to compute the message byte $x = 1 - s_0 \bmod 256$.

3.2 Fixed Points of Type II

The message reordering $r(\cdot)$ has an interesting property which allows for another type of partial state rotations.

Lemma 2. The message reordering $r(\cdot)$ does not reorder message bytes with an even index to odd-numbered positions, or vice versa. In other words,

$$\forall i, 0 \leq i < 256 : r(i) \equiv i \pmod 2 \ . \tag{8}$$

Proof. The lemma follows in a straightforward way from the definition of $r(\cdot)$ in Table 5. □

Lemma 3 (Partial state rotations of type II)**.** Consider an internal state $\langle S, 1\rangle$ of RC4-Hash with $S = \{s_0, s_1, \ldots, s_{255}\}$. Denote by $\langle S', j'\rangle$ the internal state reached after applying the compression function $\mathcal{C}$ using the message block $X = \{x_0, x_1, x_0, x_1, \ldots, x_0, x_1\}$ with $x_0 = 1 - s_0 \bmod 256$ and $x_1 = 1 - s_1 \bmod 256$:

$$\langle S', j'\rangle = \mathcal{C}\big(\langle S, j\rangle, X\big) \ . \tag{9}$$

Now, it holds that

$$j' = 1 \qquad \text{and} \qquad S'[i] = \begin{cases} s_i & 0 \leq i < 2 \\ s_{i+2} & 2 \leq i < 254 \\ s_{i-252} & 254 \leq i < 256 \end{cases} \ . \tag{10}$$

Proof. Denote by $\langle S^{(i)}, j^{(i)} \rangle$ the internal state of RC4-Hash after the i-th step of the compression function $\mathcal{C}$. Note that, because of Lemma 2 and the definition of X, $X[r(i)] = x_{i \bmod 2} = 1 - s_{i \bmod 2}$. First, we prove by induction that for every $i < 256$ it holds that

$$\begin{cases} j^{(i)} = i + 2 \bmod 256 \ , \quad \text{and} \\ S^{(i)}[i+1 \bmod 256] = s_{i+1 \bmod 2} \ , \quad \text{and} \\ S^{(i)}[i+2 \bmod 256] = s_{i \bmod 2} \ . \end{cases} \tag{11}$$

It is clear that this holds before the first step, i.e., for $i = -1$, since $j^{(-1)} = 1$, $S^{(-1)}[0] = S[0] = s_0$ and $S^{(-1)}[1] = S[1] = s_1$. Assume that the condition holds after step i ($i < 255$). Then, the update of the pointer j in the $(i+1)$-th step is

$$\begin{aligned} j^{(i+1)} &= j^{(i)} + S^{(i)}[i+1] + X[r(i+1)] \bmod 256 \\ &= (i+2) + s_{i+1 \bmod 2} + (1 - s_{i+1 \bmod 2}) \bmod 256 \\ &= i + 3 \bmod 256 \ . \end{aligned} \tag{12}$$

Thus, $S^{(i+1)}$ is found by swapping the $(i+1)$-th and $(i+3)$-th element of $S^{(i)}$. Hence, $S^{(i+1)}[i+3 \bmod 256] = S^{(i)}[i+1 \bmod 256] = s_{i+1 \bmod 2}$. Of course, $S^{(i+1)}[i+2 \bmod 256] = S^{(i)}[i+2 \bmod 256] = s_{i \bmod 2}$. This implies that the condition also holds for step $i+1$.

After 254 steps, all the elements of S have been circularly shifted over two position, i.e., $S^{(253)} = \{s_2, s_3, s_4, \ldots, s_{255}, s_0, s_1\}$. Since $j^{(254)} = 0$ and $j^{(255)} = 1$, the swaps made in the last two steps result in the following state

$$S^{(255)} = S' = \{s_0, s_1, s_4, \ldots, s_{255}, s_2, s_3\} \ . \tag{13}$$

From this, the lemma follows. □

Table 2 gives a detailed illustration of Lemma 3. The notations are the same as in Table 1. Based on this type of partial state rotations, fixed points for 127 iterations of the compression function $\mathcal{C}$ can be constructed, as is shown in the next theorem.

Theorem 2 (Fixed points of type II)**.** Consider an internal state $\langle S, 1 \rangle$ of RC4-Hash with $S = \{s_0, s_1, \ldots, s_{255}\}$. After 127 iterations of the compression function $\mathcal{C}$, each using the same message block $X = \{x_0, x_1, x_0, x_1, \ldots, x_0, x_1\}$ with $x_0 = 1 - s_0 \bmod 256$ and $x_1 = 1 - s_1 \bmod 256$, the same state is reached:

$$\langle S, 1 \rangle = \mathcal{C}^{127}\big(\langle S, 1 \rangle, X \big) \ . \tag{14}$$

Proof. The repeated application of Lemma 3 proves the theorem. □

Note that, as for fixed points of type I, the only requirement for the construction of a fixed point of type II is that the j pointer has a certain value in the starting state. There are no conditions on the contents of the array S. Constructing a fixed

Table 2 – Partial state rotations of type II.

step i	$j^{(i)}$	$S^{(i)}$								
	1	s_0	s_1	s_2	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
0	2	$\textcircled{s_2}$	s_1	$\textcircled{s_0}$	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
1	3	s_2	$\textcircled{s_3}$	s_0	$\textcircled{s_1}$	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
2	4	s_2	s_3	$\textcircled{s_4}$	s_1	$\textcircled{s_0}$	$\cdots$	s_{253}	s_{254}	s_{255}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
253	255	s_2	s_3	s_4	s_5	s_6	$\cdots$	$\textcircled{s_{255}}$	s_0	$\textcircled{s_1}$
254	0	$\textcircled{s_0}$	s_3	s_4	s_5	s_6	$\cdots$	s_{255}	$\textcircled{s_2}$	s_1
255	1	s_0	$\textcircled{s_1}$	s_4	s_5	s_6	$\cdots$	s_{255}	s_2	$\textcircled{s_3}$

point of type II, when given a suitable starting state, also requires only a negligible amount of work, i.e., two subtractions modulo 256 to compute the message bytes $x_0 = 1 - s_0 \bmod 256$ and $x_1 = 1 - s_1 \bmod 256$.

One could try to further generalise this to longer cyclic patterns. However, the message byte reordering $r(\cdot)$ prevents this as there is no $p > 2$ for which it holds that

$$\forall i, 0 \leq i < 256 : r(i) \equiv i \pmod{p} \ . \tag{15}$$

3.3 Relation to Finney States

A Finney state [4] is an RC4-state where $j = i+1$ and $S[i] = 1$. From the definition of the RC4 stream cipher, see Fig. 5, it follows that if the current state is a Finney state, the next state must also be a Finney state. Similarly, a Finney state can only arise from a Finney state. In a Finney state, the element "1" is simply moved to the next position in the array S and j is incremented. The initialisation of the RC4 pseudo-random byte generator, see Fig. 5, ensures that the initial state is not a Finney state. Hence, Finney states can never occur in RC4.

In RC4-Hash, however, we can achieve a similar pattern. This is exactly what is done in the case of partial state rotations of type I. The extra freedom coming from the message input is exploited to ensure that the element $S[i]$ is always moved to the next position, such that it is again used to update j in the next iteration. Partial state rotations of type II are a generalisation of this, using two elements in an alternating way.

4 Collisions for RC4-Hash

This section describes how to use fixed points for a number of iterations of the compression function $\mathcal{C}$ to construct colliding message pairs for RC4-Hash. In

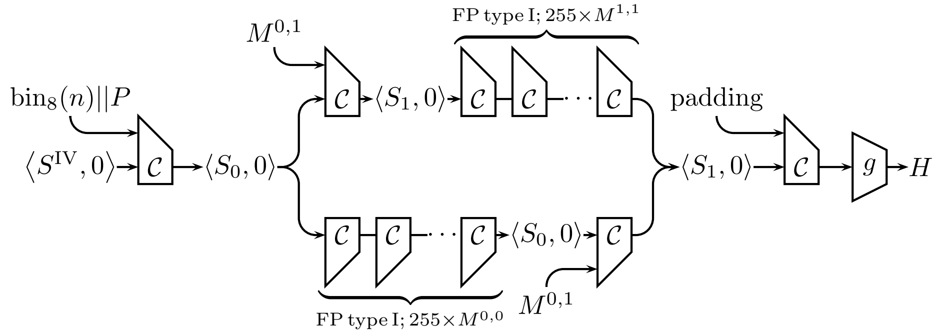


Figure 3 – A collision pair for RC4-Hash using fixed points of type I.

order to be able to construct fixed points, the value of the pointer j in the internal state of RC4-Hash has to be equal to zero (for fixed points of type I) or one (for fixed points of type II), as described in Sect. 3. Although the initial value of j is zero, we cannot make use of the first block because we do not have control over its first byte, which contains the digest length.

Consider fixed points of type I, i.e., we want $j = 0$. Since j can only take 2^8 possible values, we can simply search for a prefix block P which leads to a suitable internal state:

$$\langle S_0, 0\rangle = \mathcal{C}\big(\left\langle S^{\text{IV}}, 0\right\rangle, \text{bin}_8(n)||P\big) \ . \tag{16}$$

We expect to find a suitable prefix block after about 2^8 random trials. At this point, we can easily construct a fixed point for this state $\langle S_0, 0\rangle$ by applying Theorem 1. Denote by $M^{0,0}$ the message block that is used 255 times in this fixed point.

Then, we search for an additional message block $M^{0,1}$ which transforms the state $\langle S_0, 0\rangle$ into $\langle S_1, 0\rangle$:

$$\langle S_1, 0\rangle = \mathcal{C}\big(\langle S_0, 0\rangle, M^{0,1}\big) \ . \tag{17}$$

Again, the only condition on $M^{0,1}$ is that the value of the j pointer is not changed by the compression function $\mathcal{C}$. The expected number of random trials required to find a suitable message block is again about 2^8. For the state $\langle S_1, 0\rangle$, it is also possible to construct a fixed point of type I, using Theorem 1. Denote the message block used in this fixed point by $M^{1,1}$. Now, consider the following two messages:

$$\begin{aligned} M &= P||M^{0,1}||\overbrace{M^{1,1}||\cdots||M^{1,1}}^{255} \ , \\ M^\star &= P||\underbrace{M^{0,0}||\cdots||M^{0,0}}_{255}||M^{0,1} \ . \end{aligned} \tag{18}$$

As shown in Fig. 3, these messages form a collision. Indeed, after processing the 257-th block, the internal state of RC4-Hash is $\langle S_1, 0\rangle$ for both messages, i.e., an

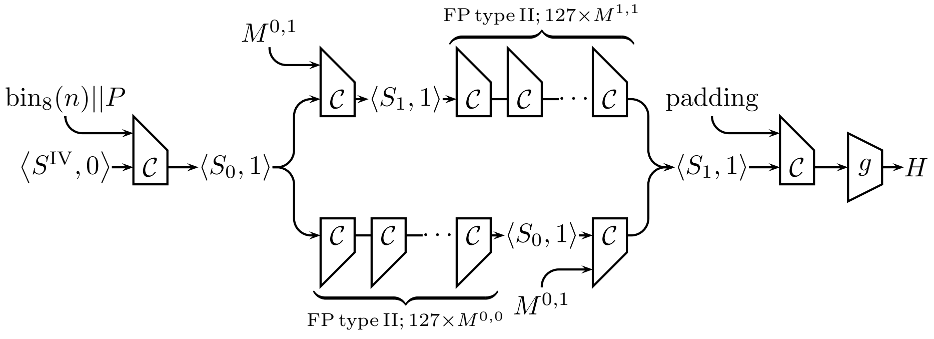


Figure 4 – A collision pair for RC4-Hash using fixed points of type II.

internal state collision is reached. The extra padding block containing the message length and the output transformation maintain the collision. The expected total time complexity is only 2^9 evaluations of the compression function $\mathcal{C}$. Note that verifying the collision requires about the same effort, since hashing M and $M^\star$ requires two times 258 calls to the compression function $\mathcal{C}$.

Using fixed points of type II, collisions can be found in a completely similar way, as Fig. 4 illustrates. The only differences are that we now require $j = 1$, and that the fixed points only contain 127 iterations of the compression function $\mathcal{C}$. The expected time complexity is also 2^9. If we do not fix in advance which type of fixed points to use, but let this depend on which kind of prefix block is found first, the expected time complexity can be lowered slightly to $2^7 + 2^8$ compression function evaluations.

There is no need to restrict the prefix block P or the message block $M^{0,1}$ to be only a single block. Using multiple blocks does not (significantly) increase the expected time complexity for finding a collision pair, if only the last block of P, resp. $M^{0,1}$, is varied in order to obtain the desired value for the pointer j. Of course, a colliding message pair can always be extended with an equal suffix.

Tables 3 and 4 give examples of colliding message pairs for RC4-Hash$_{64}$, constructed using fixed points of type I and type II, respectively. Additional constraints were imposed to arrive at meaningful messages.

5 Discussion

Kelsey-Schneier Second Preimages. Since fixed points of the compression function of RC4-Hash can be constructed very easily, one may consider to use them to mount a Kelsey-Schneier second preimage attack [6]. This involves building expandable messages, i.e., messages of varying length, which all collide on the intermediate hash result immediately after processing the message. The main problem which makes the Kelsey-Schneier second preimage attack fail for

Table 3 – Example collision pair for RC4-Hash$_{64}$, using fixed points of type I.

	M	$M^\star$
block 1 (63 bytes)	`s.␣IndestEEGE␣AnD␣B.␣pReNeEl␣-␣` `cosIc␣-␣cOlLisIoNS␣FoR␣rC4-Hash.`	
block 2 (64 bytes)	`thiS␣MEssAgE␣Is␣pArT␣oF␣a␣colLis` `ion␣EXaMpLe␣for␣RC4-HASH.␣COSIC.`	`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA` `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
blocks 3–256 (254×64 bytes)	`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` ⋮ `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`	`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA` `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA` ⋮ `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
block 257 (64 bytes)	`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`	`thiS␣MEssAgE␣Is␣pArT␣oF␣a␣colLis` `ion␣EXaMpLe␣for␣RC4-HASH.␣COSIC.`
RC4-Hash$_{64}(M)$ = RC4-Hash$_{64}(M^\star)$	`0093b4baefdc64f93d7081978808c49d1286523696e6d4a35ab64f1e42695aff` `79ce81eae91cb47673c4989238fab010f47466906fa65bed88753802c71ae82b`$_x$	

Table 4 – Example collision pair for RC4-Hash$_{64}$, using fixed points of type II.

	M	$M^\star$
block 1 (63 bytes)	`s.␣IndesTeEGE␣ANd␣b.␣pREneEl␣-␣` `cosIc␣-␣colLISioNS␣For␣Rc4-hAsH.`	
block 2 (64 bytes)	`thiS␣MesSagE␣IS␣pArT␣of␣a␣collis` `ioN␣EXAmPle␣FOr␣rc4-HASH.␣COSIC.`	`aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB` `aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB`
blocks 3–128 (126×64 bytes)	`abababababababababababababababab` `abababababababababababababababab` ⋮ `abababababababababababababababab`	`aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB` `aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB` ⋮ `aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB`
block 129 (64 bytes)	`abababababababababababababababab` `abababababababababababababababab`	`thiS␣MesSagE␣IS␣pArT␣of␣a␣collis` `ioN␣EXAmPle␣FOr␣rc4-HASH.␣COSIC.`
RC4-Hash$_{64}(M)$ = RC4-Hash$_{64}(M^\star)$	`0023dd337650ef0d9b5e77be533ea644198ff0d8f1d8190628d95b9dd04dadf5` `d9cd2c1ad8adc8555f03ea3819df4128bc96462a53c7e0cc1afffe78db3bd652`$_x$	

RC4-Hash, is the very large internal state of RC4-Hash. Because of this, the Kelsey-Schneier attack is much slower than exhaustive search in this case.

Multicollisions. A multicollision is a (large) set of messages that all hash to the same value. Multicollisions and their applications were described by Joux [5], although Coppersmith already used them in 1985 [2]. In order to obtain multicollisions for RC4-Hash, we simply concatenate the method from Sect. 4 several times. Concatenating it k times yields 2^k colliding messages. Actually, only part of the method needs to be repeated k times. Indeed, as the value of the pointer j is maintained by the fixed points, only the search for message blocks $M^{0,1}$ has to be repeated. Thus, the expected time for finding 2^k colliding messages for RC4-Hash is $2^7 + k \cdot 2^8$ compression function evaluations. Naturally, also the method of Kelsey and Schneier [6] to construct multicollisions can be applied, and both methods can even be combined.

Mitigating the Attack. The collision attack described in this paper is built on the existence of two types of fixed points of the compression function of RC4-Hash, which were described in Sect 3. These fixed points use patterns where all the (reordered) message bytes are equal (type I) or alternate between two values (type II). Replacing the message reordering $r(\cdot)$ with a message expansion that guarantees that such patterns can never occur foils the attack. Another approach would be to introduce asymmetry, for instance using intermediate rounds.

6 Conclusion

We have shown that RC4-Hash is not collision resistant. There exist two distinct types of fixed points for a number of iterations of the RC4-Hash compression function $\mathcal{C}$. These can be used to construct colliding message pairs with an expected effort of less than 2^9 compression function evaluations. This also leads to multicollisions, yielding 2^k colliding messages with an expected effort of $2^7 + k \cdot 2^8$ compression function evaluations.

Acknowledgements

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

[1] D. Chang, K. C. Gupta, and M. Nandi. RC4-Hash: A new hash function based on RC4. In R. Barua and T. Lange, editors, *Progress in Cryptology*

— *INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2006.

[2] D. Coppersmith. Another birthday attack. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 14–17. Springer, 1985.

[3] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. In X. Lai and K. Chen, editors, *Advances in Cryptology — ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[4] H. Finney. An RC4 cycle that can't happen. Newsgroup post in `sci.crypt`, Sept. 1994.

[5] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In M. K. Franklin, editor, *Advances in Cryptology — CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[6] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[7] S. Lucks. A failure-friendly design principle for hash functions. In B. K. Roy, editor, *Advances in Cryptology — ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[8] B. Schneier. *Applied Cryptography.* John Wiley & Sons, second edition, 1996.

[9] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In V. Shoup, editor, *Advances in Cryptology — CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[10] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[11] X. Wang, H. Yu, and Y. L. Yin. Efficient collision search attacks on SHA-0. In V. Shoup, editor, *Advances in Cryptology — CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

Table 5 – The message reordering $r(\cdot)$.

0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,
16,	17,	18,	19,	20,	21,	22,	23,	24,	25,	26,	27,	28,	29,	30,	31,
32,	33,	34,	35,	36,	37,	38,	39,	40,	41,	42,	43,	44,	45,	46,	47,
48,	49,	50,	51,	52,	53,	54,	55,	56,	57,	58,	59,	60,	61,	62,	63,
0,	55,	46,	37,	28,	19,	10,	1,	56,	47,	38,	29,	20,	11,	2,	57,
48,	39,	30,	21,	12,	3,	58,	49,	40,	31,	22,	13,	4,	59,	50,	41,
32,	23,	14,	5,	60,	51,	42,	33,	24,	15,	6,	61,	52,	43,	34,	25,
16,	7,	62,	53,	44,	35,	26,	17,	8,	63,	54,	45,	36,	27,	18,	9,
0,	57,	50,	43,	36,	29,	22,	15,	8,	1,	58,	51,	44,	37,	30,	23,
16,	9,	2,	59,	52,	45,	38,	31,	24,	17,	10,	3,	60,	53,	46,	39,
32,	25,	18,	11,	4,	61,	54,	47,	40,	33,	26,	19,	12,	5,	62,	55,
48,	41,	34,	27,	20,	13,	6,	63,	56,	49,	42,	35,	28,	21,	14,	7,
0,	47,	30,	13,	60,	43,	26,	9,	56,	39,	22,	5,	52,	35,	18,	1,
48,	31,	14,	61,	44,	27,	10,	57,	40,	23,	6,	53,	36,	19,	2,	49,
32,	15,	62,	45,	28,	11,	58,	41,	24,	7,	54,	37,	20,	3,	50,	33,
16,	63,	46,	29,	12,	59,	42,	25,	8,	55,	38,	21,	4,	51,	34,	17.

Table 6 – The initial value permutation S^{IV}.

145,	57,	133,	33,	65,	49,	83,	61,	113,	171,	63,	155,	74,	50,	132,	248,
236,	218,	192,	217,	23,	36,	79,	72,	53,	210,	38,	59,	54,	208,	185,	12,
233,	189,	159,	169,	240,	156,	184,	200,	209,	173,	20,	252,	96,	211,	143,	101,
44,	223,	118,	1,	232,	35,	239,	9,	114,	109,	161,	183,	88,	66,	219,	78,
157,	174,	187,	193,	199,	99,	52,	120,	89,	166,	18,	76,	241,	13,	225,	6,
146,	151,	207,	177,	103,	45,	148,	32,	29,	234,	7,	16,	19,	91,	108,	186,
116,	62,	203,	158,	180,	149,	67,	105,	247,	3,	128,	215,	121,	127,	179,	175,
251,	104,	246,	98,	140,	11,	134,	221,	24,	69,	190,	154,	253,	168,	68,	230,
58,	153,	188,	224,	100,	129,	124,	162,	15,	117,	231,	150,	237,	64,	22,	152,
165,	235,	227,	139,	201,	84,	213,	77,	80,	197,	250,	126,	202,	39,	0,	94,
42,	243,	228,	87,	82,	27,	141,	60,	160,	46,	125,	112,	181,	242,	167,	92,
198,	172,	170,	55,	115,	30,	107,	17,	56,	31,	135,	229,	40,	111,	37,	222,
182,	25,	43,	119,	244,	191,	122,	102,	21,	93,	97,	131,	164,	10,	130,	47,
176,	238,	212,	144,	41,	14,	249,	220,	34,	136,	71,	48,	142,	73,	123,	204,
206,	4,	216,	196,	214,	137,	255,	195,	26,	8,	51,	178,	2,	138,	254,	90,
194,	81,	245,	106,	95,	75,	86,	163,	205,	70,	226,	28,	147,	85,	5,	110,

```
Input: Key $K$ consisting of $\kappa$ bytes.
Output: Initial internal state of RC4, $\langle S, i, j \rangle$.
1: // RC4 Key Scheduling Algorithm (KSA)
2: $S \leftarrow \{0, 1, \cdots, 255\}$
3: $j \leftarrow 0$
4: for $i = 0$ to 255 do
5:     $j \leftarrow j + S[i] + K[i \bmod \kappa]$
6:     swap($S[i], S[j]$)
7: end for
8: return $\langle S, 0, 0 \rangle$
```

```
Input: RC4 internal state $\langle S, i, j \rangle$.
Output: One byte of keystream, updated internal state.
1: // RC4 pseudo-random byte generation (PRBG)
2: $i \leftarrow i + 1$
3: $j \leftarrow j + S[i]$
4: swap($S[i], S[j]$)
5: return $S[S[i] + S[j]]$
```

Figure 5 – The RC4 stream cipher. It consists of a key scheduling algorithm (top) and a pseudo-random byte generator (bottom). All arithmetic is done modulo 256. [8]

Publication

Coding Theory and Hash Function Design

Publication Data

Sebastiaan Indesteege and Bart Preneel. Coding theory and hash function design. In Bart Preneel, Stefan Dodunekov, Vincent Rijmen, and Svetla Nikova, editors, *Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes*, volume 23 of *NATO Science for Peace and Security Series D — Information and Communication Security*, pages 63–68. IOS Press, 2009.

Contributions

- Principal author.

Coding Theory and Hash Function Design
A Case Study: The Lane Hash Function

Sebastiaan Indesteege* and Bart Preneel

[1] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven.
Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
`sebastiaan.indesteege@esat.kuleuven.be`
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.

Abstract. We illustrate how coding theory was applied in the design of the cryptographic hash function LANE [4]. The generic structure of the LANE compression function could potentially be vulnerable to a class of meet-in-the-middle attacks. While difficult to avoid at first sight, restating the problem in the domain of error correcting codes naturally leads to a simple and elegant solution. This ensures that these attacks do not apply to LANE.

Key words: Hash Function Design, Coding Theory, Minimum Distance, LANE.

1 Introduction

Cryptographic hash functions map an input message m of variable length to a fixed length output $h(m)$. They are required to satisfy certain security properties such as collision resistance and preimage resistance. Collision resistance means that it should be difficult to find two distinct messages $m \neq m'$ that hash to the same result, i.e., $h(m) = h(m')$. There exists a generic attack based on the birthday paradox that can find collisions for any hash functions with an attack complexity of $\mathcal{O}(2^{w/2})$. Here, w is the length of the hash output in bits. Preimage resistance implies that, when given a hash result y (for which it holds that $\exists x : h(x) = y$), it is difficult to find a message m which hashes to y, i.e., $h(m) = y$. Here, the best generic attack is an exhaustive search, which has a complexity of $\mathcal{O}(2^w)$. For a secure hash function, there should exist no attacks that are significantly better than generic attacks.

Many hash functions are iterative hash functions. They apply a fixed input length compression function iteratively to process variable length input messages, for instance using the popular Merkle-Damgård construction [2, 6]. For this construction it can be proven that the iterated hash function is collision resistant if the compression function is collision resistant.

*F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

Recent advances in the cryptanalysis of hash functions have shown that widely used hash functions, such as MD5, SHA-0 and SHA-1, are not as secure as they were expected to be [3, 10–12]. In response to this, the National Institute of Standards in Technology (NIST) has started an international competition to design a new hash function standard, the SHA-3 competition [9].

1.1 Our Contribution

In this paper, we outline a design strategy for a compression function based on parallel permutations. This design strategy was used in the LANE hash function [4], which was submitted as a candidate to the NIST SHA-3 competition [9].

We show that, in order to avoid meet-in-the-middle style attacks on the compression function, we need one of its components to satisfy a certain property. Restating this problem in the domain of error correcting codes yields a simple and elegant solution: a simple requirement on the minimum distance of a linear code.

1.2 Related Work

Several parallelisable compression functions exist in the literature, among which the most famous example is the block cipher based construction MDC-2 [1, 7]. While it was originally defined for use with the DES block cipher, MDC-2 can be instantiated with any block cipher. The MDC-2 compression function contains two parallel block cipher encryptions and can be seen as a two-way parallel extension of the Matyas-Meyer-Oseas (MMO) mode [5]. Another such compression function construction, which was proposed by Nandi et al. [8], maps $3n$ bits to $2n$ bits and contains three parallel functions which each map $2n$ bits to n bits.

We consider another type of compression function construction which also consists of several parallel building blocks. In contrast to the designs mentioned above, we use permutations as building blocks rather than block ciphers or compressing functions. For a detailed discussion of the rationale behind this design, we refer to [4], which describes the design of the LANE hash function.

2 A New Parallel Compression Function Design

We propose a new type of compression function, based on parallel permutations. The structure of our compression function design is shown in Figure 1. It consists of three layers: the message expansion layer, the permutation layer and the output layer. The message expansion takes the chaining value and a message block as inputs, which we will treat uniformly as the k input blocks X_i. The outputs of the message expansion are the n expanded message blocks W_i, which form the inputs to the n permutations P_i. For simplicity, we will assume a simple output layer, consisting of exclusive OR operations. Note that a more complex output layer is used in the actual LANE hash function [4].

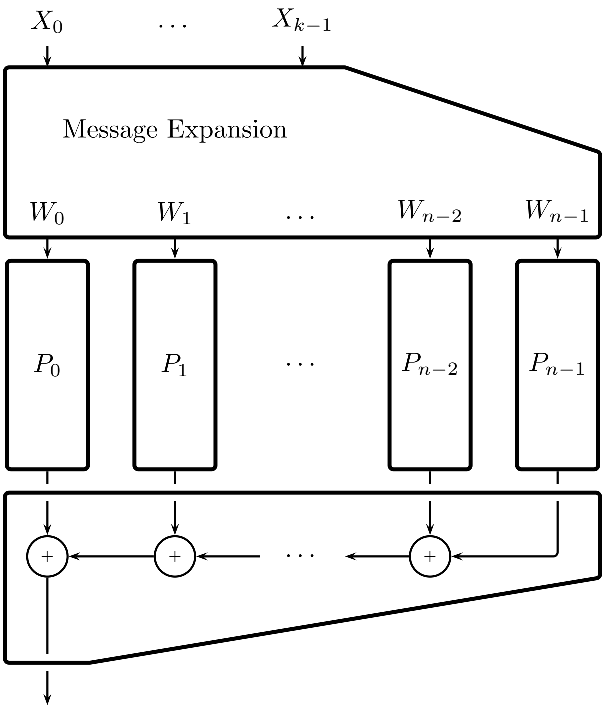


Figure 1 – A compression function design based on parallel permutations.

The ample parallelism offered by this compression function design allows for flexibility in implementation. It can use instruction level parallelism (ILP) and/or vector instructions (SIMD) in software. For hardware implementations, there is a straightforward area–speed tradeoff. At the same time, the memory requirements for a completely serial implementation remain reasonable.

The use of permutations ensures that internal state collisions can only occur in the output layer. Establishing such a collision is equivalent to satisfying a linear condition on the outputs of several permutations. Similarly, the message expansion imposes relations on the inputs of the permutations. The rationale is that, while such conditions are very simple, it is hard to maintain or even track them through the non-linear permutations.

A similar rationale applies to finding preimages for the compression function. Straightforward inversion attempts fail, as one has to ensure that the linear conditions imposed by the message expansion hold. This is again considered to be very difficult. However, as will be discussed in Sect. 3.1, this construction is potentially vulnerable to a meet-in-the-middle preimage attack. A well designed message expansion can prevent this, as will be shown in Sect. 3.2.

3 Designing the Message Expansion

In this section, we analyse the general compression function construction that was presented in Sect. 2. We show that a meet-in-the-middle preimage attack could apply, and present a simple, coding theory based method to design a message expansion which precludes such attacks.

3.1 A Meet-in-the-Middle Preimage Attack

A preimage attack on the compression function is given an output of the compression function (for which at least one input exists), and aims to find a corresponding compression function input. The best generic attack is exhaustive search, which has a complexity of $\mathcal{O}(2^w)$ for a compression function with a w-bit output.

Consider the case where the number of permutations n is two. If the inputs to the two permutations can be chosen independently, the following simple meet-in-the-middle preimage attack would apply:

1. Let H be the desired w-bit output of the compression function.
2. Choose $2^{w/2}$ random values for the first permutation input W_0, compute the permutation output $P_0(W_0)$ and store the tuple $\langle W_0, P_0(W_0)\rangle$ in a list L, which is sorted by the second element of the tuples.
3. Choose $2^{w/2}$ random values for the second permutation input W_1 and compute the permutation output $P_1(W_1)$.

4. For each value of W_1, search in the list L for a tuple containing the value $H \oplus P_1(W_1)$. If such a tuple is found, this gives the desired preimage.

This procedure allows an attacker to check 2^w compression function inputs with an effort of just $\mathcal{O}(2^{w/2})$. As the probability that a single compression function input maps to the desired output H is 2^{-w}, we expect to find one preimage. Hence, this is a preimage attack with a complexity of $\mathcal{O}(2^{w/2})$ compression function evaluations, which is significantly faster than the $\mathcal{O}(2^w)$ offered by a generic attack.

This can be generalised to the case where the number of permutations n is greater than two. If an adversary can pick two non-overlapping, independent groups of permutations, and keep the inputs to the other permutations (if any) constant, the same meet-in-the-middle principle can be applied to construct a preimage attack with a complexity of just $\mathcal{O}(2^{w/2})$. Also, if the number of input values to each group of permutations an attacker can choose is smaller than $2^{w/2}$, the attack still applies, but the attack complexity increases. In the extreme case, where the list L contains only a single value, the attack reduces to a generic, exhaustive search attack with complexity $\mathcal{O}(2^w)$.

3.2 Mitigating the Attack

We now show how the theory of error correcting codes provides us with a simple and elegant way to preclude these attacks. We map every w-bit input and output block of the message expansion to an element of $\mathrm{GF}(2^w)$. Then we use an (n, k, d) linear code over $\mathrm{GF}(2^w)$ for the message expansion. The dimension k of the linear code is equal to the number of w-bit inputs to the message expansion. The code length n is equal to the number of parallel permutations.

In order to be able to apply the meet-in-the-middle preimage attacks from Sect. 3.1, an adversary needs to be able to find two non-overlapping sets of permutations which are independent. As the linear code used for the message expansion has minimum distance d, each such set needs to consist of at least d permutation inputs. Indeed, any two valid expanded messages will differ in at least d blocks.

Now, if it holds that $d > n/2$, i.e., the minimum distance is strictly greater than half the number of permutations, it is impossible to find two non-overlapping, independent sets of permutations. Hence, if we use a linear code with minimum distance $d > n/2$ for the message expansion, these attacks no longer apply.

3.3 Assessing Resistance against Differential Cryptanalysis

This construction also provides a useful way to assess the resistance of the compression function against differential cryptanalysis. In a differential collision attack, a collision is searched among the set of pairs of messages which have a specific difference. This difference propagates through the hash function and, with a certain probability, results in a zero output difference. If this probability is too small, however, differential collision attacks will fail.

If a linear code with minimum distance d is used for the message expansion, we are ensured that in a differential attack at least d permutations will be active, i.e., have an input difference. The resistance of the compression function against differential cryptanalysis can then be assessed by combining this simple bound can with an analysis of the resistance against differential attacks offered by the permutations themselves.

4 Application: the Lane Hash Function

Lane is a cryptographic hash function which was proposed as a candidate for the NIST SHA-3 competition [9] by Indesteege et al. [4]. Lane is an iterated hash function supporting multiple digest sizes, which uses a compression function designed according to the principles outlined in this paper. Note however that the output layer in Lane is more complex that the simple XOR used in this paper. For a complete description of the Lane hash function, we refer to [4]. Here, we focus on the message expansion of Lane.

The message expansion of Lane-256 takes three blocks of 256 bits as inputs and outputs six blocks of 256 bits. Instead of using a linear code over GF(2^{256}), it is based on a (6,3,4) linear code over GF(4) which is known as the *hexacode*. The generator matrix of this code, in a standard polynomial representation of GF(4) using $X^2 + X + 1$ as the primitive polynomial, is given by

$$G = \begin{bmatrix} 1 & X & X & 1 & 0 & 0 \\ X & 1 & X & 0 & 1 & 0 \\ X & X & 1 & 0 & 0 & 1 \end{bmatrix} . \tag{1}$$

The message expansion of Lane-256 simply consists of 128 parallel applications of this code, where each application of the code uses two bits from each 256-bit block.

This offers the same guarantees as the more general approach of using an arbitrary (6,3,4) linear code over GF(2^{256}) but avoids the need for arithmetic in a large finite field, which can be costly in implementation. The Lane-256 message expansion can be computed using only XOR operations on blocks of 128 bits, which can be done very efficiently. Lane-512 is completely analogous to Lane-256, except that all word sizes are doubled.

5 Conclusion

We have demonstrated how the theory of error correcting codes was applied successfully in the design of the cryptographic hash function Lane. A simple requirement on the minimum distance of a linear code ensures that the Lane compression function is resistant to a class of meet-in-the-middle attacks.

Acknowledgements

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

[1] B. Brachtl, D. Coppersmith, M. Hyden, S. Matyas, C. Meyer, J. Oseas, S. Pilpel, and M. Schilling. Data authentication using modification detection codes based on a public one-way encryption function. US Patent #4 908 861, 1990.

[2] I. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[3] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. In X. Lai and K. Chen, editors, *Advances in Cryptology — ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[4] S. Indesteege, E. Andreeva, C. De Cannière, O. Dunkelman, E. Käsper, S. Nikova, B. Preneel, and E. Tischhauser. The LANE hash function. Submission to the NIST SHA-3 competition, Oct. 2008.

[5] S. Matyas, C. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.

[6] R. C. Merkle. One way hash functions and DES. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[7] C. Meyer and M. Schilling. Secure program load with modification detection code. In *Proc. 6th Worldwide Congress on Computer and Communications Security and Protection (SECURICOM '88)*, pages 111–130, 1988.

[8] M. Nandi, W. Lee, K. Sakurai, and S. Lee. Security analysis of a 2/3-rate double length compression function in the black-box model. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption, 12th International Workshop — FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 243–254. Springer, 2005.

[9] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, Nov. 2007.

[10] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In V. Shoup, editor, *Advances in Cryptology — CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[11] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[12] X. Wang, H. Yu, and Y. L. Yin. Efficient collision search attacks on SHA-0. In V. Shoup, editor, *Advances in Cryptology — CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

Publication

The Lane Hash Function

Publication Data

Extended abstract:

> Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser. The LANE hash function — extended abstract. Technical report, COSIC, 2008.

Full version, submitted to the NIST SHA-3 competition (included here):

> Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser. The LANE hash function. Submission to the NIST SHA-3 competition, October 2008.

Contributions

- Designer of LANE.
- Principal author of
  - Sect. 1 (Introduction),
  - Sect. 2 (Specification), and
  - Sect. 3 (Design Rationale).
- Significant contributions to
  - Sect. 4 (Security Analysis), and
  - Sect. 5 (Implementation Aspects).

The Lane Hash Function

Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser

Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven.
Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
{sebastiaan.indesteege,bart.preneel}@esat.kuleuven.be

1 Introduction

In this document, we propose the cryptographic hash function LANE as a candidate for the SHA-3 competition organised by NIST [46]. LANE is an iterated hash function supporting multiple digest sizes. Components of the AES block cipher [19, 45] are reused as building blocks. LANE aims to be secure, easy to understand, elegant and flexible in implementation.

The structure of this document is as follows. In Section 2, we give a full specification of the LANE hash function. The design rationale, including motivations for all important design choices, is discussed in Section 3. Section 4 contains an extensive security analysis, investigating the resistance of LANE against a variety of attacks. Finally, implementation aspects of LANE form the subject of Section 5.

2 Specification

2.1 Introduction

LANE is an iterated cryptographic hash function, supporting digest sizes of 224, 256, 384 and 512 bits. These four variants of LANE are referred to as LANE-224, LANE-256, LANE-384 and LANE-512, respectively. The LANE hash function reuses components from the AES block cipher [19, 45]. After introducing some preliminaries and conventions in Sect. 2.2, the building blocks are described in Sect. 2.3.

Optionally, a salt value S, can be used while computing the digest. When used, the size of this salt is 256 bits for LANE-224 and LANE-256, and 512 bits for LANE-384 and LANE-512. Refer to Table 1 for a comparison of the parameters of the various LANE variants.

Hashing a message is performed in three steps. In the first step, which is described in Sect. 2.4, the message is padded and split into message blocks of equal length. Also, the initial chaining value H_{-1} is set to the initial value $IV_{n,S}$, which depends on the digest size n and the (optional) salt value S.

Table 1 – Parameters of the LANE hash functions.

	LANE-224	LANE-256	LANE-384	LANE-512
Digest length n	224 bits	256 bits	384 bits	512 bits
Blocksize b	512 bits	512 bits	1024 bits	1024 bits
Size of chaining value	256 bits	256 bits	512 bits	512 bits
Salt length $\|S\|$	256 bits	256 bits	512 bits	512 bits

In the second step, a compression function $f(\cdot,\cdot,\cdot)$ is applied iteratively:

$$H_i = f\left(H_{i-1}, M_i, C_i\right) \quad . \tag{1}$$

Each compression function call uses a message block M_i to update the chaining value H_{i-1} to H_i. A counter C_i, which indicates the number of message bits processed so far, including the message bits in the block M_i which is currently being processed, is also input into the compression function. The compression function of LANE is described in Sect. 2.5.

The third and final step is the output transformation, described in Sect. 2.6. In this step, the digest is derived from the final chaining value, using the message length l and the (optional) salt value S as additional inputs. It consists of a single compression function call and, depending on the digest length, a truncation of the result.

Note that LANE supports hashing in 'one-pass' streaming mode. There is no need to buffer the entire message, and one can start hashing as soon as the first complete message block has been received. This property is similar to the hash functions of the SHA-family [48].

2.2 Preliminaries

This section introduces the preliminaries and conventions that will be used in this specification. For reference, the notations used in this section are summarised in Table 2.

2.2.1 Bit Strings, Bytes and States

Definition 1. A *bit string* is an ordered sequence of binary digits of arbitrary length. A bit string is written from left to right, i.e., the leftmost bit is the first bit of the sequence.

Definition 2. A *byte* is a bit string consisting of eight bits. A byte can represent an integer in the range from 0 to $2^8 - 1$. The big-endian convention is used, i.e., the first (leftmost) bit of a byte is the most significant bit.

Table 2 – The notation used in the specification of LANE.

$0^\star$	A number of zero bits, required to pad a bit string to a given length.
$\oplus$	Exclusive or (XOR).
$\|\|$	Concatenation of bit strings.
$x \gg i$	Bitwise right-shift of the word x over i bits.
$\mathrm{bin}_{32}(\cdot)$ or $\mathrm{bin}_{64}(\cdot)$	Big-endian representation of a number in 32 or 64 bits, respectively.
$\cdots_x$	A number in hexadecimal notation.
ϕ	The flag byte used in the output transformation and the derivation of $IV_{n,S}$.
b	The blocksize.
C_i	Counter indicating the number of message bits (excluding padding) in message blocks 0 up to and including i.
$f(H_{i-1}, M_i, C_i)$	The LANE compression function.
H_i	Chaining value after processing message block i.
$IV_{n,S}$ or IV_n	The initial value for digest length n and salt S (if applicable).
l	Message length in bits.
k_i	A 32-bit LANE constant.
M	A message.
M_i	Padded message block i.
n	Digest length, i.e., 224, 256, 384 or 512 bits.
P_i, Q_i	The permutations (lanes) used in LANE in the first and second layer, respectively.
r	The full round number.
S	Salt value.
$W_0,\ldots,W_5$	Expanded message blocks.
x_i	A column of an AES state, consisting of four bytes.

Definition 3. An *AES state* is a 4×4 array of bytes, corresponding to an internal state of the AES block cipher [19,45]. A sequence of 16 bytes can be mapped to an AES state, and vice versa. The sequence of 16 bytes $y_0 \,||\, \cdots \,||\, y_{15}$ is mapped to the AES state

$$\begin{bmatrix} y_0 & y_4 & y_8 & y_{12} \\ y_1 & y_5 & y_9 & y_{13} \\ y_2 & y_6 & y_{10} & y_{14} \\ y_3 & y_7 & y_{11} & y_{15} \end{bmatrix} . \tag{2}$$

Definition 4. A LANE *state* is the state used inside the LANE compression function. In LANE-224 and LANE-256, a state of 256 bits is used, which corresponds to two AES states. In LANE-384 and LANE-512, the state is 512 bits in size, corresponding to four AES states.

A sequence of 32 or 64 bytes can be mapped to two or four AES states, depending on the LANE variant. The sequence is split into 16-byte parts, each of which is mapped to an AES state as described above. The AES states are ordered in the same way as the 16-byte parts in the original byte sequence, i.e., the leftmost AES state contains the first 16 bytes of the sequence.

2.2.2 The Finite Field GF(2^8)

As LANE is based on components of the AES block cipher, it also uses arithmetic operations in the finite field GF(2^8). Elements of the finite field GF(2^8) can be represented in several ways, but all representations are isomorphic, i.e., they are simply different ways of representing the same finite field with 2^8 elements [35]. In this document, we adopt the same representation as commonly used to describe the AES block cipher [19,45].

A *byte* is used to represent an element of the finite field GF(2^8). It is useful to view the byte, consisting of bits $b_0b_1b_2b_3b_4b_5b_6b_7$ as a polynomial with coefficients 0 or 1:

$$b_0X^7 + b_1X^6 + b_2X^5 + b_3X^4 + b_4X^3 + b_5X^2 + b_6X + b_7 \ . \tag{3}$$

The *addition* of two elements of GF(2^8), represented as polynomials, is defined as component-wise addition modulo two. On the byte level, this corresponds to exclusive or (XOR). The neutral element with respect to addition is the byte $\texttt{00}_x$, and every element is its own additive inverse.

The *multiplication* of two elements of GF(2^8) is defined as the the multiplication of polynomials, reduced modulo an irreducible polynomial $m(X)$,

$$m(X) = X^8 + X^4 + X^3 + X + 1 \ . \tag{4}$$

The byte $\texttt{01}_x$ is the neutral element with respect to multiplication. Every nonzero byte has a multiplicative inverse, which can be computed using the extended Euclidean algorithm.

2.3 Building Blocks

The LANE hash functions reuse several components from the AES block cipher [19, 45]. In particular, the SubBytes, ShiftRows and MixColumns transformations are also part of LANE. In LANE, however, they are used several times in parallel, due to the larger state size.

2.3.1 SubBytes

The SubBytes transformation in LANE is identical to the corresponding component of the AES block cipher, except that it operates on a larger state. Figure 1 illustrates this for LANE-224 and LANE-256. The same non-linear substitution (S-box) is applied to each of the state bytes independently. This substitution consists of the composition of the following operations:

1. The inverse operation in the finite field $\mathrm{GF}(2^8)$, defined by the irreducible polynomial $m(X)$, given in (4). The zero element is mapped to itself.

2. An affine mapping over $\mathrm{GF}(2)$, defined by

$$\begin{bmatrix} b'_7 \\ b'_6 \\ b'_5 \\ b'_4 \\ b'_3 \\ b'_2 \\ b'_1 \\ b'_0 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \cdot \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} . \tag{5}$$

Here, $b_0,\ldots,b_7$ denote the bits of a byte representing an element of $\mathrm{GF}(2^8)$, where b_0 is the most significant bit. This is the same S-box as the one used in the AES block cipher [19, 45]. It is given in Table 3.

2.3.2 ShiftRows

The ShiftRows transformation cyclically shifts the bytes of the rows of each of the AES states that comprise the LANE state. The first, i.e., topmost row is not shifted. The second, third and fourth row are cyclically shifted to the left over one, two and three byte positions, respectively. This is identical to the ShiftRows transformation in the AES block cipher, except that it is applied two or four times in parallel, depending on the LANE variant. Figure 2 illustrates ShiftRows for LANE-224 and LANE-256.

Table 3 – The AES S-box, in hexadecimal format.

	␣0	␣1	␣2	␣3	␣4	␣5	␣6	␣7	␣8	␣9	␣a	␣b	␣c	␣d	␣e	␣f
0␣	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1␣	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2␣	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3␣	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4␣	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5␣	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6␣	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7␣	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8␣	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9␣	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a␣	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b␣	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c␣	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d␣	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e␣	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f␣	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

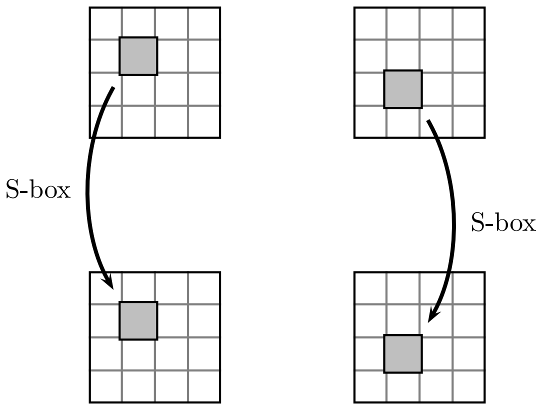


Figure 1 – The SubBytes transformation in Lane-224 and Lane-256.

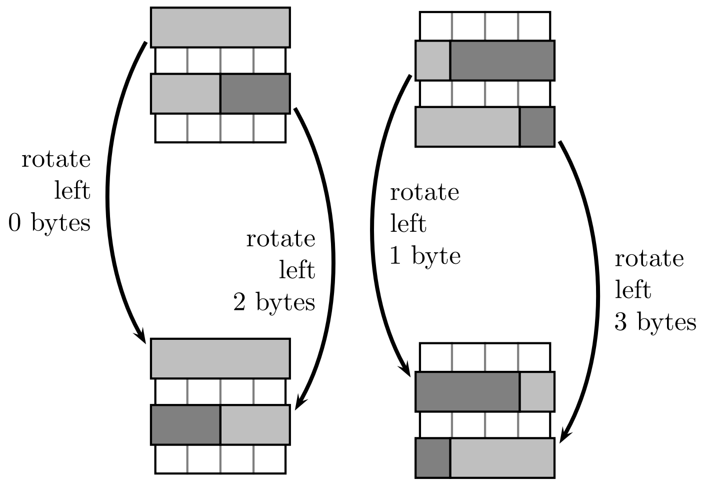


Figure 2 – The ShiftRows transformation in LANE-224 and LANE-256.

2.3.3 MixColumns

The MixColumns transformation operates on the columns of the state. Each column is viewed as a polynomial over GF(2^8), i.e., a polynomial of degree three with coefficients in GF(2^8):

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \quad \leftrightarrow \quad y_3 \cdot Y^3 + y_2 \cdot Y^2 + y_1 \cdot Y + y_0 \ . \tag{6}$$

Then, this polynomial is multiplied modulo Y^4+1 with the fixed polynomial $c(Y)$,

$$c(Y) = 03Y^3 + 01Y^2 + 01Y + 02 \ . \tag{7}$$

Even though $Y^4 + 1$ is not an irreducible polynomial over GF(2^8), implying that multiplication with a fixed polynomial is not necessarily invertible, the polynomial $c(Y)$ is such that MixColumns is an invertible operation. Equivalently, this operation can be written as a matrix multiplication

$$\begin{bmatrix} y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \end{bmatrix} = \begin{bmatrix} \mathtt{02}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{01}_x \\ \mathtt{01}_x & \mathtt{02}_x & \mathtt{03}_x & \mathtt{01}_x \\ \mathtt{01}_x & \mathtt{01}_x & \mathtt{02}_x & \mathtt{03}_x \\ \mathtt{03}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{02}_x \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \ . \tag{8}$$

Again, this is identical to the MixColumns transformation used in the AES block cipher. Figure 3 illustrates MixColumns for LANE-224 and LANE-256.

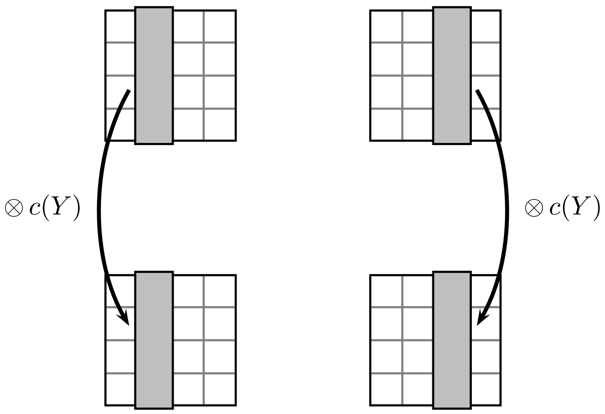


Figure 3 – The MixColumns transformation in LANE-224 and LANE-256.

2.3.4 AddConstants

The AddConstants transformation adds a 32-bit constant k_i to each column of the state. These constants k_i are generated using a linear feedback shift register (LFSR), which is described in pseudocode in Figure 4. Table 14 in Appendix A contains the values of the constants used in LANE, found using this algorithm.

Which constants are added to the state depends on the full round number r, which is given as a parameter to AddConstants. For LANE-224 and LANE-256, AddConstants is defined as

$$\begin{aligned}\mathrm{AddConstants}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_7) = \\ x_0 \oplus k_{8r} \,||\, x_1 \oplus k_{8r+1} \,||\, \cdots \,||\, x_7 \oplus k_{8r+7}\ .\end{aligned} \quad (9)$$

For LANE-384 and LANE-512, AddConstants is similarly defined as

$$\begin{aligned}\mathrm{AddConstants}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{15}) = \\ x_0 \oplus k_{16r} \,||\, x_1 \oplus k_{16r+1} \,||\, \cdots \,||\, x_{15} \oplus k_{16r+15}\ .\end{aligned} \quad (10)$$

Figure 5 shows the AddConstants transformation for LANE-224 and LANE-256.

2.3.5 AddCounter

The AddCounter transformation adds part of the counter to the state. The 64-bit counter C is split into two 32-bit words c_0 and c_1, where c_0 is the most significant and c_1 the least significant word, i.e., following the big endian convention.

Depending on the round parameter r, AddCounter adds one of these words to the fourth column of the first AES state. More formally, for LANE-224 and

```
1: $k_0 \leftarrow \texttt{07fc703d}_x$
2: for $i = 1$ to 272 (resp. 768 for LANE-384 and LANE-512) do
3:     $k_i = k_{i-1} \gg 1$
4:     if $k_{i-1} \wedge \texttt{00000001}_x$ then
5:         $k_i = k_i \oplus \texttt{d0000001}_x$
6:     end if
7: end for
```

Figure 4 – Pseudocode for generating the LANE constants.

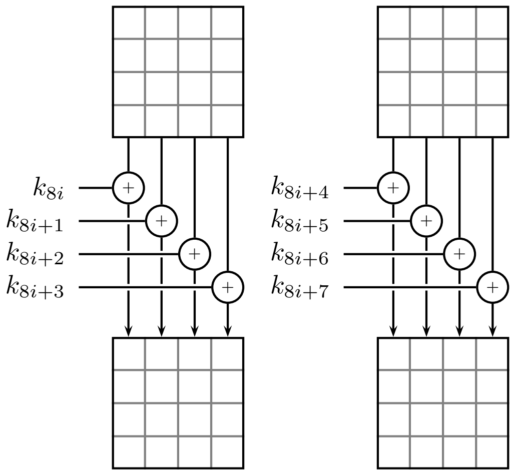


Figure 5 – The AddConstants transformation in LANE-224 and LANE-256.

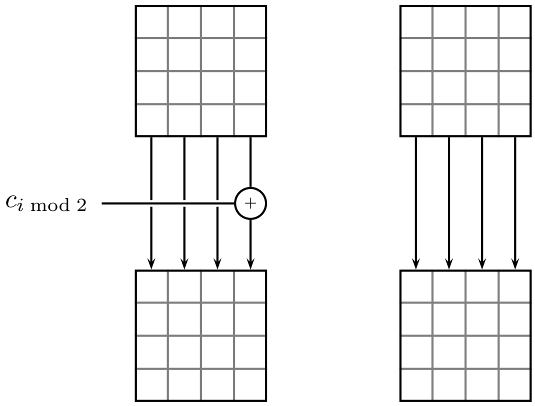


Figure 6 – The AddCounter transformation in Lane-224 and Lane-256.

Lane-256 it is given by

$$\text{AddCounter}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \,||\, \cdots \,||\, x_7) = \\ x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \oplus c_{r \bmod 2} \,||\, \cdots \,||\, x_7 \ . \quad (11)$$

Figure 6 shows the AddCounter transformation for Lane-224 and Lane-256. For Lane-384 and Lane-512, AddCounter is defined by

$$\text{AddCounter}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \,||\, \cdots \,||\, x_{15}) = \\ x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \oplus c_{r \bmod 2} \,||\, \cdots \,||\, x_{15} \ . \quad (12)$$

2.3.6 SwapColumns

The SwapColumns transformation takes a Lane state, and reorders the columns. It ensures that the AES states that comprise the Lane state are mixed among themselves. For Lane-224 and Lane-256 it is given by

$$\text{SwapColumns}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_7) = \\ x_0 \,||\, x_1 \,||\, x_4 \,||\, x_5 \,||\, x_2 \,||\, x_3 \,||\, x_6 \,||\, x_7 \ . \quad (13)$$

Figure 7 shows the SwapColumns transformation for Lane-224 and Lane-256. It can be viewed as a matrix transposition of a 2×2 matrix, where the elements are formed by pairs of state columns. For Lane-384 and Lane-512, SwapColumns is defined by

$$\text{SwapColumns}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{15}) = \\ x_0 \,||\, x_4 \,||\, x_8 \,||\, x_{12} \,||\, x_1 \,||\, x_5 \,||\, x_9 \,||\, x_{13} \,||\, x_2 \,||\, x_6 \,||\, x_{10} \,||\, x_{14} \,||\, x_3 \,||\, x_7 \,||\, x_{11} \,||\, x_{15} \ . \quad (14)$$

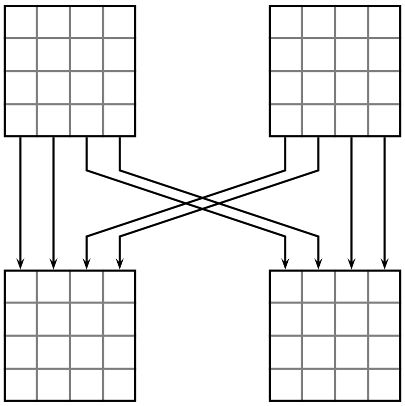


Figure 7 – The SwapColumns transformation in Lane-224 and Lane-256.

Figure 8 shows the SwapColumns transformation for Lane-384 and Lane-512. Similar to Lane-256 and Lane-224, SwapColumns can be seen as a matrix transposition, now of a 4×4 matrix, where the elements are the columns of the state.

2.4 Preprocessing

Before hashing a message using Lane, two preprocessing steps are carried out: message padding, and setting the initial chaining value.

2.4.1 Message Padding

Lane processes a message in blocks of a fixed size, the blocksize. For Lane-224 and Lane-256, the blocksize is 512 bits, and for Lane-384 and Lane-512, the blocksize is 1024 bits. To support any message length up to $2^{64} - 1$ bits (included), zero bits are appended to the message until its length is an integer multiple of the blocksize.

More formally, a message M of length l is padded as follows. Let b be the blocksize and let κ be the smallest positive integer, $0 \leq \kappa < b$, such that

$$l + \kappa \equiv 0 \pmod{b} \ . \tag{15}$$

Now, the padded message is computed as

$$\mathrm{pad}\,(M) = M \,||\, 0^{\kappa} \ . \tag{16}$$

This padding rule ensures that the padded message can be split into an integer number of blocks of b bits. Note that, if the message length l already is an integer

Table 4 – The flag byte ϕ.

	No salt used	Salt used
Output transformation	$\texttt{00}_x$	$\texttt{01}_x$
Derivation of $IV_{n,S}$	$\texttt{02}_x$	$\texttt{03}_x$

multiple of the blocksize b, no padding bits are added. The empty string is a particular example of this; no padding bits are added to it. Thus, when hashing the empty string, no message blocks are to be processed, and one proceeds immediately with the output transformation.

2.4.2 Setting the Initial Chaining Value

Every digest size supported by Lane uses a different initial value $IV_{n,S}$, which also depends on the (optional) salt. These are defined using the Lane compression function $f(H, M, C)$ itself, which will be defined in detail in Sect. 2.5.

Let n be the digest size in bits, i.e., n is 224, 256, 384 or 512. Let S be the salt value, or zero if no salt is used. The initial value $IV_{n,S}$ is then given by the output of the following compression function call using a zero input chaining value and a zero counter value:

$$IV_n = f\left(0, \phi \,||\, \mathrm{bin}_{32}(n) \,||\, 0^\star \,||\, S, 0\right) \ . \tag{17}$$

Here, $\mathrm{bin}_{32}(n)$ is the digest length n in bits, represented as a 32-bit big-endian integer. The flag byte ϕ indicates whether or not a salt value is used; see Table 4. If a salt value is not used, $\phi = \texttt{02}_x$ and the salt S is filled with zero bits. If a salt value is used, $\phi = \texttt{03}_x$. The size of the salt S is 256 bits for Lane-224 and Lane-256, and 512 bits for Lane-384 and Lane-512, as indicated in Table 1. Note that generalisations of Lane to other digest lengths can be defined in a similar way, if desired.

If no salt is being used, the initial values can also be precomputed for each digest size. Table 5 lists the initial values for supported digest sizes.

2.5 The Lane Compression Function

This section describes the Lane compression function $f(H_{i-1}, M_i, C_i)$. This function takes the following three inputs:

- The input chaining value H_{i-1} is equal to the output of the previous compression function call, or, for the first compression function call, the initial value $IV_{n,S}$. In Lane-224 and Lane-256, the size of the chaining value is 256 bits. In Lane-384 and Lane-512, a 512-bit chaining value is used.

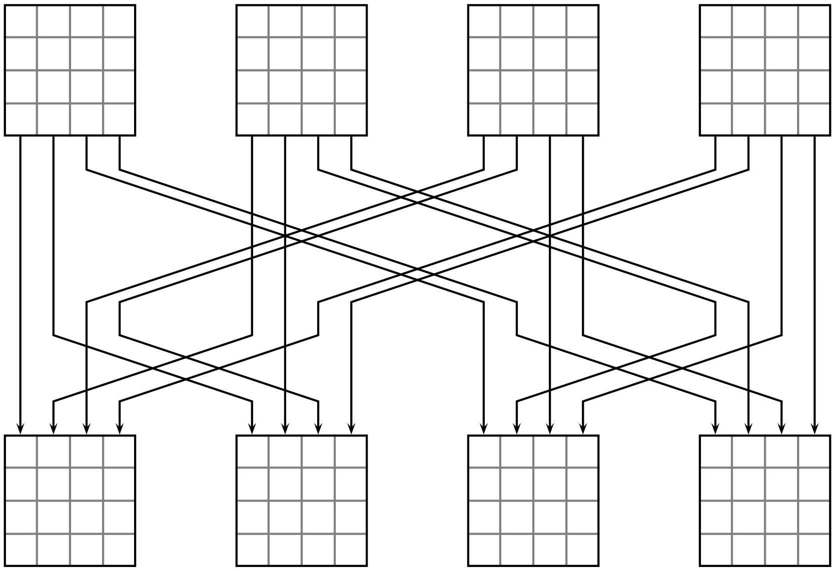


Figure 8 – The SwapColumns transformation in LANE-384 and LANE-512.

Table 5 – The LANE initial values IV_n, in big-endian notation, if no salt is used.

LANE-224	`c8245a868d733102314ddcb9f60a7ef4`$_x$
	`57b8c917eefeaec2ff4fc3be87c4728e`$_x$
LANE-256	`be292e17bb541ff2fe54b6f730b1c96a`$_x$
	`7b2592688539bdf397c4bdd649763fb8`$_x$
LANE-384	`148922ce548c300176978bc8266e008c`$_x$
	`3dc60765d85b09d94cb1c8d8e2cab952`$_x$
	`db72be8e685f0783fa436c3d4b9acb90`$_x$
	`5088dd47932f55a9a0c415c6db6dd795`$_x$
LANE-512	`9b6034811d5a931b69c4e6e0975e2681`$_x$
	`b863ba538d1be11b77340080d42c48a5`$_x$
	`3a3a1d611cf3a1c4f0a303477e56a44a`$_x$
	`9530ee60dadb05b63ae3ac7cd732ac6a`$_x$

- The message block M_i holds part of the padded message. Each message block is of a fixed size, the blocksize, which is indicated in Table 1. In Lane-224 and Lane-256, the blocksize is 512 bits. In Lane-384 and Lane-512, the blocksize is 1024 bits.
- The counter C_i holds the number of message bits hashed so far, including the message bits in the current message block M_i. The counter C_i is represented as a 64-bit unsigned integer in big-endian notation.

The structure of the Lane compression function is shown in Figure 9. It consists of a message expansion, eight permutation *lanes*, arranged in two layers, and three XOR combiners. Section 2.5.1 describes the message expansion. The permutation *lanes* are discussed in Sect. 2.5.2.

2.5.1 The Message Expansion

The message expansion of Lane takes the message block M_i and the input chaining value H_{i-1}, and expands them into six expanded message blocks, $W_0, \ldots, W_5$.

In Lane-224 and Lane-256, the six expanded message words, $W_0, \ldots, W_5$, are all 256 bits long. They are computed as follows. Split the 512-bit message block M_i into four 128-bit parts, $m_0, \ldots, m_3$:

$$m_0 \,||\, m_1 \,||\, m_2 \,||\, m_3 \leftarrow M_i \ . \tag{18}$$

Similarly, split the 256-bit input chaining value H_{i-1} into two 128-bit parts, h_0 and h_1:

$$h_0 \,||\, h_1 \leftarrow H_{i-1} \ . \tag{19}$$

Then, compute the six expanded message words, $W_0, \ldots, W_5$ as

$$\begin{array}{rcl@{\quad}c@{\quad}l}
W_0 & = & h_0 \oplus m_0 \oplus m_1 \oplus m_2 \oplus m_3 & || & h_1 \oplus m_0 \oplus m_2 \\
W_1 & = & h_0 \oplus h_1 \oplus m_0 \oplus m_2 \oplus m_3 & || & h_0 \oplus m_1 \oplus m_2 \\
W_2 & = & h_0 \oplus h_1 \oplus m_0 \oplus m_1 \oplus m_2 & || & h_0 \oplus m_0 \oplus m_3 \\
W_3 & = & h_0 & || & h_1 \\
W_4 & = & m_0 & || & m_1 \\
W_5 & = & m_2 & || & m_3
\end{array} \ . \tag{20}$$

The message expansion in Lane-384 and Lane-512 is completely analogous. The only difference is that all sizes are doubled, i.e., the 1024-bit message block M_i is split into four 256-bit parts, $m_0, \ldots, m_3$ and the 512-bit input chaining value H_{i-1} is split into two 256-bit parts. Then, (20) is used to compute the six 512-bit expanded message words $W_0, \ldots, W_5$.

2.5.2 The Permutations

The Lane compression function contains eight permutations, arranged in two layers. Each permutation consists of a number of rounds, where the number of

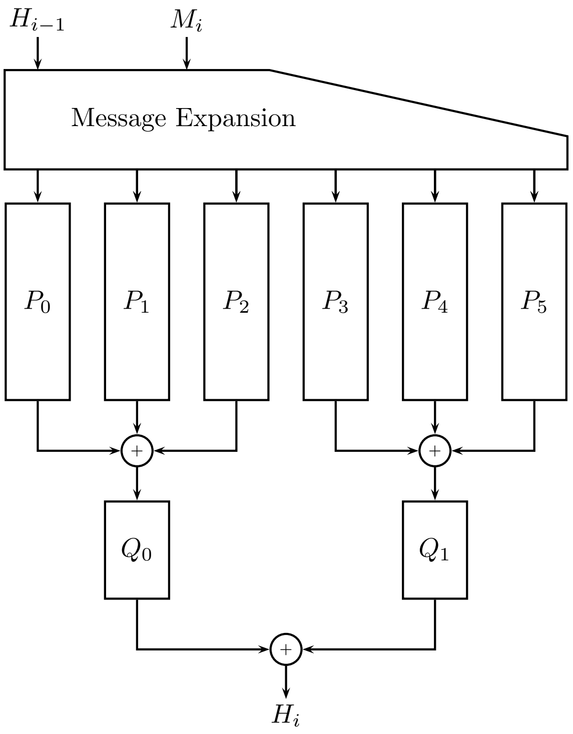


Figure 9 – The LANE compression function.

```
function Round($r$, $X$)
1: $X \leftarrow$ SubBytes($X$)
2: $X \leftarrow$ ShiftRows($X$)
3: $X \leftarrow$ MixColumns($X$)
4: $X \leftarrow$ AddConstants($r, X$)
5: $X \leftarrow$ AddCounter($r, X$)
6: $X \leftarrow$ SwapColumns($X$)
7: return $X$
```

```
function LastRound($X$)
1: $X \leftarrow$ SubBytes($X$)
2: $X \leftarrow$ ShiftRows($X$)
3: $X \leftarrow$ MixColumns($X$)
4: $X \leftarrow$ SwapColumns($X$)
5: return $X$
```

Figure 10 – Pseudocode for the LANE permutation rounds.

Table 6 – Number of rounds in the LANE permutations.

	LANE-224	LANE-256	LANE-384	LANE-512
$P_0, \ldots, P_5$	6	6	8	8
Q_0, Q_1	3	3	4	4

rounds is different for the two layers: the permutations in the first layer have twice as many rounds as those in the second layer. In the rest of the document, we use "lane" as a synonym for a single LANE permutation.

The rounds of the permutations use the building blocks described in Sect. 2.3. More in detail, a full permutation round consists of the following sequence of transformations: SubBytes, ShiftRows, MixColumns, AddConstants, AddCounter and SwapColumns. The last round of each permutation omits AddConstants and AddCounter. Figure 10 gives a pseudocode description of the LANE permutation rounds.

Note that a permutation round can be seen as two, for LANE-224 and LANE-256, or four, for LANE-384 and LANE-512, parallel invocations of a round of the AES block cipher [19,45], where the appropriate constants and counter word are used as a round key, followed by SwapColumns.

In LANE-224 and LANE-256, the first layer permutations, P_0 until P_5, consist of six rounds each. The second layer permutations, Q_0 and Q_1, have three rounds each. In LANE-384 and LANE-512, the number of rounds in increased to eight rounds for the P_i's and four rounds for the Q_i's. Table 6 summarises the number of rounds in the permutations.

A round number r is assigned to each of the full rounds across all permutations, to specify the constants and counter to use in each round. The permutations are taken in the order P_0, P_1, …, P_5, Q_0, Q_1 and only the full rounds are counted, i.e., the last round of each permutation is ignored. Table 7 lists the round numbers r in each of the permutations. Each full round is given its round number r as an extra parameter, as indicated in Figure 10. This parameter is then passed on to the AddConstants and AddCounter transformations, described in Sect. 2.3.4 and Sect. 2.3.5, respectively.

Table 7 – The full round number r.

	LANE-224	LANE-256	LANE-384	LANE-512
P_0	0—4	0—4	0—6	0—6
P_1	5—9	5—9	7—13	7—13
P_2	10—14	10—14	14—20	14—20
P_3	15—19	15—19	21—27	21—27
P_4	20—24	20—24	28—34	28—34
P_5	25—29	25—29	35—41	35—41
Q_0	30—31	30—31	42—44	42—44
Q_1	32—33	32—33	45—47	45—47

A pseudocode description of the permutations used in LANE-224 and LANE-256 is given in Figure 11, including an exact expression to compute the full round number r for each round. Figure 12 describes the permutations used in LANE-384 and LANE-512.

2.6 The Output Transformation

The output transformation of LANE takes as input the chaining value after all padded message blocks have been processed, and returns the message digest. It also includes the message length l, and the (optional) salt S, if one was used.

The transformation consists of two parts. First, a single additional compression function call is done. The counter C is set to zero, and the message input is set to

$$\phi \,||\, \mathrm{bin}_{64}(l) \,||\, 0^{\star} \,||\, S \ . \tag{21}$$

Here, $\mathrm{bin}_{64}(l)$ is the (unpadded) message length l in bits, represented as a 64-bit big-endian integer. The flag byte ϕ indicates whether or not a salt value is used; see Table 4. If a salt value is not used, $\phi = \mathtt{00}_x$ and the salt S is filled with zero bits. If a salt value is used, $\phi = \mathtt{01}_x$. The size of the salt S is 256 bits for LANE-224 and LANE-256, and 512 bits for LANE-384 and LANE-512, as indicated in Table 1.

In the second part of the output transformation, a truncation is applied to compute the final message digest. No output truncation is required for LANE-256 and LANE-512, as the size of the chaining value is equal to the required digest size. In LANE-224 and LANE-384, however, this is not the case. The digest of LANE-224 is found by taking only the first, i.e., leftmost 224 bits of the last 256-bit chaining value. Similarly, in LANE-384, only the first 384 bits of the last 512-bit chaining value are used.

More formally, the truncation operation for LANE-224 is given by

$$\mathrm{Trunc}_{224}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_6 \,||\, x_7) = x_0 \,||\, x_1 \,||\, \cdots \,||\, x_6 \ . \tag{22}$$

```
function $P_j(X)$
1: for $i = 0$ to $4$ do
2:     $r \leftarrow 5j + i$
3:     $X \leftarrow \text{Round}(r, X)$
4: end for
5: $X \leftarrow \text{LastRound}(X)$
6: return $X$
```

```
function $Q_j(X)$
1: for $i = 0$ to $1$ do
2:     $r \leftarrow 30 + 2j + i$
3:     $X \leftarrow \text{Round}(r, X)$
4: end for
5: $X \leftarrow \text{LastRound}(X)$
6: return $X$
```

Figure 11 – Pseudocode for the permutations in Lane-224 and Lane-256.

```
function $P_j(X)$
1: for $i = 0$ to $6$ do
2:     $r \leftarrow 7j + i$
3:     $X \leftarrow \text{Round}(r, X)$
4: end for
5: $X \leftarrow \text{LastRound}(X)$
6: return $X$
```

```
function $Q_j(X)$
1: for $i = 0$ to $2$ do
2:     $r \leftarrow 42 + 3j + i$
3:     $X \leftarrow \text{Round}(r, X)$
4: end for
5: $X \leftarrow \text{LastRound}(X)$
6: return $X$
```

Figure 12 – Pseudocode for the permutations in Lane-384 and Lane-512.

For Lane-384, the truncation is defined similarly as

$$\text{Trunc}_{384}(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{11} \,||\, \cdots \,||\, x_{15}) = x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{11} \ . \tag{23}$$

Note that generalisations of Lane to other digest lengths can be defined using a similar truncation, if desired.

3 Design Rationale

This section discusses the rationale behind the design of Lane. All of the important design decisions are explained. The discussion of the rationale is structured by components of the Lane hash function. The advantages and disadvantages of Lane are also discussed in this section.

3.1 The Iteration Mode

The iteration mode used in Lane was designed to be easy to understand and implement. It is based on the well-known Merkle-Damgård construction [20, 41]. For this construction, it can be proven that if the compression function is collision resistant, so is the iterated hash function built on it.

The same iteration mode supports multiple digest lengths. One simply needs to compute the initial chaining value, which depends on the digest length, and after the iteration apply a suitable truncation to the message digest. The derivation of the initial chaining value is based on the LANE compression function, for ease of implementation.

3.1.1 The Message Padding

The message padding is simplified when compared to plain Merkle-Damgård, as used in the SHA family of hash functions [48]. As LANE uses an output transformation, which is simply an additional compression function call, it is natural to include the message length, i.e., the Merkle-Damgård strengthening, in this extra block.

Because of this extra block, one can now simply pad a message with zero bits until the next block boundary. It no longer depends on the exact message length whether or not an extra padding block has to be introduced. This greatly simplifies implementation, and still results in an efficient iteration mode. If no salt is used, the initial chaining value can be precomputed, and the total number of compression function calls, including the output transformation, is

$$\#\text{calls to } f = \left\lceil \frac{l}{b} \right\rceil + 1 \ . \tag{24}$$

For plain Merkle-Damgård, assuming that the representation of the message length in the padding uses 64 bits, this is

$$\#\text{calls to } f = \left\lceil \frac{l+65}{b} \right\rceil \ . \tag{25}$$

This means that the LANE iteration mode uses at most one additional compression function call compared to plain Merkle-Damgård, but has the advantage that there is always one extra compression function call, i.e., the LANE output transformation, whose 'message' input is not under the control of the adversary, except very limited influence via choosing the message length. In plain Merkle-Damgård, an adversary could choose the message length such that almost all of the padded message bits in the last block can be chosen freely.

3.1.2 The Use of a Counter

Additionally, the LANE mode of iteration borrows the idea of including a bit counter in every compression function call from the 'HAsh Iterative FrAmework' (HAIFA) of Biham and Dunkelman [10]. This stops several attacks on the iteration, at only a very modest cost. If no counter is used, a fixed point of the compression function, if found, can be concatenated to itself to form an *expandable message* [21, 30], i.e., a set of message patterns of different lengths, all leading to the same internal hash state. Such an expandable message can for instance be used to construct efficient second preimage attacks on long messages [30]. Due to the use of a bit counter, however, it is no longer possible to concatenate a fixed point to itself, as it will only be a fixed point for a specific counter value [10].

3.1.3 The Output Transformation

An output transformation is used to offer an additional layer of protection against (first) preimage attacks. For simplicity, this output transformation is constructed based on the Lane compression function, with a message block of a fixed structure. It is straightforward to see that this structure imposed on the message block used in the output transformation drastically limits the freedom of an adversary seeking a preimage.

The output transformation also serves to protect against length-extension attacks, as it is impossible to simulate the effect of the output transformation using a regular message block. Indeed, the output transformation takes a zero counter as input, which according to the specification is not possible in a normal message block.

The output transformation also offers additional protection against distinguishing attacks, as any potential bias in the compression function is expected to be destroyed by the output transformation.

3.1.4 The Use of a Salt Value

Lane supports the use of a salt value, if this is desirable for the application. A well-known example of such an application is password hashing. If a different salt is used for every stored password, it is no longer possible to attack multiple targets in parallel in a dictionary attack or an exhaustive search. Digital signatures are another application where a salt provides a benefit. This is referred to as *randomised hashing*, after the work of Halevi and Krawczyk [27]. Consider the scenario where an attacker constructs two colliding messages, and asks the victim to sign the first message. Because the second message has the same message digest as the first, the signature is also valid for the second message. If the victim chooses a random salt before signing the message, however, the collision that was carefully crafted by the attacker is destroyed with an overwhelming probability.

When a salt is used in Lane, this salt value is included in the derivation of the initial chaining value as well as in the output transformation. Both of these operations are simply Lane compression function calls with a specific message block and a zero counter input. Apart from the salt value, this message block also includes a flag byte ϕ. The purpose of this byte is to provide domain separation. More specifically, the only compression function calls in Lane that use a zero counter C occur exactly in the derivation of the initial chaining value and in the output transformation. Hence, it is impossible to simulate these calls using a normal message block. In order to provide a similar separation for the four cases that do have a zero counter C, i.e., initial value derivation with or without salt, and output transformation with or without salt, the flag byte ϕ is used.

3.1.5 A Parallel Iteration Mode

The iteration mode used by LANE is inherently sequential. Hence, it is not possible to benefit from having multiple CPU cores to accelerate the hashing of a single, long message. There is small-scale parallelism available inside the compression function, which can be used by a single CPU, as will be explained in Sect. 3.2. But this parallelism is too fine-grained to offset the synchronisation overhead required when using multiple independent CPU cores.

In many high-performance applications, this is not a problem. Indeed, often there are many smaller messages that need to be hashed. Consider for example a web server using TLS with HMAC-LANE for data authentication. The server needs to hash every packet it sends to and receives from the network. A machine with multiple CPU cores can process all of these independent messages in parallel.

For applications that do require parallelisable hashing of a single, long message, it is beneficial to use a separate parallel iteration mode. We propose a simple and easy to implement parallel mode that is built on top of the normal, sequential LANE. This mode is based on the seminal work of Damgård [20].

Let T be the desired level of parallelism, i.e., up to T CPU cores can be utilised. Let b_{int} be the interleave factor, which defines the size of the blocks in which the message will be split. It is logical to choose b_{int} to be a multiple of the blocksize b of the underlying hash function, although this is not strictly required. Parse the message M into blocks of b_{int} bits:

$$m_0 \,||\, m_1 \,||\, \ldots \leftarrow M \ . \tag{26}$$

Then assign the blocks in turn to T streams $\mathcal{M}_0,\ldots,\mathcal{M}_{T-1}$:

$$\begin{aligned} \mathcal{M}_0 &= m_0 \,||\, m_T \,||\, m_{2\cdot T} \,||\, \ldots \\ &\vdots \\ \mathcal{M}_i &= m_i \,||\, m_{T+i} \,||\, m_{2\cdot T+i} \,||\, \ldots \\ &\vdots \\ \mathcal{M}_{T-1} &= m_{T-1} \,||\, m_{T+T-1} \,||\, m_{2\cdot T+T-1} \,||\, \ldots \end{aligned} \quad . \tag{27}$$

Finally, compute the digest of the message M as

$$\text{LANE}\Big(\text{LANE}\,(\mathcal{M}_0) \,||\, \text{LANE}\,(\mathcal{M}_1) \,||\, \cdots \,||\, \text{LANE}\,(\mathcal{M}_{T-1})\Big) \ . \tag{28}$$

There are T inner hash functions, all of which are independent and can thus be evaluated in parallel. When the message M is long, the cost of the final hash function, which combines the results from the T streams, is negligible.

Note that this mode is not interoperable with the 'normal' mode of LANE, as the computed message digest is different. Also, different values for the interleave factor b_{int} and the parallelisation degree T result in a different message digest.

3.2 The Compression Function

The Lane compression function was designed to be simple to understand and easy to analyse. This aim for simplicity can be found in virtually every aspect of the design.

The use of permutations ensures that internal collisions can only occur in certain places, i.e., at the XOR combiners. Establishing such an internal collision is equivalent to satisfying a linear condition on the outputs of several permutations. Similarly, the message expansion imposes linear relations on the inputs of the permutations. The rationale is that, while such conditions are very simple, it is hard to maintain or even track them through the rounds of the permutations.

A similar rationale applies to the problem of finding (second) preimages for the compression function. Straightforward inversion attempts fail, as one has to ensure that the linear conditions imposed by the message expansion hold. This is again considered to be very difficult.

As described in detail in Sect. 4.4.1, having only a single layer of permutations would allow for a class of distinguishers for the compression function, based on limiting the permutation inputs to a small set. The second layer of permutations not only prevents that, but also has a beneficial effect on the resistance to differential cryptanalysis. Indeed, in a collision differential, either the entire second layer must be activated, or an internal collision must be reached simultaneously on both of the XOR combiners after the first layer, i.e., on a value twice the size of the chaining value.

The ample parallelism provided by the Lane compression function allows for flexibility in implementation. In software implementations, Lane offers many opportunities for instruction level parallelism (ILP), which can be used by modern pipelined and superscalar CPU's. Also, as the same operations are carried out on many independent data values in parallel, it is possible to use vector instructions, i.e., Single Instruction Multiple Data (SIMD) instructions. On the other end of the spectrum, it is equally possible to implement Lane in a completely serial way. In such implementations, the memory requirements are kept minimal. Hardware designers implementing Lane are offered an area-speed tradeoff, making Lane suitable for both resource-constrained and very high-speed applications.

3.2.1 The Message Expansion

Even more so than other components of Lane, the message expansion was chosen to be very simple and light. Its main purpose is to introduce dependencies between the inputs of the various permutation lanes, such that they cannot be chosen independently. It also precludes straightforward inversion attempts, as it is conjectured that, however simple the linear conditions imposed by the message expansion, it is not feasible to satisfy them when only having direct control over the permutation outputs.

A similar structure, with four parallel branches, is found in the Rumba20 compression function [8]. In Rumba20, constants are used at the input to prevent finding preimages by inverting individual branches. The (linear) relations between the inputs of the various lanes of the first layer serve a similar purpose in Lane.

The message expansion is based on a (6,3,4) linear code over GF(4). The minimum distance property of this code ensures that, in a differential attack, at least four out of the six lanes in the first layer will be *active*, i.e., have a difference at the input as well as output. This property is described in more detail in Sect. 4.2.1.

Provable resistance is offered against meet-in-the-middle preimage attacks, as detailed in Sect. 4.9. In short, it is not possible to construct two independent sets of permutation lanes to use in such an attack. This follows from the minimum distance property of the linear code on which the message expansion is based.

Also for implementors, the message expansion has several interesting properties. Each output of the message expansion can be computed independently of the others, and read-only access to the current message block suffices. This implies that the message buffer can be shared with another application, eliminating the need for extra memory and costly data copying.

Finally, note that the inputs of the permutation lanes P_4 and P_5 only depend on the message block input, and not on the chaining value. This implies that those lanes can already be computed while the previous chaining value is not yet known, e.g., in parallel with the second layer of the previous compression function call. This implementation approach is described in more detail in Sect. 5.1.1. If two (or more) CPU cores are available, it is also possible to let one CPU core precompute $P_4\left(M^h\right) \oplus P_5\left(M^l\right)$, while the second CPU core takes care of the rest of the lanes. In this setting the synchronisation overhead between the CPU cores is manageable.

3.2.2 The Permutations

The permutations used in Lane are built using components of the AES block cipher [19, 45]. One motivation for this choice is that these components and their properties are well studied and hence well understood. This allows to build on existing work on the security of these components to analyse Lane.

Reusing AES components also has several practical benefits. Much effort has already been spent on efficient implementations of the AES on a wide variety of platforms. Since Lane is based on the AES, these techniques can equally be applied to Lane. Another benefit lies in resource constrained environments, requiring both a hash function and a block cipher. Using Lane together with the AES allows large parts of the implementation to be shared, yielding a substantial overall improvement.

For simplicity and ease of (parallel) implementation, all permutations in Lane are built in the same way. Different constants are thus required in each permutation lane, to ensure that any attack based on maintaining symmetry across several permutation rounds is avoided.

The permutations are keyed using the bit counter input to the compression function. This is a natural way of including the bit counter, as it is very simple and lightweight, but achieves the goal of making the whole compression function dependent on this counter. Even though scenarios where the compression function is attacked by introducing differences via the bit counter are of no immediate concern, the method by which the counter is included provides resistance against such attacks. The rationale is that the bit counter can only influence a small part of the state in each round, and those influences cannot be cancelled out immediately in the next round, but instead diffuse to affect the whole state. The fact that the same counter value is used many times in the compression function serves to further complicate such cryptanalytic attempts.

The number of rounds in the permutations in the first layer was chosen to be six rounds for Lane-224 and Lane-256, and eight rounds for Lane-384 and Lane-512. The rationale behind this choice is to use as few rounds as possible, for performance reasons, but still enough rounds to offer an adequate security margin. We refer to the discussion of truncated differential analysis in Sect. 4.3 for a more detailed analysis concerning the required number of rounds in the first layer.

Concerning the number of rounds in the second layer of permutations, recall that the main purpose of the second layer is to preclude higher order differential distinguishers, such as the distinguishers described in Sect. 4.4. Such distinguishers are based on detecting the balancedness of the intermediate values after the first layer, which is a very fragile property. Almost any non-invertible and non-linear second layer would suffice to this end, but it is reasonable to ensure that every input bit influences every output bit, i.e., to achieve full diffusion. It is also a logical choice to use the same type of permutations as in the first layer. Achieving full diffusion requires a minimum of three rounds. Hence, the second layer permutations are defined to have half as many rounds as the first layer permutations.

Unlike in the AES block cipher, the linear diffusion layer is not omitted in the last rounds of the permutations, even though its impact on the security of Lane is limited. Doing these extra operations simply makes many implementations faster and easier, both in high-performance software and hardware. Namely, we avoid handling a special case which would otherwise require multiplexers on the critical path in hardware, or extra tables or masking instructions in software. Only in applications where the MixColumns operation has to be computed explicitly, for instance in embedded software implementations, would omitting MixColumns offer a performance benefit. In the AES, another reason to omit these operations, besides a performance gain in embedded implementations, is to achieve a similar structure for the inverse cipher. But as the permutations in Lane are only ever evaluated in the forward direction, this argument does not apply to Lane.

3.2.3 The Constants

The constants serve to diversify the permutations, in order to avoid any attack based on the similarity of the parallel permutation lanes. An important design goal is that it should be possible to generate the constants on-the-fly in an inexpensive way. This avoids the need for large tables of constants in implementations where memory is limited.

A linear feedback shift register (LFSR) is a natural choice for generating constants. It is simple, and can be implemented using only very limited resources. Even though its output stream does not possess any strength in the cryptographic sense, the statistical properties are sufficiently good for the purposes of LANE. A 32-bit LFSR was chosen to match the size of the columns. The feedback polynomial is a primitive polynomial, ensuring a cycle length of $2^{32} - 1$.

The only security-related requirement on the constants is that the constants used in different permutation lanes should be different. The first constant, used to initialise the LFSR, was chosen such that no two constant bytes used in the same position of two different lanes are equal. Additionally, the number of times that two constant bytes, used in the same position in a different lane, are the one's complement of each other was minimised. An exhaustive search resulted in the conclusion that this complement property cannot be avoided. There exist ten starting states for which this happens in only a single byte. Of these ten, we picked the starting state with the lowest numerical value. The source code for this search is included in the submission package.

3.3 Advantages and Limitations of Lane

3.3.1 Advantages

- LANE design is simple. This makes LANE easy to understand and implement. Also, simplicity is an important advantage for cryptanalysis. Complex designs are often hard, or even impossible to analyse in a structured way. The design of LANE, on the other hand, allows for a relatively easy analysis of its security.

- LANE incorporates several features that can greatly improve its security, at only a modest cost in performance. In particular, LANE offers the possibility of using a salt value, uses a counter and has an output transformation.

- Components from the AES block cipher [19,45] are reused as building blocks in LANE. As discussed above, this allows existing cryptanalytic results on the AES to be used in the security analysis of LANE. Also, implementations of LANE can benefit from existing work on the implementation of the AES on a wide variety of platforms. In particular, LANE can benefit from dedicated hardware support intended to accelerate the AES, like for instance the Intel AES-NI instruction set [28].

- One of the design goals of LANE was to provide a high degree of parallelism in the compression function. At the same time, care was taken to keep the memory requirements modest for a serialised implementation. Thus, LANE is flexible in implementation and scales well across a wide range of platforms and applications.
- LANE can easily be extended to support any digest length up to 512 bits. One simply needs to derive the initial chaining value for the desired digest length, and apply a suitable truncation at the end.
- There is a clear and detailed rationale, which was presented in this section, supporting every design decision.

3.3.2 Limitations

- The iteration mode is not parallelisable. For most applications, this is not a problem. For applications where a parallelisable iteration mode is important, we suggest to use the parallel mode described in Sect. 3.1.5.
- Because the size of the intermediate chaining values was chosen to be equal to the digest length, Joux' multicollision attack [29] can be applied to LANE. Refer to Sect. 4.12 for a more in-depth treatment of multicollisions.

4 Security Analysis

In this section, we discuss the security of the LANE construction in general, as well as of the hash functions LANE-256 and LANE-512 in particular. We list known bounds on security and present attacks on reduced versions of the hash functions.

In Sect. 4.1, we suggest ways in which LANE could be reduced, to perform cryptanalysis. Sections 4.2, 4.3 and 4.4 address differential attacks and their applicability to weakened versions of LANE. As the LANE compression function shares a certain similarity with wide-block Rijndael, Sect. 4.5 summarises cryptanalytic results on Rijndael, and their relevance to LANE. Sect. 4.6 is dedicated to algebraic attacks.

Sections 4.8–4.12 discuss the resistance of LANE to various generic attacks. Sect. 4.13 summarises security arguments on the mode of operation; a technical report detailing these results is included as a separate document [1]. Sect. 4.14 concludes with a statement on the expected security of the LANE hash functions.

4.1 Reduced Versions of Lane for Cryptanalysis

This section discusses various ways in which LANE could be reduced, in order to construct weakened variants. Such variants can be useful in cryptanalysis, as they allow one to understand the margin offered by the full LANE against a particular type of attack.

A first, obvious way to reduce LANE is to vary the number of rounds used in the permutations. For example, lowering the number of rounds increases the probability of (truncated) differentials. The number of rounds in the first layer (P_i) and second layer (Q_i) can be varied independently.

Another option is to reduce the number of lanes in the first layer. A variant where a single lane is removed from the first layer, for instance, would be based on a (5,3,3) linear code, which is simply a shortening of the original code. The number of lanes can be reduced further, but then the property that in a differential attack, always more than half of the lanes are active, is lost.

Finally, LANE could be reduced by omitting the entire second layer, i.e., the Q_i permutations. The compression function output would then be found as the XOR of all six lanes, P_0 to P_5.

4.2 Standard Differential Cryptanalysis

Differential cryptanalysis was originally introduced by Biham and Shamir [11] as a means to cryptanalyse symmetric encryption primitives (block ciphers). It has also been used with success to break hash functions, e.g. [13, 22].

In a differential attack, collisions are determined by considering *pairs* of messages, which have a fixed difference, i.e., the input difference of the *characteristic*. This condition strongly reduces the search space in which the attacker looks for collisions. It is hoped that the fraction of the collisions that lies within this reduced search space is larger and/or easier to find than in the unrestricted search space.

An important difference between differential cryptanalysis of hash functions and differential cryptanalysis of block ciphers is stated in the following fact.

Fact 1. *The absence of secret keys in hash functions can be exploited by the cryptanalyst in order to reduce the complexity of a differential attack.*

For instance, instead of choosing inputs ('plaintexts'), the cryptanalyst can choose any intermediate state, and compute backwards to determine the inputs. The effect is that a number of active S-boxes can be 'bypassed', resulting in a decrease of the complexity of a differential attack.

We consider differential attacks with nonzero differences in the message only. Since each individual lane of LANE implements a permutation in the space of n-bit message inputs, collisions can be obtained only in the XOR combiners. In order to obtain a collision at the output of the compression function, either a collision must be obtained in both of the XOR combiners at the input of the second layer, or both of the lanes in the second layer will be active.

We discuss only the resistance against differential attacks provided by the first layer of lanes. If the lanes of the second layer are active, then they will increase the security against differential attacks.

4.2.1 Active Lanes in the First Layer

This section describes a property of the Lane message expansion which ensures that, in a differential attack, at least four lanes in the first layer will be *active*, i.e., have a difference. To this end, we first introduce an alternative description of the Lane message expansion, based on a linear code over GF(4).

We adopt the standard polynomial representation of GF(4) using $X^2 + X + 1$ as a primitive polynomial to define the multiplication of field elements. A string of two bits $b_0 b_1$, where b_0 is the most significant bit, can now be mapped to an element in GF(4), and vice versa

$$b_0 b_1 \quad \leftrightarrow \quad b_0 \cdot X + b_1 \ . \tag{29}$$

An alternative description of the Lane message expansion. The message expansion of Lane is based on a linear (6,3,4)-code over GF(4) which is known as the *hexacode*. Its generator matrix is given by

$$G = \begin{bmatrix} 1 & X & X & 1 & 0 & 0 \\ X & 1 & X & 0 & 1 & 0 \\ X & X & 1 & 0 & 0 & 1 \end{bmatrix} \ . \tag{30}$$

This code has length 6, dimension 3 and minimum distance 4. The minimum distance property can be easily verified by exhaustively listing all 64 codewords.

We now describe the construction of the Lane message expansion. The input chaining value H and the message block $M = M^h \,||\, M^l$ are mapped to elements of GF(4) as follows, where $0 \le i < n/2$:

$$\begin{array}{lcl} \eta_0^i & \leftrightarrow & (H)_i \cdot X + (H)_{i+n/2} \\ \eta_1^i & \leftrightarrow & (M^h)_i \cdot X + (M^h)_{i+n/2} \\ \eta_2^i & \leftrightarrow & (M^l)_i \cdot X + (M^l)_{i+n/2} \end{array} \ . \tag{31}$$

Here, $(H)_i$, $(M^h)_i$ and $(M^l)_i$ denote the i-th bit of H, M^h and M^l, respectively, where bit 0 is the most significant bit. Now, for each i, encode $\left[\ \eta_0^i \quad \eta_1^i \quad \eta_2^i\ \right]$ using the linear code described in (30):

$$\left[\ \mu_0^i \quad \mu_1^i \quad \mu_2^i \quad \mu_3^i \quad \mu_4^i \quad \mu_5^i\ \right] = \left[\ \eta_0^i \quad \eta_1^i \quad \eta_2^i\ \right] \cdot G \ . \tag{32}$$

Finally, the elements of GF(4) $\mu_0^i,\ldots,\mu_5^i$ are mapped back to the n-bit expanded message words $W_0,\ldots,W_5$ in the same way:

$$\begin{array}{lcl} \mu_0^i & \leftrightarrow & (W_0)_i \cdot X + (W_0)_{i+n/2} \\ & \vdots & \\ \mu_5^i & \leftrightarrow & (W_5)_i \cdot X + (W_5)_{i+n/2} \end{array} \ . \tag{33}$$

This entire procedure can be written as a simple partitioned matrix multiplication over GF(2), where I denotes the $n/2 \times n/2$ unity matrix:

$$[W_0 \,||\, W_1 \,||\, \dots \,||\, W_5] = \left[H \,||\, M^h \,||\, M^l\right] \cdot \begin{bmatrix} I\,0\,I\,I\,I\,I\,I\,0\,0\,0\,0\,0 \\ 0\,I\,I\,0\,I\,0\,0\,I\,0\,0\,0\,0 \\ I\,I\,I\,0\,I\,I\,0\,0\,I\,0\,0\,0 \\ I\,0\,0\,I\,I\,0\,0\,0\,0\,I\,0\,0 \\ I\,I\,I\,I\,I\,0\,0\,0\,0\,0\,I\,0 \\ I\,0\,I\,0\,0\,I\,0\,0\,0\,0\,0\,I \end{bmatrix} . \tag{34}$$

It is easy to see that this is equivalent to (20), which was used to define the message expansion in Sect. 2.5.1.

Minimum distance and active lanes. The LANE message expansion can thus be seen as a parallel application of a linear (6,3,4)-code over GF(4) to $n/2$ 'slices'. The values in each such slice must form a valid codeword. Note that the six elements of GF(4) that comprise a codeword are each input to a different first-layer lane.

Consider two different inputs to the message expansion, which yield two different sets of expanded message words, $\langle W_0, \cdots, W_5 \rangle$ and $\langle W'_0, \cdots, W'_5 \rangle$. In differential cryptanalysis terminology, we say that there is at least one *active* 'slice', i.e., at least one 'slice' has a difference.

As the minimum distance of the message expansion code is four, the Hamming distance between the two codewords in any *active* 'slice' must be at least four. This implies that at least four expanded words must have a difference. Hence, in a differential attack, there are always at least four *active* lanes. This property always holds, even when the difference is only in the chaining value.

4.2.2 Active S-boxes per Lane

Next we determine the minimum number of active S-boxes in an active lane. Each active S-box decreases the probability by a factor of at least 2^6. The input to one lane can be seen as two AES states for LANE-256, or four for LANE-512. If these states were processed independently by six, resp. eight rounds of the AES, then the minimum number of active S-boxes in one lane would be 30 resp. 50. This is the minimum for six resp. eight rounds of the AES block cipher, and could thus be achieved by only activating a single AES state.

But as the SwapColumns operation, see Sect. 2.3.6, mixes the AES states together, the minimum number of active S-boxes is in fact higher. There appears to be no elegant formula to compute the minimum number of active S-boxes. A computer search for LANE-256 and LANE-512 resulted in the lower bounds given in Table 8. The figures for 7 and 8 rounds in LANE-256 are of theoretical interest only, since we use 6 rounds only.

Table 8 – Lower bounds on the number of active S-boxes in one lane for Lane-256 and Lane-512.

Rounds	Lane-256	Lane-512
1	1	1
2	5	5
3	9	9
4	25	25
5	34	41
6	45	60
7	(52)	64
8	(65)	80

4.2.3 Breaking Reduced Versions

We describe a structure for a hypothetical differential collision attack that targets a collision at the XOR combiners after the first layer and hence will apply to Lane up to a certain number of rounds per first-layer permutations P_i, but with an arbitrary number of rounds per second-layer permutations Q_j. It is intended to demonstrate Fact. 1, i.e., that the absence of a secret key in a hash function, allows an attacker to reduce the complexity of a differential attack.

Let $\Delta = (\Delta_0 \,||\, \Delta_1)$ be any n-bit difference, where n is the digest length. Introducing the difference $(0, \Delta, \Delta)$ at the inputs (H, M^h, M^l) of the Lane compression function yields the differences $(0, \Delta', \Delta', 0, \Delta, \Delta)$ at the inputs to the six lanes, where $\Delta' = (\Delta_0 \oplus \Delta_1) \,||\, \Delta_0$. Given two suitable single-lane characteristics C_0, C_1 transforming Δ' into Δ'_o with probability p and Δ into Δ_o with probability q, the differences at the two XOR's after the first layer cancel, causing the output difference of the compression function to be zero, see Figure 13. Not that this can be extend to a larger number of compatible characteristics, i.e., characteristics that have the same input and output difference, but differ in the intermediate differences. In this more general case, the probabilities of all such characteristics should be added together. For the sake of simplicity, we will describe the case where only a single characteristic is used.

A randomly chosen pair of compression function inputs simultaneously follows all four active parallel branches of such a differential with probability $p^2 \cdot q^2$. Assume for now that this probability is large enough for a right pair to exist amongst the set of all possible inputs (see Sect. 4.2.4 for further discussion on the satisfiability of differential characteristics in Lane). Fact 1 then suggests that the complexity $p^{-2}q^{-2}$ of finding the right pair could be reduced by imposing control over the intermediate state of some lanes.

More specifically, any choice of inputs to some three lanes determines a valid input to the message expansion. For instance, the attacker can fix an intermediate state in lanes P_4 and P_5 and, calculating backwards and forwards, find M^h and

M^l in such a way that the layer of S-boxes at the round where the attacker fixes the intermediate state is passed with probability 1. If the round with the greatest number of active S-boxes within the characteristic is chosen as the starting point, this can considerably reduce the complexity of finding a right pair. Also note that the attacker can deal with P_4 and P_5 independently. For fixed M^h, M^l, P_2 becomes invertible with respect to the chaining value H, allowing exactly the same approach for finding a right pair for this lane. If this procedure for P_2, P_4, P_5 is repeated k times, and k right pairs are found for each of these three lanes, the attacker can generate k^3 input pairs by forming all combinations of the independently obtained values for H, M^h and M^l. As soon as k^3 exceeds p^{-1}, one can expect to find a right pair for P_1, which is then simultaneously a right pair for all four active lanes by construction.

As outlined in section 4.2.2, the probability of a single-lane characteristic is upper bounded by 2^{-6a}, where a is the minimum number of active S-boxes for the number of rounds per lane. During the process of finding a right pair for P_4, P_5 and P_2, a certain number of active S-boxes can be disregarded. However, the attacker has no further control over the input to the fourth active lane P_1, implying that the complexity of this attack is lower bounded by the expected complexity 2^{6a} of finding a right pair for P_1 amongst the set of $k^3 \geq 2^{6a}$ available inputs.

For the six rounds employed in LANE-256, the complexity of such an attack is at least $2^{6 \cdot 45} = 2^{270}$; for LANE-512 with eight rounds, the work factor is at least $2^{6 \cdot 62} = 2^{372}$. Both values are well above the respective birthday bounds of 2^{128} and 2^{256}.

This attack, however, breaks LANE variants faster than a standard birthday approach in case of up to 3 rounds per P-lane for the digest size $n = 256$ bits and up to 5 rounds per P-lane for the 512-bit digest version.

4.2.4 Maximum Probability of a Trail

In Section 4.2.3, we described an efficient method to determine right pairs for characteristics over reduced versions of LANE. This method works, *provided that such right pairs exist.* In this section, we show that for the full versions of LANE, the overwhelming majority of the characteristics does not exhibit a right pair.

We adopt the usual hypotheses of independence and stochastic to bound the probability of characteristics equivalence [34]. The message expansion ensures that there are always at least 4 active lanes, see Sect. 4.2.1. Each active lane has at least 45 active S-boxes for LANE-256, or at least 80 active S-boxes for LANE-512. Each active S-box has probability at most 2^{-6}. This results in the following bounds for the probability of a characteristic Q:

$$\text{LANE-256:} \qquad \Pr(Q) \leq \left(2^{-6}\right)^{4 \cdot 45} = 2^{-1080} \ , \tag{35}$$

$$\text{LANE-512:} \qquad \Pr(Q) \leq \left(2^{-6}\right)^{4 \cdot 80} = 2^{-1920} \ . \tag{36}$$

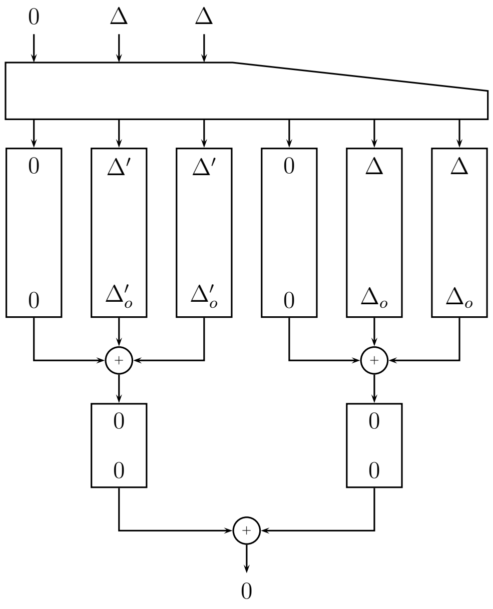


Figure 13 – A collision differential for LANE.

Since the LANE-256 and LANE-512 compression functions take only $512 + 256 + 64$, respectively $1024 + 512 + 64$, bits as input, the probability that, for a given characteristic Q, a right pair *exists*, is upper bounded by

$$\text{LANE-256:} \qquad 2^{-1080} \cdot 2^{832} = 2^{-248} \ , \tag{37}$$

$$\text{LANE-512:} \qquad 2^{-1488} \cdot 2^{1600} = 2^{-320} \ . \tag{38}$$

This suggests that for the full versions of LANE-256 and LANE-512, even with the best possible message modification techniques, it is not feasible to find a right pair, simply because with very high probability, there exists no right pair.

4.3 Truncated Differential Cryptanalysis

In the previous section, we have analysed the probability that a pair of message blocks with a fixed difference follows a single predefined characteristic. However, due to the fact that the operations used in LANE are all byte-oriented, we are likely to find a very large number of different characteristics with a comparable probability and the same or similar input and output differences. In a typical attack, each of these characteristics will be equally useful, and hence it makes sense to analyse the probability that a message pair satisfies any of them. Truncated differential cryptanalysis [31] is a technique which does exactly that.

4.3.1 Truncated Differentials

Instead of imposing specific differences in every round, a truncated differential only specifies where these differences should be zero. An example of a truncated differential for a single lane of LANE-256 is shown in Fig. 14(a). For each round, except the last one, the figure depicts the differences in the AES states before the ShiftRows transformation, after the ShiftRows transformation, and after the MixColumns transformation. Byte positions where differences are allowed are marked in grey. Since byte-equalities are preserved by all operations, except for the MixColumns transformation, this is the only stage where a reduction in probability can take place. In our example, we end up with a total probability of 2^{-96}. Note that the MixColumns transformation in the last round can be moved behind the XOR's which combine the lanes, and is therefore irrelevant if we intend to force collisions in those XOR's. This is why the last round is omitted.

4.3.2 Identifying the Optimal Truncated Differential

The probability of a truncated differential is, on its own, not a very good measure for its usefulness. As a trivial example, consider a truncated differential without any zero differences, which, despite its probability of 1, is clearly useless. To be of any use, a truncated differential should demonstrate that a pair satisfying the input difference is significantly more likely to result in the desired output difference than a random pair. In Fig. 14(a), for instance, a consistent input pair is expected

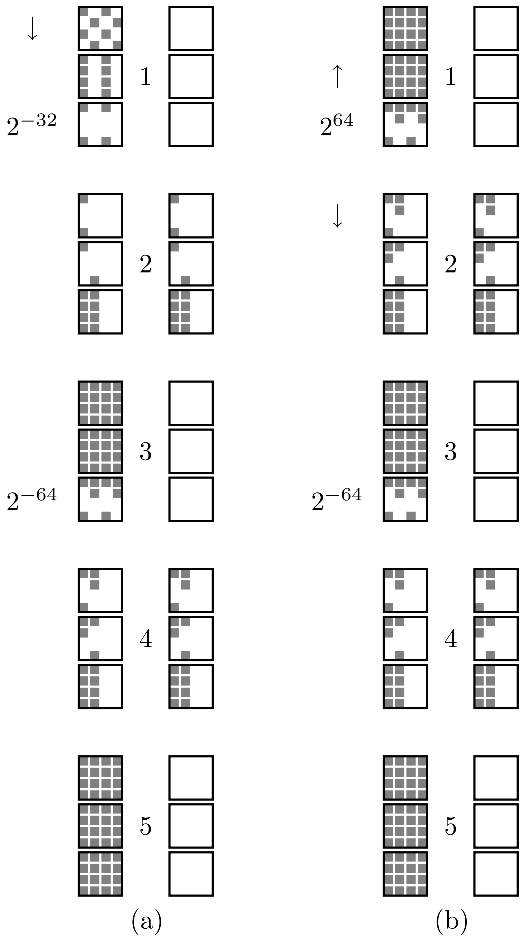


Figure 14 – Truncated differentials in one lane of LANE-256.

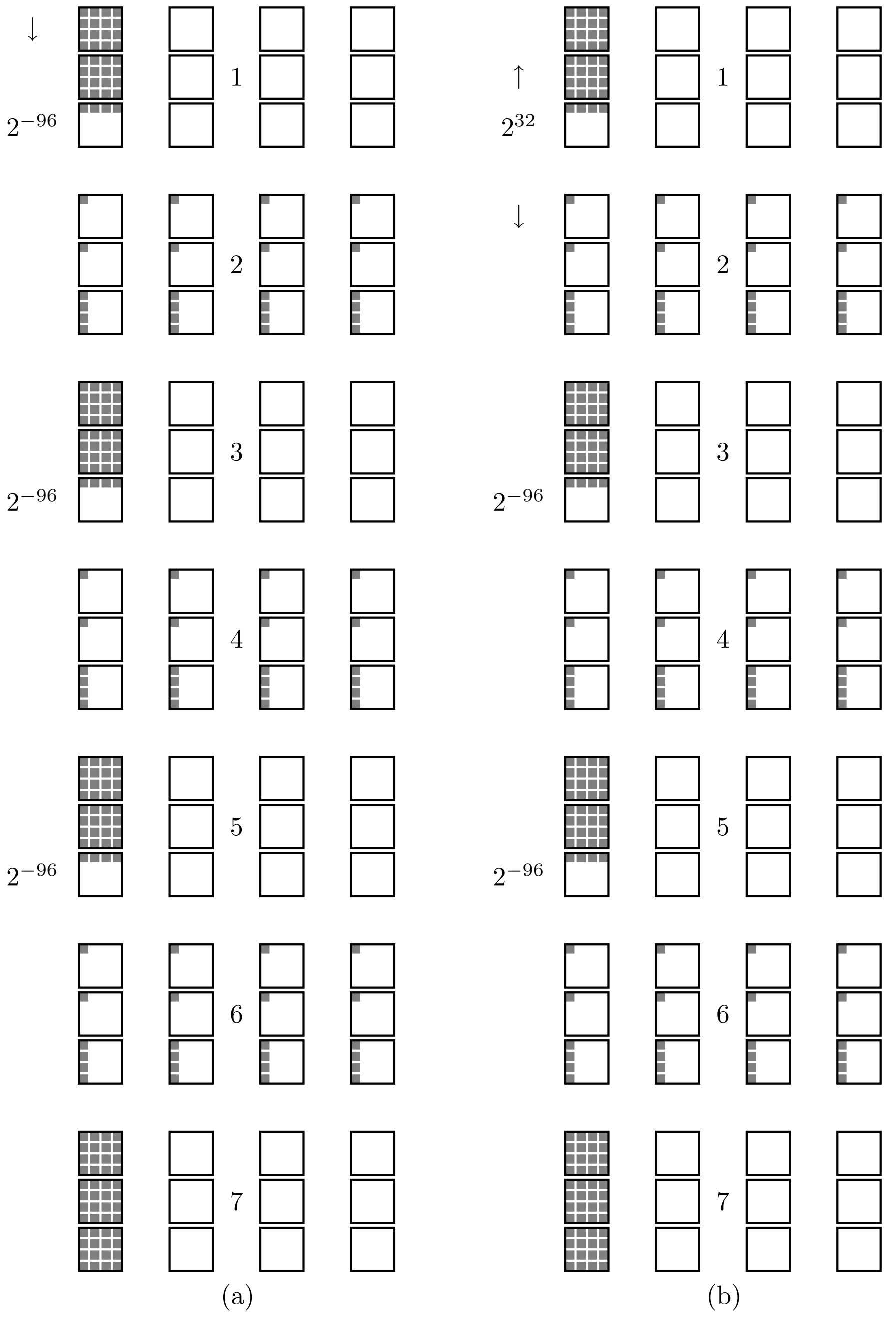


Figure 15 – Truncated differentials in one lane of LANE-512.

to result in 16 equal bytes at the output with a probability of 2^{-96}, versus 2^{-128} for a random pair. In fact, it can be shown by computer search that this factor of 2^{32} is the highest attainable gain for a single lane of LANE-256.

Another property that influences the usefulness of a truncated differential is the number of degrees of freedom that are left at the input. Without any additional restrictions, these degrees of freedom could be used to reduce the effort of finding right pairs. Consider for instance the set of all 2^{64} input states which are constant (say, zero) in all but the 8 grey bytes at the input of Fig. 14(a). By sorting the 2^{64} corresponding output states and returning all pairs which have equal values in the rightmost AES state, we would in effect have checked in the order of 2^{128} pairs. Hence, we expect to find about 2^{32} right ones, and this with an effort of only 2^{64}.

An additional optimisation which can be applied if the attacker is free to choose parts of the input state is depicted in Fig. 14(b). The approach is similar to the one described above, but this time the attacker starts after the first MixColumns, which saves a factor 2^{32} in probability, and therefore results in 2^{64} right pairs. For each of these pairs the attacker then reverses the first round in order to find the corresponding input pairs, each of which will be equal in the rightmost AES state.

The same reasoning can be applied to the two truncated differentials of LANE-512 shown in Fig. 15. However, if we start with a set of 2^{128} input states in Fig. 15(a), then we do not expect to find any right pair at all, since

$$2^{2\cdot 128} \cdot 2^{-3\cdot 96} = 2^{-32}\,.$$

In order to find a single right pair, we will therefore have to repeat this procedure 2^{32} times, resulting in a total workload of 2^{160}. Alternatively, we could start with a set of 2^{32} states after the first MixColumns, as shown in Fig. 15(b). This will have to be repeated 2^{128} times in order to compensate for the fact that

$$2^{2\cdot 32} \cdot 2^{-2\cdot 96} = 2^{-128}\,,$$

leading to the same total effort of 2^{160} as before. Note however that the first approach requires 2^{128} of memory, whereas a table of 2^{32} would suffice for the second one.

4.3.3 Using Truncated Differentials for Collision Searching

An attack using truncated differentials to construct a collision pair would then proceed as follows. First, we ensure that only 4 lanes are active, which is optimal as shown in Sect. 4.2.1. This can be achieved by choosing $\Delta m_0 = \Delta m_2$. Then, the probability that a colliding right pair exists for a given chaining input is given by

$$\text{LANE-256:} \qquad 2^{64} \cdot 2^{512} \cdot \left[\left(2^{-96}\right)^2 \cdot 2^{-128}\right]^2 = 2^{-64}\,,$$

$$\text{LANE-512:} \qquad 2^{128} \cdot 2^{1024} \cdot \left[\left(2^{-3\cdot 96}\right)^2 \cdot 2^{-128}\right]^2 = 2^{-256}\,.$$

Note that, especially for LANE-256, even if such a pair exists, we do not expect to be able to determine this with an effort less than 2^{64}, after which the pair would need to be recovered in less than 2^{128}.

4.4 Higher Order Differential Cryptanalysis

Higher order differentials where introduced as an extension of differential cryptanalysis, using higher order derivatives [31]. The i'th order derivative of a function f at the point $a_1, \ldots, a_i$ is defined as follows [33]:

$$\begin{aligned} \Delta_a f(x) &:= f(x+a) - f(x) \\ \Delta^{(i)}_{a_1,\ldots,a_i} f(x) &:= \Delta_{a_i}\left(\Delta^{(i-1)}_{a_1,\ldots,a_{i-1}} f(x)\right), \quad i > 1. \end{aligned}$$

Standard differential cryptanalysis used first-order derivatives.

The following property of the LANE message expansion is important.

Property 1. *Let $(W)_{ti}$, $0 \le t < 5$, $0 \le i < n$, denote the $6n$ output bits of the message expansion, and let $(H)_i$, $(M^h)_i$, $(M^l)_i$, $0 \le i < n$ denote the $3n$ input bits. Then for each t there exists a set of 12 binary constants a_{0t}, a_{1t}, a_{2t}, a_{3t}, a_{4t}, a_{5t}, b_{0t}, b_{1t}, b_{2t}, b_{3t}, b_{4t}, b_{5t} such that*

$$(W)_{ti} = \begin{cases} a_{0t}(H)_i + a_{1t}(M^h)_i + a_{2t}(M^l)_i \\ \quad + a_{3t}(H)_{i+n/2} + a_{4t}(M^h)_{i+n/2} + a_{5t}(M^l)_{i+n/2}, & \text{for } 0 \le i < n/2; \\ b_{0t}(H)_{i-n/2} + b_{1t}(M^h)_{i-n/2} + b_{2t}(M^l)_{i-n/2} \\ \quad + b_{3t}(H)_i + b_{4t}(M^h)_i + b_{5t}(M^l)_i, & \text{for } n/2 \le i < n. \end{cases}$$

4.4.1 A Fourth-Order Differential Distinguisher

Property 1 implies the existence of 4th order differentials that have probability 1 over the message expansion and the first layer of lanes.

Corollary 1. *Let $F(H, M^h, M^l)$ denote the function that consists of the message expansion, the first layer of lanes and the two XOR combiners following it. Let δ be an arbitrary $(n/2)$-bit difference and let*

$$\begin{aligned} a_0 &= (0,0;0,0;0,\delta) \\ b_0 &= (0,0;0,0;\delta,0) \\ a_1 &= (0,0;0,\delta;0,0) \\ b_1 &= (0,0;\delta,0;0,0) \\ a_2 &= (0,\delta;0,0;0,0) \\ b_2 &= (\delta,0;0,0;0,0). \end{aligned}$$

Then

$$\Delta_{a_i,b_i,a_j,b_j} F(H, M^h, M^l) = 0 \textit{ with probability 1,}$$

for all $i, j \in \{0, 1, 2\}$.

Proof. Property 1 implies that over the 16 inputs defined by the 4th order differential, each lane input W_t will take 1, 2 or 4 different values, and each value a multiple of 4 times. Consequently, each lane output will occur an even number of times. Hence for each of the XOR combiners it holds that the XOR of the 16 outputs equals zero. □

We could not determine a way to extend this property through the second layer of permutations.

4.4.2 Square Attacks on the Compression Function

The structure of Lane is byte oriented, suggesting a possible square attack might be applicable [18,32,36]. The attack is made slightly more complex by the message expansion and the need to track the square property over six lanes, but it is essentially the same.

For example, in Lane-256, consider 256 message blocks, for which one byte of m_0 (e.g., the first one) accepts all possible values, while all the other bytes are set to the same value. This ensures that the first byte in both halves of W_0 is active (i.e., accepts all values), the first byte of the left half of W_1 is active, as well as the first byte of both halves of W_2, and the left of W_4.

In lanes where there is only one active byte, we have after two full rounds sixteen bytes which are active (columns 0,1,4,5). Looking at the third round, we learn that all bytes are balanced (i.e., the sum of all values in these bytes is zero). In lanes where there are two active bytes (one in each half), after the second round all the bytes are active, and after three rounds, all bytes are balanced. Thus, if we reduce the length of the P_i permutations to three rounds, we know that the XOR of the outputs of these permutations is balanced. Without the Q_j permutations, we could at this point find that all the bytes are balanced. Thus, given the output values of 255 messages out of the set, we could predict the output of the missing message. This could also be used as a distinguisher to distinguish the output of the compression function from random.

We can extend the attack by one round, by considering structures of 2^{32} messages, chosen such that after one round they generate 2^{24} sets of 256 messages each, where in each such set, the first byte of the state (or the first byte of each 128-bit half of the state) obtains all possible values, while the other bytes are fixed. Similarly, we can extend the square property one more round, by taking structures of 2^{128} values. A possible structure would use

$$m_0 = m_1 = m_2 = m_3 = i \quad \text{for all} \quad 0 \leq i < 2^{128} \ . \qquad (39)$$

We conclude that the best square property of Lane-256 is of 5 rounds, and thus, after the additional round, the inputs to the Q_j's permutations have no structural property. And even if somehow the attacker succeeded in finding such a structure after 6 rounds, the Q_j's would destroy the remaining structural properties.

For Lane-512 the analysis is similar, but with an extra round. Starting from only one active byte, after four rounds we expect that all the bytes of the internal state are balanced. Hence, for Lane-512 the best square property is for 6 rounds, making Lane-512 immune to this attack as well.

4.4.3 Multiset Distinguishers

As a further generalisation, one can consider distinguishers based on (much) larger sets of inputs. As an example, consider the following case. Keep h_0, h_1, m_0 and m_1 constant and saturate m_2 and m_3, i.e. assign every possible value to them. The size of this set of messages is $2^{n/2}$. Then, we know that every expanded message word also gets assigned every possible value, except for W_3 and W_4, which are constant. Now, after the first layer, the outputs of P_0, P_1, P_2 and P_5 are saturated. After the XOR combiners, we see that the input to Q_0 sums to zero, i.e. it is balanced, and the input to Q_1 is still saturated. This implies that also the output of Q_1 is saturated, but nothing useful can be said about the output of Q_0 or the output of the compression function.

4.5 Cryptanalysis of Wide-Block Rijndael

Nakahara et al. investigated the security of wide-block Rijndael [43, 44]. Since a permutation in Lane has certain similarities with Rijndael-256 we summarise these attacks. Recall that a full permutation round in Lane consists of the following sequence of transformations: SubBytes, ShiftRows, MixColumns, AddConstants, AddCounter and SwapColumns, while a round in wide (large block) Rijndael consists of AddRoundKey, SubBytes, ShiftRows and MixColumns. Note that ShiftRows differs in Rijndael-128 (i.e. AES and Lane) and Rijndael-256. In fact, the combination of the SwapColumns and ShiftRows operations in Lane-256 can be viewed as the equivalent of the (redefined) ShiftRows operation in Rijndael-256.

There exist higher-order multiset (differential and linear) distinguishers for up to 7 rounds of Rijndael-256 [43]. These distinguishers trace the status of 128-bit words, and thus require sets of 2^{128} chosen plaintexts at a time. The rationale behind the multiset technique is to use balanced sets of bits to attack permutation mappings (cipher rounds).

Similar distinguishers can be constructed for the first layer of Lane, see for example Sect. 4.4.3. However, the second layer of permutations completely stops these distinguishers.

Impossible-differential (ID) attacks on 7-round Rijndael-256 are shown in [44]. Typical ID distinguishers follow the miss-in-the-middle technique. Two truncated differentials, one in the encryption direction and one in the decryption direction, are combined to form an impossible truncated differential. A key recovery attack can be built upon an ID distinguisher, as subkey guesses for which the impossible differential would be followed, can be eliminated with certainty.

In order to construct a distinguisher for the Lane compression function based on impossible differentials, it is not sufficient to have an impossible differential for a single Lane permutation. Assuming that the unknown keying material enters the compression function via the chaining value, four lanes in the first layer will be affected. Hence, an impossible differential needs to cover these four lanes in the first layer, as well as both lanes in the second layer. It seems very unlikely that such an impossible differential can be found.

4.6 Algebraic Attacks

In algebraic attacks, the operation of a symmetric cryptographic primitive is represented as a system of polynomial equations over $GF(2)$ or $GF(2^n)$, which is then attempted to be solved using various techniques and expression manipulations. Since Lane is based on the AES, its security with regard to algebraic attacks is closely related to that of the AES. As a single state of Lane encompasses multiple AES states, the resulting equation systems for Lane are expected to have comparable degree, but higher dimension.

There has been an extensive analysis of the equation systems corresponding to the AES, however, all techniques have so far only been successful against very small scaled variants. The approaches that are theoretically best understood are methods based on Gröbner bases [5]. Improving over Buchberger's classical algorithm [12], Faugère's F4 and F5 algorithms [23,24] are the best known methods to compute Gröbner bases. Extensive experiments indicate that those algorithms are only successful for very small AES variants, such as ten rounds of an 1×1 state or four rounds of a 2×1 state [14].

An alternative approach to solving nonlinear polynomial equations is to linearise the system by introducing new independent variables for each occurring nonlinear monomial term. Since this method can only be effective if the number of linearly independent polynomials approximately equals the number of monomials, the Extended Linearisation (XL) algorithm [16] extends the original equation system before linearisation by multiplying it with all monomials up to some degree in order to generate enough linearly independent equations. Experimental evidence indicates that the XL algorithm offers little to no advantage compared to Gröbner basis techniques [4].

Finally, the Extended Sparse Linearisation (XSL) method [17] aims at improving on the XL technique by multiplying the polynomials only by products of monomials that occur in the original system. So far, also this method has been unsuccessful in realistically-sized AES equation systems [14].

We conclude that it seems highly unlikely that algebraic attacks can be successfully applied to Lane.

4.7 Attacks Based on Reduced Query Complexity

4.7.1 General Comments

Since LANE is a permutation-based hash function, it can be studied in the *ideal permutation model* [53], which is very similar to the ideal cipher model and the random oracle model. Theorem 1 of [53] states that for a compression function $f : \{0,1\}^{sn} \rightarrow \{0,1\}^{rn}$ using k calls to n-bit permutations, collisions can be found with certainty using approximately

$$k \cdot 2^{n(1-(s-r)/k)} \tag{40}$$

permutation queries (at most). Instantiating this with the parameters for LANE ($s = 3, r = 1, k = 8$) yields *query* complexities of 2^{195} and 2^{387} for LANE-256 and LANE-512, respectively.

An interesting discussion of the merits and limitations of the ideal/random models can be found in [25]. An important observation is that there are two ways to measure the complexity of an attack. On the one hand, there is the *practical complexity*, which measures the (expected value of the) time complexity of the adversary. This is the most natural complexity measure and also the most relevant measure. On the other hand, there is the *query complexity*, which measures the number of queries that are made to the oracle. This complexity is often used in security proofs, mostly because it is easier to bound.

Since the practical complexity of an adversary is always larger than its query complexity, the ideal oracle model can be used to prove bounds on the security of hash function designs.

There are two important criticisms on this model. Firstly, the distinction between oracle queries and computations made by the adversary is artificial. A cryptographic hash function uses an instantiation of the permutation (block cipher), which is public. Hence, it can be argued that any cryptographic hash function can be broken without making a single query to the oracle. Secondly, the model ignores the practical complexity of the adversary. It is well-known that an information theoretic adversary who is given a full description of a hash function can always find collisions and preimages. On the other hand, returning to the case of hash function with the same dimensions as LANE-256 or LANE-512, the adversary has to find the actual collision in sets of at least 2^{256} or 2^{512} values, so that an acceleration in constructing these sets, such as the one given by (40), does not reduce the practical complexity of the attack.

Summarising, results on the query complexity of attacks on hash function designs do not always have a big impact on the actual security of the design. Nevertheless, they can be first steps in the development of better attacks. Therefore, we list here our results.

4.7.2 Results on Lane

The message expansion of LANE expands three inputs to six outputs which are then independently fed into the permutations of the first layer. We call a particular combination of chaining value and message block an *input* to the message expansion. If some value occurs more than once at a permutation input when hashing a set of messages, the corresponding permutation output has to be computed only once. More precisely, whenever the sum of the numbers of distinct values at each of the six outputs is lower than the number of different message expansion inputs, the output of the first layer can be computed with reduced effort, resulting in some speedup of the evaluation of the entire compression function for this set of inputs.

Property 1 leads to two corollaries which can be used to reduce the query complexity of LANE adversaries. The first corollary looks at LANE without the second layer of lanes.

Corollary 2. *It is possible to compute the inputs of the second layer of lanes for* 2^{6p} *different inputs to the compression function, using only* $6 \cdot 2^{2p}$ *queries to the* P*-lanes (exactly* 2^{2p} *queries to each of the 6* P*-lanes).*

Proof. Choose p indices j_t with $0 \leq j_t < n/2$. Consider the 2^{6p} inputs where the bits h_{j_t}, $h_{j_t+n/2}$, m_{j_t}, $m_{j_t+n/2}$, $m^*_{j_t}$, $m^*_{j_t+n/2}$ take all possible values and the remaining bits are constant. Property 1 implies that the words w_t will differ in the bits at the $2p$ positions $j_t, j_t + n/2$ only. Hence each lane needs to be queried for at most 2^{2p} different values. □

Adding the queries to the lanes of the second layer, and assuming that the lanes in the second layer are twice as fast as the lanes of the first layer, we obtain an acceleration factor given by

$$\frac{7 \cdot 2^{6p}}{6 \cdot 2^{2p} + 2^{6p}} \,, \tag{41}$$

which very rapidly converges to 7 as p approaches infinity. This simply means that hashing many messages chosen in this way can be done up to 7 times faster than the straightforward approach. This is not considered to be an issue, as it is merely a constant factor. Similar optimisations to speed up the hashing of many messages can be applied to virtually any hash function.

The following corollary will improve upon this number.

Corollary 3. *It is possible to compute the outputs of the compression function for* 2^{10p-2n} *different inputs, using only* 2^{2p+3} *queries to the lanes.*

Proof. We start by applying Property 1 twice. First, we apply it on the first layer, with p varying bits. We compute and store the 2^{6p} outputs of the first layer in list L_1. Next, we apply it on the second layer, with q varying bits. We store the 2^{4q} inputs for which we can compute the output in list L_2.

We have then made $6 \cdot 2^{2p}$ queries to the lanes of the first layer, and $2 \cdot 2^{2q}$ queries to the lanes of the second layer. The number of inputs for which we can compute the output of the compression function, equals the number of entries that appear in both lists. This number can be approximated by $2^{6p+4q-2n}$.

If we choose $p = q > (2n+3)/8$ then

$$\#\text{queries} = 2^{2p+3} = 2^{10p-(8p-3)} < 2^{10p-2n} = \#\text{outputs} \tag{42}$$

□

For $n = 256$, the number of queries drops below the number of outputs when $p = q = 65$. For $n = 512$, this happens when $p = q = 129$.

We are not aware of any method to exploit these properties to reduce the practical complexity of any attack against Lane.

4.7.3 Bounds for Query Complexity

A Lower Bound for the Query Complexity of Lane. Consider any message expansion mapping three inputs to six outputs, with the only requirement to have a minimum distance greater than half the number of lanes in order to prevent the meet-in-the-middle attack outlined in section 4.9. Assume that N different values (each comprising chaining value and message block) are input into the message expansion and denote the number of distinct values that occur at each of the six outputs by $L_0, \ldots, L_5$. The outputs of the first layer of parallel lanes for the entire set of N inputs can then be computed with $\sum_{i=0}^{5} L_i$ permutation queries.

A minimum distance of four implies that any mapping from the input to some three outputs is invertible. This in turn shows that for each $i \neq j \neq k$, the product $L_i L_j L_k$ must be at least equal to N. As the latter holds for any three outputs, we also know that $\left(\frac{1}{6}\sum_{i=0}^{5} L_i\right)^3 \geq N$, and hence

$$\sum_{i=0}^{5} L_i \geq 6\sqrt[3]{N}. \tag{43}$$

Therefore, the number of permutation queries needed to compute the first layer of permutations for N inputs is lower bounded by $6\sqrt[3]{N}$, independent of the linearity of the message expansion (only imposing the minimum distance requirement). This lower bound is tight for the message expansion of Lane and we conclude that the query strategy of Corollary 2 with $L_i = 2^{2p}$ and $N = 2^{6p}$ is in fact optimal.

Alternative linear message expansions. Finally, we discuss the relative merits of alternative linear message expansions. For the sake of clarity, we restrict the treatment to Lane-256 (and hence Lane-224). The wider variants can be handled completely analogously by considering $GF(2^{512})$ instead of $GF(2^{256})$.

In order to construct a message expansion that does not exhibit Property 1, a linear $(6, 3, 4)$ code over $GF(2^{256})$ could be used. The systematic form of the generator matrix of such a code would be

$$\begin{bmatrix} 1 & 0 & 0 & \alpha & \beta & \gamma \\ 0 & 1 & 0 & \delta & \varepsilon & \zeta \\ 0 & 0 & 1 & \eta & \theta & \iota \end{bmatrix}, \tag{44}$$

where the Greek letters denote elements of $GF(2^{256})$. Since the attacker can apply invertible transformations at both input and output, this generator matrix can always be transformed to:

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & \alpha' & \beta' \\ 0 & 0 & 1 & 1 & \gamma' & \delta' \end{bmatrix}. \tag{45}$$

Over $GF(2)$, this is a 768×1536 matrix. We show now that a weakened form of Property 1 holds also in this case.

We start by choosing an arbitrary $2p$-dimensional vector space V $(2p < n)$ and require that the inputs of each lane are in this vector space. This imposes conditions on the inputs of the message expansion. The requirements on the first 3 lanes are equivalent to: $H, M^h, M^l \in V$. This ensures that $H + M^h + M^l \in V$, which implies that also the condition on the input of the fourth lane is satisfied. The requirements on the last two lanes restrict the number of permissible M^h- and M^l-values to smaller vector spaces. We denote the dimensions of these vector spaces by $k_1, k_2 < 2p$. In this scenario, the output of the first layer for $2^{2p+k_1+k_2}$ inputs can be computed with $6 \cdot 2^{2p}$ permutation queries.

A sufficient condition on $2p$ to have both $k_1 > 0$ and $k_2 > 0$ can be obtained as follows. The conditions on the last two lanes are $H + \alpha' M^h + \gamma' M^l \in V$ and $H + \beta' M^h + \delta' M^l \in V$, respectively. If we require that, besides $M^h, M^l \in V$, also the products $\alpha' M^h, \beta' M^h, \gamma' M^l$ and $\delta' M^l$ are elements of V, then both conditions trivially hold. The three conditions on M^h form a set of $3 \cdot (n - 2p)$ equations in n unknowns, so $k_1 > 0$ if $n - 3 \cdot (n - 2p) > 0$. Hence, $2p > \frac{2}{3}n$ is a sufficient condition for having $k_1 > 0$. Analogously, $2p > \frac{2}{3}n$ implies $k_2 > 0$. In case of a linear message expansion for LANE-256, this corresponds to $6 \cdot 2^{170.7}$ permutation queries.

4.8 Wagner's Generalised Birthday Attack

Wagner [58] describes a sub-exponential algorithm for the generalised birthday problem where one is given k lists of n-bit values and wants to select one value from each list such that the selected values sum to zero. For $k = 2$ and XOR as summation operation, this is the well-known birthday problem. Provided that the lists are long enough and the list elements are independently and uniformly selected at random, it can be solved with good probability in $\mathcal{O}(2^{n/2})$ time. Under the same assumptions, Wagner's generalised algorithm has a running time of $\mathcal{O}(k \cdot 2^{n/(1+\lceil \log_2 k \rceil)})$, which in particular implies that the birthday problem with four lists can be solved in $\mathcal{O}(2^{n/3})$ time.

Lane has one XOR combiner with two inputs and two XOR combiners involving 3 inputs each. For the 3-sum problem, no algorithm faster than the birthday complexity is known. Since Wagner's algorithm assumes that k is a power of two, the best we can do is to emulate the case $k = 2$ by solving the problem $x_0 \oplus x_1 = c$, where c is a fixed, randomly selected value from the third list. Moreover, the assumption that the list entries are independent is invalid for Lane, since are linearly dependent according to the message expansion. Indeed, even though h, m, m^* can be chosen independently for the second 3-XOR, their choice immediately fixes the inputs for the first 3-XOR.

Consider now Lane without the message expansion, so that the independence assumption holds. Since selecting $k = 3$ yields no advantage compared to the birthday bound, an attacker can consider the XOR of all six P_i outputs. In this case, Wagner's algorithm would be applied with $k = 4$ (in $2^{n/3}$ time), searching for a solution to $x_0 \oplus x_1 \oplus x_2 \oplus x_3 = c_4 \oplus c_5$, where c_4, c_5 are random choices from the lists corresponding to P_4 and P_5. Once such a solution $x'_0, \ldots, x'_5$ is found, we know that $x'_0 \oplus \cdots \oplus x'_5 = 0$, which is equivalent to $x'_0 \oplus x'_1 \oplus x'_2 = x'_3 \oplus x'_4 \oplus x'_5$. Hence, the attacker can use this to obtain a *value* x such that $H_i = Q_0(x) \oplus Q_1(x)$ and is then left with the task of attacking a smaller number of AES-like rounds with identical input but different keys (the constants). Alternatively, he can apply the birthday attack on *differences* instead of values. In this scenario, the effect of the different constants cancels, so that any pair following this differential would immediately yield a collision for the entire compression function.

Summarising, both the message expansion and the second layer of permutations contribute to making attacks based upon Wagner's algorithm for the generalised birthday problem inapplicable to Lane.

4.9 Meet-in-the-Middle Attacks

A meet-in-the-middle attack can be used to construct collisions or (second) preimages by simultaneously modifying two consecutive message blocks. A basic version of the attack can be described as follows. In order to reach a certain target value for H_{i+1} from a given H_{i-1}, the attacker will determine an intermediate result V and define two maps g, g^* such that

$$g(H_{i-1}, M) = V = g^*(H_{i+1}, M^*), \tag{46}$$

with g, g^* efficiently computable functions and M, M^* two independent parts of the message input. Subsequently, the unknown V is eliminated from the equations and a solution for M and M^* is searched by constructing two lists. The first list contains output values for g; the second list contains output values for g^*. A match between both lists means that a message has been found which takes H_{i-1}

to H_{i+1}. For example, if the compression function $f(H, M)$ is invertible, then the adversary can choose $g = f$ and $g^* = f^{-1}$.

In the case of LANE, the compression function is not invertible, but a cryptanalyst could try to construct a similar attack within one application of the compression function. This can be done only if the adversary is able to partition the lanes into two disjunct sets, whose inputs can be restricted to be independent of the other set. The message expansion prevents this attack, as its minimum distance of 4 ensures that each of these sets need to comprise at least four lanes. Since there are only six lanes in total, it is not possible to find two non-overlapping, independent sets of lanes.

4.10 Long Message Second-Preimage Attacks

The standard Merkle-Damgård iteration of a compression function guarantees collision resistance of the overall construction if the compression function itself is collision-resistant. However, as discovered by Dean [21], this does not hold in the case of second preimages, if fixed points of the compression function can be found.

Definition 5. A *fixed point* of a compression function $f(\cdot, \cdot)$ is a chaining value h and message block m for which it holds that

$$h = f(h, m) \ . \tag{47}$$

Fixed points of a compression function can be concatenated to form an *expandable message*. This is a set of message patterns of different lengths which all lead to the same output chaining value. In a long-message second preimage attack [21,30], an expandable message allows an adversary to target simultaneously any intermediate chaining value instead of just a single one, reducing the expected work factor of a second preimage attack from $\mathcal{O}(2^n)$ to $\mathcal{O}(2^{n-k})$, where 2^k is the length of the message.

For hash functions based on the Davies-Meyer construction, it is very easy to find fixed points [42, 50]. Let $E_K(\cdot)$ be a block cipher with key K. Then the Davies-Meyer construction is

$$f(h, m) = E_m(h) + h \ . \tag{48}$$

Constructing a fixed point can be done by choosing an arbitrary message block m, and computing

$$h = E_m^{-1}(0) \ . \tag{49}$$

Now, it follows from (48) that this yields a fixed point.

For the LANE compression function, constructing fixed points in such a straightforward way does not seem to be possible. Even though the permutations in LANE are invertible, the structure of the compression function does not allow for the construction of fixed points, as the linear conditions on the expanded message words can only be satisfied probabilistically. Hence, finding a fixed point for the compression function of LANE should be no easier than constructing a preimage for the compression function.

Even more so, if a fixed point for the compression function of LANE could be found in an efficient way, it would still not allow for the construction of an expandable message. As discussed in [10], the inclusion of the counter in LANE prohibits the concatenation of fixed points. Also Kelsey and Schneier's multicollision-based method for constructing an expandable message [30] is not applicable thanks to the counter. Finally, the attacks on dithered hash functions by Andreeva et al. [2] are also foiled by the inclusion of the counter.

Thus, we conclude that the second preimage resistance of LANE does not degrade when the challenge message is long, and hence the iteration mode of LANE offers full n-bit security for second preimages (see also [1]).

4.11 Length-Extension Attacks

Given the hash value of a (partially) unknown message m (including padding etc.), length-extension attacks aim to infer the hash value of some message $m \,||\, x$, where the suffix x may be chosen freely by the adversary. This is an important consideration for message authentication codes (MAC's) based on hash functions, as a successful length-extension attack would lead to a forgery.

In the plain Merkle-Damgård construction, the mere knowledge of the message length l and the hash value $h(m)$ of a (partially) unknown message m enables an attacker to calculate the hash value of messages of the form $\mathrm{pad}_l(m) \,||\, x$, where x is an arbitrary suffix. Indeed, the intermediate chaining value after processing $\mathrm{pad}_l(m)$, which always aligns to a block boundary, is precisely equal to $h(m)$, which is known to the attacker.

In LANE, the output transformation, which can be regarded as another compression function call on a special padding block, is processed using the special counter value zero, which cannot occur in any regular message block. This makes it impossible for an attacker to emulate the output transformation using a regular message block. Therefore, length extension attacks are not applicable to LANE.

4.12 Multicollision Attacks

In a multicollision setting, the attacker wants to find $k > 2$ messages all hashing to the same value. Ideally, finding a k-way multicollision should require a computational effort of $\mathcal{O}(2^{n\cdot(k-1)/k})$. However, the multicollision attack by Joux [29] allows to find a 2^l-way multicollision for an n-bit hash function using the Merkle-Damgård iteration impressively faster than that. It requires an effort of only $l \cdot C(n)$, where $C(n)$ is the complexity of finding a single collision, which will be at most $2^{n/2}$ by the birthday paradox. Joux's attack works by concatenating a chain of internal collisions: For $i = 1, \ldots, l$, the attacker computes colliding pairs $\langle M_i, M'_i \rangle$ such that $h(H_{i-1}, M_i) = h(H_{i-1}, M'_i)$, where H_0 is the initial value and $H_i := h(H_{i-1}, M_i) = h(H_{i-1}, M'_i)$. Now, after appropriate padding, the 2^l messages $X_1 \,||\, \cdots \,||\, X_l$, where $X_i \in \{M, M'_i\}$, all hash to the same value, yielding a 2^l-way multicollision.

If the compression function allows for efficient calculation of fixed points, the method of Kelsey and Schneier [30] gives multicollisions of arbitrary size with $\mathcal{O}(2^{n/2})$ effort. As outlined in section 4.10, this improvement does not apply to LANE. Joux' approach, however, is applicable. In order to preclude it, a chaining value of at least $2n$ bits would be required.

Since Joux' attack combines single collisions, its complexity directly depends on the best possible single-collision attack against the hash function. Naturally, the effort of applying the multicollision attack can never be lower than that of finding a single collision. Since we require that it should be already computationally infeasible to construct a single collision, multicollisions do not present a bigger threat than collisions. In particular, LANE's security level against single collisions, which meets the theoretical bound of $2^{n/2}$, is not reduced in any regard by the fact that many messages colliding to the same hash value can be found with only little more effort.

One of the main application of multicollisions is the construction of long message second preimage attacks [30]. As was mentioned in Sect. 4.10, the inclusion of the counter precludes these attacks for LANE. Hence, we argue that, although Joux' multicollision attack does apply to LANE, it is not a threat in practice. Finally, it should be noted that the fact that this multicollision attack applies to LANE is a mere consequence of the mode of iteration and does not imply any weakness in the compression function itself.

4.13 On the Mode of Iteration

In this section, the reduction-based provable security approach is used to assess the security of the LANE iteration. More precisely, security claims on the LANE iteration under some concrete assumptions on the underlying compression function are stated. Following this approach, concrete security bounds on the computational complexity of an adversary against the LANE iteration can be shown. We also exhibit information theoretic results on the LANE iteration which indicate security against generic attacks under the assumption that the compression function is an ideal primitive. For a full security analysis on the LANE iteration, we refer to the work of Andreeva [1]. Here, we present a summary of the main results.

Similarly to the known Merkle-Damgård iterative principle [20, 41], Andreeva shows that both the non-salted and salted versions of the LANE iteration are provably collision secure. Following Rogaway's human-ignorance approach [52], the advantage of an adversary against the LANE hash function is related to that of another adversary against the LANE compression function to derive a tight security bound.

The upper bounds on the advantage of information theoretic adversaries against the second preimage and preimage security, respectively, of the non-salted LANE hash function indicate that 2^n evaluations of the underlying ideal compression function need to performed to break the respective security property. Moreover, making a (variant of) preimage security assumption on the output transformation of LANE and adopting some randomness extraction and regularity properties on the iterative portion of the non-salted LANE hash function, a tight preimage security bound on the LANE iteration is exhibited.

For the salted version of the LANE hash function a broad set of security notions is developed that capture most of the important attack scenarios of randomised hashing. In addition, the possibility of attacks under equal or distinct and known or secret salt values is taken into account. The same (as for the non-salted LANE hash function) security against information theoretic adversaries is obtained in the second preimage case, and the preimage case. The salted LANE iteration also provides second preimage security guarantees of order 2^n against adversaries who first commit to a target message and then are given a random target salt value.

The latter information theoretic results show that no generic attacks on the second preimage and preimage security on both salted and non-salted LANE hash function variants succeed in under 2^n number of evaluations of an ideal compression function.

Another important security feature of the LANE iterative design in both salted and non-salted versions is the security against extension attacks and lack of structural flaws ensured by the prefix-free property of the processed inputs. The latter design characteristic of LANE ensures its indifferentiability from a random oracle according to the work of Coron et al. [15] and pseudorandom function behaviour according to Bellare et al. [6].

The suggested parallel processing method, see Sect. 3.1.5 is also shown to be collision secure when the underlying hash function is collision secure. Under the second preimage/preimage security and some mild assumptions on the min entropy extraction properties of the hash functions, it is also shown that the parallel mode of operation is second preimage/preimage secure.

4.14 Expected Strength of Lane

To the best of our knowledge, the complexity of finding collisions, first, and second preimages for LANE is $\mathcal{O}(2^{n/2})$, $\mathcal{O}(2^n)$, and $\mathcal{O}(2^n)$, respectively. In all three cases, the complexities refer to generic approaches applicable to any hash function: the birthday attack for finding collisions and simple brute force for preimages. As discussed in this section, none of the dedicated attack attempts yielded a lower complexity than the generic attacks. This holds for any of the specified digest lengths $n = 224, 256, 384, 512$.

Length-extension attacks are generally precluded, as outlined in section 4.11. Our analysis also did not indicate any imbalance concerning the strength of individual output bits, so forming an n'-bit hash function by selecting $n' < n$ digest

Table 9 – Expected strength of LANE against cryptanalytic attacks.

Attack	LANE-224	LANE-256	LANE-384	LANE-512
Collision attacks	2^{112}	2^{128}	2^{192}	2^{256}
Preimage attacks	2^{224}	2^{256}	2^{384}	2^{512}
Second preimage attacks	2^{224}	2^{256}	2^{384}	2^{512}
Length-extension attacks	not applicable			
Output bits equally strong	yes			

bits in an arbitrary manner yields a hash function fulfilling the above criteria in terms of n' instead of n. In particular, no n'-bit truncation of an n-bit hash value is a valid n'-bit digest for the same message, since the initial value depends on the digest length n.

The claimed security levels for each of the specified digest lengths are summarised in Table 9.

5 Implementation Aspects

This section discusses implementation aspects of LANE. Several software implementations of LANE targeting general purpose CPU's have been created, as well as two hardware implementations. They are presented and evaluated in Sect. 5.1 and Sect. 5.3, respectively. Sect. 5.2 discusses the implementation aspects of LANE on 8-bit embedded systems.

5.1 General Purpose CPU's

As LANE is based on the AES block cipher [19,45], the implementation techniques that are commonly used for the AES can be applied directly to LANE. A prevailing technique is to group the SubBytes and (part of) the MixColumns operations into four 8-to-32-bit lookup tables [19]. ShiftRows can be implemented as a simple reordering of the indices, so it does not require any actual instructions. The AddConstants and AddCounter operations in LANE are implemented in the same way as AddRoundKey in the AES. Finally, like ShiftRows, also the SwapColumns operation does not require any explicit instructions, just an appropriate permutation of the indices.

There is extensive literature on fast AES implementations [3,9,37–39,55,59,60]. The design of LANE is such that entire rounds of the AES block cipher are used as components. Hence, virtually all of the fast implementation techniques and 'tricks' apply to LANE as well.

We wrote an optimised implementation of LANE in ANSI-C, using the standard, well-known techniques for the implementation of the AES [19]. In addition, we also wrote a very similar implementation in x86 assembly using the MMX instruction set as a source of eight extra registers. This reduces the register pressure, and shows a considerable improvement in performance in our test results. Finally, we developed a bitsliced implementation of LANE, inspired by the work of Matsui et al. on the AES [37,39]. Details on this implementation are given in Sect. 5.1.1.

We measured the performance of our implementations, using three different software suites: Microsoft Visual Studio, GNU GCC and the Intel C compiler. Details on our test hardware and these three software are given in Table 10. We tested both short, 64 byte messages and long, 32 kilobyte messages, and normalised the cycle count by dividing it by the message length, i.e., we use 'cycles per byte' (cpb) as a performance metric. The measurement results are presented in Table 11. As the performance of LANE-224 and LANE-384 are identical to LANE-256 and LANE-512, respectively, we only give data for the latter two. Also, our two assembly implementations currently only support LANE-256, hence no data on LANE-512 is given for these implementations.

We note that apparently, the choice of the compiler has a large impact on the performance. The Intel C compiler (ICC) achieves a speed which is very close to that of our MMX assembly implementation. We expect that further improvements can be made to these implementations. As several highly advanced, very fast AES implementations exist, e.g., the very recent work by Bernstein and Schwabe [9], we expect that (much) faster implementations of LANE can be made using these techniques.

5.1.1 Bitsliced Implementation

Similarly to the AES block cipher, the C implementation of LANE relies heavily on table lookups. Several authors have demonstrated side-channel attacks against such software implementations of AES, using cache-timing analysis to gather information on these data-dependent table lookups [7, 49]. While side-channel attacks are not relevant for hash functions in their most straightforward application, they become a potential threat once the hash function is used to process secret values such as the message authentication key in an HMAC construction [47]. Thus, we provide an alternative proof-of-concept implementation of LANE-256 that is constant-time and consequently not vulnerable to cache-timing attacks.

A Cache-Timing Resistant Implementation of Lane-256. The constant-time implementation of LANE-256 uses bitslicing to implement the AES S-box. That is, instead of using table look-ups, we compute the S-box output on-the-fly, using its representation as a concatenation of eight 8-to-1-bit Boolean functions. Such a bitsliced implementation of the AES S-box was first proposed by Matsui [37]. Later, Matsui and Nakajima [39] reported a particularly efficient implementation of bitsliced AES on the Intel Core 2 Duo processors, achieving speeds of up to 10.2 cycles per byte in modes of operation where sufficient parallelism is possible,

Table 10 – Test platform for the software implementations of LANE.

Common hardware platform	
CPU	Intel(R) Core(TM) 2 Duo T8100 2.1 GHz (supports MMX, SSE, SSE2, SSSE3, SSE4.1) 3072 KB cache memory
Memory	1024 MB
Software platform 1: GNU (64-bit)	
Operating System	Ubuntu Linux 8.04 x86_64
Compiler	GNU C compiler (GCC) version 4.2.3
Compiler flags	`-O3 -fomit-frame-pointer`
Assembler	GNU MMX assembler (GAS) version 2.18.0
Software platform 2: Microsoft (32-bit)	
Operating System	Microsoft Windows XP professional SP2
Compiler	Microsoft Visual Studio 2008 Version 9.0.21022.8 RTM
Compiler flags	`/O2`
Software platform 3: Intel (64-bit)	
Operating System	Ubuntu Linux x86_64
Compiler	Intel C compiler (ICC) 10.1 20080801
Compiler flags	`-O3`

Table 11 – Performance measurement results of our LANE implementations. The test platform, and the three software suites we used, are described in Table 10.

Implementation	Platform	64 byte message	32 kbyte message
Optimised ANSI-C implementation of LANE-256	GNU (1)	130.67 cpb	43.02 cpb
	Microsoft (2)	104.75 cpb	40.46 cpb
	Intel (3)	79.78 cpb	26.17 cpb
MMX Assembly implementation of LANE-256	GNU (1)	79.84 cpb	25.66 cpb
Bitsliced implementation of LANE-256	GNU (1)	87.19 cpb	30.20 cpb
Optimised ANSI-C implementation of LANE-512	GNU (1)	1069.97 cpb	176.97 cpb
	Microsoft (2)	923.11 cpb	152.24 cpb
	Intel (3)	864.46 cpb	145.31 cpb

Table 12 – Number of XMM instructions in one LANE round.

SubBytes	205
ShiftRows	8
MixColumns	43
AddConstants	8
AddCounter	8
SwapColumns	72
Total	344

such as the counter mode. Similarly, the parallelism available in LANE allows for an efficient bitsliced implementation.

Our implementation uses eight 128-bit XMM registers to store part of the LANE-256 state, one register for each bit position in a LANE state. That is, we collect in the first register bits from the least significant bit position in each LANE-256 byte, in the second register bits from the second bit position, and so forth. In order to fill the 128-bit XMM registers and fully utilise their width, we need to be able to process 128 bytes of state in parallel. At first glance, LANE-256 does not lend to such optimal parallelism: the six parallel P-lanes contain altogether 192 bytes, whereas the two Q-lanes in the second, sequential layer contain only 64 bytes. However, we observe that lanes P_4 and P_5 are independent of the chaining value and thus can be processed before the actual chaining value is known. This observation allows us to process the LANE-256 state in two parts of 128 bytes each. Namely, during each compression function call, we first process the Q-lanes of the previous call together with lanes P_4 and P_5. The output of the Q-lanes then provides us with the chaining value required to process lanes P_0, P_1, P_2, P_3.

Performance. We have implemented the compression function of LANE-256 in a bitsliced manner, using GNU assembly. Table 12 lists the number of instructions required in one LANE round. In addition to the compression function, our implementation contains bitslicing for input messages, and inverse bitslicing for the hash output, so that the bitsliced implementation is fully compatible with the standard implementation. Notice that apart from the input message, also the counter needs to be bitsliced on the fly for each compression function call, introducing an overhead compared to an AES implementation.

Including format conversion overhead for input, output and counters, the bitsliced implementation achieves a speed of 30.2 cycles per byte on our test platform (see Table 10 for platform details). We conclude that at least on 64-bit platforms, it is possible to implement LANE in a cache-timing resistant manner without a significant penalty in performance.

5.1.2 Intel AES-NI Instruction Set

In a white paper [28], Intel has announced AES-NI, a new set of instructions that are going to be introduced in the next generation of Intel processors, as of 2009. The AES-NI extension consists of six instructions that will provide full hardware support for the AES block cipher [19, 45].

As Lane reuses rounds of the AES as components, the Intel AES-NI instruction can also be used to create a fast implementation of Lane. Lane only uses full AES rounds in the encryption direction, hence only one of the six new instructions, AESENC, is required to implement Lane.

The AESENC instruction consists of the ShiftRows, SubBytes, MixColumns and AddRoundKey operations. The AddRoundKey operation in the AES is functionally equivalent to AddConstants in Lane. Other operations used in Lane can be implemented using instructions from the SSE/SSE2 instruction set. The message expansion and AddCounter can be implemented using the PXOR instruction, and either SHUFPD or SHUFPS can be used to implement SwapColumns, depending on the Lane variant.

As the underlying hardware that implements the AESENC instruction is fully pipelined [28], a new AES round can be started each clock cycle, provided that there are no data dependencies. The latency of the AESENC instruction is 6 cycles [28], which implies that six parallel AES rounds suffice to keep the pipeline filled. Lane offers ample opportunities for parallel AES rounds. In the first layer, there are 12 independent AESENC instructions in each round. In the second layer, there are only four parallel AES rounds. But, by scheduling the second layer of one compression function call in parallel with the P_4 and P_5 lanes of the next compression function call, as explained in Sect. 5.1.1, the parallelism can be balanced out more evenly.

As processors supporting the AES-NI instruction set are not yet available on the market, we can only estimate the performance of Lane on such a machine. For a parallel mode of AES, a throughput of about 12 cycles per block, or 1.2 cycles per AES round is claimed [28]. Counting only the AES rounds in a Lane-256 compression function call would yield a performance of $84 \cdot 1.2/64 = 1.575$ cycles per byte for Lane-256. But as other components of the Lane compression function, which are negligible in other implementations, can likely no longer be ignored, a performance of around 5 cycles per byte seems more reasonable. Still, this shows that Lane has the potential to achieve a very high performance on platforms that offer a fast, hardware accelerated way to compute AES encryptions.

5.2 Embedded Systems with an 8-bit CPU

As Lane is based on rounds of the AES block cipher, its performance on 8-bit CPU's can be estimated based on the existing literature on the implementation of the AES on these platforms. Rinne et al. [51] present an implementation of the AES on an 8-bit AVR microcontroller, inspired by the 8-bit AES code of

Gladman [26]. Their implementation has a code size (ROM size) of 3410 bytes, and is able to do an AES encryption in 3766 CPU cycles.

The message expansion of LANE-256 can be implemented using 288 8-bit XOR operations. An 8-bit XOR operation typically takes a single CPU cycle on these CPU's, hence the message expansion costs about 288 CPU cycles. An AES encryption consists of ten rounds, so the cost of a single AES round can be estimated at 377 CPU cycles. One LANE-256 compression function call contains 84 AES rounds, yielding a total of 31 668 CPU cycles. The AddCounter operation consists of four 8-bit XOR's, and is used 34 times per compression function call — 136 CPU cycles in total. Finally, the computation of the constants needs to be counted. A single step of the LFSR can be implemented in 8 CPU cycles, and it needs to be carried out 272 times, resulting in a cost of 2176 CPU cycles. Adding these components gives a total cost of 34 268 CPU cycles.

Based on this rough estimate, we can expect a real implementation of LANE-256 on this 8-bit platform to require about 35 000 CPU cycles per compression function. This corresponds to an expected performance of roughly 550 cycles per byte. As most of the program code will be the same as for an AES implementation, we expect that an implementation of LANE-256 should fit in less than 5 kilobytes of ROM.

The amount of RAM required by a LANE-256 implementation on a resource-constrained system can be estimated as follows.

- 768 bits of read/write memory to store the input chaining value H_i, and for intermediate storage. Note that it is possible to reuse the memory used for storing H_i after the fourth lane of the first layer has been started, because the last two lanes are independent of H_i. This trick allows to save 256 bits of memory.
- 512 bits to store the message block. Note that the message expansion does not need to modify this memory, so it could be shared with another application, e.g., a transmit or receive buffer.
- 64 bits for the counter.
- (optionally) 256 bits to store the salt value.
- 32 bits for computing the constants on-the-fly.

For example, a LANE-256 implementation not using salts requires 172 bytes of RAM, of which the 64 bytes containing the current message block can be shared.

Tillich et al. [57] propose to add a small hardware accelerator of only 1.1 kGates to an AVR microcontroller, which increases the performance of an AES encryption by a factor 3.6 compared to the pure software implementation of [51]. Of course, such techniques will also greatly benefit the performance of LANE. It is even more interesting for embedded systems requiring both a hash function and a block cipher. When using LANE and the AES, a single investment in additional hardware can be used both for faster hashing and faster encryption.

Table 13 – Hardware evaluation of the Lane hash function.

Design	Area [GE]	Frequency [MHz]	# of Cycles	Throughput [Mbps]
Lane-224/Lane-256[†]	16 462	100	2201	23.3
Lane-224/Lane-256[‡]	243 486	305	11	14 191
Lane-384/Lane-512[‡]	466 190	286	14	20 958

[†] Compact implementation.
[‡] High-throughput implementations.

5.3 Hardware Implementation

The hardware performance evaluation of the Lane hash function was done by synthesising the proposed designs using 0.13 μm CMOS standard cell library. The code was first written in GEZEL [54], then compiled to VHDL and synthesised using the Synopsys Design Vision tool [56]. The synthesis results are given in Table 13.

As the ample parallelism provided by Lane allows for much flexibility in high-throughput implementations, our main goal was to show that Lane can achieve a very high throughput at the cost of consumed gate area. Additionally, by implementing a compact version, we have also shown that the same algorithm can be used in more constrained environments where the available gate area is a limiting factor.

The target for the high-throughput implementations was to minimise the critical path of the design. To perform the first layer of permutations, we used 6 permutation blocks in parallel where each of them contained 2 full AES engines (4 for Lane-384 and Lane-512). The two permutations from the first layer were also reused for the second layer.

The straightforward implementation of Lane-224/Lane-256 resulted in the critical path of 5.60 ns and the cycle count of 9. The critical path was placed from the input of the message injection function, going along the permutation block and ending at the input of the second layer of permutation. As the message injection function was performed only once per input message block, we moved the state registers at the input of the permutation blocks. This approach resulted in the faster design, shortening the critical path to 4.28 ns. One more clock cycle had to be spent in order to perform the complete round, but the final throughput increased by about 20 %. The critical path was now placed along the permutation block and also contained the 3-input XOR gate at the output of the first layer permutations. By storing the output of the first layer back to the state registers and then performing an XOR operation, we introduced one more clock cycle and reduced the critical path down to only 3.28 ns (3.49 ns for Lane-384/Lane-512).

The final design achieved a high throughput of 14.2 Gbps and 21.0 Gbps for LANE-224/LANE-256 and LANE-384/LANE-512, respectively.

As can be seen from Table 13, the most compact implementation is obtained for LANE-224/LANE-256 algorithm and consumes approximately 16.5 kGE. The major part of the compact design is consumed by the message expansion, though it is performed only once per input message block. Hence, we also evaluated the circuit size assuming that the message expansion is performed outside of the hash engine. This resulted in a smaller design with a gate area of only 11.7 kGE. The compact implementation was made using only one permutation block. Inside the permutation we used a single compact AES S-box [40] and a single AES MixColumns block. This approach resulted in a large number of cycles (2201), while on the other hand it efficiently reduced the final gate count. We used three 256-bit registers to maintain the internal state. Note that our only goal for the compact implementation was to have a small die size, regardless of the circuit speed. Hence, we fixed the frequency to 100 MHz and synthesised our design.

The throughput was calculated according to the following equations:

$$\text{Throughput}_{256/224} = \frac{\text{Frequency}}{\#\text{ of Cycles}} \times 512 \ , \tag{50}$$

$$\text{Throughput}_{512/384} = \frac{\text{Frequency}}{\#\text{ of Cycles}} \times 1024 \ . \tag{51}$$

Our hardware performance figures show that the LANE cryptographic hash functions can be implemented very efficiently and provide very high throughputs, up to 21 Gbps. On the other hand, the compact implementation shows that, at the cost of speed, the LANE hash functions can be considered as a good candidate for constrained environments. We believe that in the future faster and more compact LANE designs will be announced. By exploring different levels of parallelism, one can make a number of trade-offs and choose the appropriate application-driven implementation. Compact implementation of hash functions remains a challenging task in general and hence, we expect more research effort in this direction.

Acknowledgements

I would like to thank Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, Vincent Rijmen and Elmar Tischhauser for many interesting discussions concerning the design of LANE and its predecessors, and for their continued effort on the cryptanalysis of both older and the final version of LANE. Their findings, comments and suggestions for improvements were invaluable in the design process.

I extend my gratitude to Antoon Bosselaers, Emilia Käsper, Miroslav Knežević, Nicky Mouha and Vesselin Velichkov for their work on several implementations of LANE, and for giving useful feedback from the implementor's point of view.

Additional thanks go to all of the people mentioned above, for their contributions to the writing and proofreading of this document. Finally, thanks to everyone in the COSIC research group for their support.

Sebastiaan Indesteege
October 2008

Sebastiaan Indesteege is supported by the Fund for Scientific Research Flanders (Aspirant F.W.O. Vlaanderen). This work was also supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the Interdisciplinary Institute for BroadBand Technology (IBBT), and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

[1] E. Andreeva. On LANE modes of operation. Technical report, COSIC, 2008.

[2] E. Andreeva, C. Bouillaguet, P.-A. Fouque, J. J. Hoch, J. Kelsey, A. Shamir, and S. Zimmer. Second preimage attacks on dithered hash functions. In N. P. Smart, editor, *Advances in Cryptology — EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[3] K. Aoki and H. Lipmaa. Fast implementations of AES candidates. In *Third AES Candidate Conference*, pages 106–120. National Institute of Standards and Technology, 2000.

[4] G. Ars, J.-C. Faugère, H. Imai, M. Kawazoe, and M. Sugita. Comparison between XL and Gröbner basis algorithms. In P. J. Lee, editor, *Advances in Cryptology — ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2004.

[5] T. Becker and V. Weispfenning. *Gröbner bases: A computational approach to commutative algebra*, volume 141 of *Graduate Texts in Mathematics*. Springer, 1993.

[6] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th Symposium on Foundations of Computer Science (FOCS 1996)*, pages 514–523. IEEE Computer Society, 2002.

[7] D. J. Bernstein. Cache-timing attacks on AES. Preprint, 2005. Available online at `http://cr.yp.to/papers.html#cachetiming`.

[8] D. J. Bernstein. What output size resists collisions in a XOR of independent expansions? In *ECRYPT Hash Workshop*. European Network of Excellence in Cryptology ECRYPT, May 2007.

[9] D. J. Bernstein and P. Schwabe. New AES software speed records. In D. R. Chowdhury, V. Rijmen, and A. Das, editors, *Progress in Cryptology — INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

[10] E. Biham and O. Dunkelman. A framework for iterative hash functions — HAIFA. Second NIST Hash Workshop, 2006.

[11] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology — CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[12] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965.

[13] F. Chabaud and A. Joux. Differential collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[14] C. Cid, S. Murphy, and M. Robshaw. *Algebraic Aspects of the Advanced Encryption Standard.* Springer, 2006.

[15] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In V. Shoup, editor, *Advances in Cryptology — CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[16] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In B. Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[17] N. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Y. Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[18] J. Daemen, L. R. Knudsen, and V. Rijmen. The block cipher Square. In E. Biham, editor, *Fast Software Encryption, 4th International Workshop — FSE '97*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[19] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer, 2002.

[20] I. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[21] R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Jan. 1999.

[22] H. Dobbertin. Cryptanalysis of MD4. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop — FSE '96*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 1996.

[23] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139:61–88, 1999.

[24] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In T. Mora, editor, *International Symposium on Symbolic and Algebraic Computation (ISAAC 2002)*, pages 75–83. ACM Press, 2002.

[25] P.-A. Fouque, J. Stern, and S. Zimmer. Cryptanalysis of tweaked versions of SMASH and reparation. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography — SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2009.

[26] B. Gladman. Byte oriented AES implementation. Available online at `http://www.gladman.me.uk/`.

[27] S. Halevi and H. Krawczyk. Strengthening digital signatures via randomized hashing. In C. Dwork, editor, *Advances in Cryptology — CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2006.

[28] Intel Corporation. Advanced encryption standard (AES) instructions set. White paper, July 2008. Available online at `http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set_WP.pdf`.

[29] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In M. K. Franklin, editor, *Advances in Cryptology — CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[30] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[31] L. R. Knudsen. Truncated and higher order differentials. In B. Preneel, editor, *Fast Software Encryption, Second International Workshop — FSE '94*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994.

[32] L. R. Knudsen and D. Wagner. Integral cryptanalysis. In J. Daemen and V. Rijmen, editors, *Fast Software Encryption, 9th International Workshop — FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2002.

[33] X. Lai. Higher order derivatives and differential cryptanalysis. Proc. Symposium on Communication, Coding and Cryptography, in honor of James L. Massey on the occasion of his 60'th birthday, Feb. 10-13, 1994, Monte-Verita, Ascona, Switzerland, 1994.

[34] X. Lai, J. L. Massey, and S. Murphy. Markov ciphers and differential cryptanalysis. In D. W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38. Springer, 1991.

[35] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, revised edition, 1994.

[36] S. Lucks. The saturation attack — a bait for Twofish. In M. Matsui, editor, *Fast Software Encryption, 8th International Workshop — FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.

[37] M. Matsui. How far can we go on the x64 processors? In M. J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop — FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 2006.

[38] M. Matsui and S. Fukuda. How to maximize software performance of symmetric primitives on Pentium III and 4 processors. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption, 12th International Workshop — FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[39] M. Matsui and J. Nakajima. On the power of bitslice implementation on Intel Core2 processor. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems — CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 2007.

[40] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. A systematic evaluation of compact hardware implementations for the Rijndael S-Box. In A. Menezes, editor, *Topics in Cryptology — CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 323–333. Springer, 2005.

[41] R. C. Merkle. One way hash functions and DES. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[42] S. Miyaguchi, K. Ohta, and M. Iwata. Confirmation that some hash functions are not collision free. In I. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 1991.

[43] J. Nakahara Jr., D. S. de Freitas, and R. C.-W. Phan. New multiset attacks on Rijndael with large blocks. In E. Dawson and S. Vaudenay, editors, *Progress in Cryptology — MYCRYPT 2005*, volume 3715 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 2005.

[44] J. Nakahara Jr. and I. C. Pavão. Impossible-differential attacks on large-block Rijndael. In J. A. Garay, A. K. Lenstra, M. Mambo, and R. Peralta, editors, *Information Security 10th International Conference — ISC 2007*, volume 4779 of *Lecture Notes in Computer Science*, pages 104–117. Springer, 2007.

[45] National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

[46] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, Nov. 2007.

[47] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC). Federal Information Processing Standards Publication 198-1, 2008.

[48] National Institute of Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-3, Oct. 2008.

[49] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *Topics in Cryptology — CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[50] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. In D. R. Stinson, editor, *Advances in Cryptology — CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1994.

[51] S. Rinne, T. Eisenbarth, and C. Paar. Performance analysis of contemporary light-weight block ciphers on 8-bit microcontrollers. In *SPEED – Software Performance Enhancement for Encryption and Decryption.* European Network of Excellence in Cryptology ECRYPT, June 2007.

[52] P. Rogaway. Formalizing human ignorance. In P. Q. Nguyen, editor, *Progress in Cryptology — VIETCRYPT 2006*, volume 4341 of *Lecture Notes in Computer Science*, pages 211–228. Springer, 2006.

[53] P. Rogaway and J. P. Steinberger. Security/efficiency tradeoffs for permutation-based hashing. In N. P. Smart, editor, *Advances in Cryptology — EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2008.

[54] P. Schaumont. The GEZEL development environment. `http://rijndael.ece.vt.edu/gezel2`.

[55] B. Schneier and D. Whiting. A performance comparison of the five AES finalists. In *Third AES Candidate Conference*, pages 123–135. National Institute of Standards and Technology, 2000.

[56] Synopsys Design Vision. `http://www.synopsys.com/`.

[57] S. Tillich and C. Herbst. Boosting AES performance on a tiny processor core. In T. Malkin, editor, *Topics in Cryptology — CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2008.

[58] D. Wagner. A generalized birthday problem. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

[59] R. Weiss and N. L. Binkert. A comparison of AES candidates on the Alpha 21264. In *Third AES Candidate Conference*, pages 75–81. National Institute of Standards and Technology, 2000.

[60] J. Worley, B. Worley, T. Christian, and C. Worley. AES finalists on PA-RISC and IA-64: Implementations & performance. In *Third AES Candidate Conference*, pages 57–74. National Institute of Standards and Technology, 2000.

A The Constants Used in Lane

Table 14 contains the precomputed values of the constants used in LANE.

Table 14 – The constants used in LANE.

k_0	$=$ `07fc703d`$_x$,	k_1	$=$ `d3fe381f`$_x$,	k_2	$=$ `b9ff1c0e`$_x$,	k_3	$=$ `5cff8e07`$_x$,
k_4	$=$ `fe7fc702`$_x$,	k_5	$=$ `7f3fe381`$_x$,	k_6	$=$ `ef9ff1c1`$_x$,	k_7	$=$ `a7cff8e1`$_x$,
k_8	$=$ `83e7fc71`$_x$,	k_9	$=$ `91f3fe39`$_x$,	k_{10}	$=$ `98f9ff1d`$_x$,	k_{11}	$=$ `9c7cff8f`$_x$,
k_{12}	$=$ `9e3e7fc6`$_x$,	k_{13}	$=$ `4f1f3fe3`$_x$,	k_{14}	$=$ `f78f9ff0`$_x$,	k_{15}	$=$ `7bc7cff8`$_x$,
k_{16}	$=$ `3de3e7fc`$_x$,	k_{17}	$=$ `1ef1f3fe`$_x$,	k_{18}	$=$ `0f78f9ff`$_x$,	k_{19}	$=$ `d7bc7cfe`$_x$,
k_{20}	$=$ `6bde3e7f`$_x$,	k_{21}	$=$ `e5ef1f3e`$_x$,	k_{22}	$=$ `72f78f9f`$_x$,	k_{23}	$=$ `e97bc7ce`$_x$,
k_{24}	$=$ `74bde3e7`$_x$,	k_{25}	$=$ `ea5ef1f2`$_x$,	k_{26}	$=$ `752f78f9`$_x$,	k_{27}	$=$ `ea97bc7d`$_x$,

Table 14 – The constants used in Lane. *(continued)*

k_{28} = a54bde3f$_x$,	k_{29} = 82a5ef1e$_x$,	k_{30} = 4152f78f$_x$,	k_{31} = f0a97bc6$_x$,
k_{32} = 7854bde3$_x$,	k_{33} = ec2a5ef0$_x$,	k_{34} = 76152f78$_x$,	k_{35} = 3b0a97bc$_x$,
k_{36} = 1d854bde$_x$,	k_{37} = 0ec2a5ef$_x$,	k_{38} = d76152f6$_x$,	k_{39} = 6bb0a97b$_x$,
k_{40} = e5d854bc$_x$,	k_{41} = 72ec2a5e$_x$,	k_{42} = 3976152f$_x$,	k_{43} = ccbb0a96$_x$,
k_{44} = 665d854b$_x$,	k_{45} = e32ec2a4$_x$,	k_{46} = 71976152$_x$,	k_{47} = 38cbb0a9$_x$,
k_{48} = cc65d855$_x$,	k_{49} = b632ec2b$_x$,	k_{50} = 8b197614$_x$,	k_{51} = 458cbb0a$_x$,
k_{52} = 22c65d85$_x$,	k_{53} = c1632ec3$_x$,	k_{54} = b0b19760$_x$,	k_{55} = 5858cbb0$_x$,
k_{56} = 2c2c65d8$_x$,	k_{57} = 161632ec$_x$,	k_{58} = 0b0b1976$_x$,	k_{59} = 05858cbb$_x$,
k_{60} = d2c2c65c$_x$,	k_{61} = 6961632e$_x$,	k_{62} = 34b0b197$_x$,	k_{63} = ca5858ca$_x$,
k_{64} = 652c2c65$_x$,	k_{65} = e2961633$_x$,	k_{66} = a14b0b18$_x$,	k_{67} = 50a5858c$_x$,
k_{68} = 2852c2c6$_x$,	k_{69} = 14296163$_x$,	k_{70} = da14b0b0$_x$,	k_{71} = 6d0a5858$_x$,
k_{72} = 36852c2c$_x$,	k_{73} = 1b429616$_x$,	k_{74} = 0da14b0b$_x$,	k_{75} = d6d0a584$_x$,
k_{76} = 6b6852c2$_x$,	k_{77} = 35b42961$_x$,	k_{78} = cada14b1$_x$,	k_{79} = b56d0a59$_x$,
k_{80} = 8ab6852d$_x$,	k_{81} = 955b4297$_x$,	k_{82} = 9aada14a$_x$,	k_{83} = 4d56d0a5$_x$,
k_{84} = f6ab6853$_x$,	k_{85} = ab55b428$_x$,	k_{86} = 55aada14$_x$,	k_{87} = 2ad56d0a$_x$,
k_{88} = 156ab685$_x$,	k_{89} = dab55b43$_x$,	k_{90} = bd5aada0$_x$,	k_{91} = 5ead56d0$_x$,
k_{92} = 2f56ab68$_x$,	k_{93} = 17ab55b4$_x$,	k_{94} = 0bd5aada$_x$,	k_{95} = 05ead56d$_x$,
k_{96} = d2f56ab7$_x$,	k_{97} = b97ab55a$_x$,	k_{98} = 5cbd5aad$_x$,	k_{99} = fe5ead57$_x$,
k_{100} = af2f56aa$_x$,	k_{101} = 5797ab55$_x$,	k_{102} = fbcbd5ab$_x$,	k_{103} = ade5ead4$_x$,
k_{104} = 56f2f56a$_x$,	k_{105} = 2b797ab5$_x$,	k_{106} = c5bcbd5b$_x$,	k_{107} = b2de5eac$_x$,
k_{108} = 596f2f56$_x$,	k_{109} = 2cb797ab$_x$,	k_{110} = c65bcbd4$_x$,	k_{111} = 632de5ea$_x$,
k_{112} = 3196f2f5$_x$,	k_{113} = c8cb797b$_x$,	k_{114} = b465bcbc$_x$,	k_{115} = 5a32de5e$_x$,
k_{116} = 2d196f2f$_x$,	k_{117} = c68cb796$_x$,	k_{118} = 63465bcb$_x$,	k_{119} = e1a32de4$_x$,
k_{120} = 70d196f2$_x$,	k_{121} = 3868cb79$_x$,	k_{122} = cc3465bd$_x$,	k_{123} = b61a32df$_x$,
k_{124} = 8b0d196e$_x$,	k_{125} = 45868cb7$_x$,	k_{126} = f2c3465a$_x$,	k_{127} = 7961a32d$_x$,
k_{128} = ecb0d197$_x$,	k_{129} = a65868ca$_x$,	k_{130} = 532c3465$_x$,	k_{131} = f9961a33$_x$,
k_{132} = accb0d18$_x$,	k_{133} = 5665868c$_x$,	k_{134} = 2b32c346$_x$,	k_{135} = 159961a3$_x$,
k_{136} = daccb0d0$_x$,	k_{137} = 6d665868$_x$,	k_{138} = 36b32c34$_x$,	k_{139} = 1b59961a$_x$,
k_{140} = 0daccb0d$_x$,	k_{141} = d6d66587$_x$,	k_{142} = bb6b32c2$_x$,	k_{143} = 5db59961$_x$,
k_{144} = fedaccb1$_x$,	k_{145} = af6d6659$_x$,	k_{146} = 87b6b32d$_x$,	k_{147} = 93db5997$_x$,
k_{148} = 99edacca$_x$,	k_{149} = 4cf6d665$_x$,	k_{150} = f67b6b33$_x$,	k_{151} = ab3db598$_x$,
k_{152} = 559edacc$_x$,	k_{153} = 2acf6d66$_x$,	k_{154} = 1567b6b3$_x$,	k_{155} = dab3db58$_x$,
k_{156} = 6d59edac$_x$,	k_{157} = 36acf6d6$_x$,	k_{158} = 1b567b6b$_x$,	k_{159} = ddab3db4$_x$,
k_{160} = 6ed59eda$_x$,	k_{161} = 376acf6d$_x$,	k_{162} = cbb567b7$_x$,	k_{163} = b5dab3da$_x$,
k_{164} = 5aed59ed$_x$,	k_{165} = fd76acf7$_x$,	k_{166} = aebb567a$_x$,	k_{167} = 575dab3d$_x$,
k_{168} = fbaed59f$_x$,	k_{169} = add76ace$_x$,	k_{170} = 56ebb567$_x$,	k_{171} = fb75dab2$_x$,
k_{172} = 7dbaed59$_x$,	k_{173} = eedd76ad$_x$,	k_{174} = a76ebb57$_x$,	k_{175} = 83b75daa$_x$,
k_{176} = 41dbaed5$_x$,	k_{177} = f0edd76b$_x$,	k_{178} = a876ebb4$_x$,	k_{179} = 543b75da$_x$,
k_{180} = 2a1dbaed$_x$,	k_{181} = c50edd77$_x$,	k_{182} = b2876eba$_x$,	k_{183} = 5943b75d$_x$,
k_{184} = fca1dbaf$_x$,	k_{185} = ae50edd6$_x$,	k_{186} = 572876eb$_x$,	k_{187} = fb943b74$_x$,
k_{188} = 7dca1dba$_x$,	k_{189} = 3ee50edd$_x$,	k_{190} = cf72876f$_x$,	k_{191} = b7b943b6$_x$,
k_{192} = 5bdca1db$_x$,	k_{193} = fdee50ec$_x$,	k_{194} = 7ef72876$_x$,	k_{195} = 3f7b943b$_x$,
k_{196} = cfbdca1c$_x$,	k_{197} = 67dee50e$_x$,	k_{198} = 33ef7287$_x$,	k_{199} = c9f7b942$_x$,
k_{200} = 64fbdca1$_x$,	k_{201} = e27dee51$_x$,	k_{202} = a13ef729$_x$,	k_{203} = 809f7b95$_x$,
k_{204} = 904fbdcb$_x$,	k_{205} = 9827dee4$_x$,	k_{206} = 4c13ef72$_x$,	k_{207} = 2609f7b9$_x$,
k_{208} = c304fbdd$_x$,	k_{209} = b1827def$_x$,	k_{210} = 88c13ef6$_x$,	k_{211} = 44609f7b$_x$,
k_{212} = f2304fbc$_x$,	k_{213} = 791827de$_x$,	k_{214} = 3c8c13ef$_x$,	k_{215} = ce4609f6$_x$,
k_{216} = 672304fb$_x$,	k_{217} = e391827c$_x$,	k_{218} = 71c8c13e$_x$,	k_{219} = 38e4609f$_x$,
k_{220} = cc72304e$_x$,	k_{221} = 66391827$_x$,	k_{222} = e31c8c12$_x$,	k_{223} = 718e4609$_x$,
k_{224} = e8c72305$_x$,	k_{225} = a4639183$_x$,	k_{226} = 8231c8c0$_x$,	k_{227} = 4118e460$_x$,
k_{228} = 208c7230$_x$,	k_{229} = 10463918$_x$,	k_{230} = 08231c8c$_x$,	k_{231} = 04118e46$_x$,
k_{232} = 0208c723$_x$,	k_{233} = d1046390$_x$,	k_{234} = 688231c8$_x$,	k_{235} = 344118e4$_x$,
k_{236} = 1a208c72$_x$,	k_{237} = 0d104639$_x$,	k_{238} = d688231d$_x$,	k_{239} = bb44118f$_x$,
k_{240} = 8da208c6$_x$,	k_{241} = 46d10463$_x$,	k_{242} = f3688230$_x$,	k_{243} = 79b44118$_x$,
k_{244} = 3cda208c$_x$,	k_{245} = 1e6d1046$_x$,	k_{246} = 0f368823$_x$,	k_{247} = d79b4410$_x$,
k_{248} = 6bcda208$_x$,	k_{249} = 35e6d104$_x$,	k_{250} = 1af36882$_x$,	k_{251} = 0d79b441$_x$,
k_{252} = d6bcda21$_x$,	k_{253} = bb5e6d11$_x$,	k_{254} = 8daf3689$_x$,	k_{255} = 96d79b45$_x$,
k_{256} = 9b6bcda3$_x$,	k_{257} = 9db5e6d0$_x$,	k_{258} = 4edaf368$_x$,	k_{259} = 276d79b4$_x$,
k_{260} = 13b6bcda$_x$,	k_{261} = 09db5e6d$_x$,	k_{262} = d4edaf37$_x$,	k_{263} = ba76d79a$_x$,
k_{264} = 5d3b6bcd$_x$,	k_{265} = fe9db5e7$_x$,	k_{266} = af4edaf2$_x$,	k_{267} = 57a76d79$_x$,
k_{268} = fbd3b6bd$_x$,	k_{269} = ade9db5f$_x$,	k_{270} = 86f4edae$_x$,	k_{271} = 437a76d7$_x$,
k_{272} = f1bd3b6a$_x$,	k_{273} = 78de9db5$_x$,	k_{274} = ec6f4edb$_x$,	k_{275} = a637a76c$_x$,

Table 14 – The constants used in LANE. *(continued)*

k_{276} = 531bd3b6$_x$,	k_{277} = 298de9db$_x$,	k_{278} = c4c6f4ec$_x$,	k_{279} = 62637a76$_x$,
k_{280} = 3131bd3b$_x$,	k_{281} = c898de9c$_x$,	k_{282} = 644c6f4e$_x$,	k_{283} = 322637a7$_x$,
k_{284} = c9131bd2$_x$,	k_{285} = 64898de9$_x$,	k_{286} = e244c6f5$_x$,	k_{287} = a122637b$_x$,
k_{288} = 809131bc$_x$,	k_{289} = 404898de$_x$,	k_{290} = 20244c6f$_x$,	k_{291} = c0122636$_x$,
k_{292} = 6009131b$_x$,	k_{293} = e004898c$_x$,	k_{294} = 700244c6$_x$,	k_{295} = 38012263$_x$,
k_{296} = cc009130$_x$,	k_{297} = 66004898$_x$,	k_{298} = 3300244c$_x$,	k_{299} = 19801226$_x$,
k_{300} = 0cc00913$_x$,	k_{301} = d6600488$_x$,	k_{302} = 6b300244$_x$,	k_{303} = 35980122$_x$,
k_{304} = 1acc0091$_x$,	k_{305} = dd660049$_x$,	k_{306} = beb30025$_x$,	k_{307} = 8f598013$_x$,
k_{308} = 97acc008$_x$,	k_{309} = 4bd66004$_x$,	k_{310} = 25eb3002$_x$,	k_{311} = 12f59801$_x$,
k_{312} = d97acc01$_x$,	k_{313} = bcbd6601$_x$,	k_{314} = 8e5eb301$_x$,	k_{315} = 972f5981$_x$,
k_{316} = 9b97acc1$_x$,	k_{317} = 9dcbd661$_x$,	k_{318} = 9ee5eb31$_x$,	k_{319} = 9f72f599$_x$,
k_{320} = 9fb97acd$_x$,	k_{321} = 9fdcbd67$_x$,	k_{322} = 9fee5eb2$_x$,	k_{323} = 4ff72f59$_x$,
k_{324} = f7fb97ad$_x$,	k_{325} = abfdcbd7$_x$,	k_{326} = 85fee5ea$_x$,	k_{327} = 42ff72f5$_x$,
k_{328} = f17fb97b$_x$,	k_{329} = a8bfdcbc$_x$,	k_{330} = 545fee5e$_x$,	k_{331} = 2a2ff72f$_x$,
k_{332} = c517fb96$_x$,	k_{333} = 628bfdcb$_x$,	k_{334} = e145fee4$_x$,	k_{335} = 70a2ff72$_x$,
k_{336} = 38517fb9$_x$,	k_{337} = cc28bfdd$_x$,	k_{338} = b6145fef$_x$,	k_{339} = 8b0a2ff6$_x$,
k_{340} = 458517fb$_x$,	k_{341} = f2c28bfc$_x$,	k_{342} = 796145fe$_x$,	k_{343} = 3cb0a2ff$_x$,
k_{344} = ce58517e$_x$,	k_{345} = 672c28bf$_x$,	k_{346} = e396145e$_x$,	k_{347} = 71cb0a2f$_x$,
k_{348} = e8e58516$_x$,	k_{349} = 7472c28b$_x$,	k_{350} = ea396144$_x$,	k_{351} = 751cb0a2$_x$,
k_{352} = 3a8e5851$_x$,	k_{353} = cd472c29$_x$,	k_{354} = b6a39615$_x$,	k_{355} = 8b51cb0b$_x$,
k_{356} = 95a8e584$_x$,	k_{357} = 4ad472c2$_x$,	k_{358} = 256a3961$_x$,	k_{359} = c2b51cb1$_x$,
k_{360} = b15a8e59$_x$,	k_{361} = 88ad472d$_x$,	k_{362} = 9456a397$_x$,	k_{363} = 9a2b51ca$_x$,
k_{364} = 4d15a8e5$_x$,	k_{365} = f68ad473$_x$,	k_{366} = ab456a38$_x$,	k_{367} = 55a2b51c$_x$,
k_{368} = 2ad15a8e$_x$,	k_{369} = 1568ad47$_x$,	k_{370} = dab456a2$_x$,	k_{371} = 6d5a2b51$_x$,
k_{372} = e6ad15a9$_x$,	k_{373} = a3568ad5$_x$,	k_{374} = 81ab456b$_x$,	k_{375} = 90d5a2b4$_x$,
k_{376} = 486ad15a$_x$,	k_{377} = 243568ad$_x$,	k_{378} = c21ab457$_x$,	k_{379} = b10d5a2a$_x$,
k_{380} = 5886ad15$_x$,	k_{381} = fc43568b$_x$,	k_{382} = ae21ab44$_x$,	k_{383} = 5710d5a2$_x$,
k_{384} = 2b886ad1$_x$,	k_{385} = c5c43569$_x$,	k_{386} = b2e21ab5$_x$,	k_{387} = 89710d5b$_x$,
k_{388} = 94b886ac$_x$,	k_{389} = 4a5c4356$_x$,	k_{390} = 252e21ab$_x$,	k_{391} = c29710d4$_x$,
k_{392} = 614b886a$_x$,	k_{393} = 30a5c435$_x$,	k_{394} = c852e21b$_x$,	k_{395} = b429710c$_x$,
k_{396} = 5a14b886$_x$,	k_{397} = 2d0a5c43$_x$,	k_{398} = c6852e20$_x$,	k_{399} = 63429710$_x$,
k_{400} = 31a14b88$_x$,	k_{401} = 18d0a5c4$_x$,	k_{402} = 0c6852e2$_x$,	k_{403} = 06342971$_x$,
k_{404} = d31a14b9$_x$,	k_{405} = b98d0a5d$_x$,	k_{406} = 8cc6852f$_x$,	k_{407} = 96634296$_x$,
k_{408} = 4b31a14b$_x$,	k_{409} = f598d0a4$_x$,	k_{410} = 7acc6852$_x$,	k_{411} = 3d663429$_x$,
k_{412} = ceb31a15$_x$,	k_{413} = b7598d0b$_x$,	k_{414} = 8bacc684$_x$,	k_{415} = 45d66342$_x$,
k_{416} = 22eb31a1$_x$,	k_{417} = c17598d1$_x$,	k_{418} = b0bacc69$_x$,	k_{419} = 885d6635$_x$,
k_{420} = 942eb31b$_x$,	k_{421} = 9a17598c$_x$,	k_{422} = 4d0bacc6$_x$,	k_{423} = 2685d663$_x$,
k_{424} = c342eb30$_x$,	k_{425} = 61a17598$_x$,	k_{426} = 30d0bacc$_x$,	k_{427} = 18685d66$_x$,
k_{428} = 0c342eb3$_x$,	k_{429} = d61a1758$_x$,	k_{430} = 6b0d0bac$_x$,	k_{431} = 358685d6$_x$,
k_{432} = 1ac342eb$_x$,	k_{433} = dd61a174$_x$,	k_{434} = 6eb0d0ba$_x$,	k_{435} = 3758685d$_x$,
k_{436} = cbac342f$_x$,	k_{437} = b5d61a16$_x$,	k_{438} = 5aeb0d0b$_x$,	k_{439} = fd758684$_x$,
k_{440} = 7ebac342$_x$,	k_{441} = 3f5d61a1$_x$,	k_{442} = cfaeb0d1$_x$,	k_{443} = b7d75869$_x$,
k_{444} = 8bebac35$_x$,	k_{445} = 95f5d61b$_x$,	k_{446} = 9afaeb0c$_x$,	k_{447} = 4d7d7586$_x$,
k_{448} = 26bebac3$_x$,	k_{449} = c35f5d60$_x$,	k_{450} = 61afaeb0$_x$,	k_{451} = 30d7d758$_x$,
k_{452} = 186bebac$_x$,	k_{453} = 0c35f5d6$_x$,	k_{454} = 061afaeb$_x$,	k_{455} = d30d7d74$_x$,
k_{456} = 6986beba$_x$,	k_{457} = 34c35f5d$_x$,	k_{458} = ca61afaf$_x$,	k_{459} = b530d7d6$_x$,
k_{460} = 5a986beb$_x$,	k_{461} = fd4c35f4$_x$,	k_{462} = 7ea61afa$_x$,	k_{463} = 3f530d7d$_x$,
k_{464} = cfa986bf$_x$,	k_{465} = b7d4c35e$_x$,	k_{466} = 5bea61af$_x$,	k_{467} = fdf530d6$_x$,
k_{468} = 7efa986b$_x$,	k_{469} = ef7d4c34$_x$,	k_{470} = 77bea61a$_x$,	k_{471} = 3bdf530d$_x$,
k_{472} = cdefa987$_x$,	k_{473} = b6f7d4c2$_x$,	k_{474} = 5b7bea61$_x$,	k_{475} = fdbdf531$_x$,
k_{476} = aedefa99$_x$,	k_{477} = 876f7d4d$_x$,	k_{478} = 93b7bea7$_x$,	k_{479} = 99dbdf52$_x$,
k_{480} = 4cedefa9$_x$,	k_{481} = f676f7d5$_x$,	k_{482} = ab3b7beb$_x$,	k_{483} = 859dbdf4$_x$,
k_{484} = 42cedefa$_x$,	k_{485} = 21676f7d$_x$,	k_{486} = c0b3b7bf$_x$,	k_{487} = b059dbde$_x$,
k_{488} = 582cedef$_x$,	k_{489} = fc1676f6$_x$,	k_{490} = 7e0b3b7b$_x$,	k_{491} = ef059dbc$_x$,
k_{492} = 7782cede$_x$,	k_{493} = 3bc1676f$_x$,	k_{494} = cde0b3b6$_x$,	k_{495} = 66f059db$_x$,
k_{496} = e3782cec$_x$,	k_{497} = 71bc1676$_x$,	k_{498} = 38de0b3b$_x$,	k_{499} = cc6f059c$_x$,
k_{500} = 663782ce$_x$,	k_{501} = 331bc167$_x$,	k_{502} = c98de0b2$_x$,	k_{503} = 64c6f059$_x$,
k_{504} = e263782d$_x$,	k_{505} = a131bc17$_x$,	k_{506} = 8098de0a$_x$,	k_{507} = 404c6f05$_x$,
k_{508} = f0263783$_x$,	k_{509} = a8131bc0$_x$,	k_{510} = 54098de0$_x$,	k_{511} = 2a04c6f0$_x$,
k_{512} = 15026378$_x$,	k_{513} = 0a8131bc$_x$,	k_{514} = 054098de$_x$,	k_{515} = 02a04c6f$_x$,
k_{516} = d1502636$_x$,	k_{517} = 68a8131b$_x$,	k_{518} = e454098c$_x$,	k_{519} = 722a04c6$_x$,
k_{520} = 39150263$_x$,	k_{521} = cc8a8130$_x$,	k_{522} = 66454098$_x$,	k_{523} = 3322a04c$_x$,

Table 14 – The constants used in LANE. *(continued)*

$k_{524} = \mathtt{19915026}_x$,	$k_{525} = \mathtt{0cc8a813}_x$,	$k_{526} = \mathtt{d6645408}_x$,	$k_{527} = \mathtt{6b322a04}_x$,
$k_{528} = \mathtt{35991502}_x$,	$k_{529} = \mathtt{1acc8a81}_x$,	$k_{530} = \mathtt{dd664541}_x$,	$k_{531} = \mathtt{beb322a1}_x$,
$k_{532} = \mathtt{8f599151}_x$,	$k_{533} = \mathtt{97acc8a9}_x$,	$k_{534} = \mathtt{9bd66455}_x$,	$k_{535} = \mathtt{9deb322b}_x$,
$k_{536} = \mathtt{9ef59914}_x$,	$k_{537} = \mathtt{4f7acc8a}_x$,	$k_{538} = \mathtt{27bd6645}_x$,	$k_{539} = \mathtt{c3deb323}_x$,
$k_{540} = \mathtt{b1ef5990}_x$,	$k_{541} = \mathtt{58f7acc8}_x$,	$k_{542} = \mathtt{2c7bd664}_x$,	$k_{543} = \mathtt{163deb32}_x$,
$k_{544} = \mathtt{0b1ef599}_x$,	$k_{545} = \mathtt{d58f7acd}_x$,	$k_{546} = \mathtt{bac7bd67}_x$,	$k_{547} = \mathtt{8d63deb2}_x$,
$k_{548} = \mathtt{46b1ef59}_x$,	$k_{549} = \mathtt{f358f7ad}_x$,	$k_{550} = \mathtt{a9ac7bd7}_x$,	$k_{551} = \mathtt{84d63dea}_x$,
$k_{552} = \mathtt{426b1ef5}_x$,	$k_{553} = \mathtt{f1358f7b}_x$,	$k_{554} = \mathtt{a89ac7bc}_x$,	$k_{555} = \mathtt{544d63de}_x$,
$k_{556} = \mathtt{2a26b1ef}_x$,	$k_{557} = \mathtt{c51358f6}_x$,	$k_{558} = \mathtt{6289ac7b}_x$,	$k_{559} = \mathtt{e144d63c}_x$,
$k_{560} = \mathtt{70a26b1e}_x$,	$k_{561} = \mathtt{3851358f}_x$,	$k_{562} = \mathtt{cc289ac6}_x$,	$k_{563} = \mathtt{66144d63}_x$,
$k_{564} = \mathtt{e30a26b0}_x$,	$k_{565} = \mathtt{71851358}_x$,	$k_{566} = \mathtt{38c289ac}_x$,	$k_{567} = \mathtt{1c6144d6}_x$,
$k_{568} = \mathtt{0e30a26b}_x$,	$k_{569} = \mathtt{d7185134}_x$,	$k_{570} = \mathtt{6b8c289a}_x$,	$k_{571} = \mathtt{35c6144d}_x$,
$k_{572} = \mathtt{cae30a27}_x$,	$k_{573} = \mathtt{b5718512}_x$,	$k_{574} = \mathtt{5ab8c289}_x$,	$k_{575} = \mathtt{fd5c6145}_x$,
$k_{576} = \mathtt{aeae30a3}_x$,	$k_{577} = \mathtt{87571850}_x$,	$k_{578} = \mathtt{43ab8c28}_x$,	$k_{579} = \mathtt{21d5c614}_x$,
$k_{580} = \mathtt{10eae30a}_x$,	$k_{581} = \mathtt{08757185}_x$,	$k_{582} = \mathtt{d43ab8c3}_x$,	$k_{583} = \mathtt{ba1d5c60}_x$,
$k_{584} = \mathtt{5d0eae30}_x$,	$k_{585} = \mathtt{2e875718}_x$,	$k_{586} = \mathtt{1743ab8c}_x$,	$k_{587} = \mathtt{0ba1d5c6}_x$,
$k_{588} = \mathtt{05d0eae3}_x$,	$k_{589} = \mathtt{d2e87570}_x$,	$k_{590} = \mathtt{69743ab8}_x$,	$k_{591} = \mathtt{34ba1d5c}_x$,
$k_{592} = \mathtt{1a5d0eae}_x$,	$k_{593} = \mathtt{0d2e8757}_x$,	$k_{594} = \mathtt{d69743aa}_x$,	$k_{595} = \mathtt{6b4ba1d5}_x$,
$k_{596} = \mathtt{e5a5d0eb}_x$,	$k_{597} = \mathtt{a2d2e874}_x$,	$k_{598} = \mathtt{5169743a}_x$,	$k_{599} = \mathtt{28b4ba1d}_x$,
$k_{600} = \mathtt{c45a5d0f}_x$,	$k_{601} = \mathtt{b22d2e86}_x$,	$k_{602} = \mathtt{59169743}_x$,	$k_{603} = \mathtt{fc8b4ba0}_x$,
$k_{604} = \mathtt{7e45a5d0}_x$,	$k_{605} = \mathtt{3f22d2e8}_x$,	$k_{606} = \mathtt{1f916974}_x$,	$k_{607} = \mathtt{0fc8b4ba}_x$,
$k_{608} = \mathtt{07e45a5d}_x$,	$k_{609} = \mathtt{d3f22d2f}_x$,	$k_{610} = \mathtt{b9f91696}_x$,	$k_{611} = \mathtt{5cfc8b4b}_x$,
$k_{612} = \mathtt{fe7e45a4}_x$,	$k_{613} = \mathtt{7f3f22d2}_x$,	$k_{614} = \mathtt{3f9f9169}_x$,	$k_{615} = \mathtt{cfcfc8b5}_x$,
$k_{616} = \mathtt{b7e7e45b}_x$,	$k_{617} = \mathtt{8bf3f22c}_x$,	$k_{618} = \mathtt{45f9f916}_x$,	$k_{619} = \mathtt{22fcfc8b}_x$,
$k_{620} = \mathtt{c17e7e44}_x$,	$k_{621} = \mathtt{60bf3f22}_x$,	$k_{622} = \mathtt{305f9f91}_x$,	$k_{623} = \mathtt{c82fcfc9}_x$,
$k_{624} = \mathtt{b417e7e5}_x$,	$k_{625} = \mathtt{8a0bf3f3}_x$,	$k_{626} = \mathtt{9505f9f8}_x$,	$k_{627} = \mathtt{4a82fcfc}_x$,
$k_{628} = \mathtt{25417e7e}_x$,	$k_{629} = \mathtt{12a0bf3f}_x$,	$k_{630} = \mathtt{d9505f9e}_x$,	$k_{631} = \mathtt{6ca82fcf}_x$,
$k_{632} = \mathtt{e65417e6}_x$,	$k_{633} = \mathtt{732a0bf3}_x$,	$k_{634} = \mathtt{e99505f8}_x$,	$k_{635} = \mathtt{74ca82fc}_x$,
$k_{636} = \mathtt{3a65417e}_x$,	$k_{637} = \mathtt{1d32a0bf}_x$,	$k_{638} = \mathtt{de99505e}_x$,	$k_{639} = \mathtt{6f4ca82f}_x$,
$k_{640} = \mathtt{e7a65416}_x$,	$k_{641} = \mathtt{73d32a0b}_x$,	$k_{642} = \mathtt{e9e99504}_x$,	$k_{643} = \mathtt{74f4ca82}_x$,
$k_{644} = \mathtt{3a7a6541}_x$,	$k_{645} = \mathtt{cd3d32a1}_x$,	$k_{646} = \mathtt{b69e9951}_x$,	$k_{647} = \mathtt{8b4f4ca9}_x$,
$k_{648} = \mathtt{95a7a655}_x$,	$k_{649} = \mathtt{9ad3d32b}_x$,	$k_{650} = \mathtt{9d69e994}_x$,	$k_{651} = \mathtt{4eb4f4ca}_x$,
$k_{652} = \mathtt{275a7a65}_x$,	$k_{653} = \mathtt{c3ad3d33}_x$,	$k_{654} = \mathtt{b1d69e98}_x$,	$k_{655} = \mathtt{58eb4f4c}_x$,
$k_{656} = \mathtt{2c75a7a6}_x$,	$k_{657} = \mathtt{163ad3d3}_x$,	$k_{658} = \mathtt{db1d69e8}_x$,	$k_{659} = \mathtt{6d8eb4f4}_x$,
$k_{660} = \mathtt{36c75a7a}_x$,	$k_{661} = \mathtt{1b63ad3d}_x$,	$k_{662} = \mathtt{ddb1d69f}_x$,	$k_{663} = \mathtt{bed8eb4e}_x$,
$k_{664} = \mathtt{5f6c75a7}_x$,	$k_{665} = \mathtt{ffb63ad2}_x$,	$k_{666} = \mathtt{7fdb1d69}_x$,	$k_{667} = \mathtt{efed8eb5}_x$,
$k_{668} = \mathtt{a7f6c75b}_x$,	$k_{669} = \mathtt{83fb63ac}_x$,	$k_{670} = \mathtt{41fdb1d6}_x$,	$k_{671} = \mathtt{20fed8eb}_x$,
$k_{672} = \mathtt{c07f6c74}_x$,	$k_{673} = \mathtt{603fb63a}_x$,	$k_{674} = \mathtt{301fdb1d}_x$,	$k_{675} = \mathtt{c80fed8f}_x$,
$k_{676} = \mathtt{b407f6c6}_x$,	$k_{677} = \mathtt{5a03fb63}_x$,	$k_{678} = \mathtt{fd01fdb0}_x$,	$k_{679} = \mathtt{7e80fed8}_x$,
$k_{680} = \mathtt{3f407f6c}_x$,	$k_{681} = \mathtt{1fa03fb6}_x$,	$k_{682} = \mathtt{0fd01fdb}_x$,	$k_{683} = \mathtt{d7e80fec}_x$,
$k_{684} = \mathtt{6bf407f6}_x$,	$k_{685} = \mathtt{35fa03fb}_x$,	$k_{686} = \mathtt{cafd01fc}_x$,	$k_{687} = \mathtt{657e80fe}_x$,
$k_{688} = \mathtt{32bf407f}_x$,	$k_{689} = \mathtt{c95fa03e}_x$,	$k_{690} = \mathtt{64afd01f}_x$,	$k_{691} = \mathtt{e257e80e}_x$,
$k_{692} = \mathtt{712bf407}_x$,	$k_{693} = \mathtt{e895fa02}_x$,	$k_{694} = \mathtt{744afd01}_x$,	$k_{695} = \mathtt{ea257e81}_x$,
$k_{696} = \mathtt{a512bf41}_x$,	$k_{697} = \mathtt{82895fa1}_x$,	$k_{698} = \mathtt{9144afd1}_x$,	$k_{699} = \mathtt{98a257e9}_x$,
$k_{700} = \mathtt{9c512bf5}_x$,	$k_{701} = \mathtt{9e2895fb}_x$,	$k_{702} = \mathtt{9f144afc}_x$,	$k_{703} = \mathtt{4f8a257e}_x$,
$k_{704} = \mathtt{27c512bf}_x$,	$k_{705} = \mathtt{c3e2895e}_x$,	$k_{706} = \mathtt{61f144af}_x$,	$k_{707} = \mathtt{e0f8a256}_x$,
$k_{708} = \mathtt{707c512b}_x$,	$k_{709} = \mathtt{e83e2894}_x$,	$k_{710} = \mathtt{741f144a}_x$,	$k_{711} = \mathtt{3a0f8a25}_x$,
$k_{712} = \mathtt{cd07c513}_x$,	$k_{713} = \mathtt{b683e288}_x$,	$k_{714} = \mathtt{5b41f144}_x$,	$k_{715} = \mathtt{2da0f8a2}_x$,
$k_{716} = \mathtt{16d07c51}_x$,	$k_{717} = \mathtt{db683e29}_x$,	$k_{718} = \mathtt{bdb41f15}_x$,	$k_{719} = \mathtt{8eda0f8b}_x$,
$k_{720} = \mathtt{976d07c4}_x$,	$k_{721} = \mathtt{4bb683e2}_x$,	$k_{722} = \mathtt{25db41f1}_x$,	$k_{723} = \mathtt{c2eda0f9}_x$,
$k_{724} = \mathtt{b176d07d}_x$,	$k_{725} = \mathtt{88bb683f}_x$,	$k_{726} = \mathtt{945db41e}_x$,	$k_{727} = \mathtt{4a2eda0f}_x$,
$k_{728} = \mathtt{f5176d06}_x$,	$k_{729} = \mathtt{7a8bb683}_x$,	$k_{730} = \mathtt{ed45db40}_x$,	$k_{731} = \mathtt{76a2eda0}_x$,
$k_{732} = \mathtt{3b5176d0}_x$,	$k_{733} = \mathtt{1da8bb68}_x$,	$k_{734} = \mathtt{0ed45db4}_x$,	$k_{735} = \mathtt{076a2eda}_x$,
$k_{736} = \mathtt{03b5176d}_x$,	$k_{737} = \mathtt{d1da8bb7}_x$,	$k_{738} = \mathtt{b8ed45da}_x$,	$k_{739} = \mathtt{5c76a2ed}_x$,
$k_{740} = \mathtt{fe3b5177}_x$,	$k_{741} = \mathtt{af1da8ba}_x$,	$k_{742} = \mathtt{578ed45d}_x$,	$k_{743} = \mathtt{fbc76a2f}_x$,
$k_{744} = \mathtt{ade3b516}_x$,	$k_{745} = \mathtt{56f1da8b}_x$,	$k_{746} = \mathtt{fb78ed44}_x$,	$k_{747} = \mathtt{7dbc76a2}_x$,
$k_{748} = \mathtt{3ede3b51}_x$,	$k_{749} = \mathtt{cf6f1da9}_x$,	$k_{750} = \mathtt{b7b78ed5}_x$,	$k_{751} = \mathtt{8bdbc76b}_x$,
$k_{752} = \mathtt{95ede3b4}_x$,	$k_{753} = \mathtt{4af6f1da}_x$,	$k_{754} = \mathtt{257b78ed}_x$,	$k_{755} = \mathtt{c2bdbc77}_x$,
$k_{756} = \mathtt{b15ede3a}_x$,	$k_{757} = \mathtt{58af6f1d}_x$,	$k_{758} = \mathtt{fc57b78f}_x$,	$k_{759} = \mathtt{ae2bdbc6}_x$,
$k_{760} = \mathtt{5715ede3}_x$,	$k_{761} = \mathtt{fb8af6f0}_x$,	$k_{762} = \mathtt{7dc57b78}_x$,	$k_{763} = \mathtt{3ee2bdbc}_x$,
$k_{764} = \mathtt{1f715ede}_x$,	$k_{765} = \mathtt{0fb8af6f}_x$,	$k_{766} = \mathtt{d7dc57b6}_x$,	$k_{767} = \mathtt{6bee2bdb}_x$.

Publication

Practical Collisions for EnRUPT

Publication Data

Conference version:

Sebastiaan Indesteege and Bart Preneel. Practical collisions for EnRUPT. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop — FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 246–259. Springer, 2009.

Extended journal version (included here):

Sebastiaan Indesteege and Bart Preneel. Practical collisions for EnRUPT. *Journal of Cryptology*, 2010. To appear in print. Published online at `http://dx.doi.org/10.1007/s00145-010-9058-x`.

Contributions

- Principal author.

Practical Collisions for EnRUPT[*]

Sebastiaan Indesteege[1,2,†] and Bart Preneel[1,2]

[1] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
`sebastiaan.indesteege@esat.kuleuven.be`

[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.

Abstract. The EnRUPT hash functions were proposed by O'Neil, Nohl and Henzen as candidates for the SHA-3 competition, organised by NIST. The proposal contains seven concrete hash functions, each with a different digest length. We present a practical collision attack on each of these seven EnRUPT variants. The time complexity of our attack varies from 2^{36} to 2^{40} round computations, depending on the EnRUPT variant, and the memory requirements are negligible. We demonstrate that our attack is practical by giving an actual collision example for EnRUPT-256.

Key words: EnRUPT, SHA-3 candidate, hash function, collision attack.

1 Introduction

Cryptographic hash functions are important cryptographic primitives that are employed in a vast number of applications, such as digital signatures and commitment schemes. They are expected to possess several security properties, one of which is *collision resistance*. Informally, collision resistance means that it should be hard to find two distinct messages $m \neq m'$ that hash to the same value, i.e., $h(m) = h(m')$.

Many popular hash functions, such as MD5, SHA-1 and SHA-2 share some common design principles. The recent advances in the cryptanalysis of these hash functions have raised serious concerns regarding their long-term security. This motivates the design of new hash functions, based on different design strategies. The National Institute of Standards and Technology (NIST) has decided to hold a public competition, the SHA-3 competition, to develop a new cryptographic hash function standard [6].

The EnRUPT hash functions were proposed by O'Neil, Nohl and Henzen [9] as candidates in this SHA-3 competition. The proposal contains seven concrete

[*]This paper is an extended version of [3].
[†]F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

Table 1 – EnRUPT parameters.

EnRUPT variant	digest length h	word size w	parallelisation level P	security parameter s	number of state words H
EnRUPT-128	128 bits	32 bits	2	4	8
EnRUPT-160	160 bits	32 bits	2	4	10
EnRUPT-192	192 bits	32 bits	2	4	12
EnRUPT-224	224 bits	64 bits	2	4	8
EnRUPT-256	256 bits	64 bits	2	4	8
EnRUPT-384	384 bits	64 bits	2	4	12
EnRUPT-512	512 bits	64 bits	2	4	16

EnRUPT variants, each with a different digest length. Khovratovich et al. [4] presented a theoretical preimage attack on EnRUPT, with a time complexity of 2^{480} and requiring about 2^{384} memory elements.

In this paper, we analyse EnRUPT and show that none of the proposed EnRUPT variants is collision resistant. We present a practical collision attack requiring only 2^{36} to 2^{40} EnRUPT round computations, depending on the EnRUPT variant. This is significantly less than the approximately $2^{n/2}$ hash computations required for a generic collision attack on an n-bit hash function, based on the birthday paradox.

The structure of this paper is as follows. A short description of EnRUPT is given in Sect. 2. Section 3 introduces the basic strategy used to find collisions for EnRUPT, which is based on the work on SHA by Chabaud and Joux [2] and Rijmen and Oswald [11]. Sections 4, 5 and 6 apply this basic attack strategy to EnRUPT, step by step. Our results, including an example collision for EnRUPT-256, are presented in Sect. 7. Finally, Sect. 8 concludes.

2 Description of EnRUPT

In this section, we give a short description of the seven EnRUPT variants that were proposed as SHA-3 candidates [9]. All share the same structure and use the same round function. The only differences lie in the parameters used. Table 1 gives the values of these parameters for each EnRUPT variant.

2.1 The EnRUPT Hash Functions

The structure shared by all EnRUPT hash functions can be split into four phases: preprocessing, message processing, finalisation and output. Figure 1 contains a description of the EnRUPT hash functions in pseudocode.

In the preprocessing phase (lines 2–4) the input message is padded to be a multiple of w bits, where w is the word size. Depending on the EnRUPT variant, the word size w is 32 or 64 bits, see Table 1. The padded message is then split into an integer number of w-bit words m_i.

The internal state of EnRUPT consists of several w-bit words: H state words x_i, P 'delta accumulators' d_i, and a round counter r. All of these are initialised to zero. The parameter P is equal to 2 for all seven EnRUPT variants. The value of H depends on the digest length, as indicated in Table 1.

Then, in the message processing phase (lines 5–8), the round function is called once for each w-bit padded message word m_i. Each call to the round function updates the internal state $\langle d, x, r\rangle$. A detailed description of the EnRUPT round function is given in the next section, Sect. 2.2.

After all message words have been processed, a finalisation is performed (lines 9–13). The EnRUPT round function is called once with the length of the (unpadded) message, represented as a w-bit unsigned integer[3]. Then, H blank rounds, i.e., calls to the round function with a zero message word input, are performed.

Finally, in the output phase (lines 14–18), the message digest is generated one w-bit word at a time. The EnRUPT round function is called h/w times and, after each call, the content of the 'delta accumulator' d_0 is output.

2.2 The EnRUPT Round Function

The EnRUPT round function is based entirely on a number of simple operations on words of w bits, such as bit shifts, bit rotations, exclusive OR and addition modulo 2^w. Figure 2 gives a description of the EnRUPT round function in pseudocode. The round function consists of $s \cdot P$ identical steps, where s and P are parameters of the hash function. As indicated in Table 1, $s = 4$ and $P = 2$ for all seven proposed EnRUPT variants. Thus, the EnRUPT round function consists of eight steps.

In each step, several words of the state are selected (lines 4–7) and combined into an intermediate value f (lines 9–10). Note that line 10 could equally be described as a multiplication with 9 modulo 2^w. The intermediate value f is then used to update one state word, x_γ, and one 'delta accumulator', $d_{i \bmod P}$ (lines 12–13).

After all steps have been performed, the round counter is incremented by the number of steps that were carried out, i.e., $s{\cdot}P$ (line 15). Finally, the input message word m is injected into one word of the internal state, the 'delta accumulator' d_{P-1} (line 16).

[3]Note that the EnRUPT specification [9] only states that the message length should be included, not how this is to be done exactly. The EnRUPT reference implementation uses one w-bit word for the message length, which implies that the EnRUPT variants for which $w = 32$ can only handle up to $2^{32} - 1$ bits in this implementation. Note that the results presented in this paper are independent of the details of the padding.

```
function EnRUPT ($M$)
    /* Preprocessing */
    $m_0, \cdots, m_t \leftarrow M \,||\, 1 \,||\, 0^{w-(|M|+1 \bmod w)}$   s.t.   $\forall i, 0 \le i \le t : |m_i| = w$
    $d_0, \cdots, d_{P-1}, x_0, \cdots, x_{H-1}, r \leftarrow 0, \cdots, 0$
    /* Message processing */
    for $i = 0$ to $n$ do
        $\langle d, x, r \rangle \leftarrow \mathrm{round}(\langle d, x, r \rangle, m_i)$
    end for
    /* Finalisation */
    $\langle d, x, r \rangle \leftarrow \mathrm{round}(\langle d, x, r \rangle, \mathrm{uint}_w(|M|))$
    for $i = 1$ to $H$ do
        $\langle d, x, r \rangle \leftarrow \mathrm{round}(\langle d, x, r \rangle, 0)$
    end for
    /* Output */
    for $i = 0$ to $h/w - 1$ do
        $\langle d, x, r \rangle \leftarrow \mathrm{round}(\langle d, x, r \rangle, 0)$
        $o_i \leftarrow d_0$
    end for
    return $o_0 \,||\, \cdots \,||\, o_{h/w-1}$
end function
```

Figure 1 – The EnRUPT hash function.

```
function round ($\langle d, x, r \rangle, m$)
    for $i = 0$ to $s \cdot P - 1$ do          /* An iteration of this loop is a "step" */
        /* Compute indices */
        $\alpha \leftarrow r + (i + 1 \bmod P) \bmod H$
        $\beta \leftarrow r + i + 2P \bmod H$
        $\gamma \leftarrow r + i + P \bmod H$
        $\xi \leftarrow r + i \bmod H$
        /* Compute intermediate f */
        $e \leftarrow ((x_\alpha \lll 1) \oplus x_\beta \oplus d_{i \bmod P} \oplus \mathrm{uint}_w(r + i)) \ggg w/4$
        $f \leftarrow (e \lll 3) \boxplus e$          /* Multiplication with 9 modulo $2^w$ */
        /* Update state */
        $x_\gamma \leftarrow x_\gamma \oplus f$
        $d_{i \bmod P} \leftarrow d_{i \bmod P} \oplus x_\xi \oplus f$
    end for
    $r \leftarrow r + s \cdot P$
    $d_{P-1} \leftarrow d_{P-1} \oplus m$          /* Message word injection */
    return $\langle d, x, r \rangle$
end function
```

Figure 2 – The EnRUPT round function.

3 Basic Attack Strategy

This section gives an overview of the linearisation method for finding collision differential characteristics for a hash function, which we use to attack EnRUPT in this work. This method was introduced by Chabaud and Joux [2], who applied it to SHA-0 and simplified variants thereof. Later, it was extended further and applied to SHA-1 by Rijmen and Oswald [11].

A Linear Hash Function. Consider a hypothetical hash function that consists only of linear operations over GF(2). When the input messages are restricted to a certain length, each output bit can be written as an affine function of the input bits. The *difference* in each output bit is given by a linear function of the differences in the input bits, as the constants (if any) cancel out. A message difference that leads to a collision can be found by equating the output differences to zero, and solving the resulting system of linear equations over GF(2), for instance using Gauss elimination. Any pair of messages with this difference will result in a collision.

Linearising a Nonlinear Hash Function. Actual cryptographic hash functions contain (also) nonlinear components, so this method no longer applies. However, we may still be able to *approximate* the nonlinear components by linear ones and construct a linear approximation of the entire hash function. For our purpose, a good linear approximation $\lambda(x)$ of a nonlinear function $\gamma(x)$ is such that its differential behaviour is close to that of $\gamma(x)$. More formally, the equation

$$\gamma(x \oplus \Delta) \oplus \gamma(x) = \lambda(x \oplus \Delta) \oplus \lambda(x) = \lambda(\Delta) \tag{1}$$

should hold for a relatively large fraction of values x. For instance, an addition modulo 2^w could be approximated by a simple XOR operation, i.e., ignoring the carries.

Finding Collisions. A *differential characteristic* consists of a message difference and a list of the differences in all (relevant) intermediate values. For the linear approximation, it is easy to find a differential characteristic that leads to a collision with probability one. But for the actual hash function, this probability will be (much) lower.

If the differential behaviour of all the nonlinear components corresponds to that of the linear approximations they were replaced with, i.e., if (1) holds simultaneously for each nonlinear component, we say that the differential characteristic is followed. In this case, the message pair under consideration will not only collide for the linearised hash function, but also for the original, nonlinear hash function. Such a message pair is called a *conforming* message pair.

Hence, a procedure for finding a collision for the nonlinear hash function could be to find a differential characteristic leading to collisions for a linearised variant of the hash function. Then, a message pair conforming to the differential characteristic is searched. In order to lower the complexity of the attack, it is important to maximise the probability that the differential characteristic is followed, i.e., we need to find a *good* differential characteristic.

4 Linearising EnRUPT

We now apply this general strategy to EnRUPT. Recall the description of the EnRUPT round function in Fig. 2. Note that only the modular addition in line 10 is not linear over GF(2). Indeed, the computation of the indices in lines 4–7 and the update of the round counter in line 15 do not depend on the message being hashed and can thus be precomputed. The same holds for the inclusion of the round counter in line 9, which can be seen as an XOR with a constant. The other operations are all linear over GF(2).

Replacing the modular addition in line 10 with an XOR operation yields a linearised round function, which we refer to as the EnRUPT-$\mathcal{L}$ round function. The EnRUPT-$\mathcal{L}$ hash function, i.e., the hash function built on this linearised round function, also consists solely of GF(2)-linear components.

5 The Collision Search

During the collision search phase, many collisions for EnRUPT-$\mathcal{L}$ are constructed, and a collision for EnRUPT is searched among them. Since only the modular additions (line 10 of Fig. 2) were approximated by XOR, these are the only places where the propagation of differences could differ between EnRUPT-$\mathcal{L}$ and EnRUPT. Instead of checking for a collision at the output, we can immediately check if the difference at the output of each modular addition, i.e., the difference Δf in the intermediate value f, still matches the differential characteristic.

5.1 An Observation on EnRUPT

We now make an important observation on the structure of the EnRUPT hash function. It is possible to find a conforming message pair for a given differential characteristic one round at a time.

Consider the message word m_i, which is injected into the 'delta accumulator' d_{P-1} at the end of round i. In the first $(P-1)$ steps of the next round, d_{P-1} is not used, so m_i can not influence the behaviour of the modular additions in these steps. Starting from the P-th step of round $(i+1)$, however, m_i does have an influence.

We can search for a value for m_i such that the differential characteristic is followed up to and including the first $(P-1)$ steps of round $(i+2)$. Starting with the P-th step of round $(i+2)$, the next message word, m_{i+1} also influences the modular additions. Thus, we can keep m_i fixed, and use the new freedom available in m_{i+1} to ensure that the differential characteristic is also followed for the next $s \cdot P$ steps.

This drastically reduces the expected number of trials required to find a collision. Let p_i denote the probability that the differential characteristic is followed in a block of $s \cdot P$ consecutive steps, starting at the P-th step of a round. Because we can construct a conforming message pair one word at a time, the expected number of trials is $\sum_i 1/p_i$ rather than $\prod_i 1/p_i$. In other words, the complexities associated with each block of $s \cdot P$ steps should be added together, rather than multiplied. This possibility was ignored in the security analysis of EnRUPT [9], leading to the incorrect conclusion that attacks based on linearisation do not apply.

5.2 Accelerating the Collision Search

A simple optimisation can be made to the collision search, which allows us to ignore the probability associated with one step in each round. This optimisation is analogous to Wang's 'single message modification', which was first introduced in the context of MD5 and other hash functions of the MD4-family [13].

Consider the P-th step of a round. In this step, the 'delta accumulator' d_{P-1}, to which a new message word m was XORed at the end of the previous round, is used for the first time. More precisely, it is used in line 9 of Fig. 2 to compute the intermediate value e. Note however that these computations can be inverted. We can choose the value of e, and compute backwards to find what the message word m should be to arrive at this value of e:

$$\begin{aligned} m &= d_{P-1} \oplus d_{P-1}^{\text{prev}} \\ &= (e \lll w/4) \oplus (x_\alpha \ll 1) \oplus x_\beta \oplus \text{uint}_w(r + P - 1) \oplus d_{P-1}^{\text{prev}} \ . \end{aligned} \tag{2}$$

Here, d_{P-1}^{prev} is the (known) value of d_{P-1} in the previous round, just before the message word m was added to it.

The values of e which ensure that the difference propagation of the modular addition in line 10 of Fig. 2 corresponds to that of its linear approximation can be efficiently enumerated as follows. Consider a binary tree representing all possible values for e. Each layer of the tree determines one more bit of e, starting from the least significant bit. This tree is walked in a depth-first fashion, backtracking as soon as the difference propagation is not as desired. Indeed, the difference propagation in the lower bits does not depend on the more significant bits, so this backtracking strategy effectively skips over all bad values for e.

Thus, rather than randomly picking values for m, we can efficiently sample *good* values for e in this step, and compute backwards to find the corresponding m. This ensures that the first modular addition affected by a message word m will always exhibit the desired propagation of differences. Thus, the P-th step of every round can be ignored in the estimation of the complexity of the attack.

6 Finding Good Differential Characteristics

The key to lowering the attack complexity is to find a good differential characteristic, i.e., a characteristic which is likely to be followed for the nonlinear hash function. A generic approach to this problem, based on finding low weight codewords in a linear code, was proposed by Rijmen and Oswald [11] and extended by Pramstaller et al. in [10]. In this section, we show how to apply this approach to EnRUPT.

6.1 Coding Theory

As observed by Rijmen and Oswald [11], all of the differential characteristics leading to a collision for the linearised hash function can be seen as the codewords of a linear code.

Consider the EnRUPT-$\mathcal{L}$ hash function with a h-bit output length, and the message input restricted to messages of t message words. Since it is affine over GF(2), it is possible to express the difference in the output as a linear function of the difference in the input message m:

$$[\Delta o]_{1\times h} = [\Delta m]_{1\times tw} \cdot [\mathbf{O}]_{tw\times h} \quad . \tag{3}$$

As the modular additions, or rather the multiplications with 9, in the EnRUPT round function are approximated, we are also interested in the differences that enter each of these operations. For EnRUPT restricted to t message blocks, there are $t\cdot s\cdot P$ such operations in total. Hence, we can combine the input differences to these operations in a $1 \times tsPw$ bit vector Δe. Again, for the linear approximation, Δe is simply a linear function of the message difference Δm:

$$[\Delta e]_{1\times tsPw} = [\Delta m]_{1\times tw} \cdot [\mathbf{E}]_{tw\times tsPw} \quad . \tag{4}$$

Putting this together results in a linear code described by the following generator matrix

$$\mathbf{G} = \left[\ \mathbf{I}_{tw\times tw} \mid \mathbf{E}_{tw\times tsPw} \mid \mathbf{O}_{tw\times h}\ \right] \quad . \tag{5}$$

Each codeword contains a message difference, the input differences to all approximated modular additions, and finally the output difference.

Thus, each codeword is in fact a differential characteristic for EnRUPT-$\mathcal{L}$, and all differential characteristics for EnRUPT-$\mathcal{L}$ are codewords of this code. To restrict ourselves to collision differentials, i.e., differential characteristics ending in a zero output difference, we can use Gauss elimination to force the h rightmost columns of the generator matrix G to zero.

It is well known that the differential behaviour of modular addition can be well approximated by that of XOR when the Hamming weight of the input difference, ignoring the most significant bit, is small [2, 5, 10, 11]. As the input differences to the modular additions are part of the codewords, we will attempt to find a codeword with a low Hamming weight in this part of the codeword.

6.2 Low Weight Codewords

To find low weight codewords, we used a simple and straightforward algorithm that is based on the assumption that a codeword of very low weight exists in the code. For our purposes, this is a reasonable assumption, as only a very low weight codeword will lead to an attack faster than a generic attack. The algorithm is related to the algorithm of Canteaut and Chabaud [1] and the algorithm used to find low weight codewords for linearised SHA-1 by Pramstaller et al. [10].

Let G be the generator matrix of the linear code as in (5). We randomly select a set I of (appropriate) columns of the generator matrix G and force them to zero using Gauss elimination, until only d rows remain, where d is a parameter of the algorithm. Then, the remaining space of 2^d codewords is searched exhaustively. This procedure is repeated until a codeword of sufficiently low weight is encountered. By replacing only the 'oldest' column(s) in I, instead of restarting from the beginning every time, the algorithm can be implemented efficiently in practice.

If a codeword of very low weight exists in the code, it is likely that all of the columns in the randomly constructed set I will coincide with zeroes in the codeword, which implies that the codeword will be found in the exhaustive search phase. In the case of the codes originating from the seven linearised EnRUPT variants we consider, this algorithm finds a codeword of very low weight in a matter of minutes on a PC. Repeated runs of the algorithm always find the same codewords, so it is reasonable to assume that these are indeed the best codewords we can find.

6.3 Estimating the Attack Complexity

Actually, the weight of a codeword is only a heuristic for the attack complexity resulting from the corresponding differential. Codewords with a lower weight are expected to result in a lower attack complexity, but we can easily enhance our algorithm to optimise the actual attack complexity, rather than just a crude heuristic.

The Differential Probability. The probability that a differential characteristic is followed, is determined by the differences that enter each of the multiplications with 9 (line 10 in Fig. 2), which were approximated using XOR operations. Denote by $\mathrm{DP}^{\times 9}(\Delta)$ the probability that the propagation of differences through this nonlinear operation coincides with that of its linear approximation:

$$\mathrm{DP}^{\times 9}(\Delta) = \Pr_x\Big[(x \times 9) \oplus ((x \oplus \Delta) \times 9) = \Delta \oplus (\Delta \ll 3)\Big] \ . \tag{6}$$

The differential probability of modular addition was studied by Lipmaa and Moriai [5]. Applying their results to this situation, and taking into account that the three least significant bits of $(x \lll 3)$ are always zero, we find the following estimate for $\mathrm{DP}^{\times 9}(\Delta)$:

$$\mathrm{DP}^{\times 9}(\Delta) \approx 2^{-\mathrm{wt}\left(\left(\Delta \vee (\Delta \lll 3)\right) \wedge 0111 \cdots 111000_b\right)} \ . \tag{7}$$

Even though this estimate ignores the dependency between x and $(x \lll 3)$, this confirms the intuition that a difference Δ with a low Hamming weight (ignoring the most significant bit and the three least significant bits) results in a large probability $\mathrm{DP}^{\times 9}(\Delta)$. We used this as a heuristic to find a good differential characteristic: we want to minimise the Hamming weight of the relevant parts of the differences that are input to the modular additions. In other words, we want to find a low weight codeword of the aforementioned linear code, where only the bits that impact $\mathrm{DP}^{\times 9}(\Delta)$ are counted.

Exact Computation of the Differential Probability. Computing the exact value of $\mathrm{DP}^{\times 9}(\Delta)$ for any given difference Δ can be done by counting all the values x for which the difference propagation is as predicted by the linear approximation. We now show how this can be done in an efficient way. While this is very useful for evaluating the precise attack complexity, it lacks the clear intuition we can gather from (7).

For w-bit words, the definition of $\mathrm{DP}^{\times 9}(\Delta)$ given in (6) can be restated as

$$\mathrm{DP}^{\times 9}(\Delta) = \frac{\#\left\{x \in \{0,1\}^w \mid (x \times 9) \oplus ((x \oplus \Delta) \times 9) = \Delta \oplus (\Delta \lll 3)\right\}}{2^w} \ . \tag{8}$$

Now, consider how the computation of $y = (x \times 9) = x \oplus (x \lll 3)$ is performed at the bit level. Let x_i denote the i-th bit of x, where x_0 is the least significant bit. Then the following equations can be derived:

$$\left\{ \begin{array}{rcl} y_i & = & x_i \oplus x_{i-3} \oplus c_i \\ c_{i+1} & = & \mathrm{maj}(x_i, x_{i-3}, c_i) \end{array} \right. \ . \tag{9}$$

Here, the bits c_i represent the carry bits in the modular addition. By definition, the first carry bit c_0 is zero. The majority function $\mathrm{maj}(\cdot)$ is defined by

$$\mathrm{maj}(a,b,c) = ab \oplus bc \oplus ac \ . \tag{10}$$

Let Δx_i be the XOR difference in the i-th bit of x, and similar for other differences. Then, we find

$$\left\{ \begin{array}{rcl} \Delta y_i & = & \Delta x_i \oplus \Delta x_{i-3} \oplus \Delta c_i \\ \Delta c_{i+1} & = & \mathrm{maj}(x_i, x_{i-3}, c_i) \oplus \mathrm{maj}(x_i \oplus \Delta x_i, x_{i-3} \oplus \Delta x_{i-3}, c_i \oplus \Delta c_i) \end{array} \right. \ . \tag{11}$$

Since the output difference of the multiplication is approximated by $\Delta x \oplus (\Delta x \lll 3)$, it follows from the first equation of (11) that we require $\Delta c_i = 0$ for $0 \leq i < w$.

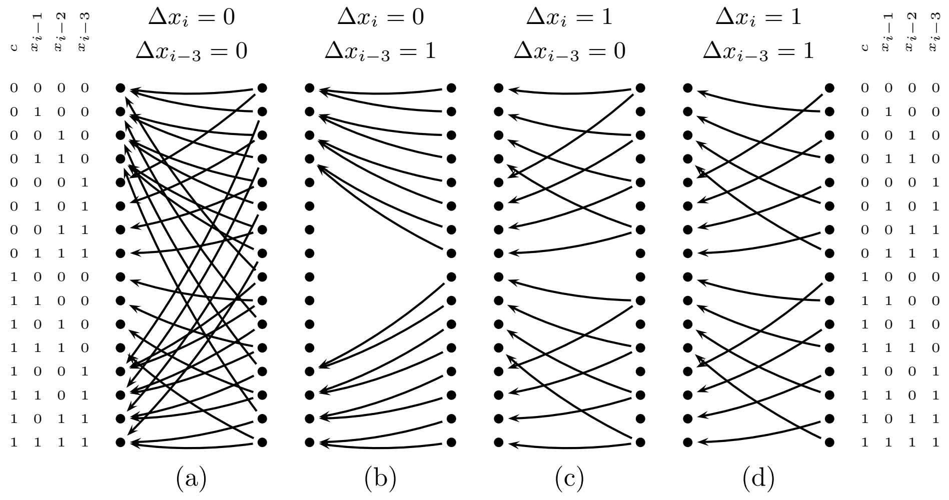

c x_{i-1} x_{i-2} x_{i-3}
$\Delta x_i = 0$ $\Delta x_{i-3} = 0$
$\Delta x_i = 0$ $\Delta x_{i-3} = 1$
$\Delta x_i = 1$ $\Delta x_{i-3} = 0$
$\Delta x_i = 1$ $\Delta x_{i-3} = 1$
0 0 0 0
0 1 0 0
0 0 1 0
0 1 1 0
0 0 0 1
0 1 0 1
0 0 1 1
0 1 1 1
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 0
1 0 0 1
1 1 0 1
1 0 1 1
1 1 1 1
(a)
(b)
(c)
(d)

Figure 3 – Trellis segments used in the calculation of $\text{DP}^{\times 9}$.

Note that only the knowledge of x_i, x_{i-3} and c_i is required to evaluate (11) when the input difference Δx is fixed, and the carry bits have no difference.

Hence, this can be represented efficiently in a trellis where the computations relating to one bit slice are represented by one segment in the trellis. Each node in the trellis represents, for a certain bit position, the values of the input carry c_i and the values of the three most recent bits, x_{i-1}, x_{i-2} and x_{i-3}. Since the differences in x are fixed a priori, and we do not allow differences in the carry, it is possible to compute the arcs in the segment as for each value of the bit x_i, the next node can be computed. Indeed, the next node is identified by c_{i+1}, which can be computed from (9) and x_i, x_{i-1} and x_{i-2}. Note however that, except for the most significant bit, we require $\Delta c_{i+1} = 0$ for our approximation to hold. This can be checked using (11), and only arcs satisfying this condition are kept. The full trellis is the concatenation of segments out of the four possibilities shown in Figure 3. The input difference Δx, which is fixed, determines which segments are used. Note that the trellis segment for the most significant bit contains all arcs regardless of Δx, as the condition prohibiting an output carry differences is no longer required there.

For each value of x for which no carry differences occur, there is a path in the trellis starting at the node $\langle 0, 0, 0, 0\rangle$ at the input of least significant bit slice, which we refer to as the source node, and ending at one of the nodes at the output of the most significant bit slice, the goal nodes. Hence, we can evaluate (8) by simply counting the number of such paths. This can be done using an algorithm that bears similarity to the Viterbi algorithm [12] and is an example of dynamic programming. Let $f(N)$ denote the number of paths from the source node to a node N. It is straightforward to see that this is equal to

$$f(N) = \sum_{\mathrm{arc}(X,N)} f(X) \ , \tag{12}$$

where $\mathrm{arc}(X, N)$ denotes that there is an arc from node X to node N. For the source node S, we initialise $f(S) = 1$. Then the recurrence (12) can be used to evaluate the number of paths to all nodes in the trellis. Finally, the sum of the number of paths to each of the goal nodes allows to compute $\mathrm{DP}^{\times 9}$ using (8).

Computing the Attack Complexity. Let $p_{r,i}$ be the differential probability associated with the modular addition in step i of round r of the differential characteristic. Recall the observation made in Sect. 5.1, i.e., finding a conforming message pair can be done one round at a time, or rather one message word at a time, as this does not coincide precisely with the round boundaries. Taking this into account, the complexity of finding the j-th word of a conforming message pair can thus be computed as

$$C_j = \left(\prod_{i=P-1}^{sP-1} \frac{1}{p_{j+1,i}} \right) \left(\prod_{i=0}^{P-2} \frac{1}{p_{j+2,i}} \right) \ . \tag{13}$$

Due to the acceleration technique presented in Sect. 5.2, we are guaranteed that the differential behaviour of the modular addition in step $P-1$ of each round will be as desired. Thus, we can set $p_{r,P-1} = 1$. With the default EnRUPT parameters ($P = 2$ and $s = 4$, see Table 1), this then becomes

$$C_j = \frac{1}{p_{j+1,2}} \cdot \frac{1}{p_{j+1,3}} \cdot \frac{1}{p_{j+1,4}} \cdot \frac{1}{p_{j+1,5}} \cdot \frac{1}{p_{j+1,6}} \cdot \frac{1}{p_{j+1,7}} \cdot \frac{1}{p_{j+2,0}} \ . \tag{14}$$

Finally, as was explained in Sect. 5.1, note that each message word can be found independently of the previous ones, due to the newly available degrees of freedom in each message word. Hence, the overall attack complexity can simply be computed as the sum of these round complexities:

$$C_{\mathrm{tot}} = \sum_{j=0}^{t} C_j \ . \tag{15}$$

Note that, given a differential characteristic, it is easy to compute the associated attack complexity. Hence, when searching for a good differential characteristic using the algorithm described in Sect. 6.2, we can use the actual attack complexity instead of the weight of the codeword. The algorithm still implicitly uses the weight of a codeword as a heuristic, but now attempts to optimise the actual attack complexity directly.

Table 2 – Summary of our attacks. Only the best attack is listed for each EnRUPT variant.

EnRUPT variant	word size w [bits]	state size H [words]	estimated time complexity [EnRUPT rounds]	length of differential [message words]
EnRUPT-128	32	8	$2^{36.04}$	6
EnRUPT-160	32	10	$2^{37.78}$	7
EnRUPT-192	32	12	$2^{38.33}$	8
EnRUPT-224	64	8	$2^{37.02}$	6
EnRUPT-256	64	8	$2^{37.02}$	6
EnRUPT-384	64	12	$2^{39.63}$	8
EnRUPT-512	64	16	$2^{38.46}$	10

7 Results and Discussion

We constructed differential characteristics for each of the seven EnRUPT variants in the EnRUPT SHA-3 proposal [9]. Table 2 lists the attack complexity and the length of the best characteristic we found for each variant. For the sake of clarity, the key parameters of each EnRUPT variant are repeated from Table 1. Recall that we fixed the length of the characteristic a priori. Note however that nothing prevents our search algorithm from proposing a shorter characteristic, padded with rounds without any difference, which we also observed in practice. We experimented with (much) longer maximum characteristic lengths, but found no better long characteristics.

The time complexities vary from 2^{36} to 2^{40} round computations, depending on the EnRUPT variant, which is remarkable. It means that the collision resistance in absolute terms of each of these EnRUPT variants is more or less the same, regardless of the digest length. Relative to the expected collision resistance of approximately $2^{n/2}$ for an n-bit hash function, however, the (relative) collision resistance of EnRUPT is much worse for the variants with a longer digest length than for those with a shorter digest length.

Tables 4–9 list our differential characteristics for each of the seven EnRUPT variants. Note that the same characteristic applies to EnRUPT-224 and EnRUPT-256 (Table 7) as both functions share the same parameter settings, see Table 1.

The format of these tables is as follows. Each line in the table corresponds to one step of the EnRUPT round function. The difference in the input (Δe) and the output (Δf) of the multiplication with 9 in that step is indicated. Also, the message word differences are shown at the end of each round. Note that the word size is 32 bits for some EnRUPT variants, and 64 bits for others. The table also includes the differential probabilities of each step, which were used to compute the

Table 3 – A collision example for EnRUPT-256.

M	13$_x$	c8$_x$	4b$_x$	45$_x$	62$_x$	70$_x$	17$_x$	6e$_x$
	04$_x$	f9$_x$	31$_x$	7e$_x$	c3$_x$	6c$_x$	e7$_x$	d3$_x$
	e1$_x$	21$_x$	78$_x$	6a$_x$	34$_x$	74$_x$	11$_x$	19$_x$
	7f$_x$	64$_x$	a3$_x$	c9$_x$	40$_x$	07$_x$	75$_x$	76$_x$
	a1$_x$	4f$_x$	90$_x$	86$_x$	fd$_x$	c7$_x$	33$_x$	4a$_x$
	41$_x$	3a$_x$	76$_x$	91$_x$	96$_x$	06$_x$	2c$_x$	a1$_x$.
M'	13$_x$	c8$_x$	4b$_x$	45$_x$	6a$_x$	70$_x$	17$_x$	6e$_x$
	04$_x$	f9$_x$	31$_x$	5c$_x$	43$_x$	6c$_x$	e7$_x$	d3$_x$
	e1$_x$	21$_x$	78$_x$	48$_x$	bc$_x$	74$_x$	11$_x$	19$_x$
	7f$_x$	64$_x$	a3$_x$	cb$_x$	48$_x$	07$_x$	75$_x$	76$_x$
	a1$_x$	4f$_x$	90$_x$	84$_x$	fd$_x$	c7$_x$	33$_x$	4a$_x$
	41$_x$	3a$_x$	76$_x$	93$_x$	96$_x$	06$_x$	2c$_x$	a1$_x$.
EnRUPT-256(M) =	bd$_x$	67$_x$	51$_x$	7c$_x$	a6$_x$	c0$_x$	41$_x$	20$_x$
EnRUPT-256(M') =	82$_x$	e0$_x$	3b$_x$	74$_x$	5f$_x$	fc$_x$	4a$_x$	64$_x$
	e9$_x$	f0$_x$	92$_x$	c2$_x$	58$_x$	c3$_x$	98$_x$	b8$_x$
	44$_x$	9a$_x$	fe$_x$	cb$_x$	7f$_x$	c8$_x$	6f$_x$	72$_x$.

attack complexity. A star ('$\star$') indicates that the differential probability can be ignored in that step because of the technique presented in Sect. 5.2. The product of the step probabilities is given for eight consecutive steps. Note that these do not coincide with the rounds, as was discussed in Sect. 6.3.

A collision example for EnRUPT-256, obtained using the characteristic from Table 7, is given in Table 3. This example was computed on a cluster of AMD Opteron 250 processors running at 2.4 GHz. The total computational effort was 237 CPU-days. This is roughly 1000 times what one would expect if one were to count just the time spent doing EnRUPT rounds using an optimised implementation of EnRUPT. This discrepancy is explained by a lack of optimisation of the implementation of the attack algorithm, a simple but somewhat wasteful approach to parallelisation, and a considerable amount of redundant work due to a bug in the initial attack implementation. However, such practical issues are to be expected in a first implementation, and we opted to balance the programming effort and CPU time, rather than pursuing the fastest possible implementation.

Table 4 – Differential characteristic for EnRUPT-128.

Round	Step	Δe	$\longrightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
inject message word difference $\Delta m_{-1} = \texttt{00000800}_x$						
0	0	$\texttt{00000000}_x$	$\longrightarrow$	$\texttt{00000000}_x$	$2^{-0.00}$	$\mathbf{2^{-0.00}}$
	1	$\texttt{00000008}_x$	$\longrightarrow$	$\texttt{00000048}_x$	$\star$	
	2	$\texttt{90000000}_x$	$\longrightarrow$	$\texttt{10000000}_x$	$2^{-0.85}$	
	3	$\texttt{48000008}_x$	$\longrightarrow$	$\texttt{08000048}_x$	$2^{-3.85}$	
	4	$\texttt{90000000}_x$	$\longrightarrow$	$\texttt{10000000}_x$	$2^{-0.85}$	
	5	$\texttt{48280008}_x$	$\longrightarrow$	$\texttt{09680048}_x$	$2^{-6.92}$	
	6	$\texttt{9002d000}_x$	$\longrightarrow$	$\texttt{10145000}_x$	$2^{-6.43}$	
	7	$\texttt{00296808}_x$	$\longrightarrow$	$\texttt{01622848}_x$	$2^{-10.39}$	
inject message word difference $\Delta m_{0} = \texttt{00228000}_x$						
1	0	$\texttt{9002d000}_x$	$\longrightarrow$	$\texttt{10145000}_x$	$2^{-6.43}$	$\mathbf{2^{-35.72}}$
	1	$\texttt{00296800}_x$	$\longrightarrow$	$\texttt{01622800}_x$	$\star$	
	2	$\texttt{9002d000}_x$	$\longrightarrow$	$\texttt{10145000}_x$	$2^{-6.43}$	
	3	$\texttt{48280000}_x$	$\longrightarrow$	$\texttt{09680000}_x$	$2^{-4.92}$	
	4	$\texttt{9002d000}_x$	$\longrightarrow$	$\texttt{10145000}_x$	$2^{-6.43}$	
	5	$\texttt{00080000}_x$	$\longrightarrow$	$\texttt{00480000}_x$	$2^{-1.85}$	
	6	$\texttt{90024000}_x$	$\longrightarrow$	$\texttt{10104000}_x$	$2^{-3.69}$	
	7	$\texttt{48092000}_x$	$\longrightarrow$	$\texttt{08402000}_x$	$2^{-5.71}$	
inject message word difference $\Delta m_{1} = \texttt{00228800}_x$						
2	0	$\texttt{90024000}_x$	$\longrightarrow$	$\texttt{10104000}_x$	$2^{-3.69}$	$\mathbf{2^{-32.73}}$
	1	$\texttt{00084800}_x$	$\longrightarrow$	$\texttt{004a0800}_x$	$\star$	
	2	$\texttt{90024000}_x$	$\longrightarrow$	$\texttt{10104000}_x$	$2^{-3.69}$	
	3	$\texttt{48096800}_x$	$\longrightarrow$	$\texttt{08422800}_x$	$2^{-8.45}$	
	4	$\texttt{90024000}_x$	$\longrightarrow$	$\texttt{10104000}_x$	$2^{-3.69}$	
	5	$\texttt{00200000}_x$	$\longrightarrow$	$\texttt{01200000}_x$	$2^{-1.85}$	
	6	$\texttt{90000000}_x$	$\longrightarrow$	$\texttt{10000000}_x$	$2^{-0.85}$	
	7	$\texttt{48200000}_x$	$\longrightarrow$	$\texttt{09200000}_x$	$2^{-3.26}$	
inject message word difference $\Delta m_{2} = \texttt{00020800}_x$						
3	0	$\texttt{90000000}_x$	$\longrightarrow$	$\texttt{10000000}_x$	$2^{-0.85}$	$\mathbf{2^{-22.65}}$
	1	$\texttt{00292000}_x$	$\longrightarrow$	$\texttt{01602000}_x$	$\star$	
	2	$\texttt{90009000}_x$	$\longrightarrow$	$\texttt{10041000}_x$	$2^{-3.70}$	
	3	$\texttt{48296800}_x$	$\longrightarrow$	$\texttt{09622800}_x$	$2^{-9.68}$	
	4	$\texttt{90009000}_x$	$\longrightarrow$	$\texttt{10041000}_x$	$2^{-3.70}$	
	5	$\texttt{00084800}_x$	$\longrightarrow$	$\texttt{004a0800}_x$	$2^{-4.59}$	
	6	$\texttt{90009000}_x$	$\longrightarrow$	$\texttt{10041000}_x$	$2^{-3.70}$	
	7	$\texttt{48080000}_x$	$\longrightarrow$	$\texttt{08480000}_x$	$2^{-3.71}$	
inject message word difference $\Delta m_{3} = \texttt{00020000}_x$						
4	0	$\texttt{90009000}_x$	$\longrightarrow$	$\texttt{10041000}_x$	$2^{-3.70}$	$\mathbf{2^{-32.76}}$
	1	$\texttt{00080008}_x$	$\longrightarrow$	$\texttt{00480048}_x$	$\star$	
	2	$\texttt{00000000}_x$	$\longrightarrow$	$\texttt{00000000}_x$	$2^{-0.00}$	
	3	$\texttt{00080008}_x$	$\longrightarrow$	$\texttt{00480048}_x$	$2^{-3.85}$	
	4	$\texttt{00000000}_x$	$\longrightarrow$	$\texttt{00000000}_x$	$2^{-0.00}$	
	5	$\texttt{48084808}_x$	$\longrightarrow$	$\texttt{084a0848}_x$	$2^{-8.47}$	

Table 4 – Differential characteristic for EnRUPT-128. *(continued)*

Round	Step	Δe	$\rightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
	6	$\texttt{00000000}_x$	$\rightarrow$	$\texttt{00000000}_x$	$2^{-0.00}$	
	7	$\texttt{48084808}_x$	$\rightarrow$	$\texttt{084a0848}_x$	$2^{-8.47}$	
inject message word difference $\Delta m_4 = \texttt{00020000}_x$						
5	0	$\texttt{00000000}_x$	$\rightarrow$	$\texttt{00000000}_x$	$2^{-0.00}$	$\mathbf{2^{-20.79}}$
	1	$\texttt{00000000}_x$	$\rightarrow$	$\texttt{00000000}_x$	$\star$	
	$\vdots$	$\vdots$	$\rightarrow$	$\vdots$	$\vdots$	
	7	$\texttt{00000000}_x$	$\rightarrow$	$\texttt{00000000}_x$	$2^{-0.00}$	

Table 5 – Differential characteristic for EnRUPT-160.

Round	Step	Δe	$\rightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
inject message word difference $\Delta m_{-1} = 00000400_x$						
0	0	00000000_x	$\rightarrow$	00000000_x	$2^{-0.00}$	$\mathbf{2^{-0.00}}$
	1	00000004_x	$\rightarrow$	00000024_x	$\star$	
	2	48000000_x	$\rightarrow$	08000000_x	$2^{-1.85}$	
	3	24000004_x	$\rightarrow$	04000024_x	$2^{-2.85}$	
	4	48000000_x	$\rightarrow$	08000000_x	$2^{-1.85}$	
	5	24140004_x	$\rightarrow$	$04b40024_x$	$2^{-5.92}$	
	6	48016800_x	$\rightarrow$	$080a2800_x$	$2^{-7.43}$	
	7	$2414b404_x$	$\rightarrow$	$04b11424_x$	$2^{-10.70}$	
inject message word difference $\Delta m_0 = 00114000_x$						
1	0	48016800_x	$\rightarrow$	$080a2800_x$	$2^{-7.43}$	$\mathbf{2^{-38.04}}$
	1	$0014b400_x$	$\rightarrow$	$00b11400_x$	$\star$	
	2	00016800_x	$\rightarrow$	$000a2800_x$	$2^{-5.59}$	
	3	$2414b400_x$	$\rightarrow$	$04b11400_x$	$2^{-9.68}$	
	4	48016800_x	$\rightarrow$	$080a2800_x$	$2^{-7.43}$	
	5	24000000_x	$\rightarrow$	04000000_x	$2^{-1.85}$	
	6	48000000_x	$\rightarrow$	08000000_x	$2^{-1.85}$	
	7	$0000b400_x$	$\rightarrow$	00051400_x	$2^{-5.59}$	
inject message word difference $\Delta m_1 = 00114400_x$						
2	0	48016800_x	$\rightarrow$	$080a2800_x$	$2^{-7.43}$	$\mathbf{2^{-39.41}}$
	1	$0004b400_x$	$\rightarrow$	00211400_x	$\star$	
	2	48012000_x	$\rightarrow$	08082000_x	$2^{-4.69}$	
	3	00042400_x	$\rightarrow$	00250400_x	$2^{-4.59}$	
	4	00012000_x	$\rightarrow$	00082000_x	$2^{-2.85}$	
	5	24042400_x	$\rightarrow$	04250400_x	$2^{-6.45}$	
	6	48012000_x	$\rightarrow$	08082000_x	$2^{-4.69}$	
	7	24042400_x	$\rightarrow$	04250400_x	$2^{-6.45}$	
inject message word difference $\Delta m_2 = 00000000_x$						
3	0	48004800_x	$\rightarrow$	08020800_x	$2^{-4.70}$	$\mathbf{2^{-34.42}}$
	1	$0010b400_x$	$\rightarrow$	00951400_x	$\star$	
	2	48004800_x	$\rightarrow$	08020800_x	$2^{-4.70}$	
	3	00100000_x	$\rightarrow$	00900000_x	$2^{-1.85}$	
	4	48000000_x	$\rightarrow$	08000000_x	$2^{-1.85}$	
	5	$0014b400_x$	$\rightarrow$	$00b11400_x$	$2^{-8.36}$	
	6	00004800_x	$\rightarrow$	00020800_x	$2^{-2.85}$	
	7	$2414b400_x$	$\rightarrow$	$04b11400_x$	$2^{-9.68}$	
inject message word difference $\Delta m_3 = 00010400_x$						
4	0	48004800_x	$\rightarrow$	08020800_x	$2^{-4.70}$	$\mathbf{2^{-33.99}}$
	1	24002400_x	$\rightarrow$	04010400_x	$\star$	
	2	48004800_x	$\rightarrow$	08020800_x	$2^{-4.70}$	
	3	00002400_x	$\rightarrow$	00010400_x	$2^{-2.85}$	
	4	48004800_x	$\rightarrow$	08020800_x	$2^{-4.70}$	
	5	00042400_x	$\rightarrow$	00250400_x	$2^{-4.59}$	

Table 5 – Differential characteristic for EnRUPT-160. *(continued)*

Round	Step	Δe	$\rightarrow$	Δf	DP$^{\times 9}$	totals
	6	48004800_x	$\rightarrow$	08020800_x	$2^{-4.70}$	
	7	00042400_x	$\rightarrow$	00250400_x	$2^{-4.59}$	
inject message word difference $\Delta m_4 = 00010000_x$						
5	0	00000000_x	$\rightarrow$	00000000_x	$2^{-0.00}$	$\mathbf{2^{-26.12}}$
	1	24042404_x	$\rightarrow$	04250424_x	$\star$	
	2	00000000_x	$\rightarrow$	00000000_x	$2^{-0.00}$	
	3	00040004_x	$\rightarrow$	00240024_x	$2^{-2.85}$	
	4	00000000_x	$\rightarrow$	00000000_x	$2^{-0.00}$	
	5	24042404_x	$\rightarrow$	04250424_x	$2^{-7.47}$	
	6	00000000_x	$\rightarrow$	00000000_x	$2^{-0.00}$	
	7	24042404_x	$\rightarrow$	04250424_x	$2^{-7.47}$	
inject message word difference $\Delta m_5 = 00010000_x$						
6	0	00000000_x	$\rightarrow$	00000000_x	$2^{-0.00}$	$\mathbf{2^{-17.79}}$
	1	00000000_x	$\rightarrow$	00000000_x	$\star$	
	$\vdots$	$\vdots$	$\rightarrow$	$\vdots$	$\vdots$	
	7	00000000_x	$\rightarrow$	00000000_x	$2^{-0.00}$	

Table 6 – Differential characteristic for EnRUPT-192.

Round	Step	Δe	$\longrightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
inject message word difference $\Delta m_{-1} = 00000800_x$						
0	0	00000000_x	$\longrightarrow$	00000000_x	$2^{-0.00}$	$\mathbf{2^{-0.00}}$
	1	00000008_x	$\longrightarrow$	00000048_x	$\star$	
	2	90000000_x	$\longrightarrow$	10000000_x	$2^{-0.85}$	
	3	48000008_x	$\longrightarrow$	08000048_x	$2^{-3.85}$	
	4	90000000_x	$\longrightarrow$	10000000_x	$2^{-0.85}$	
	5	48280008_x	$\longrightarrow$	09680048_x	$2^{-6.92}$	
	6	$9002d000_x$	$\longrightarrow$	10145000_x	$2^{-6.43}$	
	7	48296808_x	$\longrightarrow$	09622848_x	$2^{-11.70}$	
inject message word difference $\Delta m_{0} = 00228000_x$						
1	0	$9002d000_x$	$\longrightarrow$	10145000_x	$2^{-6.43}$	$\mathbf{2^{-37.03}}$
	1	48296800_x	$\longrightarrow$	09622800_x	$\star$	
	2	$0002d000_x$	$\longrightarrow$	00145000_x	$2^{-5.58}$	
	3	48296800_x	$\longrightarrow$	09622800_x	$2^{-9.68}$	
	4	$0002d000_x$	$\longrightarrow$	00145000_x	$2^{-5.58}$	
	5	48016800_x	$\longrightarrow$	$080a2800_x$	$2^{-7.43}$	
	6	90000000_x	$\longrightarrow$	10000000_x	$2^{-0.85}$	
	7	48016800_x	$\longrightarrow$	$080a2800_x$	$2^{-7.43}$	
inject message word difference $\Delta m_{1} = 00228800_x$						
2	0	90000000_x	$\longrightarrow$	10000000_x	$2^{-0.85}$	$\mathbf{2^{-37.41}}$
	1	00016800_x	$\longrightarrow$	$000a2800_x$	$\star$	
	2	$9002d000_x$	$\longrightarrow$	10145000_x	$2^{-6.43}$	
	3	00016800_x	$\longrightarrow$	$000a2800_x$	$2^{-5.59}$	
	4	$9002d000_x$	$\longrightarrow$	10145000_x	$2^{-6.43}$	
	5	48080000_x	$\longrightarrow$	08480000_x	$2^{-3.71}$	
	6	00024000_x	$\longrightarrow$	00104000_x	$2^{-2.85}$	
	7	48084800_x	$\longrightarrow$	$084a0800_x$	$2^{-6.45}$	
inject message word difference $\Delta m_{2} = 00000000_x$						
3	0	00024000_x	$\longrightarrow$	00104000_x	$2^{-2.85}$	$\mathbf{2^{-34.30}}$
	1	48092000_x	$\longrightarrow$	08402000_x	$\star$	
	2	90009000_x	$\longrightarrow$	10041000_x	$2^{-3.70}$	
	3	48092000_x	$\longrightarrow$	08402000_x	$2^{-5.71}$	
	4	90009000_x	$\longrightarrow$	10041000_x	$2^{-3.70}$	
	5	00212000_x	$\longrightarrow$	01282000_x	$2^{-4.58}$	
	6	90009000_x	$\longrightarrow$	10041000_x	$2^{-3.70}$	
	7	00200000_x	$\longrightarrow$	01200000_x	$2^{-1.85}$	
inject message word difference $\Delta m_{3} = 00000000_x$						
4	0	90009000_x	$\longrightarrow$	10041000_x	$2^{-3.70}$	$\mathbf{2^{-26.93}}$
	1	48296800_x	$\longrightarrow$	09622800_x	$\star$	
	2	00009000_x	$\longrightarrow$	00041000_x	$2^{-2.85}$	
	3	48292000_x	$\longrightarrow$	09602000_x	$2^{-6.92}$	
	4	00009000_x	$\longrightarrow$	00041000_x	$2^{-2.85}$	
	5	48000000_x	$\longrightarrow$	08000000_x	$2^{-1.85}$	

Table 6 – Differential characteristic for EnRUPT-192. *(continued)*

Round	Step	Δe	$\rightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
	6	$\mathtt{90009000}_x$	$\rightarrow$	$\mathtt{10041000}_x$	$2^{-3.70}$	
	7	$\mathtt{48004800}_x$	$\rightarrow$	$\mathtt{08020800}_x$	$2^{-4.70}$	
inject message word difference $\Delta m_4 = \mathtt{00020800}_x$						
5	0	$\mathtt{90009000}_x$	$\rightarrow$	$\mathtt{10041000}_x$	$2^{-3.70}$	$\mathbf{2^{-26.56}}$
	1	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$\star$	
	2	$\mathtt{90009000}_x$	$\rightarrow$	$\mathtt{10041000}_x$	$2^{-3.70}$	
	3	$\mathtt{00004800}_x$	$\rightarrow$	$\mathtt{00020800}_x$	$2^{-2.85}$	
	4	$\mathtt{90009000}_x$	$\rightarrow$	$\mathtt{10041000}_x$	$2^{-3.70}$	
	5	$\mathtt{48080000}_x$	$\rightarrow$	$\mathtt{08480000}_x$	$2^{-3.71}$	
	6	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$2^{-0.00}$	
	7	$\mathtt{48080000}_x$	$\rightarrow$	$\mathtt{08480000}_x$	$2^{-3.71}$	
inject message word difference $\Delta m_5 = \mathtt{00020000}_x$						
6	0	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$2^{-0.00}$	$\mathbf{2^{-17.66}}$
	1	$\mathtt{48084808}_x$	$\rightarrow$	$\mathtt{084a0848}_x$	$\star$	
	2	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$2^{-0.00}$	
	3	$\mathtt{00080008}_x$	$\rightarrow$	$\mathtt{00480048}_x$	$2^{-3.85}$	
	4	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$2^{-0.00}$	
	5	$\mathtt{48084808}_x$	$\rightarrow$	$\mathtt{084a0848}_x$	$2^{-8.47}$	
	6	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$2^{-0.00}$	
	7	$\mathtt{48084808}_x$	$\rightarrow$	$\mathtt{084a0848}_x$	$2^{-8.47}$	
inject message word difference $\Delta m_6 = \mathtt{00020000}_x$						
7	0	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$2^{-0.00}$	$\mathbf{2^{-20.79}}$
	1	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$\star$	
	$\vdots$	$\vdots$	$\rightarrow$	$\vdots$	$\vdots$	
	7	$\mathtt{00000000}_x$	$\rightarrow$	$\mathtt{00000000}_x$	$2^{-0.00}$	

Table 7 – Differential characteristic for EnRUPT-224 or -256.

Round	Step	Δe	$\rightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
inject message word difference $\Delta m_{-1} =$ 0000000008000000$_x$						
0	0	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	$\mathbf{2^{-0.00}}$
	1	0000000000000800$_x$	$\rightarrow$	0000000000004800$_x$	$\star$	
	2	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	
	3	4800000000000800$_x$	$\rightarrow$	0800000000004800$_x$	$2^{-3.70}$	
	4	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	
	5	4800280000000800$_x$	$\rightarrow$	0801680000004800$_x$	$2^{-7.28}$	
	6	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	
	7	0000280168000800$_x$	$\rightarrow$	0001680a28004800$_x$	$2^{-11.02}$	
inject message word difference $\Delta m_{0} =$ 0000002280000000$_x$						
1	0	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	$\mathbf{2^{-36.56}}$
	1	0000280168000000$_x$	$\rightarrow$	0001680a28000000$_x$	$\star$	
	2	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	
	3	4800280000000000$_x$	$\rightarrow$	0801680000000000$_x$	$2^{-5.43}$	
	4	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	
	5	0000080000000000$_x$	$\rightarrow$	0000480000000000$_x$	$2^{-1.85}$	
	6	9000000240000000$_x$	$\rightarrow$	1000001040000000$_x$	$2^{-3.70}$	
	7	4800080120000000$_x$	$\rightarrow$	0800480820000000$_x$	$2^{-6.54}$	
inject message word difference $\Delta m_{1} =$ 0000002288000000$_x$						
2	0	9000000240000000$_x$	$\rightarrow$	1000001040000000$_x$	$2^{-3.70}$	$\mathbf{2^{-34.08}}$
	1	0000080048000000$_x$	$\rightarrow$	0000480208000000$_x$	$\star$	
	2	9000000240000000$_x$	$\rightarrow$	1000001040000000$_x$	$2^{-3.70}$	
	3	4800080168000000$_x$	$\rightarrow$	0800480a28000000$_x$	$2^{-9.28}$	
	4	9000000240000000$_x$	$\rightarrow$	1000001040000000$_x$	$2^{-3.70}$	
	5	0000200000000000$_x$	$\rightarrow$	0001200000000000$_x$	$2^{-1.85}$	
	6	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	
	7	4800200000000000$_x$	$\rightarrow$	0801200000000000$_x$	$2^{-3.70}$	
inject message word difference $\Delta m_{2} =$ 0000000208000000$_x$						
3	0	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	$\mathbf{2^{-23.91}}$
	1	0000280120000000$_x$	$\rightarrow$	0001680820000000$_x$	$\star$	
	2	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	3	4800280168000000$_x$	$\rightarrow$	0801680a28000000$_x$	$2^{-11.02}$	
	4	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	5	0000080048000000$_x$	$\rightarrow$	0000480208000000$_x$	$2^{-4.70}$	
	6	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	7	4800080000000000$_x$	$\rightarrow$	0800480000000000$_x$	$2^{-3.70}$	
inject message word difference $\Delta m_{3} =$ 0000000200000000$_x$						
4	0	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	$\mathbf{2^{-34.19}}$
	1	0000080000000800$_x$	$\rightarrow$	0000480000004800$_x$	$\star$	
	2	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	3	0000080000000800$_x$	$\rightarrow$	0000480000004800$_x$	$2^{-3.70}$	
	4	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	5	4800080048000800$_x$	$\rightarrow$	0800480208004800$_x$	$2^{-8.39}$	

Table 7 – Differential characteristic for EnRUPT-224 or -256. *(continued)*

Round	Step	Δe	$\rightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
	6	0000000000000000_x	$\rightarrow$	0000000000000000_x	$2^{-0.00}$	
	7	4800080048000800_x	$\rightarrow$	0800480208004800_x	$2^{-8.39}$	
inject message word difference $\Delta m_4 = 0000000200000000_x$						
5	0	0000000000000000_x	$\rightarrow$	0000000000000000_x	$2^{-0.00}$	$\mathbf{2^{-20.49}}$
	1	0000000000000000_x	$\rightarrow$	0000000000000000_x	$\star$	
	$\vdots$	$\vdots$	$\rightarrow$	$\vdots$	$\vdots$	
	7	0000000000000000_x	$\rightarrow$	0000000000000000_x	$2^{-0.00}$	

Table 8 – Differential characteristic for EnRUPT-384.

Round	Step	Δe	$\to$	Δf	$\mathrm{DP}^{\times 9}$	totals
inject message word difference $\Delta m_{-1} =$ 0000000008000000$_x$						
0	0	0000000000000000$_x$	$\to$	0000000000000000$_x$	$2^{-0.00}$	$\mathbf{2^{-0.00}}$
	1	0000000000000800$_x$	$\to$	0000000000004800$_x$	⋆	
	2	9000000000000000$_x$	$\to$	1000000000000000$_x$	$2^{-0.85}$	
	3	4800000000000800$_x$	$\to$	0800000000004800$_x$	$2^{-3.70}$	
	4	9000000000000000$_x$	$\to$	1000000000000000$_x$	$2^{-0.85}$	
	5	4800280000000800$_x$	$\to$	0801680000004800$_x$	$2^{-7.28}$	
	6	90000002d0000000$_x$	$\to$	1000001450000000$_x$	$2^{-6.43}$	
	7	4800280168000800$_x$	$\to$	0801680a28004800$_x$	$2^{-12.87}$	
inject message word difference $\Delta m_0 =$ 0000002280000000$_x$						
1	0	90000002d0000000$_x$	$\to$	1000001450000000$_x$	$2^{-6.43}$	$\mathbf{2^{-38.41}}$
	1	4800280168000000$_x$	$\to$	0801680a28000000$_x$	⋆	
	2	00000002d0000000$_x$	$\to$	0000001450000000$_x$	$2^{-5.58}$	
	3	4800280168000000$_x$	$\to$	0801680a28000000$_x$	$2^{-11.02}$	
	4	00000002d0000000$_x$	$\to$	0000001450000000$_x$	$2^{-5.58}$	
	5	4800000168000000$_x$	$\to$	0800000a28000000$_x$	$2^{-7.43}$	
	6	9000000000000000$_x$	$\to$	1000000000000000$_x$	$2^{-0.85}$	
	7	4800000168000000$_x$	$\to$	0800000a28000000$_x$	$2^{-7.43}$	
inject message word difference $\Delta m_1 =$ 0000002288000000$_x$						
2	0	9000000000000000$_x$	$\to$	1000000000000000$_x$	$2^{-0.85}$	$\mathbf{2^{-38.75}}$
	1	0000000168000000$_x$	$\to$	0000000a28000000$_x$	⋆	
	2	90000002d0000000$_x$	$\to$	1000001450000000$_x$	$2^{-6.43}$	
	3	0000000168000000$_x$	$\to$	0000000a28000000$_x$	$2^{-5.58}$	
	4	90000002d0000000$_x$	$\to$	1000001450000000$_x$	$2^{-6.43}$	
	5	4800080000000000$_x$	$\to$	0800480000000000$_x$	$2^{-3.70}$	
	6	0000000240000000$_x$	$\to$	0000001040000000$_x$	$2^{-2.85}$	
	7	4800080048000000$_x$	$\to$	0800480208000000$_x$	$2^{-6.54}$	
inject message word difference $\Delta m_2 =$ 0000000000000000$_x$						
3	0	0000000240000000$_x$	$\to$	0000001040000000$_x$	$2^{-2.85}$	$\mathbf{2^{-34.39}}$
	1	4800080120000000$_x$	$\to$	0800480820000000$_x$	⋆	
	2	9000000090000000$_x$	$\to$	1000000410000000$_x$	$2^{-3.70}$	
	3	4800080120000000$_x$	$\to$	0800480820000000$_x$	$2^{-6.54}$	
	4	9000000090000000$_x$	$\to$	1000000410000000$_x$	$2^{-3.70}$	
	5	0000200120000000$_x$	$\to$	0001200820000000$_x$	$2^{-4.70}$	
	6	9000000090000000$_x$	$\to$	1000000410000000$_x$	$2^{-3.70}$	
	7	0000200000000000$_x$	$\to$	0001200000000000$_x$	$2^{-1.85}$	
inject message word difference $\Delta m_3 =$ 0000000000000000$_x$						
4	0	9000000090000000$_x$	$\to$	1000000410000000$_x$	$2^{-3.70}$	$\mathbf{2^{-27.87}}$
	1	4800280168000000$_x$	$\to$	0801680a28000000$_x$	⋆	
	2	0000000090000000$_x$	$\to$	0000000410000000$_x$	$2^{-2.85}$	
	3	4800280120000000$_x$	$\to$	0801680820000000$_x$	$2^{-8.28}$	
	4	0000000090000000$_x$	$\to$	0000000410000000$_x$	$2^{-2.85}$	
	5	4800000000000000$_x$	$\to$	0800000000000000$_x$	$2^{-1.85}$	

Table 8 – Differential characteristic for EnRUPT-384. *(continued)*

Round	Step	Δe	$\to$	Δf	$\mathrm{DP}^{\times 9}$	totals
	6	9000000090000000_x	$\to$	1000000410000000_x	$2^{-3.70}$	
	7	4800000048000000_x	$\to$	0800000208000000_x	$2^{-4.70}$	
inject message word difference $\Delta m_4 = 0000000208000000_x$						
5	0	9000000090000000_x	$\to$	1000000410000000_x	$2^{-3.70}$	$\mathbf{2^{-27.91}}$
	1	0000000000000000_x	$\to$	0000000000000000_x	$\star$	
	2	9000000090000000_x	$\to$	1000000410000000_x	$2^{-3.70}$	
	3	0000000048000000_x	$\to$	0000000208000000_x	$2^{-2.85}$	
	4	9000000090000000_x	$\to$	1000000410000000_x	$2^{-3.70}$	
	5	4800080000000000_x	$\to$	0800480000000000_x	$2^{-3.70}$	
	6	0000000000000000_x	$\to$	0000000000000000_x	$2^{-0.00}$	
	7	4800080000000000_x	$\to$	0800480000000000_x	$2^{-3.70}$	
inject message word difference $\Delta m_5 = 0000000200000000_x$						
6	0	0000000000000000_x	$\to$	0000000000000000_x	$2^{-0.00}$	$\mathbf{2^{-17.63}}$
	1	4800080048000800_x	$\to$	0800480208004800_x	$\star$	
	2	0000000000000000_x	$\to$	0000000000000000_x	$2^{-0.00}$	
	3	0000080000000800_x	$\to$	0000480000004800_x	$2^{-3.70}$	
	4	0000000000000000_x	$\to$	0000000000000000_x	$2^{-0.00}$	
	5	4800080048000800_x	$\to$	0800480208004800_x	$2^{-8.39}$	
	6	0000000000000000_x	$\to$	0000000000000000_x	$2^{-0.00}$	
	7	4800080048000800_x	$\to$	0800480208004800_x	$2^{-8.39}$	
inject message word difference $\Delta m_6 = 0000000200000000_x$						
7	0	0000000000000000_x	$\to$	0000000000000000_x	$2^{-0.00}$	$\mathbf{2^{-20.49}}$
	1	0000000000000000_x	$\to$	0000000000000000_x	$\star$	
	$\vdots$	$\vdots$	$\to$	$\vdots$	$\vdots$	
	7	0000000000000000_x	$\to$	0000000000000000_x	$2^{-0.00}$	

Table 9 – Differential characteristic for EnRUPT-512.

Round	Step	Δe	$\rightarrow$	Δf	DP$^{\times 9}$	totals
inject message word difference Δm_{-1} = 0000000008000000$_x$						
0	0	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	$\mathbf{2^{-0.00}}$
	1	0000000000000800$_x$	$\rightarrow$	0000000000004800$_x$	$\star$	
	2	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	
	3	4800000000000800$_x$	$\rightarrow$	0800000000004800$_x$	$2^{-3.70}$	
	4	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	
	5	4800280000000800$_x$	$\rightarrow$	0801680000004800$_x$	$2^{-7.28}$	
	6	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	
	7	4800280168000800$_x$	$\rightarrow$	0801680a28004800$_x$	$2^{-12.87}$	
inject message word difference Δm_0 = 0000002280000000$_x$						
1	0	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	$\mathbf{2^{-38.41}}$
	1	4800280168000000$_x$	$\rightarrow$	0801680a28000000$_x$	$\star$	
	2	00000002d0000000$_x$	$\rightarrow$	0000001450000000$_x$	$2^{-5.58}$	
	3	0000280168000000$_x$	$\rightarrow$	0001680a28000000$_x$	$2^{-9.17}$	
	4	00000002d0000000$_x$	$\rightarrow$	0000001450000000$_x$	$2^{-5.58}$	
	5	0000000168000000$_x$	$\rightarrow$	0000000a28000000$_x$	$2^{-5.58}$	
	6	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	7	4800000000000000$_x$	$\rightarrow$	0800000000000000$_x$	$2^{-1.85}$	
inject message word difference Δm_1 = 0000002288000000$_x$						
2	0	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	$\mathbf{2^{-27.77}}$
	1	4800000000000000$_x$	$\rightarrow$	0800000000000000$_x$	$\star$	
	2	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	
	3	4800000168000000$_x$	$\rightarrow$	0800000a28000000$_x$	$2^{-7.43}$	
	4	9000000000000000$_x$	$\rightarrow$	1000000000000000$_x$	$2^{-0.85}$	
	5	0000000168000000$_x$	$\rightarrow$	0000000a28000000$_x$	$2^{-5.58}$	
	6	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	
	7	0000000168000000$_x$	$\rightarrow$	0000000a28000000$_x$	$2^{-5.58}$	
inject message word difference Δm_2 = 0000000000000000$_x$						
3	0	90000002d0000000$_x$	$\rightarrow$	1000001450000000$_x$	$2^{-6.43}$	$\mathbf{2^{-33.16}}$
	1	4800000000000000$_x$	$\rightarrow$	0800000000000000$_x$	$\star$	
	2	00000002d0000000$_x$	$\rightarrow$	0000001450000000$_x$	$2^{-5.58}$	
	3	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	4	00000002d0000000$_x$	$\rightarrow$	0000001450000000$_x$	$2^{-5.58}$	
	5	0000080168000000$_x$	$\rightarrow$	0000480a28000000$_x$	$2^{-7.43}$	
	6	0000000090000000$_x$	$\rightarrow$	0000000410000000$_x$	$2^{-2.85}$	
	7	4800080048000000$_x$	$\rightarrow$	0800480208000000$_x$	$2^{-6.54}$	
inject message word difference Δm_3 = 0000000000000000$_x$						
4	0	0000000090000000$_x$	$\rightarrow$	0000000410000000$_x$	$2^{-2.85}$	$\mathbf{2^{-30.84}}$
	1	4800080048000000$_x$	$\rightarrow$	0800480208000000$_x$	$\star$	
	2	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	3	4800080120000000$_x$	$\rightarrow$	0800480820000000$_x$	$2^{-6.54}$	
	4	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	5	0000200120000000$_x$	$\rightarrow$	0001200820000000$_x$	$2^{-4.70}$	

Table 9 – Differential characteristic for EnRUPT-512. *(continued)*

Round	Step	Δe	$\rightarrow$	Δf	$\mathrm{DP}^{\times 9}$	totals
	6	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	7	0000200048000000$_x$	$\rightarrow$	0001200208000000$_x$	$2^{-4.70}$	
inject message word difference $\Delta m_4 =$ 0000000000000000$_x$						
5	0	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	$\mathbf{2^{-30.72}}$
	1	4800280120000000$_x$	$\rightarrow$	0801680820000000$_x$	$\star$	
	2	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	3	0000280120000000$_x$	$\rightarrow$	0001680820000000$_x$	$2^{-6.43}$	
	4	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	5	0000000048000000$_x$	$\rightarrow$	0000000208000000$_x$	$2^{-2.85}$	
	6	0000000090000000$_x$	$\rightarrow$	0000000410000000$_x$	$2^{-2.85}$	
	7	4800000048000000$_x$	$\rightarrow$	0800000208000000$_x$	$2^{-4.70}$	
inject message word difference $\Delta m_5 =$ 0000000000000000$_x$						
6	0	0000000090000000$_x$	$\rightarrow$	0000000410000000$_x$	$2^{-2.85}$	$\mathbf{2^{-19.67}}$
	1	4800000000000000$_x$	$\rightarrow$	0800000000000000$_x$	$\star$	
	2	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	3	4800000048000000$_x$	$\rightarrow$	0800000208000000$_x$	$2^{-4.70}$	
	4	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	5	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	6	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	
	7	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
inject message word difference $\Delta m_6 =$ 0000000208000000$_x$						
7	0	9000000090000000$_x$	$\rightarrow$	1000000410000000$_x$	$2^{-3.70}$	$\mathbf{2^{-19.48}}$
	1	4800000048000000$_x$	$\rightarrow$	0800000208000000$_x$	$\star$	
	2	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	3	0000000048000000$_x$	$\rightarrow$	0000000208000000$_x$	$2^{-2.85}$	
	4	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	5	0000080048000000$_x$	$\rightarrow$	0000480208000000$_x$	$2^{-4.70}$	
	6	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	7	4800080000000000$_x$	$\rightarrow$	0800480000000000$_x$	$2^{-3.70}$	
inject message word difference $\Delta m_7 =$ 0000000200000000$_x$						
8	0	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	$\mathbf{2^{-11.24}}$
	1	4800080048000800$_x$	$\rightarrow$	0800480208004800$_x$	$\star$	
	2	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	3	0000080000000800$_x$	$\rightarrow$	0000480000004800$_x$	$2^{-3.70}$	
	4	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	5	4800080048000800$_x$	$\rightarrow$	0800480208004800$_x$	$2^{-8.39}$	
	6	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	
	7	4800080048000800$_x$	$\rightarrow$	0800480208004800$_x$	$2^{-8.39}$	
inject message word difference $\Delta m_8 =$ 0000000200000000$_x$						
9	0	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	$\mathbf{2^{-20.49}}$
	1	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$\star$	
	$\vdots$	$\vdots$	$\rightarrow$	$\vdots$	$\vdots$	
	7	0000000000000000$_x$	$\rightarrow$	0000000000000000$_x$	$2^{-0.00}$	

Discussion. In response to these collision attacks, the designers of EnRUPT proposed to double the s parameter to 8, or to increase it even further to be equal to the H-parameter, see Table 1 [7,8]. As a consequence of this, the number of steps between two message word injections is at least doubled. Experiments with these EnRUPT variants indicate that this tweak seems to be effective at stopping the attacks described in this paper. For EnRUPT-256 with $s = 6$, we were still able to find a differential with an associated attack complexity of about 2^{110} EnRUPT rounds, which is still below the birthday bound. For higher values of the s parameter, all the differential characteristics we could find would result in attack complexities that are far beyond than the birthday bound, and thus should not be considered to be real attacks.

Note that the failure of this heuristic attack method for $s = 8$ or $s = H$ does not preclude the possibility of attacks based on linearisation. Our experiments only show that it is unlikely that the particular attack method used in this work can be applied directly to EnRUPT with $s \geq 8$.

8 Conclusion

We presented collision attacks on all seven variants of the EnRUPT hash function [9] that were proposed as candidates to the NIST SHA-3 competition [6]. The attacks require negligible memory and have time complexities ranging from 2^{36} to 2^{40} EnRUPT round computations, depending on the EnRUPT variant. The practicality of the attacks has been demonstrated with an example collision for EnRUPT-256.

Acknowledgements

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. The collision example for EnRUPT-256 was obtained utilising high performance computational resources provided by the University of Leuven, `http://ludit.kuleuven.be/hpc`.

References

[1] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998.

[2] F. Chabaud and A. Joux. Differential collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[3] S. Indesteege and B. Preneel. Practical collisions for EnRUPT. In O. Dunkelman, editor, *Fast Software Encryption, 16th International Workshop — FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 246–259. Springer, 2009.

[4] D. Khovratovich, I. Nikolić, and R.-P. Weinmann. Meet-in-the-middle attacks on SHA-3 candidates. In O. Dunkelman, editor, *Fast Software Encryption, 16th International Workshop — FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2009.

[5] H. Lipmaa and S. Moriai. Efficient algorithms for computing differential properties of addition. In M. Matsui, editor, *Fast Software Encryption, 8th International Workshop — FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2001.

[6] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, Nov. 2007.

[7] S. O'Neil. personal communication, Jan. 2009.

[8] S. O'Neil. EnRUPT. The First SHA-3 Candidate Conference, Leuven, Belgium, Feb. 2009. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/Feb2009/documents/EnRUPT_2009.pdf`.

[9] S. O'Neil, K. Nohl, and L. Henzen. EnRUPT hash function specification. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[10] N. Pramstaller, C. Rechberger, and V. Rijmen. Exploiting coding theory for collision attacks on SHA-1. In N. P. Smart, editor, *Cryptography and Coding, IMA International Conference*, volume 3796 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2005.

[11] V. Rijmen and E. Oswald. Update on SHA-1. In A. Menezes, editor, *Topics in Cryptology — CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

[12] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, Apr. 1967.

[13] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

Publication

Practical Preimages for Maraca

Publication Data

Sebastiaan Indesteege and Bart Preneel. Practical preimages for Maraca. In Tjalling Tjalkens and Frans Willems, editors, *Proceedings of the 30th Symposium on Information Theory in the Benelux*, pages 119–126. Werkgemeenschap voor Informatie- en Communicatietheorie, 2009.

Contributions

- Principal author.

Practical Preimages for Maraca

Sebastiaan Indesteege[1,2,*] and Bart Preneel[1,2]

[1] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
`sebastiaan.indesteege@esat.kuleuven.be`

[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.

Abstract. The cryptographic hash function Maraca was submitted to the NIST SHA-3 competition [4] by Jenkins [3]. In this work, we show a practical preimage attack on Maraca. Our attack has been implemented and verified experimentally. This shows that Maraca does not achieve several important security properties which a secure cryptographic hash function is expected to offer.

1 Introduction

Cryptographic hash functions are easy to compute, deterministic functions that map an input message of arbitrary length to a short, fixed-length digest. They are important building blocks in many cryptographic applications. Secure cryptographic hash functions are required to have several security properties, such as collision resistance and preimage resistance. In this work, we focus on the latter, preimage resistance. Informally, this notion means that, given a hash function output y, it should be difficult to find an input message x hashing to this output.

As recent cryptanalytic advances have raised serious concerns regarding the security of several widely used hash functions, such as MD5 and SHA-1, the National Institute of Standards and Technology (NIST) has recently started a public competition, the SHA-3 competition [4]. This competition aims to develop a new cryptographic hash function standard. Maraca is a hash function proposal that was submitted as a candidate to this SHA-3 competition by Jenkins [3], but it was not selected for round 1 of the competition.

Canteaut and Naya-Plasencia [1] analysed the security of Maraca with respect to collision attacks, and constructed a theoretical collision attack, requiring 2^{237} calls to the compression function and a memory of $2^{230.5}$ bits.

In this work, we propose a practical preimage attack on Maraca. After a one-time precomputation, our attack can find many (first) preimages for any hash output in just a few seconds on an average PC. We have implemented our attack, and verified it experimentally. Of course, this attack can also be used to construct collisions or second preimages for Maraca.

*F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

This paper is organised as follows. First, a description of the Maraca hash function is given in Sect. 2. The basic ideas used in our attack are presented in Sect. 3. Section 4 focuses on a component of Maraca, the Maraca S-box, as an important weakness in this component will be exploited in our attack. Section 5 puts everything together, resulting in a practical preimage attack on the Maraca hash function. The practical aspects of this attack are discussed in Sect. 6. Finally, Sect. 7 concludes.

2 Description of Maraca

Maraca is a cryptographic hash function proposed by Jenkins [3] as a SHA-3 candidate. It supports digest sizes of up to 1024 bits, and can optionally use a key. For simplicity, we only describe the unkeyed mode of Maraca here. Hashing a message with Maraca consists of three phases: message padding, message processing and digest generation.

First, the input message is padded to a multiple of 1024 bits as follows. If the message ends in a fractional byte, this byte is filled with zero bits. Then, a 16-bit tag containing the number of zero padding bits used, is appended to the message. Finally, zero bytes are used to further pad the message to an integer number of 1024-bit blocks. Note that, unlike many other hash functions, the message length is not included in the padding.

The operation of the Maraca hash function is shown in Fig. 1. Maraca has an internal state of 1024 bits, which is initialised to zero. For each 1024-bit message block W_i, a round is performed. A round consists of the following sequence of operations. First, the message block W_i is XORed into the internal state. Next, a 1024-bit permutation, which will be described in detail in Sect. 2.1, is applied once. Then, up to three message blocks are selected from a window consisting of the past 47 message blocks, where message blocks with a negative index are defined to be all zeroes. This selection of blocks is done in a way that ensures that each message block is used four times in total. Each of these three message blocks is subjected to a different fixed rotation and combined into the accumulator Acc_i using XOR. The 1024-bit quantity Acc_i is then XORed into the internal state. Finally, the same 1024-bit permutation, see Sect. 2.1, is applied twice.

After all message blocks have been processed, 47 blank rounds are performed. These are rounds where no new message blocks are used. Note that in these blank rounds, the 47 last message blocks are still reused via the accumulators Acc_i, even though there are no new message blocks. Finally, the Maraca permutation, see Sect. 2.1, is applied 28 more times. The output digest is then found by truncating the final internal state to the desired length. Thus, digest lengths of up to 1024 bits can be obtained.

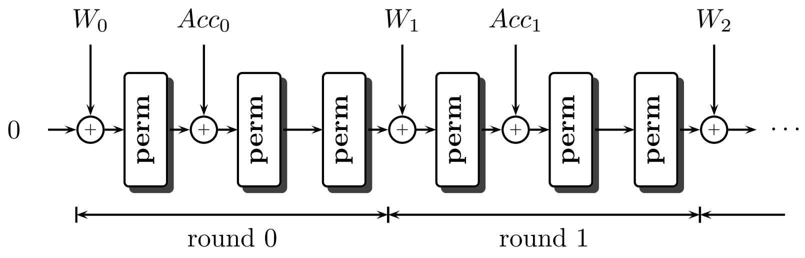


Figure 1 – The Maraca hash function.

2.1 The Maraca Permutation

The Maraca permutation is a fixed, nonlinear permutation operating on 1024 bits. It is shown schematically in Fig. 2. The input to this permutation is the 1024-bit internal state of Maraca, which is arranged as 16 words of 64 bits. First, the same nonlinear bijective 8×8 bit substitution box (S-box) is applied 128 times in parallel. Each S-box takes a single bit from eight distinct words as its input bits, as is shown in Fig. 2, and performs a table lookup as defined in Table 1. This arrangement is intended to allow for a simple bitsliced implementation of Maraca.

After the S-box layer, constants are XORed to six words. Then, the bits in each word are rotated by a fixed amount, which is different for each word. Finally, the words are shuffled, i.e., their order is changed. Note that only the S-box layer is nonlinear. Everything that comes after the S-box layer is just an XOR with constants, followed by a reordering of the 1024 bits, i.e., a bit permutation. This is clearly linear, or more precisely as there are constants, affine.

3 Basic Attack Idea

Consider a hypothetical hash function with an n-bit output which, for a certain fixed message length l, is a linear or an affine function over GF(2). Clearly, any such function can be written as

$$[y]_{n\times 1} = [\mathbf{A}]_{n\times l} \cdot [x]_{l\times 1} \oplus [b]_{n\times 1} \quad . \tag{1}$$

Here, x is a binary column vector containing the l bits of the input message and similarly, y holds the n-bit output digest. The binary matrix $\mathbf{A}$ and the binary vector b allow to express any linear or affine function over GF(2). In the remainder of this paper, we will drop the distinction between a linear and an affine function, and refer to both as linear.

Note that finding preimages for such a linear hash function is easy. Given any output y, it is easy to find a value for the message x such that $h(x) = y$. Indeed,

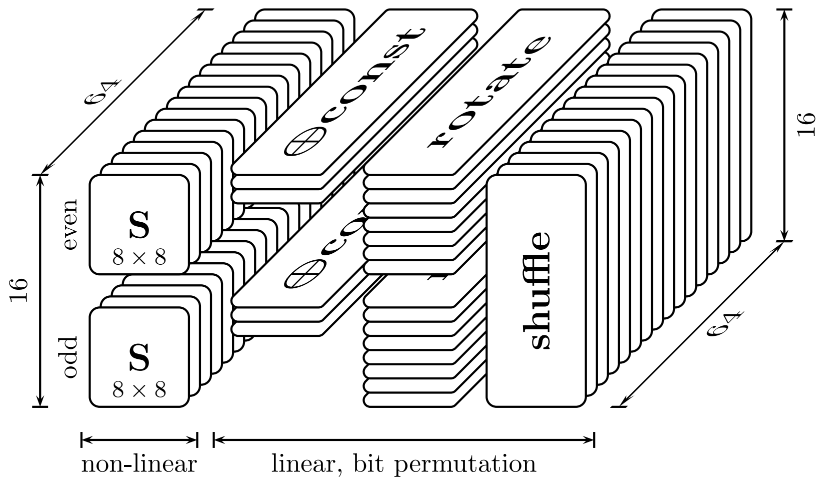


Figure 2 – The Maraca permutation.

all messages x for which $h(x) = y$ are simply the solutions of the system of linear equations over GF(2) given by:

$$\mathbf{A} \cdot x = y \oplus b \ . \tag{2}$$

Such a system of equations can be solved easily, for instance using Gaussian elimination.

Even though Maraca is clearly not a linear function, it is in essence this method that will be used to construct preimages for Maraca. In the remainder of this work, we will show that by restricting the input messages in a carefully chosen way, it is possible to turn the Maraca hash function into a linear function over GF(2).

4 Linearising the Maraca S-box

The only component in Maraca which is not linear is the S-box, which is given in Table 1. However, as was noted by Canteaut and Naya-Plasencia [1], three of the eight output bits are linear functions of the input bits.

Our aim is to linearise the Maraca S-box completely, i.e., to turn it into a linear function. The idea is to choose a linear approximation for the nonlinear S-box, and restrict the inputs to those for which the approximation holds. A trivial example of this approach is to choose any two input values to the S-box. A linear function which maps these two input values to the correct outputs can be found easily.

More concretely, the inputs of the S-box are restricted to some affine space. The reason is that such restrictions can be incorporated easily in the system of linear equations in (2). If every component of Maraca is replaced by a linear

Table 1 – The Maracа S-box (hexadecimal).

	␣0	␣1	␣2	␣3	␣4	␣5	␣6	␣7	␣8	␣9	␣a	␣b	␣c	␣d	␣e	␣f
0␣	00	41	26	4f	92	b3	94	bd	0e	47	28	49	9c	b5	9a	bb
1␣	8d	a4	8b	a2	1f	56	39	50	83	aa	85	ac	11	58	37	5e
2␣	17	76	31	78	a5	84	a3	8a	19	70	3f	7e	ab	82	ad	8c
3␣	ba	93	bc	95	08	61	2e	67	b4	9d	b2	9b	06	6f	20	69
4␣	98	d1	be	d7	4a	e3	4c	e5	96	df	b0	d9	44	ed	42	eb
5␣	55	f4	53	fa	87	46	a1	48	5b	f2	5d	fc	89	40	af	4e
6␣	8f	e6	a9	e0	6d	c4	6b	c2	81	e8	a7	ee	63	ca	65	cc
7␣	72	d3	74	dd	90	71	b6	7f	7c	d5	7a	db	9e	77	b8	79
8␣	91	10	b7	1e	43	e2	45	ec	9f	16	b9	18	4d	e4	4b	ea
9␣	5c	f5	5a	f3	8e	07	a8	01	52	fb	54	fd	80	09	a6	0f
a␣	86	27	a0	29	64	c5	62	cb	88	21	ae	2f	6a	c3	6c	cd
b␣	7b	d2	7d	d4	99	30	bf	36	75	dc	73	da	97	3e	b1	38
c␣	c9	c0	ef	c6	1b	32	1d	34	c7	ce	e1	c8	15	3c	13	3a
d␣	04	25	02	2b	d6	57	f0	59	0a	23	0c	2d	d8	51	fe	5f
e␣	de	f7	f8	f1	2c	05	2a	03	d0	f9	f6	ff	22	0b	24	0d
f␣	33	12	35	1c	c1	60	e7	6e	3d	14	3b	1a	cf	66	e9	68

approximation, then every intermediate bit is found as a linear combination of input bits and possibly a constant. Restricting a set of intermediate bits to some affine space thus corresponds to adding a number of linear equations to the system (2).

An exhaustive search through all possible affine input spaces was performed. For each such space, it was tested whether the Maraca S-box becomes a linear function when its inputs are restricted to this space. To keep the search complexity low, the problem was reformulated as a tree search, where new restrictions are added at every level of the tree. An early abort strategy was used to accelerate the search. Also, care was taken to avoid duplicate work arising from equivalent representations of the same affine spaces. Such duplicate work can be avoided by only investigating sets of conditions in reduced echelon form. The search takes just a few seconds on an average desktop PC.

The results are remarkable in the sense that imposing just three linear conditions on the input bits of the Maraca S-box can already linearise it. For comparison, at least six conditions are required for the AES [2] S-box, which is also an 8×8 bit S-box. An example of a set of conditions which linearises the Maraca S-box, is the following, where $x_7, \ldots, x_0$ denote the eight input bits:

$$\left\{ \begin{array}{rcl} x_0 & = & 0 \\ x_2 \oplus x_4 & = & 0 \\ x_7 & = & 0 \end{array} \right. . \tag{3}$$

Many such sets of three or more linear conditions which linearise the Maraca S-box were found. Actually, due to the completeness of the search algorithm used, it is guaranteed that all of them are found.

5 A Preimage Attack on Maraca

Recall from Sect. 4 that, in Maraca, imposing just three linear conditions on the input bits of an S-box can already be sufficient to turn it into a linear function. As a first attempt, consider linearising every S-box in this way, thereby linearising the entire hash function. Clearly, this will not work, as there are $3 \cdot 128$ S-boxes in each round, thus requiring a total of 1152 conditions per round. But there are only 1024 degrees of freedom available in each round, which arise from the 1024-bit message block. In other words, for each round, many more equations than unknowns would be added to the system of equations in (2). As it is very unlikely that, by accident, enough of these equations would be linearly dependent, the system of equations is not expected to have any solutions.

5.1 Making Conditions Dependent

A way to overcome this problem is to make use of the fact that there are many ways to linearise the Maraca S-box. By carefully choosing how to linearise each S-box, an attempt can be made to make as many conditions as possible dependent on each other.

Recall the structure of a Maraca round, see Fig. 1. It consists of three calls to the Maraca permutation, which was introduced in Sect 2.1. Before the first permutation, a message block W_i is XORed into the internal state. Also, after the first permutation, a combination of message blocks denoted by Acc_i is XORed into the state. But in between the second and the third permutation, no additional inputs are added to the state. Investigating the linear part of the Maraca permutation leads to the observation that all eight output bits of an S-box in the second permutation of a round are input to different S-boxes in the third permutation of that round. Since conditions on the bits of a single S-box are required in order to linearise it, only conditions involving a single bit are useful as they apply to a single S-box in both the second and the third permutation of a round.

We propose the following approach. The S-boxes in the first permutation of a round are linearised using three conditions per S-box, for instance using (3). The S-boxes in the second permutation are linearised using a set of four conditions per S-box. Depending on the constants and whether the S-box is even or odd-numbered, one of the following two sets of linearising conditions is used:

$$\begin{cases} x_1 \oplus x_3 &= 0 \\ x_2 \oplus x_5 &= 0 \\ x_4 \oplus x_5 &= 1 \\ x_6 \oplus x_7 &= 0 \end{cases} \quad \text{or} \quad \begin{cases} x_1 \oplus x_3 &= 0 \\ x_2 \oplus x_5 &= 1 \\ x_4 \oplus x_5 &= 0 \\ x_6 \oplus x_7 &= 0 \end{cases} . \tag{4}$$

Again, x_7 to x_0 denote the S-box input bits. While at first it may seem counter-productive to use more than three conditions per S-box, the advantage is that as much as four S-box output bits are also fixed to a particular value by these conditions.

For the S-boxes in the third permutation, either a set of three or a set of five linearising conditions is used, again depending on the position of the S-box. However, because of the way the S-boxes of the second permutation were linearised, most of these conditions are satisfied automatically. Only one additional condition needs to be added for the even-numbered S-boxes, i.e., just half of the S-boxes in the third permutation. Thus, the total number of conditions per round is now $128 \cdot 3 + 128 \cdot 4 + 64 \cdot 1 = 960$. As there are 1024 degrees of freedom available in each round, 64 degrees of freedom are expected to remain per round.

5.2 Maraca's Finalisation Phase

The last message block contains an amount of padding, which cannot be chosen by an adversary. However, by choosing an appropriate message length, this padding overhead can be reduced to just 16 bits whose value will be fixed and known a priori. This has the effect of reducing the available degrees of freedom in the last message block slightly, which does not cause any problems.

After all message blocks have been processed, Maraca performs 47 blank rounds. The S-boxes in these rounds also have to be linearised, but no new degrees of freedom are available. This problem can be overcome by building up enough degrees of freedom before the start of the finalisation phase. A simple estimate learns that about $47 \cdot 960 = 45120$ degrees of freedom are required for the finalisation phase. Each normal round yields on average 64 extra degrees of freedom. Thus, after 705 rounds, the required number of degrees of freedom could be achieved. In our experiments, 750 rounds were used, to provide for a reasonable margin of error.

Finally, there are 28 consecutive calls to the Maraca permutation. Note that these do not contribute to the security of Maraca in any way, as they are invertible. Indeed, each of the operations shown in Fig. 2 can be inverted easily. Note that the Maraca S-box is bijective. Starting from a Maraca digest, one first reverts the final truncation by adding arbitrary bits. Then, the final 28 permutations can be inverted in a straightforward way.

Another improvement is to not linearise the S-boxes in the 47 blank rounds, but instead fix the last 47 message blocks to some known value. Then, the 47 blank rounds can be inverted, as all the message blocks used in Acc_i in those rounds are now known. This can also be seen as moving the blank rounds 47 rounds towards the beginning, which may seem pointless at first. However, now the (fixed) message bits used in these "blank rounds" can be chosen, instead of being fixed to zeroes. Note that this principle can also be used to extend the attack to the keyed mode of Maraca. The key appears both at the beginning and the end of the padded message. For a fixed key, it is possible to remove the first and the last message block, which contain the key, and proceed as before.

5.3 Dealing with Contradictions

Up to now, it was silently assumed that none of the additional conditions that are imposed cause any contradictions. It turns out that this is mostly the case, but very sporadically, contradictions do occur. Even though they are rare, they constitute an important issue, as even a single contradiction suffices to make the entire approach fail.

However, it is possible to work around these unfortunate events at the cost of some degrees of freedom. When a contradiction is detected, one can just choose a different linearisation for the problematic S-box. For instance, a trivial linearisation using seven conditions, which is always possible, can be used. As contradictions only occur rarely, the overall impact of this procedure on the available degrees of freedom is close to being negligible.

6 Practical Aspects

Conceptually, our preimage attack on Maraca corresponds to building a system of linear equations over GF(2), and solving this system of equations. There are some small complications, such as efficiently detecting contradictions, and modifying the system of equations to circumvent them, as was discussed in Sect. 5.3. However, the most important practical obstacle is the large dimension of the system. For 750 rounds, the system of equations has more than 760 000 equations in about as many unknowns. This amounts to a memory requirement of over 67 GB just to store the system. Also, for such a large system, the cubic time complexity of straightforward Gaussian elimination is prohibitively large.

Note however that the system of equations has a block triangular structure. This is explained by the simple observation that a message block can not affect the rounds before the first use of this message block. Because of the ample diffusion in Maraca, the equations are dense otherwise. Furthermore, in order to be able to efficiently detect contradictions and work around them, it is advantageous to combine the building and the solving the system. This has the additional advantage of limiting the memory usage.

Our implementation of the attack consists of two distinct phases. First, there is a precomputation phase, which has to be done only once, and an online phase which generates a preimage. All complexity is concentrated in the precomputation phase.

6.1 The Precomputation Phase

The following approach, which is based on straightforward Gaussian elimination, was used to build and solve the system of equations simultaneously. Rather than storing equations, the solution space itself is continuously tracked throughout the rounds of Maraca. As this is an affine space, it can be represented by a single displacement vector and a set of basis vectors. These vectors contain a message of a fixed length of 750 message blocks, as well as the 1024-bit internal state at the current position. The solution space is continuously updated as follows.

1. A new message block is added. This corresponds to adding 1024 vectors to the solution space. Each contains a message with a single "1" bit in the current message block and zeroes otherwise. Then, the internal state is updated in every vector to include the XOR with the new message block.
2. Conditions on the internal state bits are imposed in order to linearise the S-boxes of the first permutation of the round. This step corresponds to performing a Gaussian elimination step on several of the internal state bits, and removing certain vectors from the solution space. Note that at this point it is easy to detect any contradictions, and to choose a different linearisation for the S-box in question.
3. Now, the internal state in each of the vectors can be updated to include the (linearised) first permutation of the round. Also, the XOR with Acc_i, which is a combination of previous message blocks, can be performed.
4. In a similar way, the conditions for linearising the S-boxes in the second and third permutation of the round are imposed, further reducing the solution space.

This procedure is repeated until all rounds have been processed. After a single round, the dimension of the solution space has been increased by 64, on average, as 1024 new degrees of freedom were introduced, but only 960 conditions were added. As explained in Sect. 5.2, the 47 last message blocks are set to a fixed value so that the blank rounds can be inverted. This corresponds to another Gaussian elimination step in these rounds, which further reduces the solution space.

In the end, an affine message space is found for which the Maraca hash function, omitting the blank rounds and the final permutations, is linear. Then, it is checked if this solution space allows to reach any value for the internal state before the blank rounds. If not, the number of rounds needs to be increased. Our experiments show that, with 750 message blocks, any state can be reached.

The precomputation phase was implemented to run distributed on a cluster of computers. Each node keeps a part of the basis vectors in memory, and most of the computations can be carried out in parallel. Running the precomputation phase took about 10.7 CPU-days and 20 GB of distributed memory on a cluster of 32 AMD Opteron nodes. The result of the precomputation is a data file of 94 MB. This is the only data that is required by the online phase.

6.2 The Online Phase

The online phase constructs arbitrary preimages using the data from the precomputation phase. The target digest is first extended to 1024 bits by adding arbitrary bits. Then, the 28 final permutations and the 47 blank rounds are inverted, to retrieve the correct internal state before the start of the blank rounds.

Finally, the appropriate basis vectors from the data file are combined using XOR to reach the desired internal state. This entire process takes just a few seconds on an average desktop PC.

As any internal state before the blank rounds can be obtained, our attack is guaranteed to succeed for any given digest value. By extending the target digest to 1024 bits in different ways, it is possible to find multiple messages hashing to a given digest, i.e., multi-preimages. Of course, our attack can also be used to construct collisions or second preimages for Maraca.

7 Conclusion

In this work, we have shown a practical preimage attack on the hash function proposal Maraca. The main weakness that we exploit lies within the Maraca S-box. We have shown that the Maraca S-box can be linearised successfully by imposing additional linear constraints on the S-box inputs. In this way, the Maraca hash function can be turned into a linear function, for which it is easy to construct preimages. Our attack has been implemented and verified experimentally. This clearly shows that Maraca is not preimage resistant nor collision resistant, and hence should not be considered to be a secure cryptographic hash function.

Acknowledgements

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. This research was conducted utilizing high performance computational resources provided by the University of Leuven, `http://ludit.kuleuven.be/hpc`.

References

[1] A. Canteaut and M. Naya-Plasencia. Internal collision attack on Maraca. In H. Handschuh, S. Lucks, B. Preneel, and P. Rogaway, editors, *Symmetric Cryptography*, number 09031 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, Germany, 2009.

[2] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard.* Springer, 2002.

[3] R. J. Jenkins Jr. Maraca: Algorithm specification. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://burtleburtle.net/bob/crypto/maraca/nist/`.

[4] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, Nov. 2007.

Publication

Cryptanalysis of Dynamic SHA(2)

Publication Data

Jean-Philippe Aumasson, Orr Dunkelman, Sebastiaan Indesteege, and Bart Preneel. Cryptanalysis of Dynamic SHA(2). In Michael J. Jacobson, Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2009.

Contributions

- Cryptanalysis of Dynamic SHA:
  - Sect. 3 (Collision attack on Dynamic SHA),
  - Sect. 4 (Preimage attack on Dynamic SHA),
  - Appendix A (Practical Results), and
  - Appendices C.1 and C.2 (Extensions to Dynamic SHA-512).
- Contributions to the cryptanalysis of Dynamic SHA2.

Cryptanalysis of Dynamic SHA(2)

Jean-Philippe Aumasson[1,*], Orr Dunkelman[2,†], Sebastiaan Indesteege[3,4,‡], and Bart Preneel[3,4]

[1] FHNW, Windisch, Switzerland.
[2] École Normale Supérieure, INRIA, CNRS, Paris, France.
[3] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven, Belgium.
[4] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.

Abstract. In this paper, we analyze the hash functions Dynamic SHA and Dynamic SHA2, which have been selected as first round candidates in the NIST hash function competition. These hash functions rely heavily on data-dependent rotations, similar to certain block ciphers, e.g., RC5. Our analysis suggests that in the case of hash functions, where the attacker has more control over the rotations, this approach is less favorable than in block ciphers. We present practical, or close to practical, collision attacks on both Dynamic SHA and Dynamic SHA2. Moreover, we present a preimage attack on Dynamic SHA that is faster than exhaustive search.

Key words: Dynamic SHA, Dynamic SHA2, SHA-3 candidate, hash function, collision attack.

1 Introduction

New generic cryptanalytic techniques for hash functions [3,4] and the recent results on MD5 and SHA-1 [1,11,12], along with the fact that the SHA-2 family of hash functions was designed with a similar structure, have led to the initiation of the NIST hash function competition [7], a public competition to develop a new hash standard, which will be called SHA-3.

The competition has sparked a great deal of submissions: 64 new hash function proposals were submitted to the competition, of which 51 were accepted as meeting the submission criteria for the first round. Among the 51 candidates, Dynamic SHA and Dynamic SHA2 stand out as a combination of the SHA family design with data-dependent rotations.

The concept of data-dependent rotations has been explored for block ciphers in several constructions, most notably in the RC5 and RC6 block ciphers [8,9]. The

*Supported by the Swiss National Science Foundation, project no. 113329.
†This author was supported by the France Telecom chaire.
‡F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

security of such block ciphers has been challenged many times, and a majority of attacks is based on guessing the distances of the rotations. In cryptanalysis of hash functions, however, the internal state is known. The attacker even has control over (parts of) the internal state, including rotations, though sometimes this control is only indirect. For example, Mendel et al. [6] exploited data-dependent rotations to find collisions for the hash function of Shin et al. [10]. Our attacks on Dynamic SHA and Dynamic SHA2 also exploit data-dependent rotations, to find (second) preimages and collisions.

2 Brief Description of Dynamic SHA and Dynamic SHA2

Dynamic SHA and Dynamic SHA2 use similar building blocks, but have different compression functions. This section gives a brief description of these algorithms.

Dynamic SHA and Dynamic SHA2 follow a classical Merkle-Damgård construction, based on a compression function that maps an 8-word chaining value and a 16-word message to a new 8-word chaining value. The 256-bit versions use 32-bit words, and the 512-bit versions use 64-bit words. We focus on the 256-bit versions, also called Dynamic SHA-256 and Dynamic SHA2-256. See [13, 14] for details on the 512-bit versions, Dynamic SHA-512 and Dynamic SHA2-512. The following presents a bottom-up description of the compression function, thus starting with its building blocks.

The symbol $\oplus$ stands for exclusive OR (XOR), $\wedge$ for logical AND, $\vee$ for logical OR, and $+$ for integer addition. Numbers in hexadecimal basis are written in typewriter font (e.g., $\texttt{FF} = 255$). We count bit indices starting from zero at the least significant bit (LSB). Thus, the first bit of a word w is written as w^0, and more generally we use the notation w^i for the bit i of the word w. The most significant bit (MSB) of w is thus w^{31} for Dynamic SHA-256, and w^{63} for Dynamic SHA-512. Note that the i-th bit of a word corresponds to the bit number $i-1$, since we start counting from zero.

2.1 Building Blocks

The function G takes as input three words x_1, x_2, x_3 and an integer $t \in \{0, 1, 2, 3\}$, and returns one word, computed as follows:

$$G_t(x_1, x_2, x_3) = \begin{cases} x_1 \oplus x_2 \oplus x_3 & \text{if } t = 0 \\ (x_1 \wedge x_2) \oplus x_3 & \text{if } t = 1 \\ (x_1 \wedge x_2) \oplus x_3 \oplus \neg x_1 & \text{if } t = 2 \\ (x_1 \wedge x_2) \oplus x_3 \oplus \neg x_2 & \text{if } t = 3 \end{cases} .$$

Note that this definition is simplified, but equivalent to the original in [13, 14].

The function R takes as input eight words $x_1, \ldots, x_8$ and an integer t, and returns one word computed as follows:

$$R(x_1, \ldots, x_8, t) = (((((((x_1 \oplus x_2) + x_3) \oplus x_4) + x_5) \oplus x_6) + x_7) \oplus x_8) \ggg t \ .$$

The function $R1$ takes as input eight words $x_1, \ldots, x_8$ and returns one word computed as follows (in the 256-bit versions):

```
$t_0 \leftarrow (((((x_1 + x_2) \oplus x_3) + x_4) \oplus x_5) + x_6) \oplus x_7$
$t_1 \leftarrow ((t_0 \gg 17) \oplus t_0) \wedge \texttt{0001FFFF}$
$t_2 \leftarrow ((t_1 \gg 10) \oplus t_1) \wedge \texttt{000003FF}$
$t_3 \leftarrow ((t_2 \gg 5) \oplus t_2) \wedge \texttt{0000001F}$
**return** $x_8 \ggg t_3$
```

Finally, the COMP function takes as input eight words $a, \ldots, h$ representing the internal state, eight message words $w_0, \ldots, w_7$, or $w_8, \ldots, w_{15}$, and an integer t. COMP updates the internal state as follows (in the 256-bit versions):

T	$\leftarrow$	$R(a, \ldots, h, w_t \bmod 32)$	T	$\leftarrow$	$R(a, \ldots, h, (w_t \gg 15) \bmod 32)$
h	$\leftarrow$	g	h	$\leftarrow$	$g + w_{t+7}$
g	$\leftarrow$	$f \ggg ((w_t \gg 5) \bmod 32)$	g	$\leftarrow$	$f \ggg ((w_t \gg 20) \bmod 32)$
f	$\leftarrow$	$e + w_{t+3}$	f	$\leftarrow$	$e + w_{t+6}$
e	$\leftarrow$	$d \ggg ((w_t \gg 10) \bmod 32)$	e	$\leftarrow$	$d \ggg ((w_t \gg 25) \bmod 32)$
d	$\leftarrow$	$G_{w_t \gg 30}(a, b, c) + w_{t+2}$	d	$\leftarrow$	$G_{t \bmod 4}(a, b, c) + w_{t+5}$
c	$\leftarrow$	b	c	$\leftarrow$	$b + w_t$
b	$\leftarrow$	a	b	$\leftarrow$	a
a	$\leftarrow$	$T + w_{t+1}$	a	$\leftarrow$	$T + w_{t+4}$

2.2 Compression Functions

Given a chaining value $h_0, \ldots, h_7$ and a message block $w_0, \ldots, w_{15}$, the compression function of Dynamic SHA (Dynamic SHA2, respectively) produces a new chaining value, as described in Fig. 1 (Fig. 2, resp.).

The compression function of Dynamic SHA is composed of an initialization, an iterative part of 48 rounds, and a feedforward of the initial chaining value. It uses three constants TT_0, TT_1, TT_2.

The compression function of Dynamic SHA2 is composed of an initialization followed by three iterative parts, and finally by a feedforward. Note that, when calling COMP with the message words $w_8, \ldots, w_{15}$ and an integer t, w_t stands for w_8, w_{t+1} stands for w_9, etc. Dynamic SHA2, surprisingly enough, uses no constants.

Initialization

$$a = h_0 \quad b = h_1 \quad c = h_2 \quad d = h_3 \quad e = h_4 \quad f = h_5 \quad g = h_6 \quad h = h_7$$

Iterative part

for $t = 0, 1 \ldots, 47$:

$$\begin{aligned} T &\leftarrow R1(a, b, c, d, e, f, g, h) \\ U &\leftarrow G(a, b, c, t \bmod 4) + w_{t \bmod 16} + TT_{t \gg 4} \\ (a, b, c, d, e, f, g, h) &\leftarrow (T, a, b, U, d, e, f, g) \end{aligned}$$

Feedforward

$$\begin{aligned} h_0 &\leftarrow h_0 + a \quad h_1 \leftarrow h_1 + b \quad h_2 \leftarrow h_2 + c \quad h_3 \leftarrow h_3 + d \\ h_4 &\leftarrow h_4 + e \quad h_5 \leftarrow h_5 + f \quad h_6 \leftarrow h_6 + g \quad h_7 \leftarrow h_7 + h \end{aligned}$$

Figure 1 – Compression function of Dynamic SHA.

Initialization

$$a = h_0 \quad b = h_1 \quad c = h_2 \quad d = h_3 \quad e = h_4 \quad f = h_5 \quad g = h_6 \quad h = h_7$$

First iterative part

$$\mathsf{COMP}\,(a, b, c, d, e, f, g, h, w_0, w_1, \ldots,\ w_7, 0)$$
$$\mathsf{COMP}\,(a, b, c, d, e, f, g, h, w_8, w_9, \ldots, w_{15}, 0)$$

Second iterative part

```
for $t = 0, 1 \ldots, 8$:
    $T \leftarrow R1(a, b, c, d, e, f, g, h)$
    $(a, b, c, d, e, f, g, h) \leftarrow (T, a, b, c, d, e, f, g)$
```

Third iterative part

```
for $t = 1, 2 \ldots, 7$ :
    $\mathsf{COMP}\,(a, b, c, d, e, f, g, h, w_0, w_1, \ldots,\ w_7, t)$
    $\mathsf{COMP}\,(a, b, c, d, e, f, g, h, w_8, w_9, \ldots, w_{15}, t)$
```

Feedforward

$$\begin{array}{lllll} h_0 & \leftarrow & h_0 + a & h_1 \leftarrow h_1 + b & h_2 \leftarrow h_2 + c & h_3 \leftarrow h_3 + d \\ h_4 & \leftarrow & h_4 + e & h_5 \leftarrow h_5 + f & h_6 \leftarrow h_6 + g & h_7 \leftarrow h_7 + h \end{array}$$

Figure 2 – Compression function of Dynamic SHA2.

3 Collision Attack on Dynamic SHA

This section describes a practical collision attack on Dynamic SHA. It builds on a 9-step local collision that exploits an important differential property of the function $R1$, which we introduce first. The same local collision pattern is repeated three times to find collisions for the entire compression function. Furthermore, these three instances of the local collision pattern can be decoupled, which drastically reduces the attack complexity. We present the attack on Dynamic SHA-256 here. We could adapt it to Dynamic SHA-512 with only minimal changes, as detailed in Appendix C.

3.1 A Differential Property of the Function $R1$

To overcome the obstacle of data-dependent rotation, our attack ensures that no difference occurs in any of the data-dependent rotation amounts. This section clarifies how to achieve this.

The data-dependent rotations are located in the 8-input function $R1$. For Dynamic SHA-256, consider the difference $\Delta = \texttt{80004000}$, i.e., only bits 31 and 14 are set. Let one of the first seven inputs to the function $R1$ have this difference, i.e., one of $x_1, \ldots, x_7$. In the first step of $R1$, an intermediary word t_0 is computed as follows:

$$t_0 \leftarrow ((((((x_1 + x_2) \oplus x_3) + x_4) \oplus x_5) + x_6) \oplus x_7 \ .$$

The difference in the MSB always propagates to t_0. Assuming that no carry occurs for bit 14, the intermediary t_0 also has the difference Δ. If t_0 has a difference Δ, this difference is then absorbed by the rest of the function $R1$. Indeed, the next step computes the intermediary word t_1 as

$$t_1 \leftarrow ((t_0 \gg 17) \oplus t_0) \wedge \texttt{0001FFFF} \ .$$

Note that $(\Delta \gg 17) \oplus \Delta = \texttt{80000000}$, which is absorbed by the logical AND operation. We note that there are other differences of Hamming weight 2 that exhibit the same property and may be used without any change in the attack, e.g., $\Delta = 80000010$

We now estimate the probability that a single Δ-difference in one of the first seven inputs of the function $R1$ is absorbed. As a Δ-difference in t_0 is absorbed with certainty, it suffices that a Δ-difference in one of the seven first inputs propagates to t_0. This happens when no carry difference occur for bit 14 in any of the modular additions. The probability that a one-bit difference in one of the summands in an addition does not cause a carry difference is $1/2$. Thus, the probability that a Δ-difference is absorbed by the function $R1$ can be estimated to 2^{-k}, where k is the number of modular additions the difference propagates through. For instance, a difference in x_3 activates two modular additions, so $k = 2$.

However, the actual probability is higher, as the undesirable effects of a carry difference in one modular addition can be reverted by another carry difference

Table 1 – A 9-step local collision for Dynamic SHA. The difference at step t is the difference in the state *before* computing step t.

t	a	b	c	d	e	f	g	h	w	Pr
0	0	0	0	0	0	0	0	0	Δ	2^{-1}
1	0	0	0	Δ	0	0	0	0	0	$2^{-1.58}$
2	0	0	0	0	Δ	0	0	0	0	2^{-1}
3	0	0	0	0	0	Δ	0	0	0	2^{-1}
4	0	0	0	0	0	0	Δ	0	0	1
5	0	0	0	0	0	0	0	Δ	0	2^{-5}
6	Δ	0	0	0	0	0	0	0	0	$2^{-2.07} \cdot 2^{-2}$
7	0	Δ	0	0	0	0	0	0	0	$2^{-2.07} \cdot 2^{-2}$
8	0	0	Δ	0	0	0	0	0	Δ	$2^{-1.58} \cdot 2^{-1}$
	0	0	0	0	0	0	0	0		

in a subsequent addition. The combination of modular additions and XOR can be represented compactly in a trellis, and a variant of the Viterbi algorithm can be used to efficiently count the probability that a Δ-difference is passed to t_0 unchanged. Our computer aided research revealed that this is indeed an important effect: For a difference in x_3 or x_4, the actual probability is $2^{-1.58}$ rather than 2^{-2}, and for a difference in x_1 or x_2, the actual probability is $2^{-2.07}$ rather than 2^{-3}. For differences in the other words, only one modular addition is affected, so no carry differences can be canceled. Hence, in those cases, the simple estimation is correct.

3.2 A 9-Step Local Collision

We present a simple 9-step local collision for Dynamic SHA in Table 1. A difference of $\Delta =$ 80004000 is introduced, then, all further diffusion of this difference is avoided. After seven more steps, the difference has rotated through the internal state of Dynamic SHA once, and can be canceled via an appropriate difference in the message word. The characteristic has probability $2^{-20.3}$.

In step 0, a Δ-difference is introduced via the message word. Note that the message word itself can contain any additive difference that can cause a Δ-difference in the state. In steps 1 to 4, the Δ-difference in one of the state variables is absorbed by the function $R1$, as described in Section 3.1. Then, at the beginning of step 5, there is a Δ-difference in the internal state word h. This word is rotated by a data-dependent amount, and thus we can require that it is rotated by zero bits, i.e., not rotated at all. In steps 6 and 7, the Δ-difference should be absorbed by the G-functions. Any G-function except XOR absorbs differences in its first two inputs with probability 1/2 per bit. Also, $R1$ should absorb the differences in these steps. Finally, in step 8, the difference in the state variable c is canceled by another Δ-difference coming from the message word.

The probability that the local collision pattern is followed is estimated by simply multiplying the probabilities of all the events discussed above. The probabilities of each step are indicated in Table 1. This yields an overall probability of $2^{-20.3}$ for the entire 9-step local collision.

3.3 The Attack

Our attack repeats the 9-step collision three times. This made possible by the simple message schedule, which consists of a simple repetition of the 16 words in a message block. Thus, the only message words that have a difference are w_0, which introduces the differences, and w_8, which cancels them.

A straightforward attack would consist of choosing an arbitrary message block, and applying a difference of $\Delta = \texttt{80004000}$ to w_0 and w_8. As the local collision is repeated three times, the complexity of this attack would be approximately $(2^{20.3})^3 = 2^{61}$. This can be improved tremendously by making the three local collisions independent. Then, the three local collision complexities can be added rather than multiplied.

The first two local collisions can be decoupled in a straightforward manner as only the message words w_0 to w_8 influence the first local collision. Therefore, once suitable values for these message words have been found, there is still enough freedom remaining in the other message words. The words w_0 to w_8 can thus be kept constant, while values for w_9 to w_{15} are searched such that the second local collision is also achieved.

Controlling Internal State Values. In each step of Dynamic SHA, the new value of the internal state word d is found as the modular addition of a message word and an intermediate depending on the internal state words a, b and c. Full control over message words allows an adversary to give the internal state word d any desired value. Indeed, it holds that

$$w_{t \bmod 16} = d_{\text{new}} - G(a, b, c, t \bmod 4) - TT_{t \gg 4} \ .$$

Applying this to eight consecutive steps allows one to almost fully control the final internal state. In every step, the new value of d is fixed to some desired value. These values then shift through the internal state words a number of times, to end up as one of the internal state words after the eighth step. However, a complication arises with the first three steps, which ends up in the state words a, b and c. Before a controlled value from d ends up in one of these three state words, it is be rotated by a data-dependent amount. An obvious way to sidestep this issue is to choose a rotation-invariant value for these three words, i.e., `00000000` or `FFFFFFFF`. Then, the data-dependent rotations have no influence.

Decoupling All Three Local Collisions. Our attack consists of three phases, each dealing with one local collision. The first phase satisfies the first local collision, using the message words w_0 to w_8. It would be possible to use message modification techniques here to find a conforming message pair quicker, but as the later phases of the attack dominate the overall complexity anyway, no significant gains can be made in this way.

To satisfy the second local collision, we use the freedom in the remaining message words. However, we do not choose the remaining message words directly, but rather choose the internal state after step 15. We then use the words w_8 to w_{15} to connect to this state, using the technique outlined earlier. We fix the values of a, b and c to zero, to make them rotation-invariant, and choose the remaining five words arbitrarily. Note that w_8 was already determined in phase 1, so it should not be modified again, but w_8 is used here to force a zero value, which ends up in the internal state word d after step 15. This issue is solved by shifting this condition on w_8 to phase 1. Instead of arbitrarily choosing w_8 there, it is computed such that the required zero is generated. This does not change the complexity of the first phase.

Finally, to satisfy the third local collision, we modify w_7. Then, only d changes after step seven. As the value in w_8, which should force d to zero after step eight, depends only on the internal state words a, b and c before step eight, modifying w_7 does not require a correction in w_8. Thus, such modifications do not change the fact that the first local collision pattern is followed. The values of w_9 to w_{15} are then updated such that the internal state after step 15 is unchanged, and so the start of the second local collision will be unaltered. For the same reasons as before, the change in w_7 also does not affect the end of the second local collision pattern.

Hence, we dispose of a modification algorithm that leaves the first two local collisions unaffected, but changes the internal state values before the third local collision randomly. This provides the required freedom to also satisfy this third and final local collision. Hence, the overall attack complexity can be estimated at about 2^{21} Dynamic SHA compression function computations. Appendix A reports on our implementation of the attack, with an example of collision.

4 Preimage Attack on Dynamic SHA

This section describes (first and second) preimage attacks on Dynamic SHA. We first describe how to find preimages for the compression function of Dynamic SHA, and then explain how to extend this to first and second preimage attacks. on the Dynamic SHA hash function. We describe how to attack Dynamic SHA-256 here, and refer to Appendix C for details on how to adapt the attack to Dynamic SHA-512.

Conceptually, our preimage attack bears some similarity to the work on SHA-0 and SHA-1 by De Cannière and Rechberger [2], for it finds a preimage bit slice per bit slice. If all data-dependent rotation amounts in Dynamic SHA are assumed to be zero, then a bit of any intermediate word cannot be influenced by any other bit of higher position. This is because, besides rotations, all operations are either bit-wise or modular additions.

4.1 Preimage Attack on the Compression Function

Assume that the rotations in a block of Dynamic SHA are all zero. Then, all words in Dynamic SHA can be divided into bit slices, as all computations are now T-functions [5]. As noted above, bit i of each word can only be influenced by bits 0 to i of other words. When bits 0 to $(i-1)$ of each word are known, bit i of all words can be determined.

In a preimage attack on the Dynamic SHA compression function, the internal state is given before step 0 and after step 47. Our attack starts by determining the LSB of each word. To determine this bit of all of the internal state words in every step, only the LSBs of the 16 message words need to be known. There are 2^{16} choices for these 16 bits. Then, it can be verified whether the LSBs of the eight internal state words after step 47 are correct. This occurs with probability 2^{-8}, so 2^8 choices are expected to survive.

We then proceed to the next bit slice. Keeping the choice for the LSB slice fixed, the same procedure can be repeated. For each choice of the LSB slice again 2^8 choices for the second LSB are expected to survive. For Dynamic SHA-256, this procedure is repeated until the 28 LSBs (bits 0–27) have been determined. At that point, one of the bits of each of the 48 rotation constants can be determined, as it does not depend on the higher bits of any word. Now, it can be verified if the initial assumption that all rotation constants are zero indeed holds. This corresponds to a 48-bit condition, i.e., for all rotation constants to be zero, surely this single bit of each rotation constant has to be zero. Any choices that do not satisfy this condition are eliminated. Then, the next bit is determined as before, after which another bit of each rotation constant can be verified. This is repeated until all bits have been determined.

4.2 Complexity Evaluation

The attack can be described as a simple tree search, where a tree level corresponds to a bit slice, and a node represents an assignment for all bits in the slice under consideration, and all LSB slices. To expand a node in the tree, one guesses the 16 message bits of the next slice, and checks that the conditions on the state words after step 47 are satisfied. As explained above, on average about 2^8 choices are expected to survive, i.e., the tree has a branching factor of 2^8. When the 28 LSB slices are known, however, the average number of child nodes drops by 2^{-48} due to the additional filtering. The cost of expanding one node is about 2^{16} Dynamic SHA compression function evaluations, as 2^{16} choices have to be investigated. The expected number of solutions is equal to the expected number of nodes at the deepest level of the tree, which is $2^{8 \cdot 32} \cdot 2^{-48 \cdot 5} = 2^{16}$. This agrees with the observation that for a given input/output chaining values of the compression function, there are expected to be 2^{256} message blocks that conform to this combination. For each of these, the probability that all the rotations are by 0 positions is 2^{-240}, so about 2^{16} remain.

As we aim to find just one solution, i.e., any node on the deepest level of the tree, a depth-first search is well suited to our application. It requires only negligible memory and can easily be parallelised. Since, for Dynamic SHA-256, 2^{16} solutions are expected, the depth-first search needs to search only about a fraction 2^{-16} of the entire tree before encountering the first solution. Due to the large branching factor, the total number of nodes in the tree is well approximated by the number of nodes on the widest level of the tree, which has $2^{8 \cdot 27} = 2^{216}$ nodes for Dynamic SHA-256. The search is thus expected to expand about 2^{200} nodes, each of which costs 2^{16} Dynamic SHA-256 compression function evaluations, resulting in a total attack complexity of 2^{216} Dynamic SHA-256 compression function evaluations.

4.3 Application to the Hash Function

Our preimage attack on the compression function directly gives a second preimage attack on the Dynamic SHA hash function with the same complexity, provided that there is at least one message block that does not contain any padding in the challenge message.

For a first preimage, the padding bits limit the control an attacker has over the message bits. It is not possible to simply copy the padding as in a second preimage attack. Thus, we use the following approach instead. First, choose a message length such that the last padded message block only contains 65 bits of padding, which is the minimum. Then, choose an arbitrary message for all but the last message block. Finally, a modified version of the attack in Section 4.1 is used to determine the last message block.

The main difference is that the last 65 bits of the message block can not be chosen by the adversary, as they are padding bits. Their contents are fixed by the choice of the message length. However, the same approach as in Section 4.1 can still be applied, except that fewer bits can be chosen in each bit slice. For Dynamic SHA-256, the expected number of solutions in the search tree now becomes $2^{6 \cdot 27} \cdot 2^{-42 \cdot 4} \cdot 2^{-43 \cdot 1} = 2^{-49}$. A solution is thus only expected to exist with probability 2^{-49}, thus the attack is repeated sufficiently many times with a different message length. The number of nodes at the widest level of the tree is $2^{6 \cdot 27}$, and the cost for expanding a single node at this level is 2^{14} Dynamic SHA compression function calls. Thus, the total attack complexity becomes approximately $2^{49} \cdot 2^{6 \cdot 27} \cdot 2^{14} = 2^{225}$ Dynamic SHA compression function evaluations.

5 Collision Attack on Dynamic SHA2

To attack Dynamic SHA2, we use similar ideas as for Dynamic SHA. Specifically, we use the control of the message to ensure that as many rotations as possible are by the amounts that we need. Moreover, as many of the rotations amounts are directly determined by the message, our task becomes easier. Our attack is based on introducing a difference in the most significant bit of two message words, w_8 and w_{14}. As a 32-bit condition is imposed on the chaining value, a two-block collision finding technique is used, where the first block is searched until a suitable chaining value is encountered. We describe our attack on Dynamic SHA2-256 here. It can be adapted to Dynamic SHA2-512, as Appendix C shows.

5.1 First Iterative Part

Given an initial value $a, \ldots, h$, the first iterative part of the compression function of Dynamic SHA2 updates the chaining value words $a, \ldots, h$ by computing

$$\mathsf{COMP}(a, b, \ldots, h, w_0, w_1, \ldots, w_7) \ ,$$

Since there is no difference in the message words $w_0, \ldots, w_7$ nor in the initial value, we have no difference at this stage.

Then, Dynamic SHA2 computes

$$\mathsf{COMP}(a, b, \ldots, h, w_8, w_9, \ldots, w_{15}) \ .$$

To follow our characteristic, the difference in w_8 and in w_{14} should lead to a difference $\Delta = \mathtt{80000000}$ in c and in f. Below, we show that, to obtain these differences, it suffices to set $w_8^{30} = 1$ and to ensure that b equals FFFFFFFF after the first COMP. These conditions are easily satisfied, and do not increase the complexity of our attack.

We note that w_{14} is used only once in the first iterative part. Thus the difference Δ in w_{14} only propagates to f, when COMP sets $f \leftarrow e + w_{14}$. The word w_8, however, is used eight times, but as only the MSB has a difference, only two of these require our attention: first, when setting $c \leftarrow b + w_8$ (which gives the difference Δ in c with probability one), and second when setting

$$d \leftarrow G_{w_8 \gg 30}(a, b, c) + w_{10} \ .$$

Here, the two MSBs of w_8 encode the index of the function used in G. Since we have a difference in the MSB of w_8, different functions are applied to (a, b, c). To obtain the same output, we require that the functions G_1 and G_3 are used, that is, we set the bit $w_8^{30} = 1$. The reason for this is that, when b equals FFFFFFFF, it is ensured that the outputs of both functions are equal, as can readily be seen from the definition of the G-functions in Section 2.1.

To summarize, a difference Δ in w_8 and w_{14} yields a difference Δ in c and f after the first iterative part. To have $b =$ FFFFFFFF, it is sufficient to start

Table 2 – Differential characteristic for the second iterative part of Dynamic SHA2. The difference at step t is the difference in the state *before* computing step t.

t	a	b	c	d	e	f	g	h
0	0	0	Δ	0	0	Δ	0	0
1	0	0	0	Δ	0	0	Δ	0
2	0	0	0	0	Δ	0	0	Δ
3	Δ	0	0	0	0	Δ	0	0
4	0	Δ	0	0	0	0	Δ	0
5	0	0	Δ	0	0	0	0	Δ
6	Δ	0	0	Δ	0	0	0	0
7	0	Δ	0	0	Δ	0	0	0
8	0	0	Δ	0	0	Δ	0	0

from a chaining values that gives at the very first COMP a T such that $T + w_1 =$ FFFFFFFF. Such a chaining value can be reached in about 2^{32} trials, and needs to be precomputed only once. That is, one first needs to find a message block leading to a chaining value that satisfies $T + w_1 =$ FFFFFFFF, before starting the actual differential attack with a second block. Actually, by using the freedom in w_0 and w_1 rather than fixing them a priori, this step can be accelerated further. However, as the other parts of the attack dominate the overall complexity, no significant gains can be made in this way.

5.2 Second Iterative Part

Table 2 describes our differential characteristic for the second iterative part of Dynamic SHA2. Note that no message word enters this part. A set of conditions that ensure that this characteristic is followed, is relatively simple. Indeed, except when $t = 2$ and $t = 5$, the two differences Δ vanish in the first step of the computation of $R1$, namely when computing

$$(((((a + b) \oplus c) + d) \oplus e) + f) \oplus g \ .$$

Therefore, particular conditions are only required for $t = 2$ and $t = 5$.

When $t = 2$, the difference in e gives a difference of 16 in the rotation amounts, and so the function $R1$ returns $h \ggg r$ and $(h \oplus \Delta) \ggg (r + 16 \bmod 32)$, respectively. In order to obtain, as required by our differential characteristic, the relation

$$(h \ggg r) \oplus \Delta = (h \oplus \Delta) \ggg (r + 16 \bmod 32) \ ,$$

a sufficient condition is to have $r = 16$, and h invariant under 16-bit rotation, i.e., $(h \ggg 16) = h$. This means that h should be of the form XYZTXYZT, which we call *symmetric*. When $t = 5$, we require similar conditions.

Now, observe that the words that should be symmetric are c and f obtained after the first iterative part. The values of c and f then directly depend on w_8 and w_{14} (see description of COMP in Section 2). We now have to find values of w_8 and of w_{14} that give symmetric c and f.

Such w_8 and w_{14} can be found as follows: first fix w_{14} to some arbitrary value, and search for a w_8 that gives a symmetric c, in 2^{16} trials. Then, fix w_8 to the value found, and search for a pair (w_5, w_{14}) that gives a symmetric f after the first iterative part. Here we need w_5 to have enough freedom, since for certain choices of w_5, there does not exist a suitable w_{14}. Again, 2^{16} trials are expected. Then we are enough degrees of freedom in the message words that do not affect c and f to find rotation $r = 16$.

Assuming symmetric c and f after the first iterative part, the characteristic is followed with probability 2^{-10}, since the condition $r = 16$ is satisfied for both $t = 2$ and $t = 5$ with probability $2^{-5} \times 2^{-5}$. By trying several values of, for example, w_9, and leaving the other message words fixed, one can thus find a conforming message pair for the first two iterative parts in about 2^{10} trials.

5.3 Third Iterative Part

Given the final difference of the second iterative part, we found a characteristic for the second round that yields no difference in the final state, thus given a collision. Table 6 in Appendix B describes our differential characteristic. Appendix B also explains in detail why the characteristic can be followed with probability 2^{-42}, given some conditions on the input.

Combining our differential characteristics with their respective conditions on the message, we obtain a method for finding a 2-block collision in about $2^{42+10} = 2^{52}$ trials. The attack succeeds with probability close to one.

6 Conclusion

In this paper we have discussed the security of the two SHA-3 candidates Dynamic SHA and Dynamic SHA2. We have analyzed their security, and found out that, despite their reliance on data-dependent rotations and in the case of Dynamic SHA2 even data-dependent functions, their security is subverted by the vast control and knowledge the adversary has while attacking a hash function. We also showed that neither Dynamic SHA nor Dynamic SHA2 are suitable to be selected as SHA-3, following their lack of security. Table 3 summarizes our results.

Acknowledgements

The research presented in this paper was performed in part while the authors were visiting Schloss Dagstuhl (`http://www.dagstuhl.de/`) in January 2009.

Table 3 – Summary of our results.

Hash Function	Attack	Complexity	Section
Dynamic SHA-256	Collision	2^{21}	3
Dynamic SHA-512	Collision	2^{22}	3,C
Dynamic SHA-256	Second preimage	2^{216}	4
Dynamic SHA-512	Second preimage	2^{256}	4,C
Dynamic SHA-256	First preimage	2^{225}	4
Dynamic SHA-512	First preimage	2^{262}	4,C
Dynamic SHA2-256	Collision	2^{52}	5
Dynamic SHA2-512	Collision	2^{85}	5,C

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

References

[1] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. In X. Lai and K. Chen, editors, *Advances in Cryptology — ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[2] C. De Cannière and C. Rechberger. Preimages for reduced SHA-0 and SHA-1. In D. Wagner, editor, *Advances in Cryptology — CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 179–202. Springer, 2008.

[3] J. Kelsey and T. Kohno. Herding hash functions and the Nostradamus attack. In S. Vaudenay, editor, *Advances in Cryptology — EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[4] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[5] A. Klimov and A. Shamir. Cryptographic applications of T-Functions. In M. Matsui and R. J. Zuccherato, editors, *Selected Areas in Cryptography — SAC 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 2004.

[6] F. Mendel, N. Pramstaller, and C. Rechberger. Improved collision attack on the hash function proposed at PKC'98. In M. S. Rhee and B. Lee, editors, *Information Security and Cryptology — ICISC 2006*, volume 4296 of *Lecture Notes in Computer Science*, pages 8–21. Springer, 2006.

[7] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, Nov. 2007.

[8] R. L. Rivest. The RC5 encryption algorithm. In B. Preneel, editor, *Fast Software Encryption, Second International Workshop — FSE '94*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 1995.

[9] R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin. RC6 as the AES. In *Third AES Candidate Conference*, pages 337–342. National Institute of Standards and Technology, 2000.

[10] S. U. Shin, K. H. Rhee, D. Ryu, and S. Lee. A new hash function based on MDx-family and its application to MAC. In H. Imai and Y. Zheng, editors, *Public Key Cryptography — PKC '98*, volume 1431 of *Lecture Notes in Computer Science*, pages 234–246. Springer, 1998.

[11] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In M. Naor, editor, *Advances in Cryptology — EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.

[12] X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[13] Z. Xu. Dynamic SHA. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[14] Z. Xu. Dynamic SHA2. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

A Practical Results

We have implemented our collision attack on Dynamic SHA. Collisions for Dynamic SHA-256 and Dynamic SHA-512 are found in a matter of seconds on an average desktop PC. A collision example for Dynamic SHA-256 is given in Table 4. An all-zero block was appended to both messages to circumvent an error in the padding routine of the Dynamic SHA reference implementation, which causes part of the last message block to be reused in the padding block.

Table 4 – Collision example for Dynamic SHA-256: two messages and their common digest.

34BC5378	1150D86C	3085EB92	7538ECEE	199FB91A	5A9614EC	4D21FB88	728FF21E
22FBFA2E	08CE50DF	95CDE61F	71E5F222	3D30C361	EB7676B8	F1AE9728	758B70AF
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
B4BC9378	1150D86C	3085EB92	7538ECEE	199FB91A	5A9614EC	4D21FB88	728FF21E
A2FBBA2E	08CE50DF	95CDE61F	71E5F222	3D30C361	EB7676B8	F1AE9728	758B70AF
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
703C40F7	9DDFE2C6	8298F6D0	8D2B45B6	664CBB71	8BAB1BE3	DD563F77	0D0901E6

B Differential Characteristic for Dynamic SHA2

This appendix describes the differential characteristic for the third iterative part of Dynamic SHA2, used in our collision attack presented in Section 5.

A transition in Table 6 has probability 1/2 when there is a difference in a or b and G_1, G_2 or G_3 is used. In this case, the difference does (not) propagate with probability 1/2. When there is a difference only in c, it always propagates to the output of the G function, independent of the function used. We also note that a difference Δ in one operand of R is always transferred to T, and thus to a, except when w_{t+1} or w_{t+4} are w_8 or w_{14}, in which case the differences vanish. When two operands of T have a difference Δ, they cancel out and yield no difference in T.

The probabilities for each step assume some conditions on the message. We will take as example the first COMP when $t = 2$: we start with a difference

$$0 \quad \Delta \quad 0 \quad \Delta \quad 0 \quad 0 \quad 0 \quad 0$$

in the chaining value $a, b, \ldots, h$. In the computation of COMP (first half), there is no difference in T, because the Δ difference in b cancels that of d. The assignment of the new values of f, g, h requires no condition on the message, for it only involves words with no difference. To obtain a difference Δ in e, we need that d is rotated by zero bit positions, that is, we need the bits 10 to 14 of w_2 to be zero. This is easy as we have direct control over w_2. Then, to obtain no difference in d, we require that the difference in b does not propagate in G. This is only possible if the Boolean function in G is not $x_1 \oplus x_2 \oplus x_3$ (see Section 2.1). Since the Boolean function is determined by the last two bits of w_2, we require $w_2^{30} \vee w_2^{31} = 1$, i.e., these bits should not be both zero. Now, the difference will not propagate in G with probability 1/2. Finally, we get a difference Δ in c with probability 1.

By applying a similar reasoning to all the steps of our differential characteristic, we obtain conditions on the message $w_0, \ldots, w_{15}$ that are sufficient to conform to the characteristic with probability 2^{-42}. Table 5 summarizes these conditions, along with the conditions for the other iterative parts.

Conditions on $w_0, \ldots, w_7$ ensure that in the first COMP of each step the rotations are by bit zero positions, and thus the difference remains in the MSB. The probabilities smaller than one are the probabilities that the function G absorbs or passes a difference in a or b. In the second COMP, we need some rotations to be zero in order the difference to stay in the MSB. This is achieved by setting conditions on the message, for example at $t = 1$, the first ten bits of w_9 should be zero. Table 5 summarizes these conditions. After satisfying all these conditions, about 200 bits of freedom remain; indeed, besides w_8 and w_{14}, the message words w_1 to w_4 have to be fixed to let the symmetric c and f unchanged after the first iterative part.

At step $t = 6$, the difference in the MSB of w_{14} implies that G will apply different functions to (a, b, c). Similarly to Section 5.1, we will require $w_{14}^{30} = 1$ and $b = \texttt{EFFFFFFF}$, which will occur with probability 2^{-32}. The MSB of b should be zero in order the difference to propagate, which will happen with probability $1/2$, thus the total probability for this step $1/2 \times 2^{-32} = 2^{-33}$

C Extensions to the 512-bit Versions

The attacks presented in this paper can be extended to the 512-bit versions of Dynamic SHA and Dynamic SHA2 in a straightforward way. This appendix details how the attacks can be adapted.

C.1 Collision Attack on Dynamic SHA

The attack on Dynamic SHA-256 can be adapted to Dynamic SHA-512 with almost no change. Due to the different $R1$ function, the difference word is $\Delta = 8000000080000000$. Also, the probability of the local collision is lowered by about 2^{-1} compared to Dynamic SHA-256, as in the fifth step six rotation bits have to be fixed to zero instead of only five.

C.2 Preimage Attack on Dynamic SHA

The preimage attack on Dynamic SHA-512 is similar to that on Dynamic SHA-256, except that the 59 LSBs are determined, instead of the 28 LSBs. Then, when building the tree, 2^{224} solutions are expected, leading to an attack complexity of 2^{256} on the compression function. Calculations for preimages on the full hash function (with correct padding bits) give a cost of of 2^{262} compression function evaluations.

C.3 Collision Attack on Dynamic SHA2

To attack Dynamic SHA2-512 we use a similar differential path. The changes are that the condition on the first block is on 64 bits (starting from a chaining

Table 5 – Conditions on the message words $w_0, \dots, w_{15}$: sufficient to follow our differential characteristic in Dynamic SHA.

Word	Condition
w_0	–
w_1	$w_1 = 0$
w_2	$w_2^{10} = \cdots = w_2^{14} = 0$, $w_2^{25} = \cdots = w_2^{29} = 0$, $w_2^{30} \vee w_2^{31} = 1$
w_3	$w_3^{30} \vee w_3^{31} = 1$
w_4	$w_4^{20} = \cdots = w_4^{29} = 0$, $w_4^{30} \vee w_4^{31} = 1$
w_5	$w_5^{5} = \cdots = w_5^{9} = 0$
w_6	$w_6^{0} = \cdots = w_6^{4} = 0$, $w_6^{15} = \cdots = w_6^{19} = 0$, $w_6^{20} = \cdots = w_6^{29} = 0$
w_7	$w_7^{5} = \cdots = w_7^{14} = 0$, $w_7^{20} = \cdots = w_7^{24} = 0$
w_8	difference in w_8^{31}, $w_8^{30} = 1$
w_9	$w_9^{0} = \cdots = w_9^{9} = 0$
w_{10}	$w_{10}^{5} = \cdots = w_{10}^{14} = 0$
w_{11}	$w_{11}^{15} = \cdots = w_{11}^{29} = 0$, $w_{11}^{30} \vee w_{11}^{31} = 1$
w_{12}	$w_{12}^{10} = \cdots = w_{12}^{24} = 0$
w_{13}	$w_{13}^{0} = \cdots = w_{13}^{4} = 0$, $w_{13}^{15} = \cdots = w_{13}^{24} = 0$
w_{14}	difference in w_{14}^{31}, $w_{14}^{10} = \cdots = w_{14}^{14} = 0$, $w_{14}^{20} = \cdots = w_{14}^{29} = 0$, $w_{14}^{30} = 1$
w_{15}	$w_{15}^{0} = \cdots = w_{15}^{9} = 0$

Table 6 – Differential characteristic for the third iterative part of Dynamic SHA2. The difference at step t is the difference in the state *before* computing step t. The column T indicates the difference in the temporary variable T. The probability on a line is the probability to reach the *next* difference, when conditions on the message are satisfied.

t	(message input)	a	b	c	d	e	f	g	h	T	prob.
1	$(w_1, \ldots, w_0)$	0	0	0	Δ	0	0	Δ	0	0	1
		0	0	0	0	Δ	0	0	Δ	0	1
1	$(w_9, \ldots, w_8)$	0	0	0	0	0	Δ	0	0	Δ	1
		Δ	0	0	0	0	0	Δ	0	0	2^{-1}
2	$(w_2, \ldots, w_1)$	0	Δ	0	Δ	0	0	0	0	0	2^{-1}
		0	0	Δ	0	Δ	0	0	0	0	1
2	$(w_{10}, \ldots, w_9)$	0	0	0	Δ	0	Δ	0	0	0	1
		0	0	0	0	Δ	0	Δ	0	0	1
3	$(w_3, \ldots, w_2)$	Δ	0	0	0	0	0	0	Δ	0	2^{-1}
		0	Δ	0	0	0	0	0	0	Δ	2^{-1}
3	$(w_{11}, \ldots, w_{10})$	Δ	0	Δ	0	0	0	0	0	0	2^{-1}
		0	Δ	0	Δ	0	Δ	0	0	Δ	2^{-1}
4	$(w_4, \ldots, w_3)$	Δ	0	Δ	0	Δ	0	Δ	0	0	2^{-1}
		0	Δ	0	Δ	0	Δ	0	Δ	0	1
4	$(w_{12}, \ldots, w_{11})$	0	0	Δ	Δ	Δ	0	Δ	0	0	1
		0	0	0	0	Δ	Δ	0	Δ	Δ	1
5	$(w_5, \ldots, w_4)$	0	0	0	0	0	Δ	Δ	0	0	1
		0	0	0	0	0	0	Δ	Δ	0	1
5	$(w_{13}, \ldots, w_{12})$	0	0	0	0	0	0	0	Δ	Δ	1
		0	0	0	0	0	Δ	0	0	Δ	1
6	$(w_6, \ldots, w_5)$	Δ	0	0	0	0	0	Δ	0	0	1
		0	Δ	0	Δ	0	0	0	Δ	Δ	2^{-1}
6	$(w_{14}, \ldots, w_{13})$	Δ	0	Δ	Δ	Δ	0	0	0	Δ	2^{-33}
		0	Δ	0	Δ	Δ	Δ	0	0	0	2^{-1}
7	$(w_7, \ldots, w_6)$	0	0	0	Δ	Δ	Δ	Δ	0	0	1
		0	0	0	0	Δ	Δ	Δ	Δ	0	1
7	$(w_{15}, \ldots, w_{14})$	0	0	0	0	0	Δ	Δ	Δ	Δ	1
		0	0	0	0	0	0	Δ	Δ	0	1
		0	0	0	0	0	0	0	0		

value with $b = \mathtt{FFFFFFFFFFFFFFFF}$), the fact that in the second iterative part the probability is 2^{-6} for each of the two transitions, the decrease in the probability only of the sixth COMP from 2^{-33} to 2^{-65}, and the different set of conditions on the message described in Table 7. Hence, the total time complexity of this attack is 2^{85}. We note that in this approach the attack fixes w_i^{60} and w_i^{61} to $i \bmod 4$ (which causes the same function to be used in this case as in the attack on Dynamic SHA2-256).

Table 7 – Conditions on the message words $w_0, \ldots, w_{15}$: sufficient to follow our differential characteristic in Dynamic SHA2-512

Word	Condition
w_0	–
w_1	$w_1 = 0$,$w_1^{18} = \cdots = w_1^{23} = 0$, $w_1^{42} = \cdots = w_1^{47} = 0$, $w_1^{60} = 1, w_1^{61} = 0$
w_2	$w_2^{18} = \cdots = w_2^{29} = 0$, $w_2^{42} = \cdots = w_2^{47} = 0$, $w_2^{60} = 0, w_2^{61} = 1$, $w_2^{62} \vee w_2^{63} = 1$
w_3	$w_3^{54} = \cdots w_3^{59} = 0$, $w_3^{60} = w_3^{61} = 1$, $w_3^{62} \vee w_3^{63} = 1$
w_4	$w_4^{6} = \cdots = w_4^{11} = 0$, $w_4^{18} = \cdots = w_4^{23} = 0$, $w_4^{42} = \cdots = w_4^{47} = 0$, $w_4^{60} = w_4^{61} = 0$, $w_4^{62} \vee w_4^{63} = 1$
w_5	$w_5^{6} = \cdots = w_5^{11} = 0$, $w_5^{60} = 1$, $w_5^{61} = 1$
w_6	$w_6^{48} = \cdots = w_6^{53} = 0$, $w_6^{60} = 0, w_6^{61} = 1$
w_7	$w_7^{6} = \cdots = w_7^{23} = 0$, $w_7^{36} = \cdots = w_7^{53} = 0$, $w_7^{60} = w_7^{61} = 1$
w_8	difference in w_8^{63}, $w_8^{62} = 1$
w_9	$w_9^{12} = \cdots = w_9^{17} = 0$,$w_9^{36} = \cdot = w_9^{41} = 0$, $w_9^{60} = 1, w_9^{61} = 0$
w_{10}	$w_{10}^{6} = \cdots = w_{10}^{11} = 0$, $w_{10}^{18} = \cdots = w_{10}^{23} = 0$, $w_{10}^{42} = \cdots = w_{10}^{47} = 0$, $w_{10}^{60} = 0$, $w_{10}^{61} = 1$
w_{11}	$w_{11}^{36} = \cdots = w_{11}^{41} = 0$, $w_{11}^{48} = \cdots = w_{11}^{59} = 0$, $w_{11}^{60} = w_{11}^{61} = 1$, $w_{11}^{62} \vee w_{11}^{63} = 1$
w_{12}	$w_{12}^{12} = \cdots = w_{12}^{23} = 0$, $w_{12}^{36} = \cdots = w_{12}^{47} = 0$, $w_{12}^{60} = w_{12}^{61} = 0$
w_{13}	$w_{13}^{36} = \cdots = w_{13}^{41} = 0$, $w_{13}^{60} = 1, w_{13}^{61} = 0$
w_{14}	difference in w_{14}^{63}, $w_{14}^{12} = \cdots = w_{14}^{23} = 0$, $w_{14}^{36} = \cdots = w_{14}^{53} = 0$, $w_{14}^{60} = 0, w_{14}^{61} = 1$
w_{15}	$w_{15}^{6} = \cdots = w_{15}^{11} = 0$, $w_{15}^{36} = \cdots = w_{15}^{41} = 0$, $w_{15}^{60} = w_{15}^{61} = 1$

Publication

Practical Collisions for SHAMATA-256

Publication Data

Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Martin Schläffer. Practical collisions for SHAMATA-256. In Michael J. Jacobson, Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography — SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009.

Contributions

- Contributions to the entire article, and in particular:
  - Sect. 2 (Description of SHAMATA),
  - Sect. 4 (Finding a Good Differential Path),
  - Sect. 5.2 (Practical Collisions for SHAMATA-256), and
  - Appendix A (Colliding Message Pair for SHAMATA-256).

Practical Collisions for SHAMATA-256

Sebastiaan Indesteege[1,2,*], Florian Mendel[3], Bart Preneel[1,2], and Martin Schläffer[3]

[1] Department of Electrical Engineering ESAT/COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.
[3] Institute for Applied Information Processing and Communications Inffeldgasse 16a, A-8010 Graz, Austria.

Abstract. In this paper, we present a collision attack on the SHA-3 submission SHAMATA. SHAMATA is a stream cipher-like hash function design with components of the AES, and it is one of the fastest submitted hash functions. In our attack, we show weaknesses in the message injection and state update of SHAMATA. It is possible to find certain message differences that do not get changed by the message expansion and non-linear part of the state update function. This allows us to find a differential path with a complexity of about 2^{96} for SHAMATA-256 and about 2^{110} for SHAMATA-512, using a linear low-weight codeword search. Using an efficient guess-and-determine technique we can significantly improve the complexity of this differential path for SHAMATA-256. With a complexity of about 2^{40} we are even able to construct practical collisions for the full hash function SHAMATA-256.

Key words: SHAMATA, SHA-3 candidate, hash function, collision attack.

1 Introduction

A cryptographic hash function H maps a message M of arbitrary length to a fixed-length hash value h. Informally, a cryptographic hash function has to fulfil the following security requirements:

- *Collision resistance:* it is infeasible to find two messages M and M^*, with $M^* \neq M$, such that $H(M) = H(M^*)$.
- *Second preimage resistance:* for a given message M, it is infeasible to find a second message $M^* \neq M$ such that $H(M) = H(M^*)$.

[*]F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

- *Preimage resistance:* for a given hash value h, it is infeasible to find a message M such that $H(M) = h$.

The resistance of a hash function to collision and (second) preimage attacks depends in the first place on the length n of the hash value. Regardless of how a hash function is designed, an adversary will always be able to find preimages or second preimages after trying out about 2^n different messages. Finding collisions requires a much smaller number of trials. Due to the birthday paradox, collisions can be found in a generic way with an effort of only about $2^{n/2}$. A hash function is said to achieve *ideal security* if these bounds are guaranteed.

In the last few years, the cryptanalysis of hash functions has become an important topic within the cryptographic community. Especially the collision attacks on the MD4 family of hash functions (MD5, SHA-1) have diminished the confidence in the security of these commonly used hash functions. Therefore, NIST has started the SHA-3 competition [7] to find a successor for the SHA-1 and SHA-2 hash functions. The goal is to find a hash function which is fast and still secure within the next few decades.

Many new and interesting hash functions have been proposed. One of them is SHAMATA [1]. Out of the 51 first round candidates, SHAMATA is one of the fastest submissions having a speed of 8–11 cycles/byte on 64-bit and 15–22 cycles/byte on 32-bit platforms [1]. It is a register based design, similar to the hash function PANAMA [4] and also bears resemblance to the sponge construction [2].

In this work, we analyse the security of the hash function SHAMATA. After a description of SHAMATA in Sect. 2, we analyse some basic differential properties of the message injection and state update function in Sect. 3. We show how to efficiently linearise SHAMATA by considering special XOR differences with an equal difference in all bytes. In Sect. 4, we construct a good differential path for the linearised variant of SHAMATA using a low-weight codeword search. Section 5 explains how basic message modification techniques allows us to construct a collision attack with a complexity of 2^{96} for SHAMATA-256 and 2^{110} for SHAMATA-512, based on this differential path. For SHAMATA-256, the attack is improved further to a complexity of only 2^{40} SHAMATA rounds using a complex guess-and-determine strategy. This attack is practical, and we show a collision example in App. A. We conclude our analysis of the hash function SHAMATA in Sect. 6.

2 Description of SHAMATA

In this section, we give a brief description of the hash function SHAMATA. SHAMATA is a register based hash function design that operates on an internal state of 2048 bits and produces a hash value of 224, 256, 384 or 512 bits. The internal state consists of two parts: the main mixing register $B_3, \ldots, B_0$ and the second mixing register $K_{11}, \ldots, K_0$. Internally, SHAMATA uses rounds of the AES block cipher [5] as building blocks.

First, the message is padded to an integer number of 128-bit blocks using classical Merkle-Damgård strengthening, like in the MD4 family. The registers comprising the internal state of SHAMATA are set to their initial values, which depend on the digest length used. Then, each 128-bit message block is used once to update the internal state as described below. Finally, the finalisation phase of SHAMATA generates the output digest from the internal state. For a detailed description of the initialisation and finalisation phases of SHAMATA, we refer to [1], as these details are not relevant to our analysis.

2.1 The Message Injection

The message injection of SHAMATA updates the internal state using a 128-bit message block. The message block M is first expanded as follows:

$$\begin{array}{rclcrcl} P & = & MC\left(M^{\mathrm{T}}\right) \ , & \quad & Q & = & MC\left(M\right) \ , \\ P' & = & P(1) \,||\, Q(0) \ , & & Q' & = & Q(1) \,||\, P(0) \ . \end{array} \tag{1}$$

Here, MC is the MixColumns operation from the AES block cipher [5] and M^{T} is the transpose of M, where M is viewed as a 4×4 matrix of bytes. The notation $P(i)$ denotes the i-th most significant 64-bit half of the 128-bit word P. Thus, P' and Q' are simply recombinations of the columns of P and Q. These expanded message words and a block counter *blockno* are then added to six words of the internal state using XOR:

$$\begin{array}{rclcrcl} B_2 & \leftarrow & B_2 \oplus P \oplus \mathit{blockno} \ , & \quad & B_3 & \leftarrow & B_3 \oplus Q \oplus \mathit{blockno} \ , \\ K_3 & \leftarrow & K_3 \oplus P' \ , & & K_5 & \leftarrow & K_5 \oplus Q \ , \\ K_7 & \leftarrow & K_7 \oplus P \ , & & K_{11} & \leftarrow & K_{11} \oplus Q' \ . \end{array} \tag{2}$$

2.2 The State Update Function

After the expanded message words have been added, the state update function updates the internal state by clocking the registers of the internal state twice, as is shown in Fig. 1. Formally, these two clockings can be written as

$$\begin{array}{rclcrcl} feedK_1 & = & ARF^r\left(B_2\right) \oplus B_0 \ , & \quad & feedB_1 & = & feedK_1 \oplus K_9 \oplus K_0 \ , \\ feedK_2 & = & ARF^r\left(B_3\right) \oplus B_1 \ , & & feedB_2 & = & feedK_2 \oplus K_{10} \oplus K_1 \ , \\ B_i & \leftarrow & B_{i+2} \quad (i = 0, 1) \ , & & K_i & \leftarrow & K_{i+2} \quad (i = 0, \ldots, 9) \ , \\ B_2 & \leftarrow & feedB_1 \ , & & K_{10} & \leftarrow & feedK_1 \ , \\ B_3 & \leftarrow & feedB_2 \ , & & K_{11} & \leftarrow & feedK_2 \ . \end{array} \tag{3}$$

The function ARF^r consists of r rounds of the AES block cipher [5], omitting subkey additions. Thus, the ARF function consists of the SubBytes, ShiftRows and MixColumns operations:

$$ARF(X) = MC\left(SR\left(SB\left(X\right)\right)\right) \ . \tag{4}$$

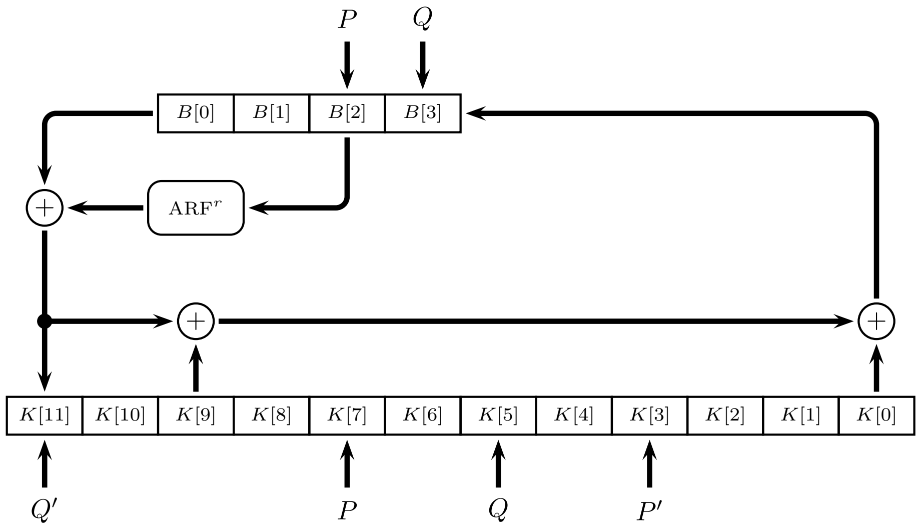


Figure 1 – The state update function of SHAMATA.

For SHAMATA-224 and SHAMATA-256, the number of rounds r is equal to one. For SHAMATA-384 and SHAMATA-512, r is two.

3 Basic Attack Strategy

In this section, we describe the basic attack strategy to construct collisions for SHAMATA. The attack is similar to the attack on PANAMA [6, 10], since we construct a collision in the internal state during the message injection phase. In this phase, the message input can be used to control the differences in the internal state. However, since the expanded message block is inserted several times into the internal state, finding a differential trail seems to be difficult at first. However, by exploiting some differential properties of the state update, we can find a differential trail for SHAMATA which results in a collision with a good probability.

3.1 Overview of the Attack

The main idea of the attack on SHAMATA is to insert special message differences Δ, which do not get changed by the message expansion and the non-linear function ARF^r. Then, the same difference Δ will be added to six positions of the internal state by the message injection. By imposing conditions on the input of ARF^r, we can ensure that the difference Δ does not get changed by this non-linear function. Hence, all parts of the state update are linear regarding the XOR difference Δ and we can search for a differential path using basic linear algebra.

3.2 Choosing the Message Difference

In the message expansion of SHAMATA, the 128-bit message word M is first arranged in a 4×4 array of bytes. Then, the MixColumns transformation is applied to both M and M^T and some columns are rearranged to get the expanded message blocks P, P', Q and Q'. All transformations are applied on the byte level and we can make the following observation.

Observation 1. A message difference Δ with equal differences in all 16 bytes, results in the same difference Δ in each of the expanded message words P, P', Q and Q'.

Transposition and rearranging columns does not change the value of byte differences. MixColumns applies the following linear transformation over GF(2^8) to each column [5]:

$$\begin{array}{lcl} b_0 & = & 2 \bullet a_0 \oplus 3 \bullet a_1 \oplus 1 \bullet a_2 \oplus 1 \bullet a_3 \\ b_1 & = & 1 \bullet a_0 \oplus 2 \bullet a_1 \oplus 3 \bullet a_2 \oplus 1 \bullet a_3 \\ b_2 & = & 1 \bullet a_0 \oplus 1 \bullet a_1 \oplus 2 \bullet a_2 \oplus 3 \bullet a_3 \\ b_3 & = & 3 \bullet a_0 \oplus 1 \bullet a_1 \oplus 1 \bullet a_2 \oplus 2 \bullet a_3 \end{array} . \tag{5}$$

If all input values are equal to some value a, we get with $2 \bullet a \oplus 3 \bullet a = 1 \bullet a$:

$$b_i = 2 \bullet a \oplus 3 \bullet a \oplus 1 \bullet a \oplus 1 \bullet a = 1 \bullet a = a \ . \tag{6}$$

and all output values are equal. Hence, for any message difference Δ with equal values in all bytes, the same difference Δ will be injected into the 6 state words B_3, B_2, K_{11}, K_7, K_5 and K_3.

3.3 Linearising ARF^r

The only non-linear part in SHAMATA is the modified AES-round ARF^r. The function ARF^r behaves linearly if a given input difference Δ results in the same output difference Δ. This is again possible for certain differences, by additionally imposing conditions on the input values of ARF^r:

Observation 2. There are input differences Δ of ARF^r with equal differences in all 16 bytes, which result in the same output difference Δ for certain conditions on the input values of ARF^r.

For example, in the case of ARF^1 (SHAMATA-256), the input difference $\Delta =$ `0xff,0xff,`... results in the same output difference $\Delta =$ `0xff,0xff,`... if all input byte values are equal to either `0x7e` or `0x81`. A more careful choice of the difference in the input bytes can improve the probability that the differential through ARF^r is followed.

For ARF^1 a careful examination of the difference distribution table (DDT) of the AES S-box reveals that the best choice is a difference of `0xc5` in each byte. Indeed, this difference passes through the S-box unchanged for input values $\{\texttt{0x00}, \texttt{0x1d}, \texttt{0xc5}, \texttt{0xd8}\}$ and hence, with an optimal probability of 2^{-6}. Using this difference, there are 4^{16} values for the input to ARF^1 which exhibit the desired differential behaviour, corresponding to a differential probability of 2^{-96}.

In the case of ARF^2 (SHAMATA-512), we can no longer view each S-box independently. Eliminating linear steps at the in- and output, ARF^2 reduces to SubBytes, followed by MixColumns and another SubBytes operation. Thus, each column is still independent here. We have performed an exhaustive search to find the best difference consisting of 16 equal bytes that passes through ARF^2 unchanged. The best choice is a difference of `0x18` in each byte, which passes through ARF^2 for $(22)^4$ values, corresponding to a differential probability of $2^{-110.16}$.

3.4 Basic Message Modification

In this section, we analyse the possibilities to fulfil the conditions on the input of ARF^r. For each active ARF^r function, the input value has to be such that the difference is passed unchanged. The probability of this event was optimised in the previous section. Note however that in each round, the expanded message word P is XORed directly to B_2. Hence, if the ARF^r function in the first clocking is active, we can simply choose M such that the input to ARF^r is X, which is fixed to one of the "good" values ensuring that the active ARF^r has the required differential behaviour:

$$M = (MC^{-1}(P))^T = (MC^{-1}(B_2 \oplus X))^T. \tag{7}$$

If the ARF^r function in the second clocking of a round is active, a similar approach can be used, as the message is also XORed to B_3 via Q, which forms the input to ARF^r in the second clocking:

$$M = MC^{-1}(Q) = MC^{-1}(B_3 \oplus X). \tag{8}$$

These basic message modification techniques do not work anymore as soon as two consecutive ARF^r functions of a single round are active. If we get a difference Δ in both B_2 and B_3 after the message injection, we can adjust only one input of the following two ARF^r functions. The main problem here is that we do not have enough freedom to fulfil the conditions on the message input imposed by both active ARF^r functions. Hence, in this case, one of them has to be satisfied probabilistically. The best probability is 2^{-96} for ARF^1 and $2^{-110.16}$ for ARF^2, as was shown in Sect. 3.3.

Hence, we will aim for a differential path with a low number of consecutive active ARF^r functions (see Sect. 4). Unfortunately, in any differential path, we always get a difference in both, B_2 and B_3 after the first message injection. However, in Sect. 5.2, we show how we can still fulfil both conditions for SHAMATA-256 with much less effort, such that the attack becomes practical.

4 Finding a Good Differential Path

In this section, we first show how to find an efficient collision path for SHAMATA. Recall from Sect. 3.4 that the new message freedom in each round of SHAMATA allows an adversary to linearise the ARF^r function in one of the two clockings in a round. Thus, we aim to find a collision differential path that activates the ARF^r function in at most one clocking of each round as well. However, it was already pointed out in Sect. 3.4 that it is impossible to avoid this in the round where the first difference is introduced, but we can aim to avoid this in all the other rounds. We describe two methods to achieve this. The first method is based on searching low-weight codewords of a linear code and the second method is a simple exhaustive search. The former is more general and can also be used to find differential paths spanning a long message. The latter is only feasible for short messages, but it is simpler. In the case of SHAMATA, either of the methods can be used to achieve the same result.

4.1 Low-Weight Codewords

For a fixed number of message blocks, all differential paths under consideration can be seen as the codewords of a linear code. We show that searching for low-weight codewords in this code is a useful tool to construct good differential paths for SHAMATA. The use of low-weight codeword search techniques to construct differential paths was proposed by Rijmen and Oswald [9] and extended by Pramstaller et al. in [8].

A codeword of the code under consideration contains, for each round, the message difference and the differences in the internal state registers immediately after the new message block was added. As we consider only Δ differences, each of these differences is represented by a single bit. Let $\Delta m^{(i)}$, $\Delta b_3^{(i)}, \ldots, \Delta b_0^{(i)}$ and $\Delta k_{11}^{(i)}, \ldots, \Delta k_0^{(i)}$ denote these bits for round i. With N the fixed number of message blocks used, a codeword of the code is then given by

$$\left[\ \Delta m^{(1)} \cdots \Delta m^{(N)} \quad || \quad \Delta b_3^{(1)} \cdots \Delta k_0^{(1)} \quad || \quad \cdots \quad || \quad \Delta b_3^{(N)} \cdots \Delta k_0^{(N)}\ \right] . \qquad (9)$$

We now construct the generator matrix $\mathbf{G}$ of this code. The differences in a SHAMATA state immediately after the message addition in round i can be represented by an 1×16 binary vector $\Delta s^{(i)}$,

$$\Delta s^{(i)} = \left[\ \Delta b_3^{(i)} \quad \cdots \quad b_0^{(i)} \quad || \quad k_{11}^{(i)} \quad \cdots \quad k_0^{(i)}\ \right] . \qquad (10)$$

As the ARF^r function is assumed to behave linearly with respect to the Δ difference, the state difference vector in round i, $\Delta s^{(i)}$, can be written in function of the state differences vector in round $i-1$, $\Delta s^{(i-1)}$, as follows

$$\Delta s^{(i)} = \Delta s^{(i-1)} \cdot \mathbf{A} \oplus \Delta m^{(i)} \cdot w \ . \qquad (11)$$

Here, w is a 1×16 vector indicating to which positions of the internal state a new message block is added. It is easy to see that

$$w = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} . \tag{12}$$

The 16×16 matrix $\mathbf{A}$ is a transition matrix corresponding to the two clockings in the round. It is given by

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & & & 0 & & & & & & & \\ 1 & & 1 & & 1 & & & & & & & \\ 0 & & & 1 & 0 & & & & & & & \\ 1 & & & & 1 & & & & & & & \\ 0 & & & & 0 & 1 & & & & & & \\ 0 & & & & 0 & & 1 & & & & & \\ 1 & & & & 0 & & & 1 & & & & \\ 0 & & & & 0 & & & & 1 & & & \\ \vdots & & & & \vdots & & & & & \ddots & & \\ 0 & & & & 0 & & & & & & 1 & \\ 1 & & & & 0 & & & & & & & 1 \end{bmatrix}^2 . \tag{13}$$

Now, consider the $N \times 17N$ generator matrix $\mathbf{G}_{\text{all}}$ given by

$$\mathbf{G}_{\text{all}} = \left[\begin{array}{c|ccccc} & w & w\mathbf{A} & w\mathbf{A}^2 & \cdots & w\mathbf{A}^{N-1} \\ & & w & w\mathbf{A} & \cdots & w\mathbf{A}^{N-2} \\ I_{N \times N} & & & w & & \vdots \\ & & & & \ddots & w\mathbf{A} \\ & & & & & w \end{array} \right] . \tag{14}$$

This is the generator matrix of a linear code that contains all length N differential paths of the type we consider. As we are only interested in collision differentials, it is required that the last internal state has no difference. This can be achieved by using Gaussian elimination to force zeroes in the last 16 columns of $\mathbf{G}_{\text{all}}$. This gives the generator matrix $\mathbf{G}$, which generates a linear code containing all differential paths that result in a collision.

Due to the possibility of message modification in either of the clockings in a SHAMATA round, but not both (see Sect. 3.4), a good differential path for SHAMATA activates the ARF^r function in at most one clocking per round. As was already noted, it is impossible to avoid activating ARF^r in both clockings of the round where a difference is first introduced. But we aim to avoid this in the remainder of the differential path.

Intuitively, a codeword with a low weight in Δb_2 and Δb_3, which are the input differences to ARF^r, is more likely to satisfy this property than a random codeword. Thus, we look for low-weight codewords in this code, considering only the weight of these bits, using an algorithm similar to that of Canteaut and

Chabaud [3]. For each codeword below a certain threshold weight, we check if it satisfies the condition mentioned above. If it does, a suitable collision differential path has been found. If not, the search is simply continued. Note that this search method can find collision differential paths shorter than N rounds. Indeed, nothing prevents the search from padding a shorter differential path to N rounds by adding rounds without a difference, as we indeed observed. The shortest collision differential path we found is shown in Table 1. It consists of 25 rounds and, except for the first round, only activates ARF^r in at most one of the clockings of a round.

4.2 An Alternative Approach

Note that, for a given length of N rounds, there are only 2^N possible differential paths of the type we consider. Indeed, as each message block can only have a Δ difference or no difference at all, there are only 2^N possible message differences. Given the message difference, exactly one differential path follows. Hence, when N is not too large, a simple brute force search can also be a viable approach.

As the more general approach given above resulted in a differential path of only 25 rounds, a brute force approach is indeed practically feasible. We have exhaustively searched all differential paths of length up to 25 rounds. As expected, this search also found the differential path given in Table 1. Moreover, there is only one differential path of 25 rounds, and no shorter differential paths of this type exist. Hence, the differential path in Table 1 is optimal.

5 Collision Attack on SHAMATA

In this section, we put together the various pieces that were introduced, and present our collision attack on SHAMATA. We search for a message pair which follows the differential path in Table 1.

5.1 Collisions for SHAMATA-256 and SHAMATA-512

In rounds where none of the ARF^r functions is active, the differential path is always followed, regardless of the message block. Hence, in those rounds, we make an arbitrary choice for the message block. In rounds with exactly one active ARF^r function, the message modification technique presented in Sect. 3.4 is used to deterministically construct a message block that ensures that the differential path is followed. This takes only negligible time, i.e., no more than computing a single round of SHAMATA.

However, in the first round where a difference is introduced, the ARF^r function is active in both clockings. The message modification technique of Sect. 3.4 can only deterministically satisfy the conditions for one of them. As discussed in Sect. 3.4, the probability that the path is still followed is 2^{-96} for ARF^1

Table 1 – The differential path for 25 rounds of SHAMATA with differences after each clocking. For differences at the input of ARF^r (word B_1, grey column), the differential probabilities of each round are given in the last two columns for SHAMATA-256 (ARF^1) and SHAMATA-512 (ARF^2).

round	M	B_3	B_2	B_1	B_0	K_{11}	K_{10}	K_9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	ARF^1	ARF^2
1	Δ	Δ	Δ	Δ		Δ	Δ				Δ		Δ		Δ				
		Δ	Δ	Δ	Δ	Δ	Δ	Δ				Δ		Δ		Δ		2^{-192}	$2^{-220.32}$
2	Δ				Δ	Δ		Δ	Δ		Δ						Δ		
		Δ				Δ	Δ		Δ	Δ		Δ							
3	Δ	Δ		Δ		Δ		Δ		Δ					Δ				
		Δ	Δ		Δ		Δ		Δ		Δ					Δ		2^{-96}	$2^{-110.16}$
4	Δ	Δ				Δ	Δ	Δ		Δ	Δ	Δ	Δ		Δ		Δ		
			Δ				Δ	Δ	Δ		Δ	Δ	Δ	Δ		Δ			
5				Δ		Δ		Δ	Δ	Δ		Δ	Δ	Δ	Δ		Δ		
					Δ		Δ		Δ	Δ	Δ		Δ	Δ	Δ	Δ		2^{-96}	$2^{-110.16}$
6		Δ				Δ		Δ		Δ	Δ	Δ		Δ	Δ	Δ	Δ		
			Δ				Δ		Δ		Δ	Δ	Δ		Δ	Δ	Δ		
7	Δ	Δ	Δ				Δ	Δ		Δ	Δ	Δ		Δ	Δ	Δ	Δ		
		Δ	Δ	Δ		Δ		Δ	Δ		Δ	Δ	Δ		Δ	Δ	Δ	2^{-96}	$2^{-110.16}$
8	Δ				Δ				Δ	Δ	Δ	Δ		Δ	Δ	Δ	Δ		
						Δ				Δ	Δ	Δ	Δ		Δ	Δ	Δ		
9		Δ					Δ				Δ	Δ	Δ	Δ		Δ	Δ		
		Δ	Δ					Δ				Δ	Δ	Δ	Δ		Δ		
10	Δ						Δ		Δ		Δ			Δ		Δ			
								Δ		Δ		Δ			Δ		Δ		
11									Δ		Δ		Δ			Δ			
										Δ		Δ		Δ			Δ		
12		Δ									Δ		Δ		Δ				
			Δ									Δ		Δ		Δ			
13	Δ		Δ				Δ				Δ						Δ		
				Δ		Δ		Δ				Δ						2^{-96}	$2^{-110.16}$
14		Δ			Δ		Δ		Δ				Δ						
		Δ	Δ			Δ		Δ		Δ				Δ					
15	Δ	Δ							Δ				Δ						
			Δ							Δ				Δ					
16		Δ		Δ		Δ					Δ				Δ				
			Δ		Δ		Δ					Δ				Δ		2^{-96}	$2^{-110.16}$
17	Δ	Δ	Δ			Δ	Δ	Δ			Δ				Δ		Δ		
		Δ	Δ	Δ		Δ	Δ	Δ	Δ			Δ				Δ		2^{-96}	$2^{-110.16}$
18	Δ	Δ			Δ			Δ	Δ	Δ	Δ				Δ		Δ		
		Δ	Δ			Δ			Δ	Δ	Δ	Δ				Δ			
19	Δ									Δ		Δ			Δ		Δ		
		Δ									Δ		Δ			Δ			
20			Δ									Δ		Δ			Δ		
				Δ		Δ							Δ		Δ			2^{-96}	$2^{-110.16}$
21					Δ		Δ							Δ		Δ			
		Δ				Δ		Δ							Δ		Δ		
22			Δ				Δ		Δ							Δ			
		Δ		Δ		Δ		Δ		Δ							Δ	2^{-96}	$2^{-110.16}$
23			Δ		Δ		Δ		Δ		Δ								
				Δ				Δ		Δ		Δ						2^{-96}	$2^{-110.16}$
24		Δ			Δ				Δ		Δ		Δ						
		Δ	Δ			Δ				Δ		Δ		Δ					
25	Δ																		

(SHAMATA-256) and $2^{-110.16}$ for ARF^2 (SHAMATA-512). A prefix with no difference is used to provide the required message freedom.

Thus, a conforming pair for the first round of the differential path can be found by performing about 2^{96} trials for SHAMATA-256 and about 2^{110} trials for SHAMATA-512. Once such a pair has been found, a colliding message pair can be constructed with negligible additional effort. Thus, the overall complexity of our attack is about 2^{96} SHAMATA rounds for SHAMATA-256, and about 2^{110} SHAMATA rounds for SHAMATA-512. The attack requires only negligible memory and is easily parallelisable. Hence, for both variants of SHAMATA, the attack is significantly faster than a brute force attack. Note that the attack also applies to SHAMATA-224 and SHAMATA-384.

5.2 Practical Collisions for SHAMATA-256

In the case of SHAMATA-256, a more efficient approach exists to control the values which are input to the ARF^r function in both clockings of a round. This approach exploits the fact that in SHAMATA-256 only a single AES round is used, i.e., $r = 1$. Hence, this method can not be applied to SHAMATA-512, where $r = 2$.

Assume we aim to fix the inputs to the ARF^1 function in both clockings of round i to X_1 and X_2, respectively. Let $B^{(i)}$ denote the B-register at the beginning of round i. Then, this requirement can be written as

$$\left\{ \begin{array}{lcl} B_2^{(i)} \oplus P^{(i)} \oplus i & = & X_1 \\ B_3^{(i)} \oplus Q^{(i)} \oplus i & = & X_2 \end{array} \right. . \tag{15}$$

Using the definition of the state update function of SHAMATA in (1)–(3), this can be rewritten in a function of the internal state at the beginning of round $i-1$ and the message blocks M_{i-1} and M_i, yielding the following

$$\left\{ \begin{array}{lcl} M_{i-1} & = & MC^{-1}\left(D_1 \oplus \left(C_1 \oplus SR^{-1}\left(M_i\right)\right)\right) \\ {M_{i-1}}^{\mathrm{T}} & = & MC^{-1}\left(D_2 \oplus \left(C_2 \oplus SR^{-1}\left({M_i}^{\mathrm{T}}\right)\right)\right) \end{array} \right. , \tag{16}$$

where C_1, C_2, D_1 and D_2 are constants defined by

$$\begin{array}{lcl} C_1 & = & SR^{-1}\left(MC^{-1}\left(B_0^{(i-1)} \oplus K_9^{(i-1)} \oplus K_0^{(i-1)} \oplus i \oplus X_1\right)\right) , \\ C_2 & = & SR^{-1}\left(MC^{-1}\left(B_1^{(i-1)} \oplus K_{10}^{(i-1)} \oplus K_1^{(i-1)} \oplus i \oplus X_2\right)\right) , \\ D_1 & = & B_2^{(i-1)} \oplus (i-1) , \\ D_2 & = & B_3^{(i-1)} \oplus (i-1) . \end{array} \tag{17}$$

These constants only depend on the internal state of SHAMATA-256 at the beginning of round $i-1$, and are thus known. Now, we search for message blocks M_{i-1} and M_i such that the conditions of (16) are satisfied.

A straightforward approach to find the message blocks M_{i-1} and M_i would be to guess one of them, compute the other using the first equation of (16) and then, check if the second equation of (16) holds as well. This procedure is expected to find a solution after about 2^{128} trials. We propose a guess-and-determine approach which performs significantly better. Our approach is as follows

1. Assume we know the four bytes of M_i indicated in the pattern in Fig. 2 (a). Note that this pattern is symmetric, i.e., it is invariant under matrix transposition. This implies that also the same pattern of bytes of ${M_i}^{\mathrm{T}}$ is known.

   Note that in (16), M_i and ${M_i}^{\mathrm{T}}$ are input to the inverse ShiftRows operation or SR^{-1}. This operation performs a circular right shift of the rows of the state over 0, 1, 2 or 3 bytes for the first, second, third and fourth row, respectively. Hence, the bytes of M_i indicated in Fig 2 (a) form the first column of $SR^{-1}(M_i)$. Similarly, the first column of $SR^{-1}\left({M_i}^{\mathrm{T}}\right)$ is known.

   All other operations in (16) treat the four columns independently, so knowledge of the first columns of $SR^{-1}(M_i)$ and $SR^{-1}\left({M_i}^{\mathrm{T}}\right)$ suffices to compute the first columns of M_{i-1} and ${M_{i-1}}^{\mathrm{T}}$. The latter is equal to the first row of M_{i-1}, which overlaps with the first column of M_{i-1} in exactly one byte.

   Thus, we investigate all 2^{32} guesses for four bytes of M_i as indicated in Fig. 2 (a). For each guess, we compute the first column and the first row of M_{i-1} using (16). Then, we verify if the overlapping byte matches, and if so, we save the candidate in a list L_1. As this imposes an 8-bit condition, about 2^{24} candidates are expected to remain.

2. The same procedure is repeated with the patterns in Fig. 2 (b), Fig. 2 (c) and Fig. 2 (d). Each pattern is invariant under matrix transposition, and results in one column after applying the SR^{-1} operation. This results in four lists, L_1, L_2, L_3 and L_4 of about 2^{24} elements each.

3. An element of the list L_1 contains candidate values of the first row and column of M_{i-1}. Similarly, an element of the list L_2 contains the second row and column of M_{i-1}. Note that these overlap in two byte positions. Thus, we can merge both lists and store all matching combinations in a new list, L_A. The expected number of entries in the new list L_A is $2^{24} \times 2^{24} \times 2^{-16} = 2^{32}$. If the lists L_1 and L_2 are sorted according to the overlapping bytes, this merge operation can be performed very efficiently.

4. The same procedure is used to merge the lists L_3 and L_4, resulting in a new list L_B which is also expected to contain about 2^{32} entries.

5. Finally, the lists L_A and L_B are merged. The entries in these lists overlap in eight byte positions, which corresponds to a 64-bit condition. Again, if both lists are sorted according to these bytes, merging them can be done efficiently. The number of expected matches is $2^{32} \times 2^{32} \times 2^{-64} = 1$.

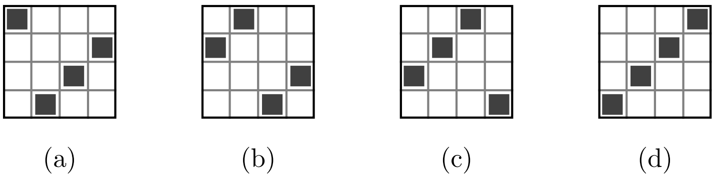


Figure 2 – Patterns used in the guess-and-determine phase.

It is easy to verify that each final match will satisfy (16), and also that every solution to (16) will be found by this procedure. The time complexity of this algorithm is dominated by the merging of lists L_A and L_B, which takes 2^{32} operations. Using hash tables as the data structure to store the lists, an explicit sorting step can be avoided. The memory complexity is determined by one of the lists L_A or L_B, as only one of them really needs to be stored in memory, while the elements of the other can be computed on-the-fly. This corresponds to a memory requirement of about 2^{32} AES states.

For a practical implementation, it is better to reduce the memory requirements of the algorithm, at the expense of an increase in its time complexity. This can be done by, for instance, fixing the byte in the first row and last column of M_{i-1} a priori. Then, the lists L_1 and L_4 are only expected to contain 2^{16} elements each, and the lists L_A and L_B are reduced to about 2^{24} elements. Thus, the total memory complexity is reduced to about 2^{24} AES states, or 256 MB. However, as one byte was fixed a priori, the entire procedure has to be repeated 2^8 times, increasing the time complexity to 2^{40} operations. We have implemented our attack. The guess-and-determine phase was run on a cluster using 256 jobs with a running time of about 5 minutes each. The rest of the attack takes only negligible time using message modification, as explained in Sect. 3.4. A collision example for SHAMATA-256 is given in App. A.

6 Conclusion

In this paper, we have presented a practical collision attack on the SHA-3 submission SHAMATA. Due to weaknesses in the message injection and state update function of SHAMATA it is possible to find certain message differences, that do not get changed by the message expansion or the non-linear part of the state update function. These symmetric XOR differences need to be equal in each byte of the 128-bit words. Using these differences, the non-linear ARF^r function behaves linearly and we can search for a differential path using a linearised variant of SHAMATA. Moreover, since we use the same difference in every 128-bit word, we can represent each word of the internal state by a single bit.

The main weakness in SHAMATA is the relatively light message injection followed by a low number of register clockings. The message injection allows us to efficiently fulfil many conditions using basic message modification. This results in an attack complexity of about 2^{96} for SHAMATA-256 and 2^{110} for SHAMATA-512. Using an efficient guess-and-determine technique we are able to improve the complexity of the attack on SHAMATA-256 to about 2^{40} round computations and present a practical collision for SHAMATA-256. Possible improvements for SHAMATA include increasing the number of times the internal registers are clocked and the use of constants to avoid the use of symmetric differences.

Acknowledgements

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. The collision example for SHAMATA-256 was obtained utilizing high performance computational resources provided by the University of Leuven, `http://ludit.kuleuven.be/hpc`.

References

[1] A. Atalay, O. Kara, F. Karakoç, and C. Manap. SHAMATA hash function algorithm specifications. Submission to the NIST SHA-3 competition, Oct. 2008. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. In *ECRYPT Hash Workshop*. European Network of Excellence in Cryptology ECRYPT, May 2007.

[3] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998.

[4] J. Daemen and C. S. K. Clapp. Fast hashing and stream encryption with PANAMA. In S. Vaudenay, editor, *Fast Software Encryption, 5th International Workshop — FSE '98*, volume 1372 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 1998.

[5] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer, 2002.

[6] J. Daemen and G. Van Assche. Producing collisions for PANAMA, instantaneously. In A. Biryukov, editor, *Fast Software Encryption, 14th International Workshop — FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007.

[7] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, Nov. 2007.

[8] N. Pramstaller, C. Rechberger, and V. Rijmen. Exploiting coding theory for collision attacks on SHA-1. In N. P. Smart, editor, *Cryptography and Coding, IMA International Conference*, volume 3796 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2005.

[9] V. Rijmen and E. Oswald. Update on SHA-1. In A. Menezes, editor, *Topics in Cryptology — CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

[10] V. Rijmen, B. V. Rompay, B. Preneel, and J. Vandewalle. Producing collisions for PANAMA. In M. Matsui, editor, *Fast Software Encryption, 8th International Workshop — FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2002.

A Colliding Message Pair for SHAMATA-256

```
m1 =
00000000: 10 37 fd e7 65 30 1c c0 e3 61 6e 41 24 6f cb b9 |.7..e0...anA$o..|
00000010: 7f 28 81 17 81 4a d1 3f bf 4e ca da 92 f5 35 d0 |.(...J.?.N....5.|
00000020: f0 f0 dc 19 73 d5 a7 07 8c 0b bc 3d b6 85 46 57 |....s......=..FW|
00000030: 02 92 d1 24 00 df 40 67 ca 2c fa 5b 9d 70 2c ce |...$..@g.,.[.p,.|
00000040: de 38 51 f5 01 3c 3b aa d8 ba 38 0e a1 40 b1 91 |.8Q..<;...8..@..|
00000050: 7b 18 18 24 cc d9 76 c0 f7 4a 61 28 86 06 30 8e |{..$..v..Ja(..0.|
00000060: 30 8d ab a3 62 52 aa ee 5d 66 2b 13 ec 71 6b ca |0...bR..]f+..qk.|
00000070: e3 29 f2 2c b3 ed 3d 7e f7 f2 fd 0b 1e c7 d6 e5 |.).,..=~........|
00000080: aa bc bf ab f9 fb 56 d1 b5 8e df 57 ce 90 e8 fe |......V....W....|
00000090: 1e 93 a2 80 e6 4c 6f 43 b3 9a 57 9f 0c c2 69 b6 |.....LoC..W...i.|
000000a0: 7e 29 61 77 24 b7 48 d9 45 27 30 13 b8 19 12 d6 |~)aw$.H.E'0.....|
000000b0: ac b4 56 92 00 c5 d6 b3 60 2d 52 6c ef bc 22 6d |..V.....`-Rl.."m|
000000c0: e5 83 e5 09 3b 2d e2 80 55 13 94 0d 2c a6 e3 d8 |....;-..U...,...|
000000d0: 53 e9 01 66 72 ae 8d cf 68 25 8a b6 ae 64 e7 c1 |S..fr...h%...d..|
000000e0: 5a 39 6b 5a ff 41 0e 5f 6e 60 cb 5d 1c ed ca 01 |Z9kZ.A._n`.]....|
000000f0: 70 af 0a ab dd ed 2c 32 00 c0 3f 2c 66 22 04 c0 |p.....,2..?,f"..|
00000100: 3b 97 65 9d 01 64 98 7b e6 63 d4 d6 4b 77 00 bb |;.e..d.{.c..Kw..|
00000110: bb ac 35 e3 27 66 55 34 0c 0f db d7 2f 16 19 ae |..5.'fU4..../...|
00000120: 5b 6f 1a 5a b0 28 b9 1e 89 84 7b a5 71 46 a7 e2 |[o.Z.(....{.qF..|
00000130: f5 b1 8d d2 9e b9 04 9e 79 43 ca df 65 cf 9f c1 |........yC..e...|
00000140: bb f6 43 f9 cd 88 af 13 ea 2f 93 e8 cd 39 8c a0 |..C....../...9..|
00000150: 3e ba 1b ef e2 d5 0d 6b 59 89 11 cb cf b8 ad c4 |>......kY.......|
00000160: 1a 3f 2f 9d a3 1d 82 3c e0 75 9d 83 b2 ac 3c bf |.?/....<.u....<.|
00000170: e0 27 0c c5 af b0 be a9 94 1e de 9d 50 69 10 cb |.'..........Pi..|
00000180: 69 3a 97 08 f4 9b a6 6d df 71 4d 44 40 ec 05 7e |i:.....m.qMD@..~|
00000190: a6 21 6d 89 f6 7b f4 4f 04 05 1a d3 bd c7 97 27 |.!m..{.O.......'|

SHAMATA-256(m1) =
00000000: 6e a3 b1 a1 29 75 8d 3f f5 60 f8 1b 6b 11 02 9a |n...)u.?.`..k...|
00000010: 14 b9 b2 d9 b3 2a b6 02 2a f5 83 ab e3 4c 1a 2a |.....*..*....L.*|
m2 =
00000000: 10 37 fd e7 65 30 1c c0 e3 61 6e 41 24 6f cb b9 |.7..e0...anA$o..|
00000010: 80 d7 7e e8 7e b5 2e c0 40 b1 35 25 6d 0a ca 2f |..~.~...@.5%m../|
00000020: 0f 0f 23 e6 8c 2a 58 f8 73 f4 43 c2 49 7a b9 a8 |..#..*X.s.C.Iz..|
00000030: fd 6d 2e db ff 20 bf 98 35 d3 05 a4 62 8f d3 31 |.m... ..5...b..1|
00000040: 21 c7 ae 0a fe c3 c4 55 27 45 c7 f1 5e bf 4e 6e |!......U'E..^.Nn|
00000050: 7b 18 18 24 cc d9 76 c0 f7 4a 61 28 86 06 30 8e |{..$..v..Ja(..0.|
00000060: 30 8d ab a3 62 52 aa ee 5d 66 2b 13 ec 71 6b ca |0...bR..]f+..qk.|
00000070: 1c d6 0d d3 4c 12 c2 81 08 0d 02 f4 e1 38 29 1a |....L........8).|
00000080: 55 43 40 54 06 04 a9 2e 4a 71 20 a8 31 6f 17 01 |UC@T....Jq .1o..|
00000090: 1e 93 a2 80 e6 4c 6f 43 b3 9a 57 9f 0c c2 69 b6 |.....LoC..W...i.|
000000a0: 81 d6 9e 88 db 48 b7 26 ba d8 cf ec 47 e6 ed 29 |.....H.&....G..)|
000000b0: ac b4 56 92 00 c5 d6 b3 60 2d 52 6c ef bc 22 6d |..V.....`-Rl.."m|
000000c0: e5 83 e5 09 3b 2d e2 80 55 13 94 0d 2c a6 e3 d8 |....;-..U...,...|
000000d0: ac 16 fe 99 8d 51 72 30 97 da 75 49 51 9b 18 3e |.....Qr0..uIQ..>|
000000e0: 5a 39 6b 5a ff 41 0e 5f 6e 60 cb 5d 1c ed ca 01 |Z9kZ.A._n`.]....|
000000f0: 8f 50 f5 54 22 12 d3 cd ff 3f c0 d3 99 dd fb 3f |.P.T"....?.....?|
00000100: 3b 97 65 9d 01 64 98 7b e6 63 d4 d6 4b 77 00 bb |;.e..d.{.c..Kw..|
00000110: 44 53 ca 1c d8 99 aa cb f3 f0 24 28 d0 e9 e6 51 |DS........$(...Q|
00000120: a4 90 e5 a5 4f d7 46 e1 76 7b 84 5a 8e b9 58 1d |....O.F.v{.Z..X.|
00000130: 0a 4e 72 2d 61 46 fb 61 86 bc 35 20 9a 30 60 3e |.Nr-aF.a..5 .0`>|
00000140: bb f6 43 f9 cd 88 af 13 ea 2f 93 e8 cd 39 8c a0 |..C....../...9..|
00000150: 3e ba 1b ef e2 d5 0d 6b 59 89 11 cb cf b8 ad c4 |>......kY.......|
00000160: 1a 3f 2f 9d a3 1d 82 3c e0 75 9d 83 b2 ac 3c bf |.?/....<.u....<.|
00000170: e0 27 0c c5 af b0 be a9 94 1e de 9d 50 69 10 cb |.'..........Pi..|
00000180: 69 3a 97 08 f4 9b a6 6d df 71 4d 44 40 ec 05 7e |i:.....m.qMD@..~|
00000190: 59 de 92 76 09 84 0b b0 fb fa e5 2c 42 38 68 d8 |Y..v.......,B8h.|

SHAMATA-256(m2) =
00000000: 6e a3 b1 a1 29 75 8d 3f f5 60 f8 1b 6b 11 02 9a |n...)u.?.`..k...|
00000010: 14 b9 b2 d9 b3 2a b6 02 2a f5 83 ab e3 4c 1a 2a |.....*..*....L.*|
```

Curriculum Vitae

Sebastiaan Indesteege was born on 9th April, 1984 in Hasselt, Belgium. He received the Master's degree in Electrical Engineering (ICT — Telecommunications) from Katholieke Universiteit Leuven, Belgium in July 2006. His Master's thesis on side-channel attacks on cryptographic chips received an award from Ubizen.

In October 2006, he joined the research group COSIC (COmputer Security and Industrial Cryptography) at the Department of Electrical Engineering (ESAT) of K.U.Leuven. His PhD research was sponsored by the Fund for Scientific Research, Flanders (FWO-Vlaanderen). At the ISC 2008 conference, he received the Best Student Paper Award for the article 'Collisions for RC4-Hash'.

He visited the Krypto research group at IAIK, Technische Universität Graz (TU Graz), Austria in March 2008. From June to September 2009, he visited the Security Technologies Group of Sony corp. in Tokyo, Japan.

Arenberg Doctoral School of Science, Engineering & Technology

Faculty of Engineering

Department of Electrical Engineering (ESAT)

COmputer Security and Industrial Cryptography (COSIC)

Kasteelpark Arenberg 10 — 2446, 3001 Heverlee, Belgium