Optimized Algorithms and Architectures for Fast Non-cryptographic Hash Functions in Hardware

Arish Sateesan[a], Jelle Biesmans[a], Thomas Claesen[a], Jo Vliegen[a] and Nele Mentens[a,b]

[a] imec-COSIC/ES&S, ESAT, KU Leuven, Belgium
[b] LIACS, Leiden University, The Netherlands

ARTICLE INFO

Keywords:
FPGA
Network security
Non-cryptographic hash function
Lightweight cipher

ABSTRACT

The design and implementation of eight novel non-cryptographic hash functions is proposed in this paper. Those can suitably be employed in networking and security applications that require fast lookup and/or counting architectures. Our approach is inspired by the design of the existing non-cryptographic hash function Xoodoo-NC, which is constructed through the concatenation of several Xoodoo permutations. In a similar way, this work concatenates several rounds of symmetric-key ciphers to construct non-cryptographic hash functions. The goal is to achieve high performance while maintaining the avalanche properties. Meeting the avalanche criteria is necessary to yield a significant change in the output of hash functions even for the slightest change in the input. The number of rounds required to achieve satisfactory avalanche scores for each cipher is simulated. We implement the corresponding non-cryptographic hash functions on an FPGA and we synthesize them on ASIC to evaluate the occupied resources and the performance. This paper evaluates the performance and area of ASIC implementations optimized for high operating speed. This evaluation is done on the 65nm TSMC technology. One of the proposed non-cryptographic hash functions, namely PHOTON-NC, outperforms all previously proposed non-cryptographic hash functions in terms of throughput and latency on FPGA, in exchange for an acceptable increase in FPGA resources, while Xoodoo-NC proves to have the highest performance on ASIC.

1. Introduction

Hashing is the practice of taking a message of arbitrary length and converting it into a digest of fixed length. A hash function is a one-way function, which means recovering the original message from the digest is not feasible in practice. Hash functions can be cryptographic or non-cryptographic. Non-cryptographic hash functions are preferred for applications where speed of operation is of prime importance and where security properties such as pre-image resistance are not important. Hash functions form an important building block in many network security applications. In this work, our focus is mainly on the application of hash functions for network security applications. In most of the network security applications such as network intrusion detection, anomaly detection, and heavy-hitter detection, the network flow identifiers are required to be stored in or queried from a data structure. A flow identifier (flow ID) represents a network flow and it is extracted from the packet header. Typically, a flow ID is characterized by the 5-tuple (<source IP address, source port, destination IP address, destination port, transport protocol>), but can be any combination of source and destination IP addresses and ports [1].

Dictionaries and associative arrays are the most commonly used storage data structures for storing or querying flow IDs. On hardware, architectures such as content addressable memories (CAM) are analogous to associative arrays. However, storing the complete flow IDs in dictionaries or associative arrays would result in increased memory overhead and lookup time. Hence, storage architectures such as associative arrays always become a bottleneck, especially in high-speed networks where processing data at line rate is vitally important. Containing the storage architecture within the high-speed internal memory significantly reduces the lookup time, but the large memory footprint of associative array-based storage makes it impossible to fit into the limited amount of cache. Approximate membership query/lookup data structures provide fast lookup and low memory overhead compared to dictionaries and associative arrays, but with a tolerable error. Non-cryptographic hash functions are the principal constituent of such approximate/probabilistic data

arish.sateesan@kuleuven.be (A. Sateesan); jelle.biesmans@kuleuven.be (J. Biesmans); thomasclaesen@live.be (T. Claesen); jo.vliegen@esat.kuleuven.be (J. Vliegen); nele.mentens@kuleuven.be (N. Mentens)
ORCID(s): 0000-0002-8197-0097 (A. Sateesan); 0000-0003-0653-1956 (J. Biesmans); 0000-0003-4258-2208 (J. Vliegen); 0000-0001-8753-7895 (N. Mentens)

structures. Advantages such as speed and low memory footprint of probabilistic data structures like Bloom filters [2] and sketches [3] make them favourable for fast membership queries and network flow measurement for networking applications.

In order to serve the security guarantees of certain applications, cryptographic hash functions offer high levels of security. However, certain security properties of cryptographic hash functions such as preimage resistance, second preimage resistance, and collision resistance [4] are of minor importance and are not necessary for probabilistic data structures employed in network security applications. Since the throughput of the probabilistic data structures is dependent on the execution speed of the hash computations, the speed of the hash function becomes a crucial factor. Employing cryptographic hash functions in these data structures would make it impossible to perform the processing of packets at line-rate in high-speed networks. Non-cryptographic hashes, on the other hand, are faster and more suitable for the aforementioned applications. Keeping this in mind, this paper concentrates on the design of novel non-cryptographic hash functions for probabilistic data structures that can support high-speed networking applications. The prime requirement of non-cryptographic hashes employed in Bloom filters, hash tables, and sketches is that they must be fast, uniformly distributed and must possess excellent avalanche properties [5, 3, 6]. The latter indicates how well a hash function succeeds in minimizing the correlation between the input text and the generated digest. Security properties such as pre-image resistance are of minor importance.

Universal classes of hash functions [7] are most commonly employed in network applications but at the cost of increased latency. One way to reduce the latency in network applications is to employ a single hash function having a large enough output size and then split/sample the output to generate multiple hash values. This technique imposes hardly any effect on the false positive rate of probabilistic data structures if the hash function is uniformly distributed and there is no correlation between the bit-fields. Recent works on bloom filters and sketches [8, 9] have shown that this technique can significantly improve the speed to match Terabit networks without any adverse effects on false positive rates and estimation errors. Yet, choosing a suitable non-cryptographic hash function which can satisfy the requirements is a cumbersome task. Well-known non-cryptographic hash functions such as FNV and Murmur series hash functions [10, 11] show excellent avalanche properties, but exhibit high latency with larger inputs. The latency of these hashes is dependent on the size of the input key. Also, the hardware suitability of FNV and Murmur hashes, like many other non-cryptographic hashes, is limited, because they target efficient execution in software [8]. Recent works have proven that non-cryptographic hashes constitute a significant proportion of the overall resource requirements of data structures on hardware [12].

This paper extends the work presented in [13]. In this paper, we target high-speed implementations of non-cryptographic hash functions on FPGA and ASIC, in order to serve lookup and counting applications for network security. Also, the applications we target exploit the parallelism of hardware solutions and possess the ability to work as a standalone unit. The trend of moving security and networking applications to FPGAs in order to achieve a higher throughput is motivated by the work in [14]. In order to achieve higher throughput and low latency on hardware, we propose to use a concatenation of the rounds in symmetric-key ciphers as hashes. We reduce the number of rounds in order to optimize the performance on hardware while preserving the avalanche properties. Inspired by Xoodoo-NC [8], we consider 96-bit hash functions because network (security) applications typically process source and destination addresses and ports that constitute a total of 96 bits of the header of incoming network packets. We evaluate the resource utilization and performance of the resulting non-cryptographic hash functions to finally conclude that we outperform all previously proposed work in terms of speed. Evaluation results of all the implemented algorithms are presented on both FPGA and ASIC.

This paper is organized as follows: Sect. 2 includes the background information and relevant related work on existing non-cryptographic hashes and lightweight cryptographic ciphers. Sect. 3 provides a brief discussion on all the lightweight ciphers we have used in this work. Sect. 4 presents the avalanche analysis of all the proposed non-cryptographic hashes. The hardware evaluation and the results on FPGA and ASIC are discussed in Sect. 5. Finally, conclusions are drawn in Sect. 6.

2. Background and related work

Non-cryptographic hashes are fast and resource-efficient compared to cryptographic hashes, which makes it favourable for membership queries in networking applications. In such applications, the hash functions must be fast, independent, and uniform [11]. Hence, non-cryptographic hashes must satisfy the avalanche criteria to ensure no correlation between bit-fields, and have low logical depth to be fast. Most of the well-known non-cryptographic hashes

exhibit satisfactory avalanche properties. However, most of these hash functions are designed for software and result in inferior performance on hardware in terms of implementation complexity and speed.

Popular non-cryptographic hash functions such as FNV-1a [10] and Murmur3 [11] are also optimized for implementation in software and do not perform efficiently on hardware. FNV-1a and Murmur3 employ multiplication as the core operation, which is an expensive operation in hardware in terms of resource utilization and latency. Moreover, FNV-1a processes 8 bits per cycle and Murmur process 32 bits per cycle, which also causes the execution delay to be dependent on the message size. Also, the size of the hash output is limited in many of these hashes, for example, 16-bits for SipHash [15] and XORHash [16]. Increasing the hash output size will result in further lowering the speed. Non-cryptographic hash implementations on FPGA such as XORHash prove to be faster than FNV-1a and Murmur3 in terms of operating frequency, but the overall execution delay is not fast enough for high-speed applications such as Terabit Ethernet network applications. Additionally, the output hash size of XORHash is also limited to 16-bits. Grochol and Sekanina proposed various versions of NSGAHash [17], which are also capable of achieving a higher operating frequency due to their low logical depth. However, detailed avalanche analysis shows that there exists a very high correlation between the bit-fields. NSGAHash is designed using evolutionary computation based on the collision resistance on a specific dataset. Such a design based on collision resistance is highly data-dependent [11], which causes a predictable correlation between the bit-fields.

Since cryptographic hash functions have high logical depth and computational overhead, there has been a lot of research focused on the design of lightweight cryptographic hashes. Cryptographic hash functions based on lightweight symmetric-key algorithms address the issues such as high computational overhead and poor resource efficiency. These lightweight algorithms provide both area efficiency and security. Nevertheless, these hashes are typically slower than non-cryptographic hash functions because they require a relatively large number of rounds to satisfy all security requirements. Many of the recent works employ FPGAs as the target platform for running the lightweight cryptographic algorithms [18, 19, 20, 21, 20]. Most of these implementations are proven to be efficient in terms of resource utilization and power consumption. Still, high latency persisted to be a major drawback of these implementations. The implementation of the Photon lightweight hash function presented in [21, 20, 18, 20] shows a minimum cycle count of 12 and a maximum cycle count of 1680. Similarly, for Spongent hash [21, 20, 19], the minimum cycle count is 45 and the maximum cycle count is 1980. One thing which is common in most of the lightweight cryptographic hash functions is the high latency. This is not acceptable in high-speed network security applications where processing at line-rate is a matter of utmost concern. Reducing the number of rounds of such ciphers helps in achieving a faster execution time, but such reduced-round versions result in deteriorated security properties. However, maintaining the security properties is not important for non-cryptographic hash functions as long as the required avalanche properties are satisfied. Hence, reduced-round versions of lightweight cryptographic hashes which satisfies the avalanche properties are excellent replacements for hardware-friendly non-cryptographic hashes. Xoodoo-NC [8] is the result of such an effort. It is a reduced-round version of the cryptographic permutation Xoodoo [22] and is proven to be very efficient on hardware. In this work, we evaluate a number of non-cryptographic hash functions that are based on reduced-round symmetric-key algorithms with respect to avalanche properties, performance, and hardware resource utilization.

3. Lightweight ciphers

For our analysis, we arbitrarily choose eight symmetric-key ciphers: Pyjamask [23], GIFT [24], SKINNY [25], AES [26], SPECK [27], PHOTON [28], Sparkle [29], and TinyJAMBU [30]. TinyJAMBU is a stream cipher, whereas all the others are block ciphers. In this section, the algorithmic details of all the chosen ciphers are concisely summed up. It is pointed out that any special modifications to the first or last round in these algorithms are not discussed here nor are they implemented. Only the regular round function of the ciphers is considered. For a more in-depth and complete description, we forward the interested reader to the specifications. Some of the ciphers have a 96-bit block size, which matches our goal to design 96-bit non-cryptographic hash functions. Some ciphers have a 128-bit block size. For those, the avalanche properties shown in Sect. 4 are calculated for the entire 128-bit output, but the hardware implementation results are generated for only 96 bits, while fixing 32 bits to 0. PHOTON and Sparkle use a 256-bit block size, but to meet our design requirements, the 256-bit block is trimmed down to 144 and 128 bits, respectively, while keeping the avalanche properties intact.

3.1. SPECK

The SPECK block cipher family holds 10 different variations of the SPECK algorithm, having block sizes of 32, 48, 64, 96, and 128 with varying key sizes ranging from 64 to 256.

This paper focuses only on the cipher with a block size and key size of 96 bits. SPECK is an add-rotate-xor cipher, which means it only uses these three operations to calculate the ciphertext. SPECK with a block size and key size of 96-bits consists of 28 rounds. During the rounds, the key and the plaintext are split into two parts of 48 bits. The most significant bits of the plaintext (PT) and the key (K) are stored in PT_1 and K_1 respectively, while the least significant bits are stored in PT_2 and K_2.

Round function: In SPECK, each round performs three operations: Rotation, Addition, and XOR. First, PT_1 is rotated 8 bits to the right, this is followed by an addition with PT_2. Then, PT_1 is XORed with K_2. For PT_2, the operations start after the addition with PT_1, with rotating 3 bits to the left followed by an XOR with PT_1. Figure 1 shows how this is done for a single round. After each round, the output is connected to the input of the next round.

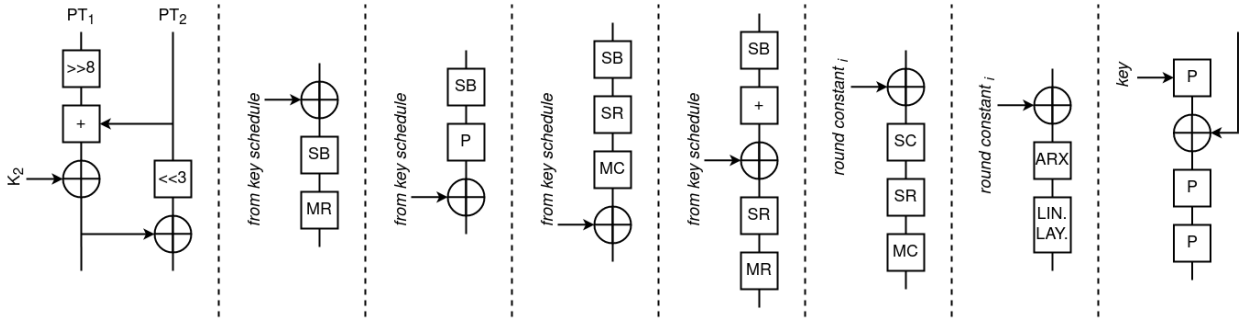

Figure 1: Functional representation of a single round of the 7 chosen ciphers: SPECK, Pyjamask, GIFT, AES, SKINNY, PHOTON, Sparkle, and TinyJAMBU.

Key schedule: The key changes every round for SPECK, this round function works similarly to the encryption round for the plaintext. The only difference is the XOR, where the second operand is the round number, starting from 0 (the first round) and incrementing by 1 in every round.

3.2. Pyjamask

The Pyjamask block cipher family [23] contains two algorithms which support both 96-bit and 128-bit block sizes. Both algorithms use a 128-bit key, perform 14 rounds and rely on a Substitution-Permutation Network (SPN) structure to transform the plaintext into the ciphertext. The plaintext is structured in a bit-by-bit left-right, top-down structure, where each row consists of 32 bits. Each row R_i starts from $32 \times i$ and increments by 1, where $i \in 0, 1, 2, 3$. Depending on the block size of the chosen algorithm, 3 or 4 rows are used. The cell which holds the lowest index represents the most significant bit of the plaintext and the cell with the highest index represents the least significant bit. The key is represented in the same way as the plaintext. Only this time, there are 4 rows used of which each contain 32 bits. Once the plaintext enters the algorithm, it will be referred to as the internal state.

Round function: Each round is composed of three operations, which happen in sequential order: AddRoundKey, SubBytes and MixRows. AddRoundKey first XORs n bits of the key with the internal state. n is 128 for both Pyjamask-128 and Pyjamask-96, however, in case of the latter only the 96 most significant bits are addressed. The result of the XOR is passed through an Sbox. Finally, the internal state is multiplied with a constant matrix.

Key schedule: Throughout the algorithm, different keys are used in each AddRoundKey. These keys are called subkeys, and all originate from the original secret key. To receive these subkeys, three operations are executed in each round on the current round key: MixColumns, MixRows and Constant Addition. Because Pyjamask-96 and Pyjamask-128 both use 128-bit keys, the same operations are used to create the subkeys.

3.3. GIFT

The GIFT block cipher [24] family contains two algorithms: GIFT-64 and GIFT-128. Both use a key size of 128 bits, while the former uses a block size of 64 bits and the latter of 128 bits. In this paper, GIFT-128 is used. Like Pyjamask, GIFT is an SPN cipher containing 40 rounds.

Round function: Each round of GIFT is based on three operations, namely SubCells, PermBits, and AddRoundKey. Similar to SubBytes in Pyjamask, each 4-bit column is compared to an Sbox and replaced by the appropriate value. The PermBits operation shuffles all the bits in the internal state according to a specific permutation. Finally, the AddRoundKey operation adds the roundkey to the internal state through an XOR.
Key Schedule: GIFT follows a simple key schedule operation to minimize the area in hardware. The key is split in to blocks K_i of 32 bits each as in the AddRoundKey operation, where $i \in \{0, 1, 2, 3\}$. K_0, K_1 and K_2 are then shifted towards the least significant bits of the key state, while K_3 now holds the most significant bit. K_3 is then split again into two 16-bit blocks, $K_{3,1}$ and $K_{3,2}$. These two blocks are then right-rotated over 2 and 12 positions, respectively.

3.4. AES

AES is probably the most popular and well-known cipher. It has a block size of 128 bits and the key size can be either 128, 192 or 256 bits. Depending on the key size, the number of rounds is 10, 12 and 14, respectively. This paper focuses on the algorithm having a key size of 128 bits. Similar to GIFT, the input plaintext is structured in a top-down,left-right manner. The key is also structured in a similar fashion.
Round function: Each round consists of 4 operations: SubBytes, ShiftRows, MixColumns and AddRoundKey. AddRoundKey performs a simple bitwise XOR of the key with the internal State. Each round uses a different round key. These round keys are calculated in the key schedule. In the next step, each byte in the internal state is replaced through an Sbox lookup. After the Sbox, the rows of the state are shifted cyclically in a specific manner. Finally, in the MixColumns step, a matrix multiplication is done. The state is updated with the resulting product.
Key schedule: The key schedule generates a new round key, starting from the initial key. Every key is split in four equally sized parts. One of these parts is rotated and subsequently passes another Sbox. The result of the Sbox is then XORed with a round constant and another part of the incoming key. The final two parts are finally XORed in turn with the previous result.

3.5. SKINNY

The SKINNY block cipher family holds six algorithms, namely SKINNY-64-64, SKINNY-64-128, SKINNY-64-196, SKINNY-128-128, SKINNY-128-256, and SKINNY-128-384. The first numerical value in the name is the block size used in the algorithm, and the second value is the key size. In this paper, we focus only on SKINNY-128-128. The plaintext and key are initialized similarly to AES but in a left-right top-down manner. The number of rounds of the SKINNY cipher varies with the algorithm used. For SKINNY-128-128, the number of rounds required is 40.
Round function: Each round consists of 5 different operations: SubBytes, AddConstants, AddRoundKey, ShiftRows and MixColumns. The SubBytes operation follows the exact operation of SubBytes in AES, the only difference being the values in the Sbox. In the AddConstants operation, the round constants are XORed to the first column of the internal State. The round constant is a 6-bit number and is generated using an LFSR, which changes every round. The key state is added to the internal state in the AddRoundKey operation. Unlike the AddRoundKey function in AES, only the 8 most significant bytes of the key are XORed to the 8 most significant bytes of the internal state. The ShiftRows function works the same way as the ShiftRows function in AES, the only difference is that the rows are shifted in the opposite direction. The MixColumns function is the final round operation. Similar to MixColumns in Pyjamask and AES, each column of the internal state is multiplied by a constant Matrix M.
Key schedule: Just like other ciphers, the key is updated in every round into different subkeys. In SKINNY, the key updates are similar to the key schedule in GIFT.

3.6. PHOTON

We make use of the underlying PHOTON_{256} [31] permutation in PHOTON-Beetle to construct a non-cryptographic hash function. The number of rounds is 12, and the input state to the permutation is represented as a 8x8 matrix with each element consisting of 4 bits. This leads to an input size of 256 bits. However, to meet our requirements, we trim down the input state from a 8x8 to a 6x6 matrix, and the corresponding elements from the Sbox, Round Constant matrix, and MixColumn matrix are ignored, i.e., the MixColumn matrix of size 8x8 is also trimmed down to a 6x6 matrix and the remaining constant elements are ignored. The Round Constant matrix of size 8x12 is trimmed down to a size of 6x12.
Permutation (Round function): Each round of the permutations consists of four rounds: AddConstant, SubCells, ShiftRows, and MixColumn. The AddConstant operation is very much similar to the AddConstant/AddRoundKey operations of the algorithms stated above. In each round, a 4-bit constant from the Round Constant matrix of size 8x12

is XORed with all the elements in the first column of the input state matrix. Similar to the SubBytes operation stated in other algorithms, the SubCells operation replaces the elements in the input state with the values from the Sbox. The ShiftRows and MixColumn operations are also similar, where the elements are shifted cyclically during the ShiftRows operation and multiplied with constant matrix M during the MixColumn operation.

The PHOTON permutation does not use a key, but uses pre-stored round constants in each round.

3.7. Sparkle

The Sparkle family consists of 3 permutations: SPARKLE256$_{n_s}$, SPARKLE384$_{n_s}$, and SPARKLE512$_{n_s}$, each having block sizes of 256, 384 and 512 bits, respectively. n_s denotes the number of steps of the permutation and is defined for any $n_s \in N$. Sparkle uses *slim* and *big* instances of sparkle, and these instances differ only in terms of number of steps of the permutations, where n_s=7 for the *slim* instance and $n_s = 10$ for the *big* instance. n_b defines the number of branches where each branch represents a block of 64-bits. n_b must be even and Sparkle uses 4, 6, or 8 branches, and 8 is the maximum number of branches that can be used. We employ the *slim* instance of SPARKLE256 permutation, but to reduce the output size of the hash, we take n_b=2, which keeps the output bits to 128.

Permutation (Round function): The Sparkle permutation is a classical Substitution-Permutation Network (SPN) construction in which the S-boxes are different in each branch. The 64-bit block cipher *Alzette* plays the role of the S-box which is iterated in each step of the permutation, and the constants (keys) used in Alzette are branch dependant. Alzette (ARX-box layer) is followed by a *linear layer* in each step. A step-dependant constant is XORed to the state in the beginning of each step prior to ARX-box layer and linear layer.

Key schedule: The Sparkle permutation itself does not use a key, but the 64-bit block cipher Alzette, which is employed as the ARX-box layer, uses a 32-bit key. The keys are pre-stored as round constants. They are different in each step and dependant on the number of branches.

3.8. TinyJAMBU

TinyJAMBU is specified for three different key sizes: 128, 192 and 256 bits. Each variant uses a 96-bit nonce, 128-bit state, and 64-bit authentication tag. TinyJambu employs a keyed permutation P_n, where n defines the number of rounds. In each round of the permutation, a 128-bit non-linear feedback shift register (NFSR) is used to update the state. The permutation uses two different rounds, N_1 and N_2, for state update during different stages, where N1 is 640 (P_{640}) and N2 is 1024 (P_{1024}). Since TinyJAMBU is a stream cipher, the avalanche properties cannot be satisfied when using the plaintext block as the input to the hash function where the plaintext is XORed only once to the generated keystream. To make use of TinyJAMBU as a non-cryptographic hash function, the *Associated Data (AD)* input of TinyJAMBU is used as the input to the hash and the plaintext value is tied to 0. The authentication tag or Message Authentication Code (MAC) is used as the output and it can be any multiple of 32 bits.

Round function: The TinyJambu-128 keystream generation and encryption round consist of 4 stages: initialization, associated data processing, encryption, and finalization. The initialization stage consists of a key setup and a nonce setup phase. The key setup sets the 128-bit state to zero and the state is updated with the keyed permutation using N2 rounds. The nonce setup consists of three rounds where in each round a constant *FrameBitsIV* is XORed with the state, followed by a state update using the keyed permutation with N1 rounds, and an XOR of the 32-bit nonce with the state. The associated data (AD) is our hash input and there are $\frac{adlen}{32}$ steps in AD processing, where $adlen$ is the length of AD in bits. In each step of AD processing, a constant *FrameBitsAD* is XORed with the state, followed by a state update using the keyed permutation with N1 rounds, and finished by XORing the 32 bits of AD with the state. Two different steps are followed for full blocks (32 bits) of AD and partial blocks (if the last block is not a full block) of AD. In the encryption stage, there are $\frac{mlen}{32}$ steps, where $mlen$ is the length of the plaintext in bits. The encryption stage is similar to AD processing with the difference being the constant *FrameBitsPC*. The ciphertext is generated by XORing the plaintext with a part of the state. In our implementation, we do not generate the ciphertext, and the plaintext value is taken as 0. However, the length of the plaintext determines whether the encryption stage needs to be executed or not. Hence keeping the $mlen$ to zero ignores the encryption stage, whereas keeping $mlen$ to non-zero helps in achieving the avalanche properties with less rounds. The finalization or MAC stage involves XORing the state with the constant *FrameBitsFinalization* followed by state update using the keyed permutation with N1 rounds, and finally assigning state[64:95] to the first 32 bits of the tag. This step is repeated again with N2 rounds of permutation to generate the next 32 bits of the tag. Since the tag is taken as our hash output, we have repeated this step once more to generate the 96-bit tag output. To optimize the implementation, 128 rounds of the permutation are performed in parallel and hence the values of N1 and N2 are taken as 5 and 8, respectively, instead of 640 and 1024.

Key schedule: TinyJAMBU uses a 128-bit key. The key is used to update the state using the keyed permutation P_n. The key-nonce pair used must be unique for each message.

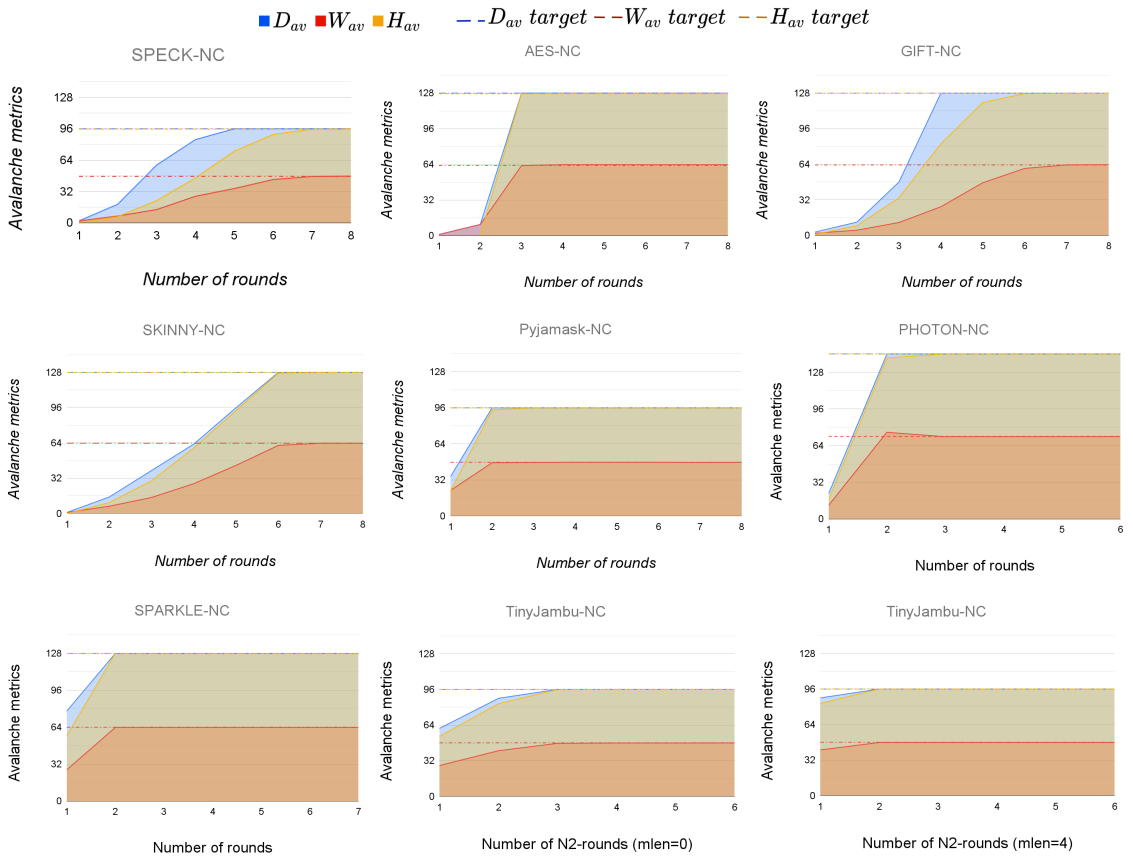

Figure 2: The avalanche metrics for SPECK-NC, Pyjamask-NC, GIFT-NC, AES-NC, SKINNY-NC, PHOTON-NC, Sparkle-NC, and TinyJAMBU-NC respectively. The areas show the obtained metric while the horizontal lines indicate the target. The legend indicates which colors are the Avalanche Dependence (D_{av}), the Avalanche Weight (w_{av}), and the Avalanche Entropy (H_{av}).

4. Analysis of non-cryptographic hash functions

The ciphers mentioned in the previous section have proven to be secure, but not fast enough for applications such as Bloom filters and sketching architectures. Our goal is to use these ciphers as hash functions by reducing the number of rounds and to analyze the avalanche properties [32] to make sure that they are still satisfactory. The use of keys does not cause any difference in avalanche properties but allows us to easily change the hash function by changing the key value. The avalanche performance can be assessed by analyzing to which extent the output value changes for a slight change in the input of the hash function. If the hash function does not show a good avalanche performance, the randomization is poor for the hash function, which leaves the function vulnerable to attackers. The avalanche properties are determined here using three avalanche metrics put forward by Daemen et al. [33]: Avalanche dependence, Avalanche weight and Avalanche entropy, which are defined below.

Avalanche dependence: For a single-bit change in the input, the number of bits in the output that may flip defines the avalanche dependence. The avalanche dependence is defined as:

$$D_{av} = n - \sum_{i} g(p[i]) \tag{1}$$

Here n is the number of bits in the output, p is the probability vector, and $p[i]$ represents the probability that the bit i of the output flips for a single bit change in the input. $g(p) = 1$ if $p = 0$ and $g(p) = 0$ otherwise.

The Avalanche dependence is satisfied when $D_{av} = n$ for all outputs as a result of a single bit change at the input.

Avalanche weight: The avalanche weight generalizes the avalanche criterion and is defined as the expected Hamming weight of the output difference for a single bit change in the input. It is defined as:

$$w_{av} = \sum_{i} p[i] \tag{2}$$

The condition is satisfied when the avalanche weight is equal to 50% of the number of output bits i.e., $w_{av} \approx \frac{n}{2}$, for all inputs with a single bit change (Hamming weight = 1).

Avalanche entropy: The avalanche entropy visualizes the uncertainty of whether the bits in output flip or not for a single bit change in the input. It is defined as:

$$H_{av} = \sum_{i} (-p[i].log_2(p[i]) - (1 - p[i]).log_2(1 - p[i])) \tag{3}$$

The H_{av} generalizes the strict avalanche criterion. The avalanche entropy is said to be satisfactory if it fulfils the condition $H_{av} \approx n$ for all inputs with a single bit change.

In our analysis, to make sure that the avalanche metrics are satisfactory in any given condition, the worst-case values are considered. To have high precision for the avalanche metrics, we set the number of iterations M as large as 250'000, and $\frac{1}{\sqrt{M}}$ is the expected standard deviation of each element in the probability vector p [33]. In each iteration, all n bits are flipped one by one and the avalanche metrics are calculated. This ensures that the avalanche metrics obtained are stable and precise.

The plot of the avalanche metrics of each cipher as a function of the number of rounds is shown in Figure 2. We add NC (Non-Cryptographic) to the name of the cipher to indicate that we are considering the reduced-round version in order to construct the non-cryptographic hash function. While SPECK has 28 rounds, SPECK-NC only requires 7 rounds to achieve satisfactory avalanche properties. Pyjamask normally requires 14 rounds, but the avalanche metrics plot shows that only 2 or 3 rounds are required for Pyjamask-NC to meet the avalanche properties. GIFT-NC requires only 6 to 7 rounds to achieve the avalanche property requirements as shown in the Figure, in contrast to the actual 40 rounds in GIFT. The number of rounds to have sufficient avalanche properties for AES-128 can also be reduced from 10 rounds to 3 rounds in AES-NC. Similar to GIFT, SKINNY-128-128 also uses 40 rounds but requires only 6 rounds to meet the avalanche criteria in SKINNY-NC. The cryptographic PHOTON permutation requires 12 rounds in total, but only 3 rounds are needed to meet the avalanche requirements. Even with 2 rounds, D_{av} of PHOTON is 144, and the values of W_{av} and H_{av} are satisfactory with 75.64 and 140.53, respectively. The number of rounds for Sparkle can be reduced from 7 to 2 to meet the avalanche properties.

Because TinyJAMBU is a stream cipher, the avalanche calculations slightly differ from the calculations for the block ciphers. Even though we have tied the plaintext value to 0, the length of the plaintext, *mlen*, plays an important role in the output generation. If $mlen = 0$ bytes, the encryption stage, as mentioned in Section 3.8, is ignored and if $mlen > 0$, the encryption stage is entered. Hence we have plotted the avalanche properties for $mlen = 0$ and $mlen = 4$ bytes. Instead of using N_1 and N_2 to define each round as mentioned in Section 3.8, we have accumulated all the rounds, where each round is 128 rounds of the permutation performed in parallel. In this work, one round of TinyJAMBU is defined as a 128-round permutation performed in parallel. $mlen = 8$ or $mlen = 12$ is not plotted because only one round of N_1 and N_2 is enough to achieve the required avalanche properties. Based on our analysis, incrementing N_1 has a negligible effect on the improvement of the avalanche properties, hence we have only incremented N_2 while calculating the avalanche properties. The chart shows only the incremental rounds of N_2. The cryptographic version of TinyJAMBU requires a total of 80 rounds ($N_1 = 5, N_2 = 8$) for *mlen* equal to 12 bytes, whereas only 12 rounds with $mlen = 8$ are required to meet the avalanche criteria. This analysis shows that we can use reduced-round versions of cryptographic ciphers as non-cryptographic hash functions. The number of rounds can be reduced by 70-85% while maintaining excellent avalanche properties. With a reduced number of rounds, the latency can be improved keeping the hardware resource requirement to a minimum. A summary of the avalanche metrics analysis as a function of the number of rounds is shown in Table 1. For TinyJAMBU, the accumulated rounds of 128-round permutations performed in parallel are given for r and r_{av} in the Table.

Table 1
The number of rounds needed for meeting the avalanche criteria

Cipher	D_{av}	w_{av}	H_{av}	r	r_{av}
SPECK-NC (96-bit)	96	47.45	95.68	28	7
Pyjamask-NC (96-bit)	96	47.73	95.96	14	3
TinyJAMBU-NC (96-bit)	96	47.58	95.89	80	12*
GIFT-NC (128)	128	63.39	127.95	40	7
AES-NC (128-bit)	128	63.0	127.43	10	3
SKINNY-NC (128-bit)	128	63.72	127.95	40	7
Sparkle-NC (128-bit)	128	63.86	127.99	7	2
PHOTON-NC (144-bit)	144	71.84	143.99	12	3

r: Number of rounds in the original cipher
r_{av}: Number of rounds to meet avalanche criteria
*Accumulated rounds for $N_1 = 1, N_2 = 1, mlen = 8$

5. Hardware Evaluation

5.1. Experimental setup

The non-cryptographic hash functions are implemented on an FPGA and synthesized for ASIC to evaluate the resource requirements and operating speed. On FPGA, the non-cryptographic hash functions are evaluated on both Zynq and Virtex Ultrascale+ FPGAs. All the values presented in the paper are the implementation results on Virtex Ultrascale+ XCVU7P-FLVB2104-2-i device. On ASIC, the non-cryptographic hash functions are synthesized for TSMC 65nm.

The general block diagram of the hardware implementation of the reduced-round ciphers as hash functions is shown in Figure 3. This Figure shows that the implementations are unrolled with only a register before and after the rounds. Therefore, the cycle count is fixed to a single clock cycle. Pipelining is not applied in the implementation and the whole execution of all the rounds is completed in this single clock cycle, which provides the worst-case operating frequency. The plaintext is the input to the hash function and the resulting ciphertext is the output of the hash. Adding a key is not really necessary and the addition of key does not affect the avalanche properties. In textbook hash functions, the digest should always be the same, given the same input. In some applications, however, it is useful to have multiple, independent hash values for the same input, like in Bloom filters. Adding a key to the block cipher allows the generation of multiple independent hash values using the same hash function but with different keys. In hardware implementations without the key, we eliminated all the computational logic involving the key and manually removed all the hardware logic that is used to process the key bits.

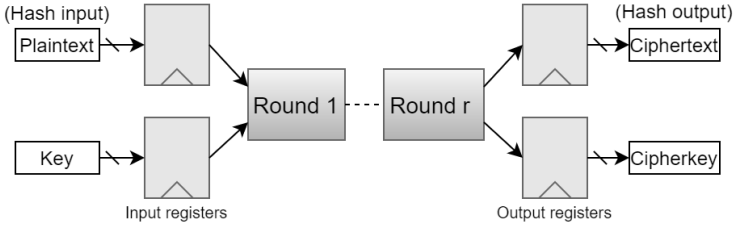

Figure 3: General hardware setup of hash implementation

5.2. FPGA results

The performance of the implemented non-cryptographic hash functions is evaluated mainly in terms of maximum operating frequency, latency, resource requirements, and throughput. The number of rounds r for each cipher has been chosen corresponding to the number of rounds required to meet the sufficient avalanche properties (see Table 1). The results are also compared with some of the well-known existing non-cryptographic hashes.

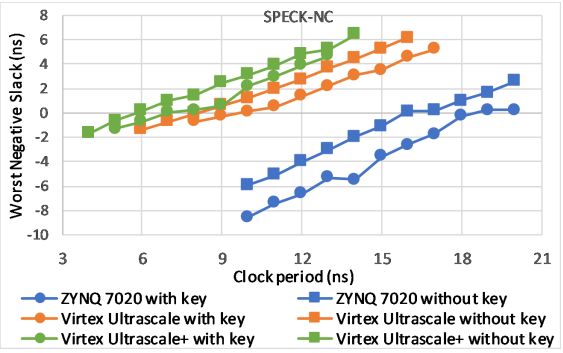

Figure 4: Timing analysis of SPECK-NC

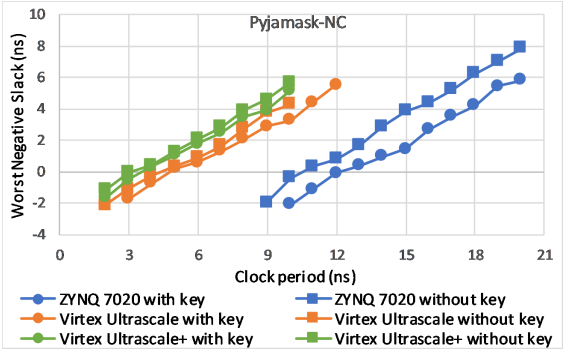

Figure 5: Timing analysis of Pyjamask-NC

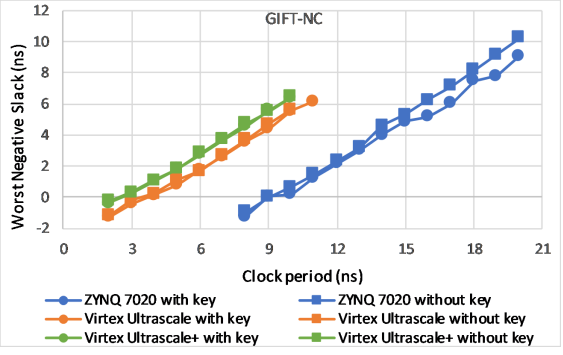

Figure 6: Timing analysis of GIFT-NC

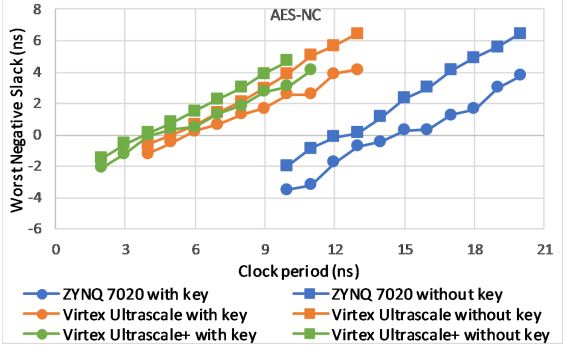

Figure 7: Timing analysis of AES-NC

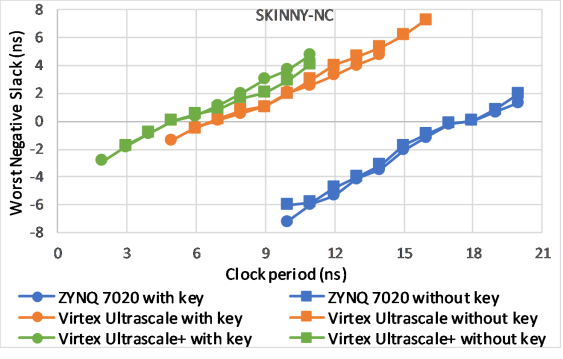

Figure 8: Timing analysis of SKINNY-NC

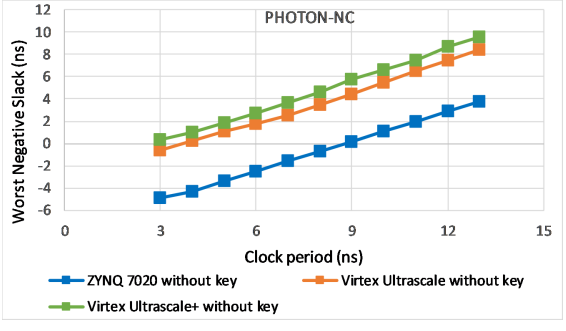

Figure 9: Timing analysis of PHOTON-NC

5.2.1. Latency and throughput

To analyze the timing, the worst negative slack is plotted against varying clock periods and is shown in Figures figs. 4 to 11. For each cipher, the timing results with and without adding the key are plotted. In general, it can be seen from the Figures that the implementation with key is slightly slower compared to the implementation without key. This is because of the fact that all the calculations involving the key are completely eliminated in the implementation without key. It is also evident from the plots that for larger FPGAs, the latency is lower. The maximum possible frequency for each implementation on hardware is given in Table 2. Compared to each other, GIFT-NC and PHOTON-NC exhibit the best operating frequency and prove to be faster than the other ciphers. With and without using the key, the variations in latency of GIFT-NC, Pyjamask-NC, and SKINNY-NC are negligible. However, adding

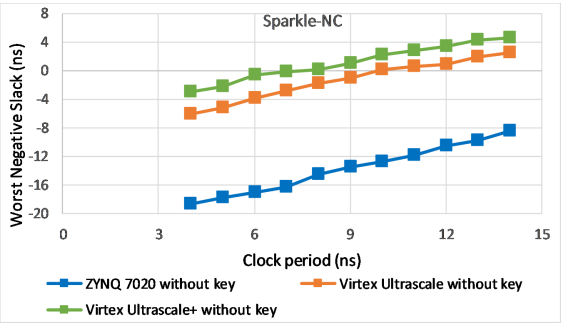

Figure 10: Timing analysis of Sparkle-NC

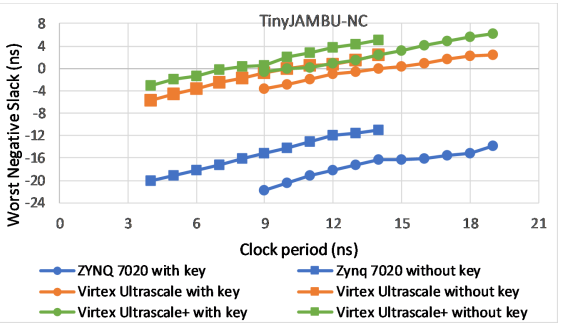

Figure 11: Timing analysis of TinyJAMBU-NC

Table 2
Maximum operating frequency of hash implementations

Hash Function	Max. Frequency (without key)	Max. Frequency (with key)
SPECK-NC	171.6 MHz	144.8 MHz
Pyjamask-NC	279.4 MHz	271.6 MHz
GIFT-NC	369.7 MHz	365.8 MHz
AES-NC	255.9 MHz	214.8 MHz
SKINNY-NC	202.4 MHz	202.5 MHz
PHOTON-NC	384.6 MHz	NA
Sparkle-NC	133.3 MHz	NA
TinyJAMBU-NC	138.9 MHz	104.2 MHz

NA-Not applicable

Table 3
Resource utilization

Hash function	LUTs (without key)	Flip Flops (without key)	LUTs (with key)	Flip Flops (with key)
SPECK-NC	432	192	1273	384
Pyjamask-NC	811	448	1615	448
GIFT-NC	546	512	665	512
AES-NC	2225	512	3402	512
SKINNY-NC	2176	512	2348	512
PHOTON-NC	1338	436	NA	NA
Sparkle-NC	811	388	NA	NA
TinyJAMBU-NC	1521	387	1919	515

NA-Not applicable

the key in TinyJAMBU adds a 33% increase in latency whereas SPECK-NC and AES-NC add around 17% latency, which is caused by additional computational steps and the delay involving the computations.

The throughput of the hash function is a function of the latency and input block size. The throughput here is calculated using the equation $\frac{Block\ size}{latency\ in\ cycles} \times f_{max}$, where f_{max} is the maximum operating frequency. The throughput is of the order of ten thousand as the latency is only a single clock cycle thanks to the unrolling.

5.2.2. Resource requirements

The resource utilization of the implementations for the best achievable latency is shown in Table 3. GIFT-NC proves to be the most resource-efficient of all the compared ciphers when the key is included. SPECK-NC uses the lowest resources when the key is not added. AES-NC and SKINNY-NC are the most expensive in terms of resources while showing moderate operating frequency. SPECK-NC, even though showing efficiency in resource utilization without the key, requires ≈3× more resources when the key is added. GIFT shows the least variation in resources whether the key is added or not.

Table 4
Comparison of maximum frequencies, throughput (Tp), throughput per LUT (Tp/LUT), and delay with related work

Design	Block Size (In/Out)	Maximum Frequency	Tp (Mbps)	Tp / LUT (Mbps/LUT)	Latency (ns)
Murmur3	96/64	120.6 MHz	2573	4.54	24.87
FNV-1a	96/128	122.9 MHz	925	1.63	130.08
SipHash [17]	96/16	182.8 MHz	1463	1.38	21.88
XORHash [17]	96/16	**627.3 MHz**	2868	9.86	11.13
NSGAHash7 [17]	96/16	184.1 MHz	1473	18.41	21.72
Xoodoo-NC [8]	96/192	363.6 MHz	34'906	**112.96**	2.75
SPECK-NC (ours)	96/96	171.6 MHz	16'473	38.13	5.82
Pyjamask-NC (ours)	96/96	279.4 MHz	26,822	32.70	3.58
GIFT-NC (ours)	96/128	369.7 MHz	35'491	65.00	2.70
AES-NC (ours)	96/128	255.9 MHz	24'566	11.04	3.91
SKINNY-NC (ours)	96/128	202.4 MHz	19'440	8.93	4.93
PHOTON-NC (ours)	96/144	384.6 MHz	**36'922**	27.59	**2.6**
Sparkle-NC (ours)	96/128	133.3 MHz	12'797	15.78	7.5
TinyJAMBU-NC (ours)	96/96	138.9 MHz	13'334	8.77	7.2

5.2.3. Comparison with related work

In this work, we focus on reduced-round, reduced-logic implementations of non-cryptographic hash functions based on existing ciphers. Hence, a straightforward comparison with the existing full-fledged implementations of the same ciphers on FPGA is not fair. For example, the throughput of the FPGA implementation of AES-128 and SPECK-96 given by Diehl et al. [34] are 119 Mbps and 1622 Mbps, respectively, which is 206× and 10× lower than our reduced-round implementations. Similarly, the high-throughput implementation of SKINNY-128 on FPGA [25] has a Throughput/LUT of 3.06 Mbps, which is 3 times lower than our reduced-round implementation. To have a fair comparison, we have also implemented FNV-1a and Murmur3 hash on FPGA and compared them with the reduced-round ciphers. The comparison of some of the fast non-cryptographic hashes with our implementations is shown in Table 4.

As shown in Table 4, the throughput of the previously proposed non-cryptographic hash functions is significantly lower compared to our implementations. Only the throughput of Xoodoo-NC is comparable, which is also a reduced round implementation of the Xoodoo permutation. XORHash [17] shows the highest operating frequency of all related work, but the throughput compared to Sparkle-NC is still 4.5× lower, where Sparkle-NC has the lowest throughput among our reduced round implementations. PHOTON-NC shows the best throughput, which is 12.9× higher than that of XORHash and slightly higher than GIFT-NC and Xoodoo-NC. Most of the operations in PHOTON-NC are 4-bit operations which prove to be advantageous for PHOTON-NC on hardware. Besides the throughput per LUT, which was already given in Table 4, we calculate another metric that takes into account both the latency and the FPGA resources. It is calculated as $1/(latency * LUTs)$ and shown in Figure 12 together with the $throughput/LUT$. The higher these two parameters, the better the performance of the hash function.

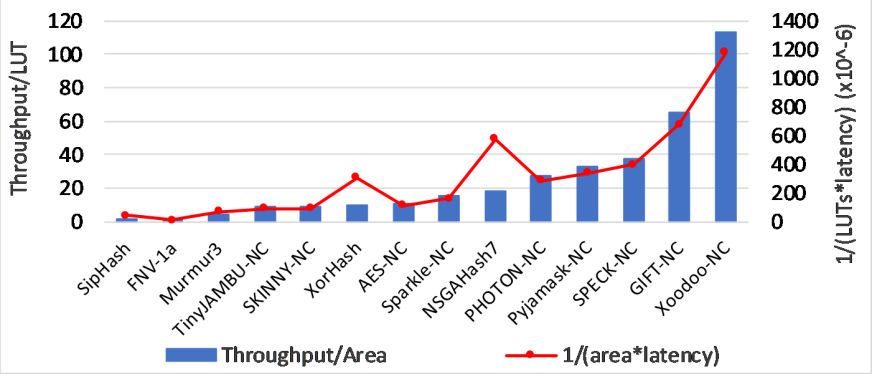

Figure 12: Throughput and latency with respect to resource utilization

Xoodoo-NC [8] exhibits the best throughput per LUT, which is 1.7× higher than GIFT-NC and 4× higher than PHOTON-NC. Hence for applications where area plays an important role, Xoodoo-NC should be preferred over GIFT-NC or PHOTON-NC. Another observation when evaluating throughput per area is that Murmur3, FNV-1a, and NSGAHash7 employ 29, 38, and 3 DSP blocks, respectively, respectively, in their implementations, whereas our reduced-round ciphers do not use any DSP slices. PHOTON-NC has the lowest execution delay among all the implementations. The area-latency performance of hashes shows similar characteristics as throughput-area for all hashes except for NSGAHash7 and XorHash. Both NSGAHash7 and XorHash show increased area-latency performance, thanks to the relatively lower area-to-latency ratio, while all other hashes' area-latency performance closely follows their throughput-area characteristics. The other versions of NSGAHash deliver much higher operating frequency and a higher area-latency performance. However, NSGAHash do not satisfy the avalanche criteria as mentioned in Sect. 2 and NSGAHash7 is the best version of NSGAHash in terms of collision resistance.

5.3. ASIC results

Synthesis has been done to have an indication of the ASIC performance of the algorithms. This is done for the 65nm TSMC manufacturing node. The typical-condition timing constraints of nominal voltage and 25 degree Celsius are used. The synthesis tool used is Genus 20.10 from Cadence. The first synthesis has been done to find the maximum frequency, and timing and area are reported. Then, synthesis has been done at 100 MHz to find a minimum area. It is important to know that the synthesis tool will always try to optimize for the area. This is why an algorithm synthesized for 100 MHz is smaller than one synthesized for 500 MHz. The synthesis tool has a lot of standard cells available, some are smaller but have a lower drive strength while others are larger and have a higher drive strength.

All algorithms are unrolled and all rounds are executed in a single clock cycle. This way the algorithm with the highest frequency also has the lowest delay. For the implementations in this work that have an optional key, synthesis has been done on both versions, with and without the key. The algorithms Xoodoo-NC, PHOTON-NC and Sparkle-NC do not use a key.

Figure 13 shows the maximum frequency of the algorithm implementation and Figure 14 shows the required area to achieve this frequency. We can clearly see that Xoodoo-NC has the highest frequency and the lowest area, which makes it the best option of the tested algorithms.

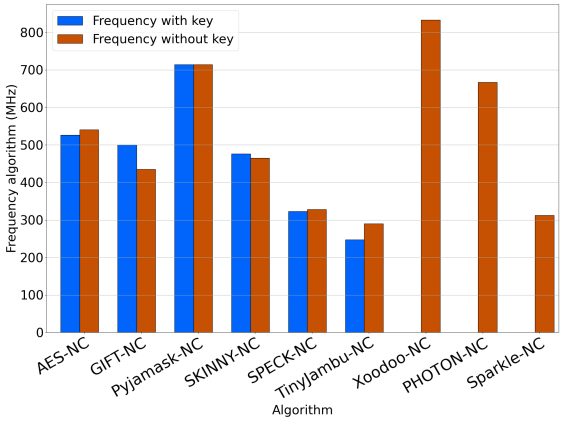

Figure 13: Max. frequency of the ASIC implementations

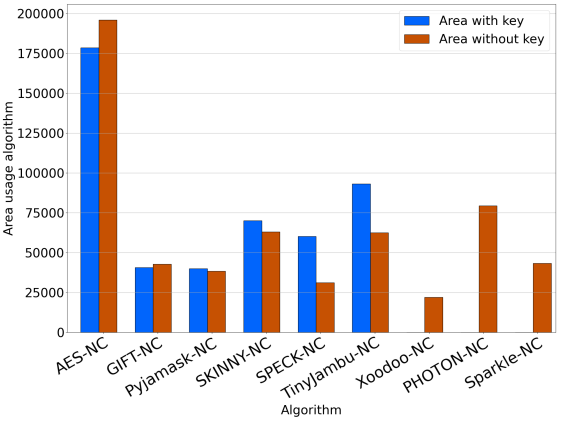

Figure 14: ASIC area at max. frequency

Figure 15 shows the area results after synthesis for the TSMC 65nm technology. It clearly shows that AES-NC is the largest evaluated implementation, whereas Xoodoo-NC is the smallest. GIFT, Pyjamask, and PHOTON are promising algorithms as well on ASIC, considering the fact that these algorithms deliver high operating frequency with considerably low area requirements.

6. Conclusion

This paper investigates the use of reduced-round versions of existing lightweight ciphers to serve as non-cryptographic hash functions. In order to do so, we simulate the avalanche properties of each cipher as a function

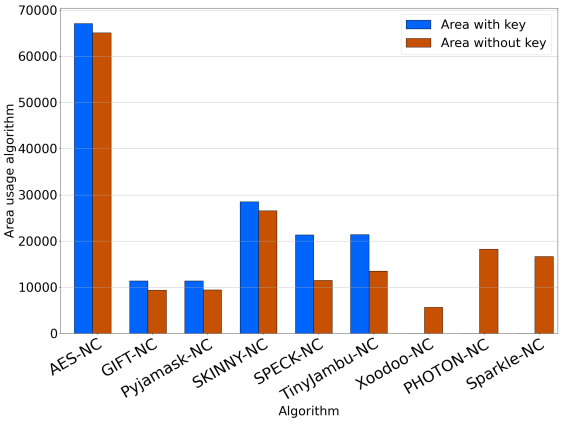

Figure 15: Area of the ASIC implementations at 100 MHz

of the number of rounds. Based on the minimum number of rounds required to satisfy the avalanche requirements, we evaluate the resource utilization and latency on an FPGA and ASIC, assuming completely unrolled and non-pipelined architectures. We show that the newly proposed PHOTON-NC outperforms all previously proposed hash functions in terms of throughput and latency on FPGA. However, Xoodoo-NC is the most efficient algorithm on ASIC in all aspects. Also, the other non-cryptographic hash functions that we evaluate, are valuable alternatives for PHOTON-NC, which allows having a pool of different non-cryptographic hash functions with desirable throughput and latency. This is important for applications that need more than one different non-cryptographic hash function in high-speed lookup or counting architectures.

Acknowledgement

This work is supported by the ESCALATE project, funded by FWO (G0E0719N) and SNSF (200021L_182005), and by Cybersecurity Research Flanders (VR20192203).

References

[1] Marcelo Bagnulo, Philip Matthews, and Iljitsch van Beijnum. Stateful nat64: Network address and protocol translation from ipv6 clients to ipv4 servers. Technical report, 2011.

[2] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006.

[3] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[4] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[5] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] MV Ramakrishna. Practical performance of bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237–1239, 1989.

[7] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[8] Arish Sateesan, Jo Vliegen, Joan Daemen, and Nele Mentens. Novel Bloom filter algorithms and architectures for ultra-high-speed network security applications. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 262–269. IEEE, 2020.

[9] Arish Sateesan, Jo Vliegen, Simon Scherrer, Hsu-Chun Hsiao, Adrian Perrig, and Nele Mentens. Speed records in network flow measurement on fpga. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 219–224. IEEE, 2021.

[10] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald Eastlake, and Tony Hansen. The FNV non-cryptographic hash algorithm. *Ietf-draft*, 2011.

[11] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*, 44(6):681–698, 2014.

[12] Amit Kulkarni, Monica Chiosa, Thomas B Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. Hyperloglog sketch acceleration on fpga. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 47–56. IEEE, 2020.

[13] Thomas Claesen, Arish Sateesan, Jo Vliegen, and Nele Mentens. Novel non-cryptographic hash functions for networking and security applications on fpga. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 347–354. IEEE, 2021.

[14] Stephen M Trimberger and Jason J Moore. FPGA security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

[15] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[16] Zhiruo Cao and Zheng Wang. Flow identification for supporting per-flow queueing. In *Proceedings Ninth International Conference on Computer Communications and Networks (Cat. No. 00EX440)*, pages 88–93. IEEE, 2000.

[17] David Grochol and Lukas Sekanina. Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 257–263. IEEE, 2018.

[18] Mohammed Omar Awadh Al-Shatari, Fawnizu Azmadi Hussin, Azrina Abd Aziz, Gunawan Witjaksono, and Xuan-Tu Tran. FPGA-Based Lightweight Hardware Architecture of the PHOTON Hash Function for IoT Edge Devices. *IEEE Access*, 8:207610–207618, 2020.

[19] Carlos Andres Lara-Nino, Miguel Morales-Sandoval, and Arturo Diaz-Perez. Small lightweight hash functions in FPGA. In *2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 1–4. IEEE, 2018.

[20] N Nalla Anandakumar, Thomas Peyrin, and Axel Poschmann. A very compact FPGA implementation of LED and PHOTON. In *International Conference on Cryptology in India*, pages 304–321. Springer, 2014.

[21] Bernhard Jungk, Leandro Rodrigues Lima, and Matthias Hiller. A systematic study of lightweight hash functions on FPGAs. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6. IEEE, 2014.

[22] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. *IACR Cryptol. ePrint Arch.*, 2018:767, 2018.

[23] Dahmun Goudarzi, Jérémy Jean, Stefan Kölbl, Thomas Peyrin, Matthieu Rivain, Yu Sasaki, and Siang Meng Sim. Pyjamask: Block Cipher and Authenticated Encryption with Highly Efficient Masked Implementation. *IACR Transactions on Symmetric Cryptology*, pages 31–59, 2020.

[24] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.

[25] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. Skinny-aead and skinny-hash. *IACR Transactions on Symmetric Cryptology*, pages 88–131, 2020.

[26] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced Encryption Standard (AES), 2001-11-26 2001.

[27] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[28] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. Photon-beetle authenticated encryption and hash family. *NIST Lightweight Compet. Round*, 1:115, 2019.

[29] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Qingju Wang, and Alex Biryukov. Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST round*, 2, 2019.

[30] Hongjun Wu and Tao Huang. Tinyjambu: A family of lightweight authenticated encryption algorithms. the nist lightweight cryptography (lwc) standardization project (a round-2 candidate), 2019. *URL https://csrc. nist. gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/TinyJAMBU-spec-round2. pdf*.

[31] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In *Annual Cryptology Conference*, pages 222–239. Springer, 2011.

[32] Réjane Forrié. The strict avalanche criterion: spectral properties of boolean functions and an extended definition. In *Conference on the Theory and Application of Cryptography*, pages 450–468. Springer, 1988.

[33] Joan Daemen, Seth Hoffert, G Van Assche, and R Van Keer. The design of xoodoo and xoofff. 2018.

[34] William Diehl, Farnoud Farahmand, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. Comparison of hardware and software implementations of selected lightweight block ciphers. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.