A Lightweight 1.16 pJ/bit Processor for the Authenticated Encryption Scheme KetjeSR

Yun-Wen Lu[1], Antoon Purnal[2], Simon Vandenhende[2], Chen-Yi Lee[1], Ingrid Verbauwhede[2], and Hsie-Chia Chang[1]

Email: *ewan41311.ee05g@nctu.edu.tw,* {*antoon,simon,ingrid*}.{*purnal,vandenhende,verbauwhede*}*@kuleuven.be*
cylee@si2lab.org, hcchang@mail.nctu.edu.tw
[1]Department of Electronics Engineering and Institute of Electronics, National Chiao Tung University, Taiwan
[2]Department of Electrical Engineering, K.U.Leuven, Belgium

***Abstract*—This paper presents the results of the first ASIC implementation of the authenticated encryption scheme KetjeSR. The design covers the encryption and decryption operation in combination with a handshake protocol for the data transfer. The chip implementation was done in a TSMC 90nm GUTM process. The encryption/decryption module has an area footprint of 12.2kGE. The processor reaches an end-to-end throughput of 2.08 Gbps when running at a clock frequency of 130 MHz. The design was further optimized for low power and consumes 2.421 mW. The optimization is based on the reuse of the permutation function in combination with extensive pipelining. In terms of energy, the encryption operation costs 1.16 pJ/bit.**

I. Introduction

As 5G paves the way for wireless communication at an extremely high speed, demand for transferring information securely through encrypted channels is growing at a fast pace. Also, the ability to authenticate the received data is of paramount importance to both parties. In order to meet these security objectives in applications subject to strict power constraints, novel algorithms have been proposed that securely integrate confidentiality and authentication in an efficient manner.

Authenticated encryption combines both confidentiality and authentication into a single algorithm. The encryption module acts on the plaintext P. Next to encrypting the plaintext, the module synthesizes a tag T_1. The tag is derived from P and allows the decryption module to verify the authenticity of the encrypted message. The decryption module acts on the ciphertext C and the tag T_1. During the decryption operation, the module generates its own tag T_2. In case T_2 differs from T_1, the module acknowledges that the received message must have been altered and refrains from returning the plaintext. Authenticated encryption has as advantage that the encryption and authentication operation can share part of their functionality, thereby reducing the computational complexity. Also, since encryption and authentication are combined, there is no need for an additional interface between both entities in the hardware design.

For the adoption of new authenticated encryption ciphers, the cryptographic community follows the tradition of focused, open competitions (cf. AES and SHA-3). The CAESAR competition [2], launched in 2013, aims to find a novel authenticated encryption scheme that performs well in terms of software and hardware efficiency, security, applicability and robustness.

KETJE [4] is a third round CAESAR candidate. This sponge-based cipher builds on top of the KECCAK permutation [3], which is standardized by NIST as SHA-3. The KETJE submission paper suggests several instantiations, providing the user with different security strengths. This article covers the implementation of KETJESR, a lightweight solution that achieves long-term security with a 128-bit security level. We target a high throughput implementation, while holding on to a design that is energy efficient. Our realization is based on the reuse of the permutation function in combination with extensive pipelining.

Our work resulted in the first ASIC implementation of KETJESR. The following section contains the necessary background information on KETJESR. The third section describes the hardware architecture. The final section concludes our work with a discussion on the experimental results.

II. Background on KetjeSR

This section contains a bottom-up description of KetjeSR and identifies the different modules of the design.

A. The KECCAK-*p** permutation

KETJESR relies on the KECCAK-*p* permutation* [3] for modifying a state of $b = 400$ bits, from which the ciphertext/plaintext and tags are derived. The KECCAK-p* permutation in itself is found from repeatedly applying a round function. The hardware design is round-based and implements the round function as a combinational block. It takes the state and a round constant as input. The round constant is defined by the standard and depends on the index of the round that is executed.

B. MonkeyDuplex

The *MonkeyDuplex* construction, as defined in Algorithm 1, specifies three operations - $MD.Start()$, $MD.Step()$, $MD.Stride()$ - which build on the KECCAK-p* permutations. The main differences between the operations are the number of rounds in the permutation and the possibility of having an additional padding operation. In Algorithm 1, the function $f\,(nr, s)$ denotes the application of KECCAK-p* with nr rounds on the state s.

$MD.Start()$ takes an initial package I, containing a padded version of the key and nonce, and applies n_{start} iterations of the round function. The result initializes the state register, which is manipulated in subsequent $MD.Step()$ and

$MD.Stride()$ operations to obtain the ciphertext and authentication tag.

The second instruction, $MD.Step()$, absorbs the associated data and plaintext, which are provided in blocks of size r at the σ data input. The instruction updates the state register by applying the round function n_{step} times and outputs an l-bit truncated part of the state.

The third instruction, $MD.Stride()$, initiates the squeeze of the authentication tag. It serves to increase the difficulty of forgery ($n_{stride} > n_{step}$).

KETJESR considers the following numbers:

$$n_{start} : n_{step} : n_{stride} = 12 : 1 : 6$$

.

Algorithm 1 The MonkeyDuplex construction

Require: $2 < r < b$, $n_{step} < n_{stride}$ and a state $s \in \mathbb{Z}_2^b$

Start

MD.start(I) with $I \in \mathbb{Z}_2^{\leq b-2}$

$s = Padding\,(b, |I|)$

$s = f\,(n_{start}, s)$

Step

Z = MD.step(σ, l) with $l \leq r, \sigma \in \mathbb{Z}_2^{\leq r-2}, Z \in \mathbb{Z}_2^l$

$P = \sigma || Padding\,(r, |\sigma|)$

$s = s \oplus \big(P || 0^{b-r}\big)$

$s = f\,(n_{step}, s)$

return $\lfloor s \rfloor_l$

Stride

Z = MD.stride(σ, l) with $l \leq r, \sigma \in \mathbb{Z}_2^{\leq r-2}, Z \in \mathbb{Z}_2^l$

$P = \sigma || Padding\,(r, |\sigma|)$

$s = s \oplus \big(P || 0^{b-r}\big)$

$s = f\,(n_{stride}, s)$

return $\lfloor s \rfloor_l$

C. MonkeyWrap

The top level operation is named *MonkeyWrap*. The module is responsible for producing the ciphertext C and the authentication tag T, given associated data A and plaintext P. Similarly, *MonkeyUnwrap* acts on the ciphertext C and tag T, for which it constructs a second tag T' and plaintext P. For the algorithm, we refer to [4]. The plaintext is only valid when $T = T'$. From an implementation point of view, MonkeyWrap's packs the input blocks and manages the dataflow to the MonkeyDuplex construction.

Four stages can be identified in MonkeyWrap, described in algorithm 2. The first stage is the keypack operation which pads the concatenation of the secret key K and nonce N to a fixed length. The keypack is served as input to $MD.Start()$, in order to initialize the state registers. In the second stage, the associated data A is processed by means of the $MD.Step()$ operation. Consequently, the value of the state register will depend on A. In the third stage, the plaintext is absorbed and encrypted through the $MD.Step()$ operation. The final stage implements the $MD.Stride()$ operation. The remaining part of the authentication tag is synthesized by means of the $MD.Step()$ operation.

Algorithm 2 MonkeyWrap

Let the block size $\rho = 32$ for KETJESR.

Initialization

KeyInit(K, N) with key $K \in \mathbb{Z}_2^{\leq b-18}$, $|K| \mod 8 = 0$ and nonce $N \in \mathbb{Z}_2^{b-|K|-18}$

MD.start(keypack$(K, |K| + 16)\,||N)$

MonkeyWrap

(C, T) = MonkeyWrap (A, P) with A, P, $C \in \mathbb{Z}_2^\star$ and $l \geq 0, T \in \mathbb{Z}_2^l$:

for $i = 0$ to $||A|| - 2$ **do**

MD.step$(A_i||00, 0)$;

end for

$Z = MD.\text{step}\big(A_{||A||-1}||01, |P_0|\big)$

$C_0 = Z \oplus P_0$

for $i = 0$ to $||P|| - 2$ **do**

$Z = MD.\text{step}(P_i||11, |P_{i+1}|)$;

$C_{i+1} = P_{i+1} \oplus Z$

end for

$T = MD.\text{stride}\big(P_{||P||-1}||10, \rho\big)$

while $|T| < l$ **do**

$T = T||MD.\text{step}(0, \rho)$

end while

return (C, T)

III. OUR PROPOSED KETJESR PROCESSOR

A. Security recommendations

We recommend to use KETJESR with a key length of 128 bits. Based on [4] this provides the user with 128 bits confidentiality, which is recommended for long term protection [1]. For integrity, the security claim of 128-bit forgery resistance holds for the proposed key length in combination with a 128 bits tag. The size of the nonce determines the key agility (i.e. the number of executions with the same key). Hence, we propose a length of at least 64 bits, but this can be tailored towards the specific application.

B. Unrolling the rounds?

As with many symmetric-key cryptographic algorithms, KETJESR repeatedly applies a given round function. We compare between a round-based implementation and a six-fold unrolled implementation, where six is the greatest common divisor of n_{start} and n_{stride}. In the average case, the $MD.Step()$ calls dominate. Using the official test vectors from [4], we construct a benchmark based on a representative set of message lengths. We compared two designs featuring a different number of permutation blocks. Table I shows the synthesis and benchmarking results.

Reusing the round function results in the fastest execution time. In algorithm 2, both $MD.Start()$ and $MD.Stride()$ are called only once. Hence, their cycle cost can be amortized as long as the message is large enough. In the general case, $MD.Step()$ calls will dominate the execution, motivating a round-based implementation with a much shorter critical path. Furthermore, due to the sequential nature of the sponge construction, the subsequent applications of the round function

cannot be pipelined. Finally, having multiple permutations is clearly more expensive in terms of area. As a result, the implementation considers a round-based implementation, which is most efficient in terms of throughput, area and power.

TABLE I: Implementation strategy comparison

	Unrolled (6 rounds)	Round-based
Critical path	3.86 ns	1.42 ns
Cycles	110,901	127,471
Total time	428 µs	181 µs

C. MonkeyDuplex Engine

The MonkeyDuplex datapath is shown in Fig. 1. It contains one KECCAK-p* permutation function. Furthermore, the round constants for the next permutation are calculated parallel to the current permutation. In case the $MD.Start()$ operation is executed, the state is XORed with zeros. This is because only $MD.Step()$ and $MD.Stride()$ require to XOR the state with an input block. A MUX is used to select the appropriate input to the XOR, rather than performing an XOR with the input block before selecting the correct input to the permutation. In doing so, the load of the MUX is placed on the input register instead of the state register. Using the zero signal in the $MD.Start()$ operation reduces the load on a node in the critical path, which starts and ends in the state register.

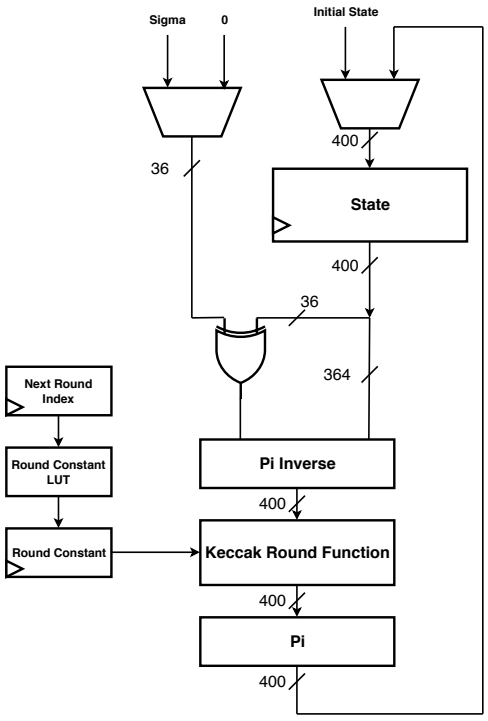

Fig. 1: MonkeyDuplex core

D. MonkeyWrap & MonkeyUnwrap Core

Because of their similarity, MonkeyWrap and MonkeyUnwrap are integrated into a single module. The MonkeyWrapUnwrap module comprises the MonkeyDuplex construction, the padding operation and an XOR operation. Fig. 2 shows the datapath for the integrated module. When performing MonkeyWrap, the XOR operation for the current plaintext block is executed parallel to padding the next plaintext block. The MonkeyUnwrap operation does not feature this parallelism due to a feedback loop, although data forwarding could mitigate this at the expense of tearing apart the modular design.

As Fig. 3 shows, the KETJESR module consists of three main parts. Note that this article describes a standalone encryption-decryption module. To limit the I/O ports and overhead, the implementation employs a 16 bit input/output interface. Given that KETJESR has a block size of 32 bits, the module is I/O-bound and can only output one full block every 2 cycles. When integrating this module in a larger system, this constraint should no longer hold. As a result, the speed of the implementation can easily be doubled. Future implementations can also avoid this behavior by turning to a larger I/O pad.

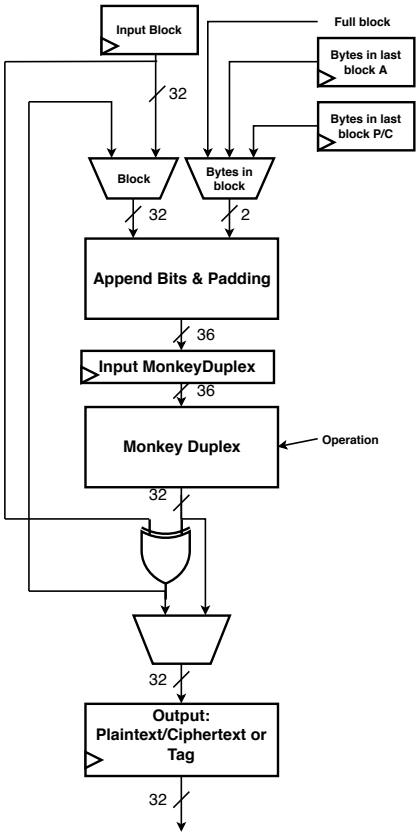

Fig. 2: MonkeyWrapUnwrap Control Flow

IV. IMPLEMENTATION & MEASUREMENT RESULT

The proposed design for the 128-bit authenticated encryption scheme KETJESR was fabricated in 90 nm GUTM CMOS technology. The circuit is able to perform KETJESR in the range of 10 MHz to 130 MHz at an operating voltage of 1.0 V. Fig. 4 depicts the crossover region of the Shmoo plot. The micrograph of the processor is shown in Fig. 5. The core module *MonkeyWrapUnwrap* comprises 12.2kGE, whereas the interface modules contain 4.8kGE in total.

To stimulate further research and allow for a fair comparison with this implementation, we adopt the performance metrics of

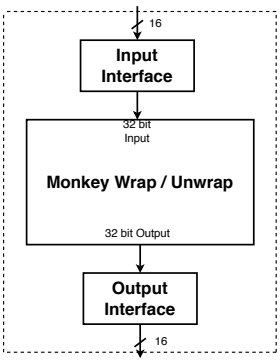

Fig. 3: KETJESR Interface Buffers

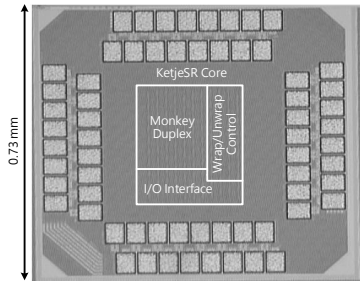

Fig. 5: Chip Micrograph of KETJESR Processor

[5]. We discriminate between short and long messages to explicitly account for the overhead of the one-time $MD.Start()$ and $MD.Stride()$ calls, as well as the synthesis of the tag. As a short message, we encrypt a 256-bit plaintext. Long messages can be chosen arbitrarily large. The tag length for both messages is 128 bit.

Table II summarizes the performance of the circuit. We do not consider the situation where the chip is idle. In this case, it is most efficient to power gate the circuit except for the internal state register.

TABLE II: Circuit performance

Operating Voltage [V]	1.0
Clock Freq [MHz]	130
Max TP - Encrypt [Gbps]	2.08
Max TP - Decrypt [Gbps]	1.39
Power [µW]	2421
Energy/bit (Short) $\left[\mathrm{pJ\,bit^{-1}}\right]$	2.69
Energy/bit (Long) $\left[\mathrm{pJ\,bit^{-1}}\right]$	1.16
HW Efficiency (TP/Gate) [Kbps/GE]	170

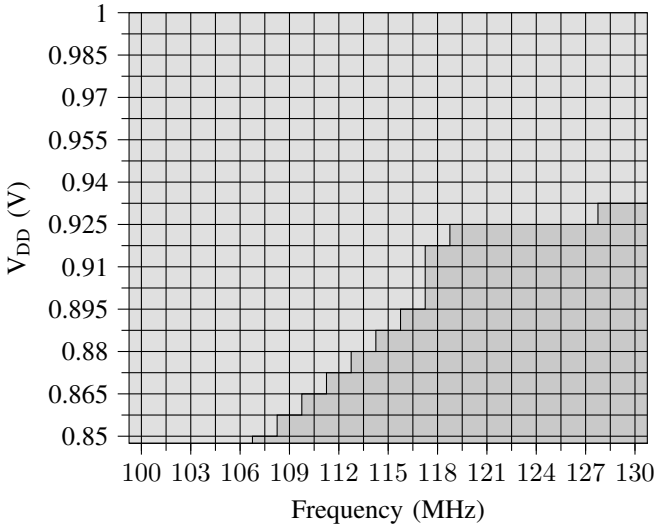

Fig. 4: Shmoo plot showing the pass/fail domain.

V. CONCLUSION & FUTURE WORK

This paper presented the first ASIC implementation of the lightweight authenticated encryption scheme KETJESR. Both encryption and decryption are integrated in a single module. An implementation in 90 nm resulted in an energy efficiency of $1.16\,\mathrm{pJ\,bit^{-1}}$ and 2.08 Gbps throughput. We report different performance metrics to provide reference value for future implementations. In future research, other trade-offs for KETJE can be explored. For instance, the MonkeyWrap module can be further pipelined, given the knowledge that $MD.Step()$ only takes one cycle. As explained in section III-D, the throughput of the decryption operation can be doubled as well.

ACKNOWLEDGMENT

This work was funded by MOST and MOE, Taiwan, under grants MOST 105-301-E-009004 and MOE 107W210. The authors thank TSMC for the chip fabrication and CIC for providing measurement facilities.

REFERENCES

[1] *Algorithms, Key Size and Protocols Report (2018)*. Tech. rep. ECRYPT-CSA, 2018.

[2] Daniel J Bernstein. *Caesar: Competition for authenticated encryption: Security, applicability, and robustness*. 2014.

[3] Guido Bertoni et al. "Keccak sponge function family main document". In: *Submission to NIST (Round 2)* 3.30 (2009).

[4] G Bertoni et al. *CAESAR submission: Ketje v2. online at http://ketje. noekeon. org/Ketje-1.1. pdf*. 2014.

[5] Miroslav Knezevic et al. "Fair and consistent hardware evaluation of fourteen round two SHA-3 candidates". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.5 (2012), pp. 827–840.