On the Implementation Aspects of Sponge-Based Authenticated Encryption for Pervasive Devices

Tolga Yalçın and Elif Bilge Kavun

Horst Görtz Institute for IT-Security,
Ruhr-Universität Bochum, Germany
{tolga.yalcin,elif.kavun}@rub.de

Abstract. Widespread use of pervasive devices has resulted in security problems which can not be solved by conventional algorithms and approaches. These devices are not only extremely resource-constrained, but most of them also require high performance – with respect to available resources – in terms of security, speed and latency. Especially for authenticated encryption, such performance can not be achieved with a standard encryption-hash algorithm pair or even a "block cipher mode of operation" approach. New ideas such as permutation-based authenticated encryption have to be explored. This scheme has been made possible by the introduction of sponge functions. Implementation feasibility of such an approach has yet to be explored. In this study, we make such an attempt by implementing the new SpongeWrap authenticated encryption schemes on all existing sponge functions and show that it is possible to realize a low-latency scheme in less than $6K$ gate equivalents at a throughput of 5 Gbps with a 128-bit claimed security level.

Keywords: Pervasive computing, data security, authenticated encryption, sponge functions, Keccak, Photon, Quark, Spongent.

1 Introduction

Pervasive computing is everywhere. We have, for quite sometime, got used to the idea of using all sorts of computing devices almost in every moment of our lives. These devices range from smart phones to smart cards with varying levels of computing power and usability [1]. A smart card on an ATM card may be used only as a means to draw cash from an ATM. On the other hand, a smart phone may be used to perform complex image processing in unprecedented speeds compared to the most powerful desktop computers of a decade ago. While more pervasive devices are introduced for new applications, these devices themselves also introduce more and more new application areas autonomously. However, it is not just new application areas they introduce, but unforeseen problems as well. Among these problems, security, perhaps, is the most important one, not just from an engineering point of view, but also from the user perspective.

Security of data has to be guaranteed in various levels: During computation from outside observers (also known as side-channel attackers [2]), during communication from third parties, and during storage from unauthorized users. Looking

S. Mangard (Ed.): CARDIS 2012, LNCS 7771, pp. 141–157, 2013.

at the specific example of smart cards, one may consider them to be the most pervasive computing platform. We carry several cards in our wallets for various applications such as banking, identification, access control, mobile phones, loyalty schemes [3]. All of these applications rely on personal and sometimes even critical data of the user (health information, pin codes, biometric information, etc.), which is not only a security nightmare to the user but even worse for the designers of such systems.

Data stored on such systems has to be secured via an encryption algorithm. There are several algorithms and standards designed and extensively analyzed for this purpose. The internationally accepted Advanced Encryption Standard, AES, is perhaps the most widely used encryption algorithm. It is very well-analyzed, tested, and proved to be secure – not just for today, but for the coming decades as well. Moreover, it has been implemented on countless number of platforms for almost all imaginable performance targets. While a high performance version of AES can process 55.5 Gbps of data on tens of thousands of ASIC gates [4], a lightweight version of it can fit into only about 2 thousands gates at reduced performance of a few tens of Kbps.

However, encryption alone is not sufficient. Another important operation that has to be performed on the secure data is authentication. As in the case of encryption, authentication is also very well-established and standardized as Secure Hash Algorithms, SHA-1 and SHA-2 by NIST. Competition for a third standard, SHA-3, is underway.

Authenticated encryption is a technique, which combines both authentication and encryption in order to provide confidentiality, integrity and authenticity of the data, simultaneously. While, the same functionality can be achieved via an encryption algorithm and an authentication algorithm running in parallel, it is not always the preferred solution, especially on resource-limited devices. Therefore, authenticated encryption is introduced as a block cipher mode of operation, where the same cipher block performs both functionalities. The most commonly used (also standardized) modes are CCM, CWC, OCB, EAX and GCM.

More recently, use of sponge-based hash functions as authenticated encryption primitives has been proposed [5]. With its arbitrarily long input and output sizes, the sponge construction allows building various cryptographic primitives such as a hash function, a stream cipher or a MAC [6], which, if properly combined, can lead to a "Do-It-All-Cipher". Although, sponge functions are still quite young, already a few number of sponge-based hash functions have been introduced. Among these, PHOTON, QUARK and SPONGENT are mostly targeted for lightweight applications, which KECCAK, a SHA-3 finalist, though not specifically designed so, can be tweaked to operate in lightweight mode. Furthermore, SPONGEWRAP modes of KECCAK and QUARK have also been presented to be used as authenticated encryption primitives [7]. Especially, the MONKEYDUPLEX and DONKEYSPONGE constructions show a lot of promise to be used in lightweight applications.

With its single round streaming mode, DonkeySponge construction can be effectively used in data storage applications as a low-latency cipher at the penalty of using a nonce in addition to the key, while MonkeyDuplex can perform the same functionality without nonce at the expense of more rounds per encryption. Even in that case, low-latency can be achieved by means of an unfolded design, in a similar fashion as presented in [8]. But there are open questions: From a mathematical point of view, these modes are yet to be extensively studied. The number of rounds per each sponge function in the proposed modes have to be determined together with their security claims, as done for Keccak. On the other hand, from a hardware point of view, efficiency of each sponge function in the target modes have to be determined. A new sponge-based proposal is only acceptable if it offers more (or less – in terms of gate count and power consumption) than the existing block cipher based systems.

In our study, we try to bring some answers, or more literally performance figures to the second question. We implement both DonkeySponge and MonkeyDuplex constructions on the existing sponge functions – Keccak, Photon, Quark and Spongent. In our implementations, we target low-latency data encryption, which is a realistic design target for pervasive applications, especially for data storage security. Furthermore, we choose data wordlength of 32 bits, which also is a realistic figure, considering data storage solutions on pervasive devices. We then select variants of sponge functions that can provide this data rate. These are Keccak-200, Photon-196, Quark-176 and Spongent-176, all of which provide around 80 bits of generic security with a target key length of 128 bits. Since the round numbers for MonkeyDuplex and DonkeySponge duplex mode are given only for Keccak, we obtain round numbers for other sponge functions by simple proportioning (i.e. with respect to the original proposed round numbers). In all fairness, we present the performance figures, for both these proportional round numbers and ones identical to those of Keccak.

The rest of the paper is organized as follows. In the next section, we give a brief introduction about sponge functions, which also includes specific details of each of the target sponge functions. It is followed by MonkeyDuplex and DonkeySponge constructions. In the following section, we present our implementations for both constructions together with performance figures on all hash functions. In the last section, we summarize our results and propose future directions for research.

2 Sponge Functions

Sponge functions can be used to generalize cryptographic hash functions to more general functions with arbitrary output lengths. They are based on the sponge construction, which is a repetitive construction to build a function F with variable-length input and arbitrary-length output based on a fixed-length permutation f operating on a fixed number of b bits, which is called the width. The sponge construction operates on a state of $b = r + c$ bits. r is called the

bit rate and c is called the capacity. In the first step, the bits of the state are all initialized to zero. Then, the input message is padded and cut into blocks of r-bit. The construction consists of two phases, namely the absorbing phase and the squeezing phase.

- In the absorbing phase, the r-bit input message blocks are XORed with the first r-bit of the state, then interleaved with the function f. After processing all of the message blocks, the squeezing phase begins.
- In the squeezing phase, the first r-bit of the state is returned as output blocks, and then interleaved with the function f. The number of output blocks is chosen by the user. The block diagram of the sponge construction is shown in Figure 1.

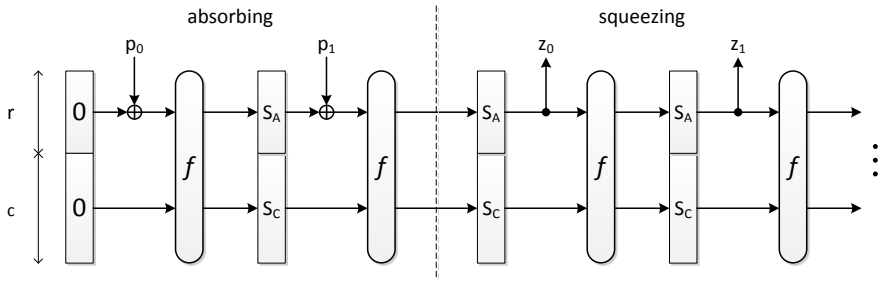

Fig. 1. Sponge construction

The existing sponge functions are summarized in the following subsections.

2.1 Keccak Sponge Function

Keccak [9] is a cryptographic hash function submitted to the NIST SHA-3 hash function competition. It is a family of hash functions based on the sponge construction that is used as a building block of a permutation from a set of seven permutations denoted by Keccak-$f[b]$, where $b \in 25, 50, 100, 200, 400, 800, 1600$ is the width of the permutation. The width b of the permutation is also the width of the state in the sponge construction. The state is organized as an array of 5×5 lanes, each of length w bits, where $w \in 1, 2, 4, 8, 16, 32, 64$ ($b = 25w$). Depending on the selected permutation width, the Keccak-f permutation consists of a number of simple rounds with logical operations and bit permutations. The number of rounds n_r depends on the permutation width which is calculated by $n_r = 12 + 2l$, where $2^l = w$. This yields 12, 14, 16, 18, 20, 22, 24 rounds for Keccak-$f[25]$, Keccak-$f[50]$, Keccak-$f[100]$, Keccak-$f[200]$, Keccak-$f[400]$, Keccak-$f[800]$, Keccak-$f[1600]$, respectively. The Keccak-$[r, c, d]$ sponge function can be obtained by applying the sponge construction to Keccak-$f[r+c]$ with the parameters capacity c, bit rate r and diversifier d and also padding the message input specifically.

The KECCAK iterated round function is explained in Algorithm 1, where all of the operations on the indices are done in *modulo* 5. A denotes the complete permutation state array, and $A[x, y]$ denotes a particular lane in that state. $B[x, y]$, $C[x]$, $D[x]$ are intermediate variables, the constants $r[x, y]$ are the rotation offsets and $RC[i]$ are the round constants. $ROT(w, r)$ is the bitwise cyclic shift operation which moves the bit from position i into position $i + r$, in the modulo lane size.

Algorithm 1. Pseudo-code of KECCAK-f

KECCAK-$f[b](A)$

- $for\ i\ in\ 0 \dots n_r - 1$
  $A = Round[b](A, RC[i])$
- $return\ A$

$Round[b](A, RC)$

- θ step:
  $C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4], \forall x\ in\ 0 \dots 4$
  $D[x] = C[x - 1] \oplus ROT(C[x + 1], 1), \forall x\ in\ 0 \dots 4$
  $A[x, y] = A[x, y] \oplus D[x], \forall (x, y)\ in\ (0 \dots 4, 0 \dots 4)$
- ρ and π steps:
  $B[y, 2x + 3y] = ROT(A[x, y], r[x, y]), \forall (x, y)\ in\ (0 \dots 4, 0 \dots 4)$
- χ step:
  $A[x, y] = B[x, y] \oplus ((NOTB[x + 1, y]ANDB[x + 2, y], \forall (x, y)\ in\ (0 \dots 4, 0 \dots 4)$
- χ step:
  $A[0, 0] = A[0, 0] \oplus RC$
- $return\ A$

2.2 PHOTON Sponge Function

PHOTON [10] is a sponge construction with an AES-like permutation. The internal state size $t = (c + r)$ depends on the hash output size and can take five distinct values. Therefore, internal permutation P_t is defined for each internal state size. PHOTON starts with the initialization phase where the message is padded and cut into blocks of t bits. Then the t-bit state is processed by P_t permutation in absorption phase. Finally, in the squeezing phase, the n-bit hash value is returned.

The internal permutation P_t is an N_r-round transform of the t-bit state. Note that the state organization is defined according to the hash output size. However, the number of rounds is the same for all t values and the round function is iterated as the number of rounds. The round function is similar to AES round function as shown in Algorithm 2. It starts with an *AddConstants* step instead of the key

addition of AES. Here, round constants and internal constants are XORed to the state. Round constants are defined for each round and the internal constants depend on the state organization. It is followed by *SubCells* and *ShiftRows* steps. In the substitution layer, the PRESENT Sbox is used in the case of 4-bit cells and the AES Sbox is used in the case of 8-bit cells. In *MixColumnsSerial* step, the final mixing layer is applied to each of the columns of the internal state. The coefficients and the irreducible polynomials used in this step again depend on the permutation type.

Algorithm 2. Pseudo-code of PHOTON

- $for\ i = 1\ to\ R\ do$
  $State \leftarrow AddConstant(State)$
  $State \leftarrow SubCells(State)$
  $State \leftarrow ShiftRows(State)$
  $State \leftarrow MixColumnsSerial(State)$
- $end\ for$

2.3 QUARK Sponge Function

QUARK [11] uses the sponge construction and a b-bit permutation P. Three different instances of QUARK are specified. Each instance is parameterized by a rate r, capacity c, and hash length n. The size of the internal state is b bits ($b = r + c$). The QUARK sponge construction processes a message in three steps: Initialization, absorption and squeezing. As in the case of KECCAK, the message is first padded and cut into r-bit blocks. These blocks are then XORed with the last r bits of the state and interleaved with permutation (P) applications. In the end, the last r bits of the state are returned as output, interleaved with permutation applications, until n bits are returned.

The permutation P is inspired by the stream cipher Grain [12] and the block cipher KATAN [13] (see Figure 2). The internal state of P is viewed as three feedback shift registers – two nonlinear and one linear. P proceeds in three stages for a given b-bit input: Initialization of the internal state, status update with f, g, and h functions, and finally the computation of the output similar to initialization. Note that functions f, g, and h are defined separately for each instance.

2.4 SPONGENT Sponge Function

SPONGENT [14] is a sponge construction based on a wide PRESENT-type [15] permutation. SPONGENT produces an n-bit hash value for a given finite number of input bits – it is a simple iterated design that takes a variable-length input and can produce an output of an arbitrary length based on a permutation π_b

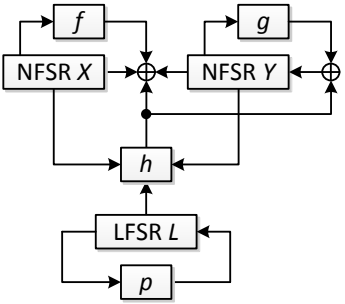

Fig. 2. Permutation of QUARK

operating on a state of b bits (where $b = r + c \geq n$, r rate and c capacity). Hashing starts with an initialization phase where the message is padded and cut into blocks of r bits. In the following phase (absorption), the r-bit message blocks are XORed with the first r bits of the state and then processed in π_b permutation. Finally, in the squeezing phase, the first r bits of the state are returned as output, interleaved with applications of π_b, until n bits are returned.

The permutation π_b is an R-round transform of the b-bit state. The round function is iterated as the number of rounds (R), which depends on the SPONGENT variant used. It is similar to the PRESENT round function, but a wider version. Also, instead of key addition, a counter value depending on an LFSR is added. The substitution and permutation layers are the same; however, they are defined for larger states. Algorithm 3 shows the π_b permutation.

Algorithm 3. Pseudo-code of SPONGENT

- *for* $i = 1$ *to* R *do*
  $State \leftarrow lCounter_{reversed\ bit-order} \oplus State \oplus lCounter$
  $State \leftarrow sBoxLayer(State)$
  $State \leftarrow pLayer(State)$
- *end for*

3 Permutation-Based Authenticated Encryption

Mainstream symmetric cryptography has been dominated by block ciphers, which offer inverse function capability. However, an inverse cipher is only needed in specific modes such as electronic codebook (ECB), cipher block chaining (CBC) and offset codebook mode (OCB) in authenticated encryption. There are several modes of operation where the cipher block is used in forward (encryption) mode only. From a designer's point of view, an n-bit block cipher is nothing but a

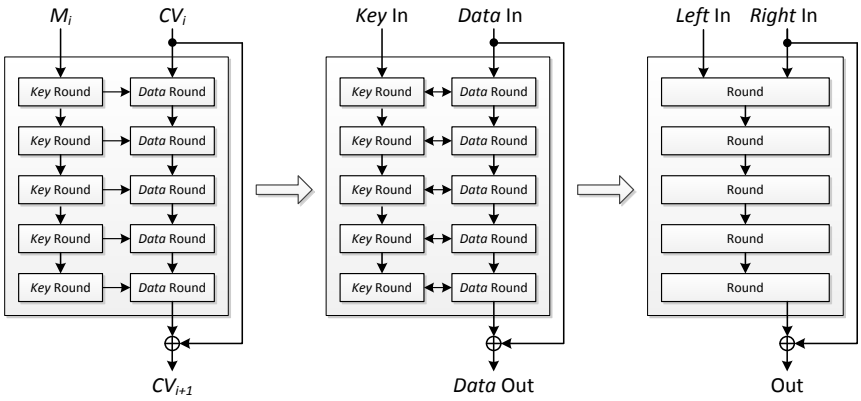

Fig. 3. Evolution from Davies-Meyer construction (left) to iterated permutation mode (right)

b-bit permutation ($b = n + |K|$, $|K|$ being the key size) with no diffusion from data state to key state. A simple example of use of a block cipher in this mode is the hashing via Davies-Meyer compression function. Since for hashing there is no need to limit diffusion, one can use a block cipher in iterated permutation mode, where the internal state is composed of both the left and right states as shown in Figure 3.

This construction vanishes the need for separate key schedule and replaces the n-bit block cipher by a b-bit permutation. This is, in fact, a block cipher without inverse, which has the capability to perform not only encryption but also message authentication – or simply, authenticated encryption (AE). In the following subsections, we will see how sponge functions can be used to perform resource-efficient and secure permutation-based authenticated encryption.

3.1 Authenticated Encryption Mode SpongeWrap

SpongeWrap [5] construction realizes authenticated encryption as shown in Figure 4. Upon initialization, key, K, is loaded into the state. Next, padded header A (also referred to as additional authentication data – AAD) is *absorbed* into the state. This is followed by the encryption (or decryption) phase, which is run in duplex mode, i.e. for each input data block (plaintext or ciphertext), an output data block (ciphertext or plaintext, respectively) is generated. The output (in case of encryption) or input (in case of decryption) ciphertext is also *absorbed* into the state, thereby running hashing in parallel with encryption. Upon completion of processing of all input data blocks, the sponge is run (with zero input data) until all the l-bit tag, T is *squeezed* from internal the state. In decryption mode, the *squeezed* tag is compared with the received tag in order to check if the received tag is valid. In SpongeWrap mode, every key, header and plain/cipher-text block is extended with a so-called *frame bit*.

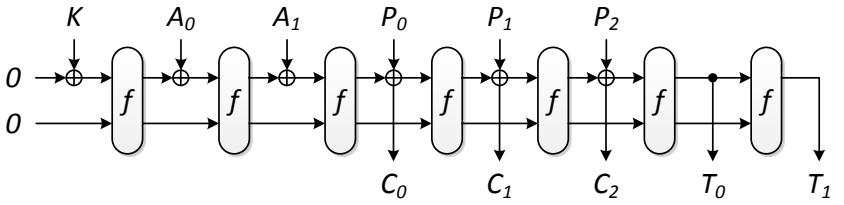

Fig. 4. SpongeWrap authenticated encryption

It is proven that the sponge and duplex constructions are secure against generic attacks with complexity below $2^{c/2}$ [16]. However, when a sponge function or duplex object is used in conjunction with a key, more refined bounds can be defined taking into account the data complexity. If the data complexity is limited to 2^a r-bit blocks, the keyed mode withstands generic attacks with time complexity up to 2^{c-a} calls of the underlying permutation. If $a < c/2$, this results in an increase of the security strength from $c/2$ to $c - a$. It should be noted that when the memory requirements of a pervasive system are considered, a is usually limited to 32 or less, which is in perfect agreement with the requirement of $a < c/2$.

3.2 DonkeySponge Construction

DonkeySponge mode of operation [17] (shown in Figure 5) can be summarized as follows:

- The b-bit state is initialized with the key and run through the round function, f, n_{init} time, resulting in the secret state. The number of rounds, n_{init} must be chosen such that all bits of the secret state depend on the MAC key.
- The b-bit blocks of the message are XORed into the secret state, interleaved with n_{absorb}-round permutations. The number n_{absorb} must be chosen to make the success probability of generating inner collisions negligible.

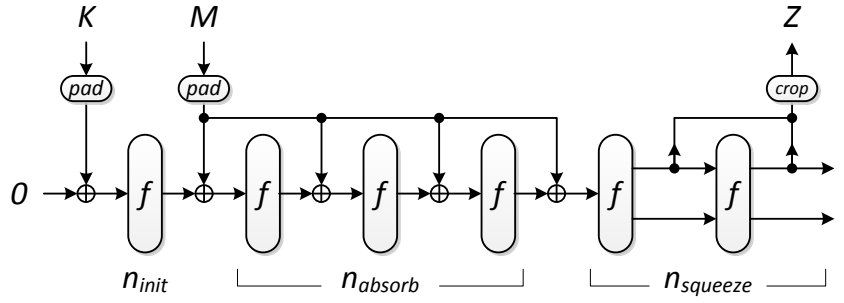

Fig. 5. DonkeySponge construction

– The tag is obtained by applying an $n_{squeeze}$-round permutation to the secret state and truncating the result to l bits. The number of rounds $n_{squeeze}$ should be high enough to prevent an adversary in reconstructing the inner state from outputs observed for chosen inputs. The number $b - l$ must be large enough to prevent state reconstruction by exhaustive search, namely, $b - l \geq k$.

In [17], $n_{init} = 3$, $n_{absorb} = 6$ and $n_{squeeze} = 12$ are chosen for KECCAK-f[200]. This choice of parameters are mainly influenced by propagation experiments. Therefore, a realistic selection of these parameters for all other sponge functions can only be possible after similar experiments and/or analyses.

3.3 MonkeyDuplex Construction

MonkeyDuplex mode of operation [17] (shown in Figure 6) is a modified version of the authenticated encryption with associated data (AEAD) mode SpongeWrap based on the duplex construction in [5]. The original version can guarantee confidentiality if for the same key and different messages the associated data is unique. In other words, the associated data should behave as a nonce.

MonkeyDuplex mode removes this restriction by using a unique nonce, which makes it more fragile. However, it results in a considerable security gain. Furthermore, it allows data encryption in stream mode with $n_{duplex} = 1$, resulting in extremely high rates.

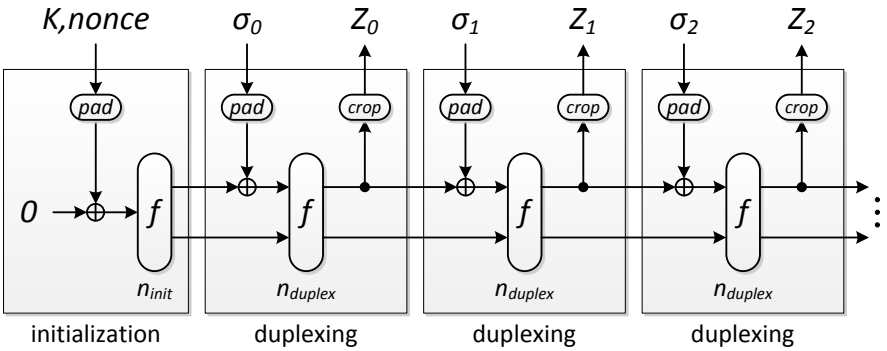

Fig. 6. MonkeyDuplex construction

4 Implementation Aspects

In this section, we summarize our implementation of the DonkeySponge and MonkeyDuplex cores on which we evaluate our performance figures. We start

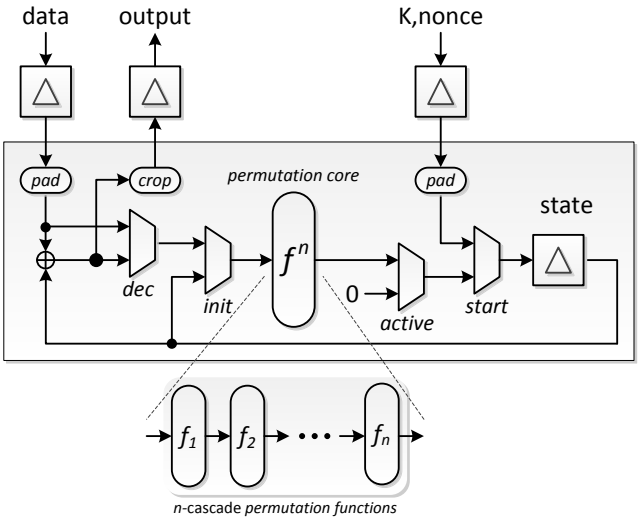

Fig. 7. DonkeySponge and MonkeyDuplex wrapper

by building a generic wrapper, into which any permutation function of a sponge, can be embedded. As shown in Figure 7, the wrapper is build up of a key register, data register, length registers (for header/AAD and data), and a state register. In agreement with our initial design target, data register is a 32-bit register, and the key register is a 192-bit register designed to collect 128-bit key and 64-bit nonce read in 32-bit blocks from input. The length registers are also 32-bit registers in order to support header and input data of up to $2^{32} \times 32$ bits each. The size of the state register depends on the state size of the chosen sponge (permutation) function. The core module is the sponge permutation function, f, module. Its input is determined depending on the phase of authenticated encryption – key/nonce absorption (initialization), header absorption, duplex, tag extraction.

The wrapper has two variants for DonkeySponge and MonkeyDuplex modes, respectively. The main differences between the two variants are the padding circuitry (which has negligible effect on the gate count) and the input data size. The DonkeySponge wrapper is designed to support input data width of up to 64 bits. However, in our designs, we run it in 32-bit mode. Both wrappers are fully functional blocks including control circuitry, and their functionalities have been verified by Modelsim SE v6.5b.

The operation of the authenticated encryption wrappers can be summarized in five phases:

– **Length/Key/Nonce Loading Phase.** Upon reception of a start command, the wrapper requests for initialization data – lengths, key, and optionally nonce – from the input interface. It then loads the corresponding

registers with the received data, and instructs the internal sponge core to start operation.
- **Initialization.** This is the phase where key/nonce is hashed to generate the secret internal state of the sponge. This phase takes n_{init} rounds.
- **Header Absorption.** In this phase, the wrapper requests header data from the input interface. Each received data word is padded with and loaded into the data register, whose output is then processed by the internal sponge core (absorbed into the state). In case, the header length is zero, this phase is bypassed.
- **Data Absorption.** In this phase, the wrapper requests plaintext/ciphertext data from the input interface. Again, each received data word is padded with and loaded into the data register, whose output is then processed by the internal sponge core (absorbed into the state). Additionally the leading b bits of the new state is XORed with the input data word to generate the corresponding ciphertext/plaintext, respectively. This mode of operation is known as the duplex mode. The output word is written into the output register which flags the readiness of the output with an "output enable" flag. Processing of each word in this phase and the previous phase takes n_{absorb} rounds, which is realized in a single clock cycle (via unfolding) in order to guarantee low-latency operation.
- **Tag Extraction.** This is the final phase, where the sponge core is run with zero input data in order to generate l-bit tag (where l can be 64 to 128 bits wide). Depending on the target tag length, the internal sponge core can be run several times, squeezing 32 bits of tag at every run. In DonkeySponge wrapper, each run takes $n_{squeeze}$ rounds, where number of clock cycles is determined by the ratio of $n_{squeeze}$ to n_{absorb}. For the MonkeyDuplex, the core is run in duplex mode with zero input data; therefore $n_{squeeze} = n_{absorb}$. As in the data absorption phase, the output tag words are loaded into the output register and signaled with an "output enable" flag.

In the implementation of f-module, we refer to our initially set design targets. We want to primarily achieve low-latency encryption of 32-bit words with (about) 80-bit generic security. Therefore, the rate, r, of the sponge function should be at least $(32 + 2)$ bits (including the padding); and the capacity, c, around 160 bits, resulting in total minimum state size of 194 bits. With these parameters in mind, we choose Keccak-200, Photon-196, Quark-176 and Spongent-176 variants, with the parameters summarized in Table 1. Although not all the variants provide equal design parameters, we have to make a choice in order to limit our design space. However, since we provide the performance figures for different unfolding options, it is still possible to make realistic guesses with the chosen variants.

4.1 Performance Comparison

For the performance evaluation, we run our syntheses using Cadence RTL Synthesizer v08.10-s222 with Nangate 45 nm generic and UMC 90 nm low-leakage

Table 1. Parameters for the chosen sponge functions

Function	b (bits)	c (bits)	r (bits)	Security (bits) preimage $(c-r)$	2nd preimage $(c/2)$	collision $(c/2)$	Generic security (bits) (keyed mode, $a=32$) $(c-a)$
KECCAK-200	200	164	36	128	82	82	132
PHOTON-196	196	160	36	124	80	80	128
QUARK-176	176	160	16	144	80	80	128
SPONGENT-176	176	160	16	144	80	80	128

cell libraries. The gate counts are provided for both constrained and unconstrained syntheses, while power figures are only provided for 50 MHz constrained syntheses.

Figure 8 shows the area comparison for 50 MHz constrained (left) and unconstrained (right) DONKEYSPONGE wrapper designs, respectively. n in the figures corresponds to the number of unfolded levels of permutations within the f-block. In the case of KECCAK, this corresponds to n_{absorb}. However, for all other sponge functions, n's are chosen arbitrarily in order to present illustrative numbers. For example, in the C-QUARK proposal [7] (a QUARK based SPONGEWRAP instance), n is chosen to be 64. As a reference value, gate counts for KECCAK are 13.6 KGE and 15.4 KGE for 90 nm and 45 nm libraries, respectively.

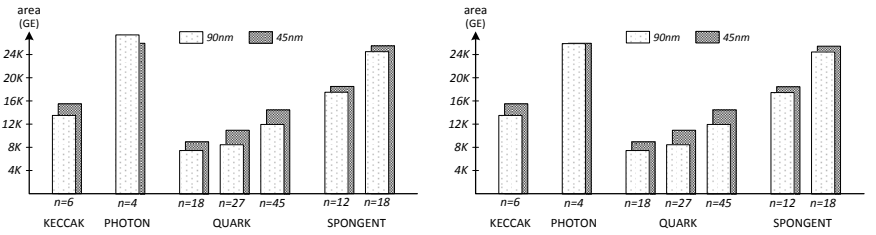

Fig. 8. Area comparison of DONKEYSPONGE wrappers for 50 MHz constrained (left) and unconstrained (right) cases

An important observation we can make from the figure is how close the gate counts are for constrained and unconstrained cases. This means timing targets are met even in the presence of several unfolded rounds, which is a major advantage of permutation based encryption schemes over conventional block ciphers.

Figure 9 shows the power comparison for 50 MHz constrained wrapper designs for the same values of n as in the area comparison. Power figures are provided only for 90 nm library. It should also be noted that these are synthesized power figures, and are *not* as much realistic as simulated figures. As a reference value, average power consumption for KECCAK is 13.6 mW.

Figure 10 shows the area comparison for 50 MHz constrained (left) and unconstrained (right) MONKEYDUPLEX wrapper designs, respectively. Again, n in the figures corresponds to the number of unfolded levels of permutations within

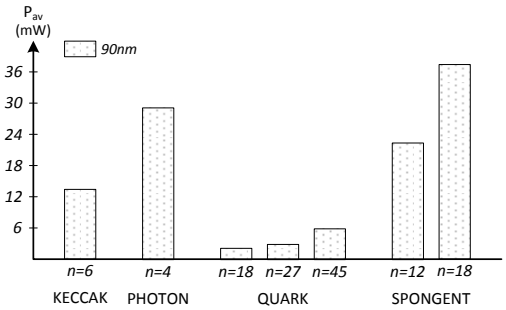

Fig. 9. Power comparison of DonkeySponge wrappers (50 MHz constrained)

the f-block. In the case of Keccak, this corresponds to $n_{duplex} = 1$. However, for all other sponge functions, n's are chosen arbitrarily in order to present illustrative numbers. As a reference value, gate counts for Keccak are 5.9 KGE and 7.4 KGE for 90 nm and 45 nm libraries (constrained), respectively.

In the case of MonkeyDuplex wrappers, Quark loses its area advantage coming from its simple internal structure, since registers dominate the overall area.

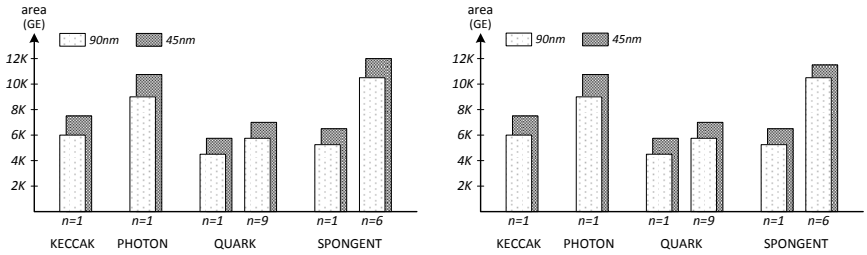

Fig. 10. Area comparison of MonkeyDuplex wrappers for 50 MHz constrained (left) and unconstrained (right) cases

Figure 11 shows the power comparison for 50 MHz constrained wrapper designs for the same values of n as in the area comparison. Again, power figures are provided only for 90 nm library, and they are synthesized power figures. As a reference value, average power consumption for Keccak is 2.1 mW.

We furthermore present the gate counts corresponding to different values of n for each design in separate graphs in Figure 12. For our synthesis ranges, gate counts do not vary in the existence of timing constraint for 45 nm designs, while it is not the case for other designs (except Quark). The gate counts are presented for only MonkeyDuplex constructions. The differences between DonkeySponge and MonkeyDuplex areas are negligible.

We have also checked the highest clock frequencies for all the designs in 90 nm (for $n = 1$). These are 154.2, 154.1, 83.4, and 154.1 MHz for Keccak, Quark,

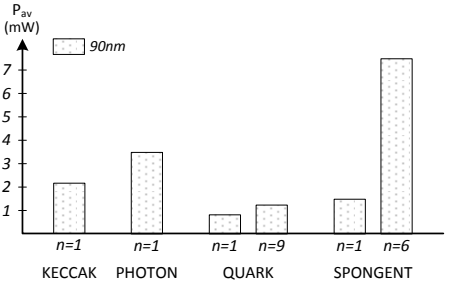

Fig. 11. Power comparison of MonkeyDuplex wrappers (50 MHz constrained)

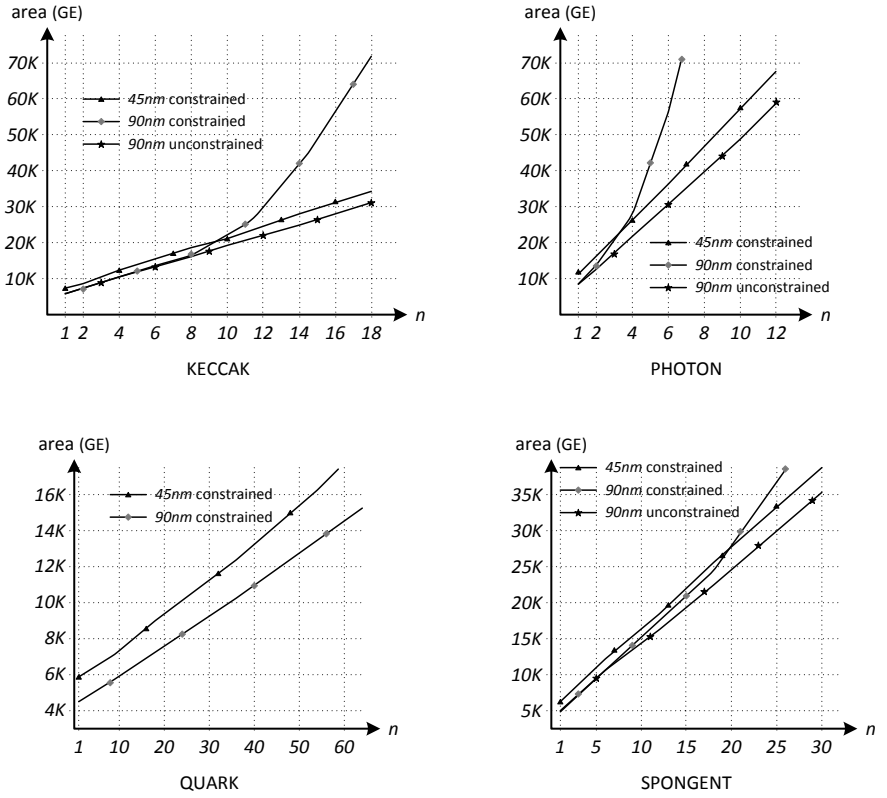

Fig. 12. Area figures for Keccak (upper left), Photon (upper right), Quark (lower left) and Spongent (lower right) MonkeyDuplex wrappers

Photon and Spongent, respectively. This means, except for Photon, sponge permutations do not determine the critical delay path. Instead, the wrapper around determines the overall speed of the design.

With a 32-bit datapath (as in the current implementations), MonkeyDuplex scheme can achieve authenticated encryption throughput of 4.9 Gbps at a gate count of 5.9 KGE, in other words almost 1 Mbps/GE. This number is much higher than any reported figure for existing block cipher based schemes.

5 Conclusion

In this study, we have implemented the newly proposed sponge-based DonkeySponge and MonkeyDuplex authenticated encryption schemes for all known sponge functions in the literature. We have demonstrated the simplicity and effectiveness of these schemes in terms of resource usage and latency, both of which are important design parameters for pervasive computing systems. Especially in data storage security, low-latency is of top priority. MonkeyDuplex scheme achieves low-latency at an incredibly high throughput of 4.9 Gbps within only 5.9 KGE area, with a 128-bit claimed security.

This is still a very new area of research. The security of these schemes have yet to be studied in deep. On the other hand, the performance results we have obtained shows the value of such research. As the next step, we will evaluate these ciphers with different technologies and parameters (using different variants of each sponge function). We will also provide simulated power figures, and compare our results with block cipher based authenticated encryption schemes, as well.

References

[1] Hansmann, U., Merk, L., Nicklous, M.S., Stober, T.: Pervasive Computing: The Mobile World. Springer (August 2003)
[2] Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security). Springer-Verlag New York, Inc. (2007)
[3] Allied Technique. Smart Cards (June 2012), http://www.alliedtechnique.com/smartcards/
[4] Soliman, M.I., Abozaid, G.Y.: FPGA Implementation and Performance Evaluation of a High Throughput Crypto Coprocessor. J. Parallel Distrib. Comput. 71(8), 1075–1084 (2011)
[5] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 320–337. Springer, Heidelberg (2012)
[6] Saarinen, M.-J.O., Engels, D.W.: A Do-It-All-Cipher for RFID: Design Requirements (Extended Abstract). IACR Cryptology ePrint Archive, 2012:317 (2012)
[7] Aumasson, J.-P., Knellwolf, S., Meier, W.: Heavy Quark for secure AEAD. In: DIAC - Directions in Authenticated Ciphers, Sweden, July 5-6 (2012)
[8] Ege, B., Kavun, E.B., Yalçın, T.: Memory Encryption for Smart Cards. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 199–216. Springer, Heidelberg (2011)
[9] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak Specifications (2009)

[10] Guo, J., Peyrin, T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 222–239. Springer, Heidelberg (2011)
[11] Aumasson, J.-P., Henzen, L., Meier, W., Naya-Plasencia, M.: QUARK: A Lightweight Hash. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 1–15. Springer, Heidelberg (2010)
[12] Hell, M., Johansson, T., Meier, W.: Grain: A Stream Cipher for Constrained Environments. Int. J. Wire. Mob. Comput. 2(1), 86–93 (2007)
[13] De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg (2009)
[14] Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varıcı, K., Verbauwhede, I.: SPONGENT: A Lightweight Hash Function. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 312–325. Springer, Heidelberg (2011)
[15] Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007)
[16] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
[17] Daemen, J.: Permutation-based Encryption, Authentication and Authenticated Encryption. In: DIAC - Directions in Authenticated Ciphers, Sweden, July 5-6 (2012)