Duplexing the sponge: single-pass authenticated encryption and other applications

Guido Bertoni[1], Joan Daemen[1], Michaël Peeters[2], and Gilles Van Assche[1]

[1] STMicroelectronics
[2] NXP Semiconductors

Abstract. This paper proposes a novel construction, called duplex, closely related to the sponge construction, that accepts message blocks to be hashed and—at no extra cost—provides digests on the input blocks received so far. It can be proven equivalent to a cascade of sponge functions and hence inherits its security against generic attacks. The main application proposed here is an authenticated encryption mode based on the duplex construction. This mode is readily usable in, e.g., key wrapping and is single-pass, namely, enciphering and authenticating requires only a single call to the underlying permutation per block. The duplex construction can be used to efficiently realize other primitives, such as a reseedable pseudorandom sequence generators and a sponge variant that overwrites part of the state with the input block rather than to XOR it in.

Keywords: sponge functions, duplex construction, authenticated encryption, key wrapping, provable security, PRNG, Keccak

1 Introduction

With its arbitrarily long input and output sizes, the sponge construction allows building various primitives such as a stream cipher or a hash function [8]. In the former, the input is short (typically the key and a nonce) while the output is as long as the message to encrypt. In contrast, the latter takes a message of any length at input and produces a digest of small length.

Some applications can take advantage of both a long input and a long output size. For instance, authenticated encryption combines the encryption of a message and the generation of a message authentication code (MAC) on it. It could be implemented with one sponge function call to generate a key stream (long output) for the encryption and another call to generate the MAC (long input). However, in this case, encryption and authentication are separate processes without any synergy.

The duplex construction is a novel way to use a fixed permutation (or transformation) to allow the alternation of input and output blocks at the same rate as the sponge construction, like a full-duplex communication. In fact, the duplex construction can be seen as a particular way to use the sponge construction, hence it inherits its security properties. By using the duplex construction, authentication encryption can be single-pass, i.e., only one call to the underlying permutation (or tranformation) is needed per message block. In a nutshell, the input blocks of the duplex are used to input the key and the message blocks, while the intermediate output blocks are used as key stream and the last one as a MAC.

Authenticated encryption (AE) has been extensively studied in the last ten years and many modes have been proposed, e.g., [2,23,27,38,35,4,28,32,39,36]. It is a popular way to provide simultaneously both integrity and condifentiality in a secure way. Note that all the modes cited above are based on a block cipher as underlying primitive, whereas our construction is the first one based on a permutation. An important efficiency parameter of an AE mode is the number of calls to the block cipher or to the permutation per block. While encryption or authentication alone can be single-pass, i.e., one call per block, some AE modes can remain single-pass. The duplex construction naturally provides a good basis for building a single-pass AE mode.

Authenticated encryption can also be used to transport secret keys in a confidential way and to ensure their integrity. This task, called key wrapping, is very important in key management and can be implemented with our construction if each key is associated to a unique identifier.

The duplex construction can be used for other applications as well, such as a reseedable pseudo-random number generator (PRNG). A mode based on sponge functions was proposed in [12], while in this paper

we revisit it to make direct use of the duplex construction. Finally, we can use the properties of the duplex construction to prove the security of an "overwrite" mode where the input block overwrites part of the state (instead of XORing it in).

The number of bits added by the padding used in the sponge function has an impact on the block size of the duplex construction. For this reason, we introduce a new padding function, which is as compact as possible and which allows one to deploy a set of sponge functions calling a common permutation f with different bitrates. Hence, the duplex construction inherits the flexibility of the sponge construction in terms of security/speed trade-offs.

1.1 Organization of the paper

The remainder of this paper is organized as follows. First, we propose a model for authenticated encryption in Section 2. Then in Section 3, we review the sponge construction. The core concept of this paper, namely the duplex construction, is defined in Section 4. Its use for authenticated encryption is given in Section 5 and for other applications in Section 6. A new compact padding scheme is given in Section 7, together with a proof that it is suitable for a set of sponge functions with different bitrates. Finally, Section 8 discusses the applicability of duplexing on other hash function constructions.

2 Modeling authenticated encryption

We consider authenticated encryption as a process that takes as input a key K, a data header A and a data body B and that returns a cryptogram C and a tag T. We denote this operation by the term *wrapping* and the operation of taking a data header A, a cryptogram C and a tag T and returning the data body B if the tag T is correct by the term *unwrapping*.

The cryptogram is the data body enciphered under the key K and the tag is a MAC computed under the key K over both header A and body B.

We assume the wrapping and unwrapping operations as such to be deterministic. Hence two inputs (A, B) and (A', B') that are equal will under the same key K give rise to the same output (C, T). If this is a problem, it can be tackled by expanding A with a counter or a random value.

2.1 Security requirements

For a key K chosen secretly and uniformly over $|K|$ bits, the security requirements for authenticated encryption are the following:

Key recovery infeasibility The success probability of finding the key in an attack with effort equivalent to trying N keys values is not above $N2^{-|K|}$.

Tag forgery infeasibility In the absence of key recovery, the success probability of tag forgery for any chosen (A, B) is $2^{-|T|}$, even for an adversary that is given the corresponding cryptogram C and is given the outputs (C_i, T_i) corresponding to any set of adaptively chosen inputs (A_i, B_i) with the only restriction that $(A_i, B_i) \neq (A, B)$.

Plaintext recovery infeasibility The most efficient method to gain information about B (excluding its length), given an output (C, T) corresponding to input (A, B) with chosen A but unknown B, is key recovery, even for an adversary that is given the outputs (C_i, T_i) corresponding to adaptively chosen inputs (A_i, B_i) with $A_i \neq A$.

Plaintext recovery infeasibility as defined above relies on the fact that there are no collisions in (K, A), namely, for a given K there are no two inputs with equal data header A and different data body B. Hence, it is up to the application to ensure that for a given key K, the data header A behaves as a nonce. Note that tag forgery does not rely on this.

2.2 An ideal system

We can define a reference system that satisfies these requirements using a pair of *random oracles* $(\mathcal{RO}_1, \mathcal{RO}_2)$. We use the definition of random oracle from [3]. A random oracle, denoted $\mathcal{RO}$, takes as input binary strings of any length and returns for each input a random infinite string, i.e., it is a map from $\mathbb{Z}_2^*$ to $\mathbb{Z}_2^\infty$, chosen by selecting each bit of $\mathcal{RO}(x)$ uniformly and independently, for every input.

Encryption and tag computation are now implemented as follows:

Encryption This is done by XORing B with a key stream. This key stream is the output of a random oracle $\mathcal{RO}_1$ to a string s_k computed from (K, A) with an injective encoding function: $s_k = s_k(K, A)$. If (K, A) is a nonce, key streams for different data inputs are the result of calls to $\mathcal{RO}_1$ with different input strings s_k and hence one key stream gives no information on another.

Tag computation The tag is the output of a random oracle $\mathcal{RO}_2$ to a string h_t computed from (K, A, B) with an injective encoding function: $h_t = h_t(K, A, B)$. Tags computed over different messages will be the result of calls to $\mathcal{RO}_2$ with a different input string.

Key stream sequences give no information on tags as they are obtained by calls to different random oracles. Additionally, as the key is only used as input to random oracles, the key recovery requirement is satisfied. Note that we can implement the two random oracles $\mathcal{RO}_1$ and $\mathcal{RO}_2$ above with a single $\mathcal{RO}$ using domain separation, for instance, $\mathcal{RO}_1(x) = \mathcal{RO}(x||0)$ and $\mathcal{RO}_2(x) = \mathcal{RO}(x||1)$.

The sponge construction has the same interface as a random oracle and hence one can build a practical system by replacing $\mathcal{RO}$ by a sponge function.

3 The sponge construction

The sponge construction [8] builds a function $\text{SPONGE}[f, \text{pad}, r]$ with variable-length input and arbitrary output length using a fixed-length permutation (or transformation) f, a padding function "pad" and a parameter *bitrate* r. A sponge function, that is, a function implementing the sponge construction, provides a particular way to generalize hash functions and has the same interface as a random oracle.

For the padding function we use the following notation: the padding of a message M to a sequence of x-bit blocks is denoted by $M||\text{pad}[x](|M|)$, where $|M|$ is the length of M. This notation highlights that we only consider padding functions that append a bitstring that is fully determined by the length of M and the block length x. We may omit $[x]$, $|M|$ or both if their value is clear from the context. For the sponge construction to be secure (see Section 3.2), the padding function pad must satisfy the following (easy to realize) requirements:

Reversible Given a padded string, it shall be possible to reconstruct the string before padding.
Non-empty A padded string shall consists of at least one block.
Non-zero last block The last block of a padded string shall differ from an all-zero block.

3.1 Definition

The permutation f operates on a fixed number of bits, the *width* b. The sponge construction has a state of b bits. First, all the bits of the state are initialized to zero. The input message is padded with the function $\text{pad}[r]$ and cut into r-bits blocks. Then it proceeds in two phases: the *absorbing phase* followed by the *squeezing phase*:

Absorbing phase The r-bit input message blocks are XORed into the first r bits of the state, interleaved with applications of the function f. When all message blocks are processed, the sponge construction switches to the squeezing phase.

Squeezing phase The first r bits of the state are returned as output blocks, interleaved with applications of the function f. The number of iterations is determined by the requested number of bits.

Finally the output is truncated to the requested length. The sponge construction is illustrated in Figure 1, and Algorithm 1 provides a formal definition. In our algorithms and figures we denote the bitlength of a string x by $|x|$ and truncation to its ℓ first bits by $\lfloor x \rfloor_\ell$.

The value $c = b - r$ is called the *capacity*. The last c bits of the state are never directly affected by the input blocks and are never output during the squeezing phase. The capacity c actually determines the attainable security level of the construction [9,11].

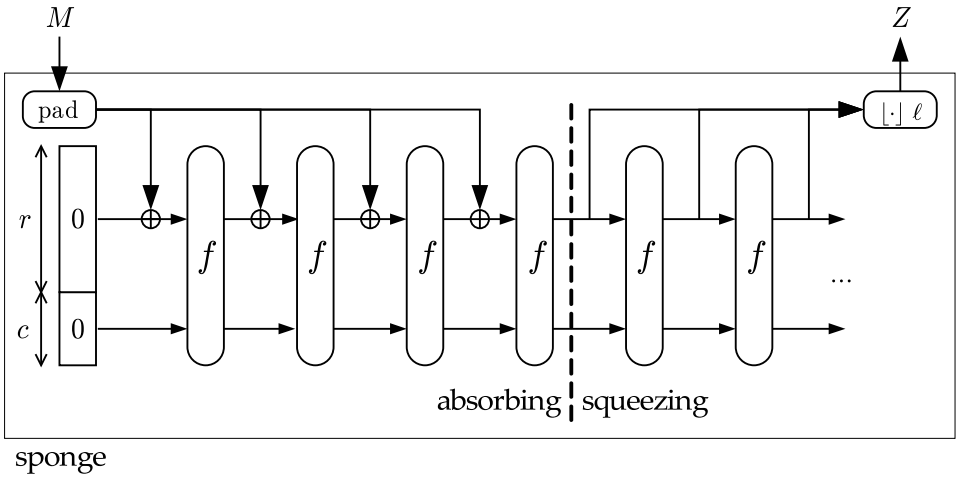

Fig. 1. The sponge construction

Algorithm 1 The sponge construction SPONGE$[f, \text{pad}, r]$

Require: $r < b$

Interface: $Z = \text{sponge}(M, \ell)$ with $M \in \mathbb{Z}_2^*$, integer $\ell > 0$ and $Z \in \mathbb{Z}_2^\ell$
$P = M||\text{pad}[r](|M|)$
Let $P = P_0||P_1||\dots||P_w$ with $|P_i| = r$
$s = 0^b$
for $i = 0$ to w **do**
  $s = s \oplus (P_i||0^{b-r})$
  $s = f(s)$
end for
$Z = \lfloor s \rfloor_r$
while $|Z| < \ell$ **do**
  $s = f(s)$
  $Z = Z||\lfloor s \rfloor_r$
end while
return $\lfloor Z \rfloor_\ell$

3.2 Security

Cryptographic functions are often designed in two steps. In the first step, one chooses a construction that uses a cryptographic primitive with fixed input and output size (e.g., a compression function or a permutation) and builds a function that can take inputs and or generate outputs of arbitrary size. If the security of this construction can be proven, it guarantees that any potential flaw can only come from the underlying cryptographic primitive, and thereby reduces the scope of cryptanalysis.

We have taken this approach for the sponge construction. In [9] we have proven that the sponge construction is *indifferentiable* from a random oracle. Indifferentiability is a concept developed by Maurer, Renner and Holenstein and allows one to compare the security of a system to that of an ideal object, such as the random oracle [33]. The system can use an underlying cryptographic primitive (e.g., a compression function or a permutation) as a public subsystem.

In [9] we have proven that the success probability of any generic attack for differentiating the sponge construction calling a random permutation or transformation from a random oracle is upper bounded by $2^{-(c+1)}N^2$. Here N is the number of calls to the underlying permutation or its inverse. This results in a lower bound for the expected complexity of about $2^{c/2}$. Note that this is true independently of the output

length. For example, finding collisions for output lengths shorter than c has for a random sponge the same expected complexity as for a random oracle.

In [11], we address the security of the sponge construction when the message is prefixed with a key, as it will be done in the mode of Section 5. In this specific case, the security proof goes beyond the $2^{c/2}$ complexity if the number of input or output blocks for which the key is used (data complexity) is upper bounded by $M < 2^{c/2-1}$. In that case, distinguishing the keyed sponge from a random oracle has time complexity of at least $2^{c-1}/M > 2^{c/2}$. Hence, for keyed modes, one can reduce the capacity c for the same targeted security level.

Note that the proofs mentioned above only cover generic attacks, namely, attacks that do not exploit specific properties of f. The natural design goal for any concrete f to be used in a sponge function is hence the absence of properties exploitable in attacks. In the *hermetic sponge strategy* [10] this is addressed by building a permutation f that should not have structural distinguishers.

3.3 Implementing authenticated encryption

As said, the simplest way to build an actual system that behaves as the reference system described above would be to replace the random oracle $\mathcal{RO}$ by a sponge function. The indifferentiability proof in [9] guarantees the result is secure if the permutation f of the sponge function has no structural distinguishers.

However, such a solution requires two sponge function executions: one for the generation of the key stream and one for the generation of the tag, while we aim for a single-pass solution. To achieve this, we define a variant where the key stream blocks and tag are the responses of a sponge function to input sequences that are each other's prefix. This introduces a new construction that is closely related to the sponge construction: the duplex construction. Subsequently, we build an authenticated encryption mode on top of that.

4 The duplex construction

Like the sponge construction, the *duplex construction* DUPLEX$[f, \text{pad}, r]$ uses a fixed-length transformation (or permutation) f, a padding function pad and a parameter bitrate r. Unlike a sponge function that is stateless in between calls, the duplex construction accepts calls that take an input string and return an output string depending on all inputs received so far. We call an instance of the duplex construction a *duplex object*, which we denote D in our descriptions. We prefix the calls made to a specific duplex object D by its name D and a dot.

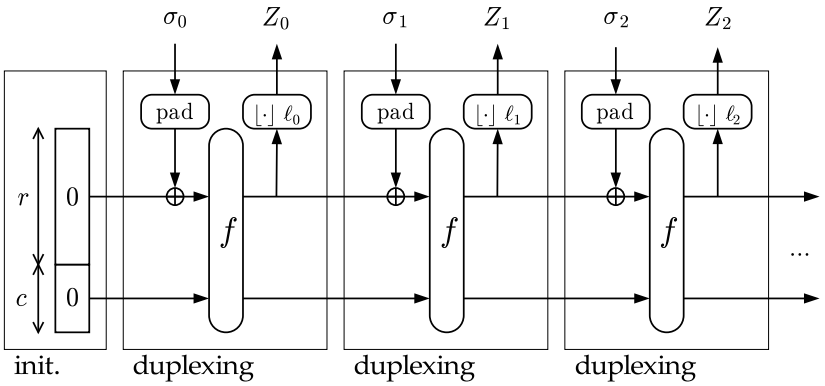

Fig. 2. The duplex construction

The duplex construction works as follows. A duplex object D has a state of b bits. Upon initialization all the bits of the state are set to zero. From then on one can send to it D.duplexing(σ, ℓ) calls, with σ an input string and ℓ the requested number of bits.

Algorithm 2 The duplex construction DUPLEX$[f, \text{pad}, r]$

Require: $r < b$
Require: $\rho_{\max}(\text{pad}, r) > 0$

Interface: D.initialize()
$s = 0^b$

Interface: $Z = D.\text{duplexing}(\sigma, \ell)$ with $\ell \leq r$, $\sigma \in \bigcup_{n=0}^{\rho_{\max}(\text{pad},r)} \mathbb{Z}_2^n$, and $Z \in \mathbb{Z}_2^\ell$
$P = \sigma || \text{pad}[r](|\sigma|)$
$s = s \oplus (P || 0^{b-r})$
$s = f(s)$
return $\lfloor s \rfloor_\ell$

The maximum number of bits ℓ one can request is r and the input string σ shall be short enough such that after padding it results in a single r-bit block. We call the maximum length of σ the *maximum duplex rate* and denote it by $\rho_{\max}(\text{pad}, r)$. Formally:

$$\rho_{\max}(\text{pad}, r) = \max\{x : x + |\text{pad}[r](x)| \leq r\}. \tag{1}$$

Upon receipt of a $D.\text{duplexing}(\sigma, \ell)$ call, the duplex object pads the input string σ and XORs it into the first r bits of the state. Then it applies f to the state and returns the first ℓ bits of the state at the output. We call a *blank call* a call with σ the empty string, and a *mute call* a call without output, $\ell = 0$. The duplex construction is illustrated in Figure 2, and Algorithm 2 provides a formal definition.

The following lemma links the security of the duplex construction DUPLEX$[f, \text{pad}, r]$ to that of the sponge construction SPONGE$[f, \text{pad}, r]$. Generating the output of a D.duplexing() call using a sponge function is illustrated in Figure 3.

Lemma 1. [Duplexing-sponge lemma] *If we denote the input to the i-th call to a duplex object by (σ_i, ℓ_i) and the corresponding output by Z_i we have:*

$$Z_i = D.\text{duplexing}(\sigma_i, \ell_i) = \text{sponge}(\sigma_0 || \text{pad}_0 || \sigma_1 || \text{pad}_1 || \dots || \sigma_i, \ell_i)$$

with pad_i *a shortcut notation for* $\text{pad}[r](|\sigma_i|)$.

Proof. The proof is by induction on the number of input strings σ_i.

First consider the case $i = 0$. We must prove $D.\text{duplexing}(\sigma_0, \ell_0) = \text{sponge}(\sigma_0, \ell_0)$. The state of the duplex object before the call has value 0^b, the same as the initial state of the sponge function. Both in the case of the sponge function and the duplex object the input string is padded with pad resulting in a single r-bit block P. Then, in both cases P is XORed to the first r bits of the state and f is applied to the state. At this point the sponge function and the duplex object have the same state and both return the first $\ell \leq r$ bits of the state as output string. Since the sponge function does not do any additional iterations of f on the state, the state of the duplex object after the call $D.\text{duplexing}(\sigma_0, \ell_0)$ is equal to the state of the sponge construction after absorbing a single block $\sigma_0 || \text{pad}_0$.

Now assume that after the call $D.\text{duplexing}(\sigma_{i-1}, \ell_{i-1})$ the duplex object has the same state as the sponge function after absorbing $\sigma_0 || \text{pad}_0 || \sigma_1 || \text{pad}_1 || \dots || \sigma_{i-1} || \text{pad}_{i-1}$. During the call $D.\text{duplexing}(\sigma_i, \ell_i)$, the block $\sigma_i || \text{pad}_i$ is XORed into the first r bits of the state and subsequently f is applied to the state. It follows that the state of the duplex object D after the call $D.\text{duplexing}(\sigma_i, \ell_i)$ is equal to the state of the sponge function after absorbing $\sigma_0 || \text{pad}_0 || \sigma_1 || \text{pad}_1 || \dots \sigma_i || \text{pad}_i$. As the output just consists of the first ℓ_i bits of the state, this proves Lemma 1.

□

Thanks to the duplexing-sponge lemma the output of a duplexing call is the output of a sponge function with an input $\sigma_0 || \text{pad}_0 || \sigma_1 || \text{pad}_1 || \dots || \sigma_i || \text{pad}_i$ and from this input the exact sequence $\sigma_0, \sigma_1, \dots, \sigma_i$ can be recovered. As such, the duplex construction is as secure as the sponge construction with the same parameters. In the following sections we will show that the duplex construction is a powerful tool for building modes of use.

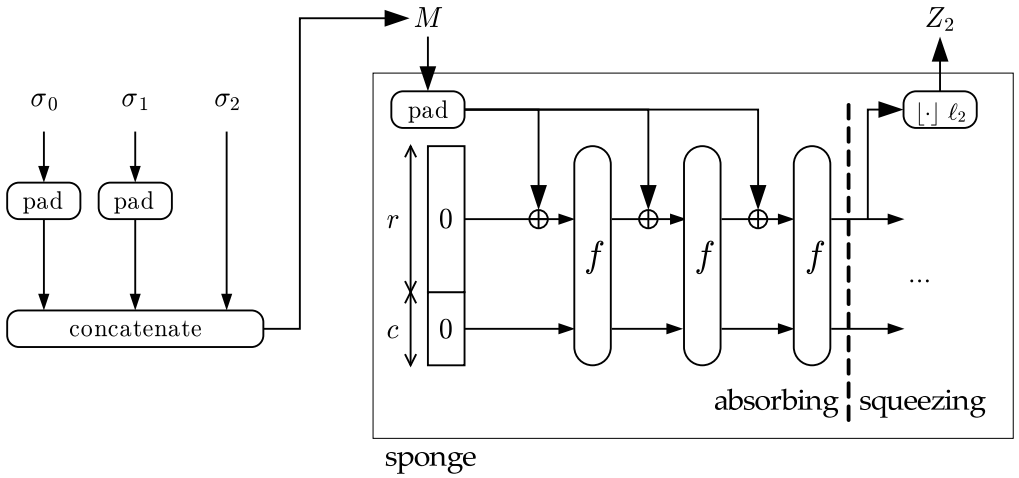

Fig. 3. Generating the output of a duplexing call with a sponge

5 The authenticated encryption mode SpongeWrap

We propose an authenticated encryption mode SpongeWrap that realizes a generalization of the authenticated encryption process defined in Section 2. Similarly to the duplex construction, we call an instance of the authenticated encryption mode a SpongeWrap object.

Upon initialization of a SpongeWrap object, it loads the key K. From then on one can send requests to it for wrapping and/or unwrapping data. The key stream blocks used for encryption and the tags depend on the key K and the data sent in all previous requests. The process defined in Section 2 can be implemented with the SpongeWrap mode using only a single wrap or unwrap request.

5.1 Definition

A SpongeWrap object W internally uses a duplex object D with parameters f, pad and r. Upon initialization of a SpongeWrap object, it initializes D and forwards the (padded) key blocks K to D using mute D.duplexing() calls.

When receiving a W.wrap(A, B, ℓ) request, it forwards the blocks of the (padded) header A and the (padded) body B to D. It generates the cryptogram C block by block $C_i = B_i \oplus Z_i$ with Z_i the response of D to the previous D.duplexing() call. The ℓ-bit tag T is the response of D to the last body block (possibly extended with the response to additional blank D.duplexing() calls in case ℓ is large). Finally it returns the cryptogram C and the tag T.

When receiving a W.unwrap(A, C, T) request, it forwards the blocks of the (padded) header A to D. It decrypts the data body B block by block $B_i = C_i \oplus Z_i$ with Z_i the response of D to the previous D.duplexing() call. The response of D to the last body block (possibly extended) is compared with the tag T received as input. If the tag is valid, it returns the data body B; otherwise, it returns an error. Note that in implementations one may impose additional constraints, such as SpongeWrap objects dedicated to either wrapping or unwrapping. Additionally, the SpongeWrap object may impose a minimum length for the tag received before unwrapping.

Before being forwarded to D, every key, header, data or cryptogram block is extended with a so-called *frame bit*. The rate ρ of the SpongeWrap mode determines the size of the blocks and hence the maximum number of bits processed per call to f. Its upper bound is $\rho_{\max}(\text{pad}, r) - 1$ due to the inclusion of one frame bit per block. A formal definition of SpongeWrap is given in Algorithm 3.

Algorithm 3 The authenticated encryption mode SpongeWrap$[f, \text{pad}, r, \rho]$.

Require: $\rho \leq \rho_{\max}(\text{pad}, r) - 1$
Require: $D = \text{duplex}[f, \text{pad}, r]$

Interface: W.initialize(K) with $K \in \mathbb{Z}_2^*$
Let $K = K_0||K_1||\dots||K_u$ with $|K_i| = \rho$ for $i < u$, $|K_u| \leq \rho$ and $|K_u| > 0$ if $u > 0$
D.initialize()
for $i = 0$ to $u - 1$ **do**
  D.duplexing$(K_i||1, 0)$
end for
D.duplexing$(K_u||0, 0)$

Interface: $(C, T) = W$.wrap(A, B, ℓ) with $A, B \in \mathbb{Z}_2^*$, $\ell \geq 0$, $C \in \mathbb{Z}_2^{|B|}$ and $T \in \mathbb{Z}_2^\ell$
Let $A = A_0||A_1||\dots||A_v$ with $|A_i| = \rho$ for $i < v$, $|A_v| \leq \rho$ and $|A_v| > 0$ if $v > 0$
Let $B = B_0||B_1||\dots||B_w$ with $|B_i| = \rho$ for $i < w$, $|B_w| \leq \rho$ and $|B_w| > 0$ if $w > 0$
for $i = 0$ to $v - 1$ **do**
  D.duplexing$(A_i||0, 0)$
end for
$Z = D$.duplexing$(A_v||1, |B_0|)$
$C = B_0 \oplus Z$
for $i = 0$ to $w - 1$ **do**
  $Z = D$.duplexing$(B_i||1, |B_{i+1}|)$
  $C = C||(B_{i+1} \oplus Z)$
end for
$Z = D$.duplexing$(B_w||0, \rho)$
while $|Z| < \ell$ **do**
  $Z = Z||D$.duplexing$(0, \rho)$
end while
$T = \lfloor Z \rfloor_\ell$
return (C, T)

Interface: $B = W$.unwrap(A, C, T) with $A, C, T \in \mathbb{Z}_2^*$, $B \in \mathbb{Z}_2^{|C|} \cup \{\text{error}\}$
Let $A = A_0||A_1||\dots||A_v$ with $|A_i| = \rho$ for $i < v$, $|A_v| \leq \rho$ and $|A_v| > 0$ if $v > 0$
Let $C = C_0||C_1||\dots||C_w$ with $|C_i| = \rho$ for $i < w$, $|C_w| \leq \rho$ and $|C_w| > 0$ if $w > 0$
Let $T = T_0||T_1||\dots||T_x$ with $|T_i| = \rho$ for $i < x$, $|C_x| \leq \rho$ and $|C_x| > 0$ if $x > 0$
for $i = 0$ to $v - 1$ **do**
  D.duplexing$(A_i||0, 0)$
end for
$Z = D$.duplexing$(A_v||1, |C_0|)$
$B_0 = C_0 \oplus Z$
for $i = 0$ to $w - 1$ **do**
  $Z = D$.duplexing$(B_i||1, |C_{i+1}|)$
  $B_{i+1} = C_{i+1} \oplus Z$
end for
$Z = D$.duplexing$(B_w||0, \rho)$
while $|Z| < \ell$ **do**
  $Z = Z||D$.duplexing$(0, \rho)$
end while
if $T = \lfloor Z \rfloor_\ell$ **then**
  return $B_0||B_1||\dots B_w$
else
  return Error
end if

5.2 Security

This modes follows the ideal construction of Section 2.2, with two differences: first, the random oracle is replaced by a sponge function (via the duplexing-sponge lemma) and second, we allow the key stream to depend on previous blocks. For the former, the security of the SpongeWrap mode thus depends on the security of the underlying sponge function. The introduction of the dependency on previous blocks does not reduce the security of the ideal construction but is required to match the interface of the duplex construction. Hence, the security of the SpongeWrap mode reduces to the ability to have injective encoding functions s_k and h_t.

Let us first review the properties of the frame bit. It serves two purposes:

Domain separation The duplex (or equivalently, sponge) inputs to generate key stream blocks and those to generate tag blocks are in separate domains. Every duplex response that is used to encipher the next block has as input a string ending with a frame bit 1, whereas every duplex response that is used to form a tag has as input a string ending with a frame bit 0.

Decodability The key, header and body blocks can be recovered from the duplex input sequence. This implies that two different sequences K, $A^{(0)}$, $B^{(0)}$, $A^{(1)}$, $B^{(1)}, \ldots$ and K', $A'^{(0)}$, $B'^{(0)}$, $A'^{(1)}$, $B'^{(1)}, \ldots$ cannot lead to two equal duplex input sequences.

Hence, it is easy to see that SpongeWrap$[f, \mathrm{pad}, r, \rho]$ satisfies the security requirements defined in Section 2.1 on the condition that the sponge function sponge$[f, \mathrm{pad}, r]$ is secure. This follows from the following properties:

- For different inputs, tag blocks are the responses of sponge calls with distinct input strings.
- If the (first of a sequence) header $A^{(0)}$ is a nonce, all key stream blocks are the responses of sponge calls with distinct input strings.
- Tag blocks and key stream blocks are the responses of sponge calls for input strings in separate domains.
- The usage of the key is limited to serving as a prefix to all input strings to sponge calls.

5.3 Advantages and limitations

The authenticated encrpyption mode SpongeWrap has the following unique combination of advantages:

- While most other authenticated encryption modes require a block cipher, SpongeWrap only requires a fixed-length permutation.
- It supports the alternation of strings that require authenticated encryption and strings that only require authentication.
- It can provide intermediate tags after each W.wrap(A, B, ℓ) request.
- It has a strong security bound against generic attacks with a very simple proof, that relies on the indifferentiability of the sponge construction (or the security of keyed sponge functions specifically) and on the sponge-duplexing lemma.
- It is single-pass: it requires only a single call to the permutation f per ρ-bit block.
- It is flexible as the bitrate can be freely chosen as long as the capacity is larger than some lower bound.
- The encryption is not expanding.

As compared to some block cipher based authenticated encryption modes, it has some limitations. First, the mode as such is serial and cannot be parallelized at algorithmic level. Some block cipher based modes do actually allow parallelization, for instance, the offset codebook (OCB) mode [37]. Yet, SpongeWrap can support parallel streams in a fashion similar to tree hashing, but with some overhead.

Second, if a system does not impose the nonce requirement on A, an attacker may send two requests (A, B) and (A, B') with $B \neq B'$. In this case, the first differing blocks of B and B', say B_i and B'_i, will be enciphered with the same key stream, making their bitwise XOR available to the attacker. Some block cipher based modes are *misuse resistant*, i.e., they are designed in such a way that in case the nonce requirement is not fulfilled, the only information an attacker can find out is whether B and B' are equal or not [39]. Yet, many applications already provide a nonce, such as a packet number or a key ID, and can put it in A.

5.4 An application: key wrapping

Key wrapping is the process of ensuring the secrecy and integrity of cryptographic keys in transport or storage, e.g., [34,20]. A *payload key* is wrapped with a *key-encrypting key* (KEK). We can use the SpongeWrap mode with K equal to the KEK and let the data body be the payload key value. In a sound key management system every key has a unique identifier. It is sufficient to include the identifier of the payload key in the header A and two different payload keys will never be enciphered with the same key stream. When wrapping a private key, the corresponding public key or a digest computed from it can serve as identifier.

6 Other applications of the duplex construction

Authenticated encryption is just one application of the duplex construction. In this section we further illustrate it by providing two more examples: a pseudo random sequence generator and a sponge-like construction that overwrites part of the state with the input block rather than to XOR it in.

6.1 A reseedable pseudo random sequence generator SpongePRNG

In various cryptographic applications and protocols, random numbers are used to generate keys or unpredictable challenges. While randomness can be extracted from a physical source, it is often necessary to provide many more bits than the entropy of the physical source. In this context, a pseudo-random number generator (PRNG) provides a way to do so. It is initialized with a seed, generated in a secret or truly random way, and it then expands the seed into a sequence of bits.

For cryptographic purposes, it is required that the generated bits cannot be predicted, even if subsets of the sequence are revealed. In this context, a PRNG is similar to a stream cipher. A PRNG is also similar to a cryptographic hash function when gathering entropy coming from different sources. Finally, some applications require a pseudo-random number generator to support forward security: The compromise of the current state does not enable the attacker to determine the previously generated pseudo-random bits [5,19].

It is convenient for a pseudo-random number generator to be reseedable, i.e., one can bring an additional source of entropy after pseudo-random bits have been generated. Instead of throwing away the current state of the PRNG, reseeding combines the current state of the generator with the new seed material. From a user's point of view, a reseedable PRNG can be seen as a black box with an interface to request pseudo-random bits and an interface to provide fresh seed material.

In [12] we have defined a reseedable PRNG based on the sponge construction that implements the required functionality. The ideas behind that PRNG are very similar to the duplex construction. The goal of this section is to show that a PRNG can be easily defined on top of the duplex construction.

First note that a duplex object can readily be used as a reseedable PRNG. Seed material can be fed via the σ inputs in D.duplexing() call and the responses can be used as pseudo-random bits. If pseudo-random bits are required and there is no seed available, one can simply send blank D.duplexing() calls. The only limitation of this is that the user must split his seed material in strings of at most $\rho_{\max}$ bits and that at most r bits can be requested in a single call.

Then, we propose a more sophisticated mode called SpongePRNG. This mode is similar to the one proposed in [12] in that it minimizes the number of calls to f, although explicitly based on the duplex construction.

The SpongePRNG mode works as follows. Internally it makes use of a duplex object D and it has two buffers: an input buffer B_{in} and an output buffer B_{out}. During feed requests it accumulates seed material in B_{in} and, if it has received at least ρ bits, it forwards them to D in a D.duplexing() call. Any surplus seed string is kept in the input buffer. Upon a fetch request, if the input buffer is not empty, it empties it by forwarding any remaining seed to D and returns the requested number of bits, performing more duplexing calls if necessary, each requesting ρ bits. The surplus of produced bits are kept in B_{out}, which will be returned first upon the next fetch request. Note that at any moment, one of B_{in} and B_{out} is empty.

As such, the operation of a SpongePRNG object is based on a permutation and revealing the state allows the attacker to backtrack the generation back to the most recent unknown seed fed into it. Still, forward security can be explicitly enforced by means of a P.forget() request. The effect of a P.forget() request is the resetting to zero of the first ρ bits of the state and a subsequent application of f. This is done $\lfloor c/\rho \rfloor$ times.

Guessing the state before this operation given the state afterwards requires guessing at least c bits and hence is infeasible for reasonable values of c.

The SpongePRNG mode is defined in Algorithm 4. Note that the buffers do not require separate storage but can be implemented merely as pointers to the state: The input buffer requires a pointer to the state indicating from where on new bits must be XORed into the state, while the output buffer pointer points in the state where the next output bit must be taken. The storage is thus limited to the b-bit state and two integers.

It is clear that every bit returned by P.fetch() is part of the output of the sponge presented with a string that contains all seed material presented so far. The SpongePRNG mode does not allow reconstructing the individual blocks σ_i but does allow reconstructing their concatenation.

Algorithm 4 Pseudo random sequence generator mode SpongePRNG$[f, \text{pad}, r, \rho]$

Require: $\rho \leq \rho_{\max}(\text{pad}, r)$
Require: $D = \text{DUPLEX}[f, \text{pad}, r]$

Interface: P.initialize()
D.initialize()
B_{in} = empty string
B_{out} = empty string

Interface: P.feed(σ) with $\sigma \in \mathbb{Z}_2^*$
$M = B_{\text{in}}||\sigma$
Let $M = M_0||M_1||\ldots||M_w$ with $|M_i| = \rho$ for $i < w$ and $0 \leq |M_w| < \rho$
for $i = 0$ to $w - 1$ **do**
  D.duplexing($M_i, 0$)
end for
$B_{\text{in}} = M_w$
B_{out} = empty string

Interface: $Z = P$.fetch(ℓ) with integer $\ell \geq 0$ and $Z \in \mathbb{Z}_2^\ell$
while $|B_{\text{out}}| < \ell$ **do**
  $B_{\text{out}} = B_{\text{out}}||D$.duplexing($B_{\text{in}}, \rho$)
  B_{in} = empty string
end while
$Z = \lfloor B_{\text{out}} \rfloor_\ell$
B_{out} = last $(|B_{\text{out}}| - \ell)$ bits of B_{out}
return Z

Interface: $Z = P$.forget()
$Z = D$.duplexing(B_{in}, ρ)
B_{in} = empty string
for $i = 1$ to $\lfloor c/r \rfloor$ **do**
  $Z = D$.duplexing(Z, ρ)
end for
B_{out} = empty string

6.2 The mode Overwrite

In [25] sponge-like constructions were proposed and cryptanalyzed. In some of these constructions, absorbing is done by overwriting part of the state by the message block rather than XORing it in. A concrete function that follows such a construction is the hash function Grindahl [31].

These overwrite functions have the advantage over sponge functions that between calls to f, only c bits must be kept instead of b. This may not be useful when hashing a message in a continuous fashion, as b bits

must be processed by f anyway. However, when hashing a partial message, then putting it aside to continue later on, storing only c bits may be useful on some platforms.

It turns out that an overwrite function is equivalent to a function based on the duplex construction. If the first ρ bits of the state are known to be Z, overwriting them with a message block P_i is equivalent to XORing in $Z \oplus P_i$. This idea is also used in the forget call of the SpongePRNG mode and is formally implemented in Algorithm 5. In practice, of course, the implementation can just overwrite the first ρ bits of the state by a message block. As a matter of fact, Algorithm 5 can be rewritten to call f directly, similar to the sponge construction. We leave this as an exercise for the reader.

We define the mode Overwrite on top of the duplex construction. An Overwrite function internally uses a duplex object D. It pads the message M and splits it in ρ-bit blocks. Then it makes a sequence of D.duplexing() calls, each time with a message block XORed with the response of the previous D.duplexing() call and with a frame bit appended to it. This frame bit is equal to 1 for the last block and 0 for all other blocks. If the requested number of output bits ℓ is larger than ρ, additional D.duplexing() calls are done where each time the response of the previous D.duplexing() call is fed back to D.

The coding using the frame bits allows, for any input sequence of D, finding the last block and the length of the original message M. To recover the message M from the input sequence, one can start with the first block. Since $Z = 0^\rho$ in the first block, the first block in the D.duplexing() call allows recovering the first block of M. Then, this block allows determining the output Z that was XORed into the next block, and so on. This, together with the sponge-duplexing lemma proves that Overwrite$[f, \text{pad}, r, \rho]$ is as secure as sponge$[f, \text{pad}, r]$.

We have thus proven that thanks to the duplexing-sponge lemma the security of Overwrite is equivalent to that of the sponge construction with the same parameter, but at a cost of 2 bits of bitrate (or equivalently, of capacity).

Algorithm 5 The construction Overwrite$[f, \text{pad}, r, \rho]$

Require: $\rho \leq \rho_{\max}(\text{pad}, r) - 1$
Require: $D = \text{duplex}[f, \text{pad}, r]$

Interface: $Z = \text{Overwrite}(M, \ell)$ with $M \in \mathbb{Z}_2^*$, integer $\ell > 0$ and $Z \in \mathbb{Z}_2^\ell$
$P = M||\text{pad}[\rho](|M|)$
Let $P = P_0||P_1||\dots||P_w$ with $|P_i| = \rho$ for $i \leq w$
D.initialize()
$Z = 0^\rho$
for $i = 0$ to $w - 1$ **do**
  $Z = D.\text{duplexing}((P_i \oplus Z)||0, \rho)$
end for
$Z = D.\text{duplexing}((P_w \oplus Z)||1, \rho)$
$B_{\text{out}} = Z$
while $|B_{\text{out}}| < \ell$ **do**
  $Z = D.\text{duplexing}(Z||1, \rho)$
  $B_{\text{out}} = B_{\text{out}}||Z$
end while
return $\lfloor B_{\text{out}} \rfloor_\ell$

7 A compact padding scheme suitable for sponge functions

An ideal padding scheme should be compact and should be suitable for a family of sponge functions with different rates.

For a given capacity and width, the padding reduces the maximum bitrate of the duplex construction, as in Eq. (1). To minimize this effect, especially when the width of the permutation is relatively small, we recommend choosing a compact padding scheme. The padding scheme that satisfies the criteria listed in Section 3 with the smallest overhead is the well-known *simple reversible padding*, which appends a single

1 and the smallest number of zeroes such that the length of the result is a multiple of the required block length. We denote it by $\text{pad10}^*[r](M)$. It satisfies $\rho_{\max}(\text{pad10}^*, r) = r - 1$ and hence has only one bit of overhead.

When considering the security of a set of sponge functions that make use of the same permutation f but with different bitrates, simple reversible padding is not sufficient. The indifferentiability proof of [9] actually only covers the indifferentiability of a single sponge function instance from a random oracle. For instance, the padding of the sponge function family KECCAK explicitly encodes the bitrate (in bytes). It was proven that this padding scheme can be used to make a set of sponge functions indifferentiable from a set of random oracles [10, Theorem 1].

In practice, it is useful to allow bitrates to have specific values. In many applications one prefers to have block lengths that are a multiple of 8 or even higher powers of two to avoid bit shifting or misalignment issues. With modes using the duplex construction, one has to distinguish between the mode-level block size and the bitrate of the underlying sponge function. For instance in the authenticated encryption mode SPONGEWRAP, the block size is at most $\rho_{\max}(\text{pad}, r) - 1$. To have a block size with the desired value, it suffices to take a slightly higher value as bitrate r; hence, the sponge-level bitrate may no longer be a multiple of 8 or of a higher power of two. This motivated us to consider the security of a set of sponge functions with common f and different bitrates, including bitrates that are not multiples of 8 or of a higher power of two.

As a solution, we propose the *minimal sponge padding*, denoted $\text{pad10}^*1[r](|M|)$, which returns a bitstring 10^q1 with $q = (-|M| - 2) \bmod r$. This padding satisfies the three requirements listed in Section 3 and has $\rho_{\max}(\text{pad10}^*1, r) = r - 2$. Hence, this padding scheme is compact as the duplex-level maximum rate differs from the sponge-level rate by only two bits. Furthermore, it is sufficient for the indifferentiability of a set of sponge functions as shown in Theorem 1. The intuitive idea behind this is that, with the pad10^*1 padding scheme, the last block absorbed has a bit with value 1 at position $r - 1$, while any other function of the family with $r' < r$ this bit has value 0.

Regarding the indifferentiability of a set of sponge functions, it is clear that the best one can achieve is bounded by the strength of the sponge construction with the lowest capacity (or, equivalently, the highest bitrate), as an adversary can always just try to differentiate the weakest construction from a random oracle. The next theorem states that we achieve this bound by using the minimal sponge padding.

Theorem 1. *Given a random permutation (or transformation) f, differentiating the array of sponge functions* $\text{SPONGE}[f, \text{pad10}^*1, r]$ *with* $0 < r \leq r_{\max}$ *from an array of independent random oracles* $(\mathcal{RO}_r)$ *has the same success probability as differentiating* $\text{SPONGE}[f, \text{pad10}^*1, r_{\max}]$ *from a random oracle.*

Proof. An array $(\mathcal{RO}_r)$ of random oracles can be implemented by having a single random oracle $\mathcal{RO}$ and algorithms I_r that pre-process the input strings, so that $\mathcal{RO}_r(M) = \mathcal{RO}(I_r(M))$. To simulate independent random oracles, each I_r must produce a different range of output strings, i.e., provide domain separation. This reasoning is also valid if the output of the random oracles is processed by some algorithm O_r that extracts bits at predefined positions, so that $\mathcal{RO}_r(M) = O_r(\mathcal{RO}(I_r(M)))$.

We simulate the array of sponge functions $\text{SPONGE}[f, \text{pad10}^*1, r]$, via a single sponge function $\text{sponge}_{\max} = \text{SPONGE}[f, \text{pad10}^*, r_{\max}]$ using a pre-processing function applied to the input. We then rely on the indifferentiability proof in [9] for the indifferentiability between $\text{sponge}_{\max}$ and $\mathcal{RO}$.

We define a blockwise appending function $x = I_r(M)$ that consists of the following steps:

- apply the minimal sponge padding: $x = M||\text{pad10}^*1[r](|M|)$;
- split the result in r-bit blocks x_i;
- append to all blocks $r_{\max} - r$ zeroes, $x_i \leftarrow x_i||0^{r_{\max}-r}$;
- concatenate the blocks again: $x = x_0||x_1||\dots||x_u$;
- remove the last $(r_{\max} - r + 1)$ bits, $x \leftarrow \lfloor x \rfloor_{|x|-r_{\max}+r-1}$, and return x.

Notice that $|x| \bmod r_{\max} = r - 1$.

Similarly, we define a blockwise truncation function $z = O_r(y)$ that consists of first splitting y in $r_{\max}$-bit blocks and then truncating each block to r bits.

It is easy to verify that

$$\text{SPONGE}[f, \text{pad10}^*1, r](M) = O_r(\text{sponge}_{\max}(I_r(M))).$$

It is clear that the blockwise appending function $x = I_r(M)$ realizes domain separation, as $|x| \bmod r_{\max} = r - 1$ and $r_{\max} \geq r$. Once the bitrate r has been determined, it is easy to reconstruct M from x by simply removing the zero blocks and the reversible padding at the end.

Hence, differentiating the array $\text{SPONGE}[f, \text{pad10*1}, r]$ from the array $(\mathcal{RO}_r)$ comes down to differentiating $\text{sponge}_{\max}$ from $\mathcal{RO}$, where $\text{sponge}_{\max}$ has capacity $c_{\min} = b - r_{\max}$. □

8 Duplexing iterated functions in general

The duplex construction can be seen as a way to use a sponge function in a cascaded way. The central idea is that a duplex object keeps a state equal to that of a sponge function that has absorbed the combination of all inputs to the duplex object so far. Clearly, the same principle can be applied to most other sequential hash function constructions that consist of the iterated application of a compression function or permutation f.

In general, a duplex-like object corresponding to such a hash function would work as follows. Its state is the chaining value resulting from hashing all previous inputs and possibly a counter (e.g., if the hash function requires the message length for the padding or as input in the compression function). Upon presentation of an input σ, it performs two tasks. First, it generates an output: It pads σ with the padding rule of the hash function, applies the final compression function f or an output transformation g, and returns the result. Second, it updates its state by padding σ with reversible padding, applying f and updating the counter.

The disadvantage of this method is that, in general, a single duplexing call to the object requires two calls to f, or in case of an output transformation g, one call to f and one to g. In contrast, for a sponge function, the generation of the output and the update of the state can be done in a single call to f.

Three main obstacles may hinder the efficiency of duplexing.

- First, as already mentioned, the special processing done after the last block prevents to update the state and produce output at the same time. For instance, some constructions have an output transformation, which must be applied before producing output, while the main compression function is applied to update the state. The same problem occurs in the HAIFA framework [13], which enforces domain separation between the final call to f and the previous ones. In some constructions, blank iterations are applied at the end, which must be performed every time output is requested.
- Second, the overhead due to the padding reduces the number of bits that can be input in a duplexing call. If the input block size is fixed to a power of two (or a small multiple of it), the place taken by the padding can break the alignment of input blocks. Flexibility on the input block size is thus an advantage in this respect, as it can restore their alignment.
- Third, the output length of the hash function may be smaller than the input block size. This can be another slowdown factor, as in the case of the SpongeWrap mode, since as many output bits are needed as input bits. The last compression function, output transformation or blank iterations have then to be performed several times to produce output bits like in a mask generating function. Another possible solution is just to use shorter input blocks.

The chop-MD construction [18,17] is a good candidate for duplexing. Producing output and updating the state can be made in the same operation. However, for the duplexing to be as fast as hashing, the output length should be as large as the message block and the padding should be as compact as possible.

Table 1 compares the SHA-3 second-round candidates, without modifications, with respect to the efficiency of duplexing. The cost factor tells how much slower duplexing would be compared to plain hashing of long messages. Note that the effect of the padding size is ignored, that is, we assume the number of bits taken by the padding is small compared to the block size, except if it is explicitly specified to be as big as a whole block (JH and SIMD).

The numbers vary greatly from one to the other. Besides Keccak, Hamsi and Luffa-512 have a moderate cost factor. For instance, Hamsi needs 8 iterations of 32 bits each to update the state with a 256-bit input block. Then the output of 256 bits can be performed at the cost of an output transformation equivalent to 2 iterations. Hence, the cost factor is $(8 + 2)/8 = 1.25$. As another example, functions such as BLAKE, ECHO-512, and SHAvite-3 have a block size twice the output size. It costs 1 iteration to update the state with one block. Because of the last block domain separation and of the block size output size ratio, the output must be produced by 2 additional iterations of the compression function, hence a total cost of 3.

Candidate	Features	Cost factor
BLAKE [1]	LBDS, OS/BS	3
BMW [24]	OS/BS	2
Cubehash-16/32 [7]	BI	6
ECHO-256 [6]	LBDS, OS/BS	7
ECHO-512 [6]	LBDS, OS/BS	3
Fugue [26]	BI, OT	≈ 2
Grøstl [22]	OT, OS/BS	3
Hamsi [29]	OT	1.25
JH [41]	BI due to padding	2
Keccak [10]		1
Luffa-256 [16]	BI	2
Luffa-512 [16]	BI	1.5
Shabal [15]	BI	4
Shavite-3 [14]	LBDS, OS/BS	3
SIMD [30]	BI due to padding, OS/BS	3
Skein [21]	LBDS	2

Table 1. Features hindering the efficiency of duplexing (same output size as input size) on hash functions, and its cost factor compared to plain hashing for long messages. LBDS = last block domain separation; OS/BS = output size smaller than block size; OT = output transformation; BI = blank iteration(s).

9 Conclusions

We have defined a new construction, namely the duplex construction, and showed that its security is equivalent to that of a sponge function with the same parameters. This construction was then used to give an efficient (single-pass) authenticated encryption mode. We proposed a reseedable pseudo-random number generator as another application of the duplex construction and used it to prove the security of the Overwrite construction. We have showed that the duplex construction inherits the flexibility of the sponge construction in terms of security/speed trade-offs thanks to a new padding function. Finally, we have shown through examples that duplexing with other hash function constructions is not always as efficient as with the sponge construction.

References

1. J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, *SHA-3 proposal BLAKE*, Submission to NIST, 2008.
2. M. Bellare and C. Namprempre, *Authenticated encryption: Relations among notions and analysis of the generic composition paradigm*, Asiacrypt (T. Okamoto, ed.), Lecture Notes in Computer Science, vol. 1976, Springer, 2000, pp. 531–545.
3. M. Bellare and P. Rogaway, *Random oracles are practical: A paradigm for designing efficient protocols*, ACM Conference on Computer and Communications Security 1993 (ACM, ed.), 1993, pp. 62–73.
4. M. Bellare, P. Rogaway, and D. Wagner, *The EAX mode of operation*, in Roy and Meier [40], pp. 389–407.
5. M. Bellare and B. Yee, *Forward-security in private-key cryptography*, Cryptology ePrint Archive, Report 2001/035, 2001, `http://eprint.iacr.org/`.
6. R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin, *SHA-3 proposal: ECHO*, Submission to NIST (updated), 2009.
7. D. J. Bernstein, *CubeHash specification (2.b.1)*, Submission to NIST (Round 2), 2009.
8. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Sponge functions*, Ecrypt Hash Workshop 2007, May 2007, also available as public comment to NIST from `http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html`.
9. ———, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, `http://sponge.noekeon.org/`, pp. 181–197.
10. ———, Keccak *sponge function family main document*, NIST SHA-3 Submission (updated), June 2010, `http://keccak.noekeon.org/`.

11. ———, *On the security of the keyed sponge construction*, Second SHA-3 candidate conference, August 2010.
12. ———, *Sponge-based pseudo-random number generators*, CHES (S. Mangard and F.-X. Standaert, eds.), Lecture Notes in Computer Science, vol. 6225, Springer, August 2010, pp. 33–47.
13. E. Biham and O. Dunkelman, *A framework for iterative hash functions—HAIFA*, Second Cryptographic Hash Workshop, Santa Barbara, August 2006.
14. ———, *The SHAvite-3 hash function*, Submission to NIST (Round 2), 2009.
15. E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau, *Shabal, a submission to NIST's cryptographic hash algorithm competition*, Submission to NIST, 2008.
16. C. De Canniere, H. Sato, and D. Watanabe, *Hash function Luffa: Specification*, Submission to NIST (Round 2), 2009.
17. D. Chang and M. Nandi, *Improved indifferentiability security analysis of chopMD hash function*, Fast Software Encryption (K. Nyberg, ed.), Lecture Notes in Computer Science, vol. 5086, Springer, 2008, pp. 429–443.
18. J. Coron, Y. Dodis, C. Malinaud, and P. Puniya, *Merkle-Damgård revisited: How to construct a hash function*, Advances in Cryptology – Crypto 2005 (V. Shoup, ed.), LNCS, no. 3621, Springer-Verlag, 2005, pp. 430–448.
19. A. Desai, A. Hevia, and Y. L. Yin, *A practice-oriented treatment of pseudorandom number generators*, Advances in Cryptology – Eurocrypt 2002 (L. R. Knudsen, ed.), Lecture Notes in Computer Science, vol. 2332, Springer, 2002, pp. 368–383.
20. M. Dworkin, *Request for review of key wrap algorithms*, Cryptology ePrint Archive, Report 2004/340, 2004, `http://eprint.iacr.org/`.
21. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, *The Skein hash function family*, Submission to NIST (Round 2), 2009.
22. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, *Grøstl – a SHA-3 candidate*, Submission to NIST, 2008.
23. V. D. Gligor and P. Donescu, *Fast encryption and authentication: XCBC encryption and XECB authentication modes*, Fast Software Encryption 2001 (M. Matsui, ed.), Lecture Notes in Computer Science, vol. 2355, Springer, 2001, pp. 92–108.
24. D. Gligoroski, V. Klima, S. J. Knapskog, M. El-Hadedy, J. Amundsen, and S. F. Mjølsnes, *Cryptographic hash function Blue Midnight Wish*, Submission to NIST (Round 2), 2009.
25. M. Gorski, S. Lucks, and T. Peyrin, *Slide attacks on a class of hash functions*, Asiacrypt (J. Pieprzyk, ed.), Lecture Notes in Computer Science, vol. 5350, Springer, 2008, pp. 143–160.
26. S. Halevi, W. E. Hall, and C. S. Jutla, *The hash function Fugue*, Submission to NIST (updated), 2009.
27. C. S. Jutla, *Encryption modes with almost free message integrity*, Advances in Cryptology – Eurocrypt 2001 (B. Pfitzmann, ed.), Lecture Notes in Computer Science, vol. 2045, Springer, 2001, pp. 529–544.
28. T. Kohno, J. Viega, and D. Whiting, *CWC: A high-performance conventional authenticated encryption mode*, in Roy and Meier [40], pp. 408–426.
29. Ö. Küçük, *The hash function Hamsi*, Submission to NIST (updated), 2009.
30. G. Leurent, C. Bouillaguet, and P.-A. Fouque, *SIMD is a message digest*, Submission to NIST (Round 2), 2009.
31. L.Knudsen, C. Rechberger, and S. Thomsen, *The Grindahl hash functions*, FSE (A. Biryukov, ed.), Lecture Notes in Computer Science, vol. 4593, Springer, 2007, pp. 39–57.
32. S. Lucks, *Two-pass authenticated encryption faster than generic composition*, Fast Software Encryption (H. Gilbert and H. Handschuh, eds.), Lecture Notes in Computer Science, vol. 3557, Springer, 2005, pp. 284–298.
33. U. Maurer, R. Renner, and C. Holenstein, *Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology*, Theory of Cryptography - TCC 2004 (M. Naor, ed.), Lecture Notes in Computer Science, no. 2951, Springer-Verlag, 2004, pp. 21–39.
34. NIST, *AES key wrap specification*, November 2001.
35. ———, *NIST special publication 800-38C, recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality*, July 2007.
36. ———, *NIST special publication 800-38D, recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC*, November 2007.
37. P. Rogaway, M. Bellare, and J. Black, *OCB: A block-cipher mode of operation for efficient authenticated encryption*, ACM Trans. Inf. Syst. Secur. **6** (2003), no. 3, 365–403.
38. P. Rogaway, M. Bellare, J. Black, and T. Krovetz, *OCB: A block-cipher mode of operation for efficient authenticated encryption*, CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security (New York, NY, USA), ACM, 2001, pp. 196–205.
39. P. Rogaway and T. Shrimpton, *A provable-security treatment of the key-wrap problem*, Eurocrypt (S. Vaudenay, ed.), Lecture Notes in Computer Science, vol. 4004, Springer, 2006, pp. 373–390.
40. B. K. Roy and W. Meier (eds.), *Fast software encryption, 11th international workshop, FSE 2004, Delhi, India, February 5-7, 2004, revised papers*, Lecture Notes in Computer Science, vol. 3017, Springer, 2004.
41. H. Wu, *The hash function JH*, Submission to NIST (updated), 2009.