Sponge-based pseudo-random number generators

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

[1] STMicroelectronics
[2] NXP Semiconductors

Abstract. This paper proposes a new construction for the generation of pseudo-random numbers. The construction is based on sponge functions and is suitable for embedded security devices as it requires few resources. We propose a model for such generators and explain how to define one on top of a sponge function. The construction is a novel way to use a sponge function, and inputs and outputs blocks in a continuous fashion, allowing to interleave the feed of seeding material with the fetch of pseudo-random numbers without latency. We describe the consequences of the sponge indifferentiability results to this construction and study the resistance of the construction against generic state recovery attacks. Finally, we propose a concrete example based on a member of the Keccak family with small width.

Keywords: pseudo-random numbers, hash function, stream cipher, sponge function, indifferentiability, embedded security device, Keccak

1 Introduction

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. 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 pretty similar to a stream cipher. If the key is unknown, it must be infeasible to infer anything on the key stream, even if it is partially known.

The state of the PRNG must have sufficient entropy, from the point of view of the adversary, so that the prediction of the output bits cannot rely on simply guessing the state. Hence, the seeding material must provide sufficient entropy. Physical sources of randomness usually provide seeding material with relatively low entropy rate due to imbalance of or correlations between bits. To increase entropy, one may use the seeding material from several randomness sources. However, this entropy must be transferred to the finite state of the PRNG. Hence, we need a way to gather and combine seeding material coming from several sources into the state of the PRNG. Loading different seeds into the

PRNG shall result in different output sequences. The latter implies that different seeds result in different state values. In this respect, a PRNG is similar to a cryptographic hash function that should be collision-resistant.

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 seeding 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 seeds.

The remainder of this paper is organized as follows. We continue our introduction with the advantages and limitations of our construction and an illustrative example of a pseudo-random number generator mode of a hash function. We then define the reference model of a reseedable PRNG in Section 2 and specify and motivate our sponge-based construction in Section 3. We discuss the security aspects of our proposal in Section 4 and provide a concrete example in Section 5.

1.1 Advantages and limitations of our construction

With their variable-length input and variable-length output, sponge functions combine in a unified way the functionality of hash functions and stream ciphers. They make therefore a natural candidate for building PRNGs, taking the seeding material as input and producing a sequence of pseudo-random bits as output.

In this paper, we provide a clean and efficient way to construct a reseedable PRNG with a sponge function. The main idea is to integrate in the same construction the combination of the various sources of seeding material and the generation of pseudo-random output bits. The only requirement for seeding material is to be available as bit sequences, which can be presented as such without any additional preprocessing. So both seeding and random generation can work in a continuous fashion, making the implementation simple and avoiding extra iterations when providing additional seeding material.

In the context of an embedded security device, the efficiency and the simplicity of the implementation is important. In our construction we can keep the state size small thanks to two reasons. First, the use of a permutation preserves the entropy of the state (see Section 1.2). Second, we have strong bounds on the expected complexity of generic state recovery attacks (see Section 4.2).

Making sure that the seeding material provides enough entropy is out of scope of this paper. This aspect has been studied in the literature, e.g., [10,16] and is fairly orthogonal to the problem of combining various sources and generating pseudo-random bits.

In our construction, forward security must be explicitly activated. Forward security (also called forward secrecy) requires that the compromise of the current state does not enable the attacker to determine the previously generated pseudo-random bits [2,9]. As our construction is based on a permutation, revealing the state immediately allows the attacker to backtrack the generation up to the previous combination of that state and seeding material. Nevertheless, reseeding

regularly with sufficient entropy already prevents the attacker from going backwards. Also, an embedded security device such as a smartcard in which such a PRNG would be used is designed to protect the secrecy of keys and therefore reading out the state is expected to be difficult. Yet, we propose in Section 4.3 a simple solution to get forward secrecy at a small extra cost. Hence, if forward security is required, one can apply this mechanism at regular intervals.

1.2 Using a hash function for pseudo-random number generation

Sponge functions are a generalization of hash functions and using the latter for generating pseudo-random bits is not new, e.g., [12,14]. For instance, NIST published a recommendation for random number generation using deterministic random bits generators [14]. They specify how to implement a PRNG using a hash function, a keyed hash function, a block cipher or an elliptic curve. When using a hash function H, the state of the PRNG is essentially determined by two values, V and C, each of the size of the input block of H.

- At initialization, both V and C are obtained by hashing the seeding material, a nonce and an optional personalization string. If V and C are larger than the output size of H, a specific derivation function is used to produce a longer digest.
- The pseudo-random bits are produced by hashing V. If more than one output block is requested, further blocks are produced by hashing $V + i$, where i is the index of the produced output block. The value of V is then updated by combining it with, amongst others, $H(V)$ and C. The value C is not modified in this process.
- When reseeding, the new value of V is obtained by hashing the old value of V together with the new seeding material. The value C is derived from the new value of V by hashing.

For a PRNG based on a hash function, there are two aspects we wish to draw attention to.

First, due to the requirements they must satisfy, cryptographic hash function are not injective. Iterating the function, i.e., computing $H(H(\dots H(x))\dots)$ reduces the size of the range resulting in entropy loss. To prevent this, one can for instance keep the original seed along with the evolving state. In the hash function based PRNG specified in [14], the value V evolves by iterated hashing every time output bits are produced, but the value C does not and therefore keeps the full entropy of the seed. This comes at the cost of keeping a state twice the block size of the hash function.

Second, when reseeding, the current state or the original seed must be hashed together with the seeding material. However, the current state V and the seed C are already the result of a hashing process.

The sponge-based construction we propose below addresses these two aspects more efficiently. First, by using a P-sponge, i.e., a sponge function based on a permutation, no entropy is lost when iterating the permutation and this allows

one to have a smaller state for the same security level. Second, the current state of our construction is precisely the state of the sponge function. Hence, reseeding is more efficient than in the example above, as the current state can be reused immediately instead of being hashed again.

Finally, the use of a sponge function for PRNG is conceptually simpler than existing constructions.

2 Modeling a reseedable pseudo-random number generator

We define a reseedable PRNG as a stateful entity that supports two types of requests, in any order:

- *feed* request, `feed`(σ), injects a seed consisting of a non-empty string $\sigma \in \mathbb{Z}_2^+$ into the state of the PRNG;
- *fetch* request, `fetch`(l), instructs the PRNG to return l bits.

The *seeding material* is the concatenation of the σ's received in all `feed` requests.

Informally, the requirements for a reseedable PRNG can be stated as follows. First, its output (i.e., responses to fetch requests) must depend on all seeding material fed (i.e., payload of feed requests). Second, for an adversary not knowing the seeding material and that has observed part of the output, it must be infeasible to infer anything on the remaining part of the output.

To have more formal security requirements, one often defines a reference system that behaves ideally. For sponge functions, hash functions and stream ciphers the appropriate reference system is the random oracle [1]. For reseedable PRNG we cannot just use a random oracle as it has a different interface. However, we define an ideal PRNG as a particular *mode of use* of a random oracle.

The mode we define is the following. It keeps as state the sequence of all feed and fetch requests received, the *history* h. Upon receipt of a feed request `feed`(σ), it updates the history by incorporating it. Upon receipt of a fetch request `fetch`(l), it queries the random oracle with a string that encodes the history and returns the bits z to $z + l - 1$ of its response to the requester, with z the number of bits requested in the fetch requests since the last feed request. Hence, concatenating the responses of a run of fetch requests is just the response of the random oracle to a single query. This is illustrated in Figure 1. We call this mode the *history-keeping* mode with encoding function $e(h)$. The definition of a history-keeping mode hence reduces to the definition of this encoding function.

As the output of the PRNG must depend on the whole seeding material received, the encoding function $e(h)$ must be injective in the seeding material. In other words, for any two sequences of requests with different seeding materials, the two images through $e(h)$ must be different. We call this property *seed-completeness*. With a seed-complete encoding function, the response of the mode to a fetch request corresponds with non-overlapping parts of the response of the random oracle to different input strings. It follows that the PRNG returns independent and a priori uniformly distributed bits.

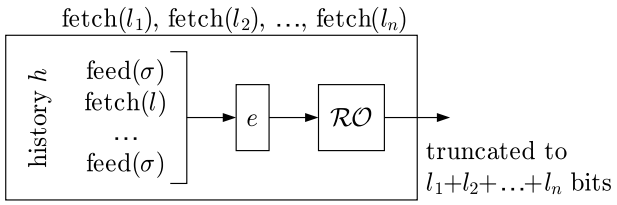

Fig. 1. Response of an ideal reseedable PRNG to fetch requests

We thus propose the following definition of an ideal PRNG. In the sequel, we will use PRNG to indicate a reseedable pseudo-random number generator.

Definition 1. *An* ideal PRNG *is a history-keeping mode calling a random oracle with an encoding function $e(h)$ that is seed-complete.*

3 Constructing a PRNG using a sponge function

In general, the history-keeping mode is not practical as it needs to store all past queries and hence requires ever growing amounts of memory. In this section we will show that if we use a sponge function instead of a random oracle we can define an encoding function that can work with a limited amount of memory.

3.1 The sponge construction

The sponge construction [3] is a simple iterated construction for building a function $\mathcal{S}[f]$ with variable-length input and arbitrary output length based on a fixed-length transformation (or permutation) f operating on a fixed number b of bits. Here b is called the width. A sponge function, i.e., a function implementing the sponge construction provides a particular way to generalize hash functions and has the same interface as a random oracle.

For given values of r and c, the sponge construction operates on a state of $b = r+c$ bits. The value r is called the *bitrate* and the value c the *capacity*. First, all the bits of the state are initialized to zero. The input message is padded and cut into blocks of r bits. The sponge construction then proceeds in two phases: the *absorbing phase* followed by the *squeezing phase*.

- In the 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.
- In the squeezing phase, the first r bits of the state are returned as output blocks, interleaved with applications of the function f. The number of output blocks is chosen at will by the user.

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 [4].

3.2 Reusing the state for multiple feed and fetch phases

It seems natural to translate the feed of seeding material into the absorbing phase and the fetch of pseudo-random numbers into the squeezing phase of a sponge function, as illustrated in Figure 2. However, as such, a sponge function execution has only one absorbing phase (i.e., one input), followed by a single squeezing phase (i.e., one output, of arbitrary length), and thus cannot be used to provide multiple "absorbing" phases and multiple "squeezing" phases.

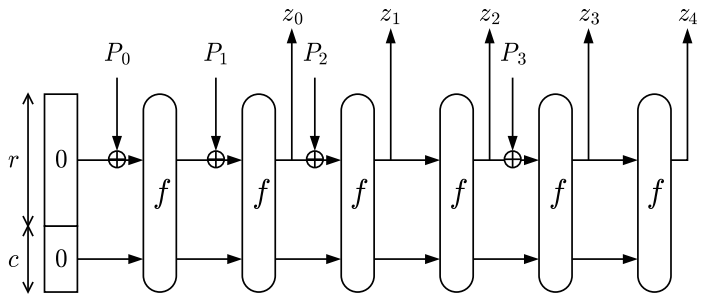

Fig. 2. The sponge construction with multiple feed and fetch phases.

This apparent difficulty is easy to circumvent. Conceptually, it suffices to consider that each time pseudo-random bits are fetched, a different execution of the sponge function is queried with a different input, as illustrated in Figure 3. When entering the squeezing phase of each of these queries (so before pseudo-random bits are requested), one must thus guarantee that the data absorbed so far compose a valid sponge input, i.e., the input is properly padded [3]. This can be achieved by defining an encoding function adapted to the particular sponge.

In the sponge construction, an input message $m \in \mathbb{Z}_2^*$ must be cut into blocks of r bits and padded. Let us denote as $p(m)$ the function that does this, and we assume that this function only appends bits after m (as in the padding of most, if not all, practical hash functions). Let us assume that we wish to reuse the state of the sponge whose input was the string m_1 and from which $l > 0$ output bits have been squeezed. The state of the sponge function at this point is as if the partial message $m'_1 = p(m_1)||0^{r(\lceil l/r \rceil - 1)}$ was absorbed. Note that the zero blocks account for the extra iterations due to the squeezing phase. Restarting the sponge from this point means that the input is going to be a message m_2 of which m'_1 is a prefix.

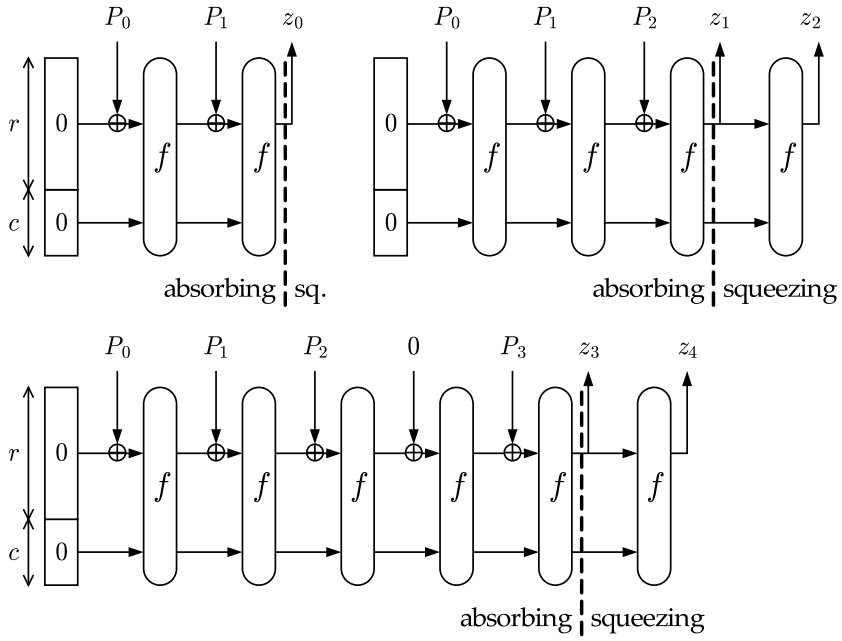

Fig. 3. The multiple feed and fetch phases of Figure 2 can be viewed as a sponge function queried multiple times, each having only one absorbing and one squeezing phase. In this example, $P_0||P_1$, $P_0||P_1||P_2$ and $P_0||P_1||P_2||0^r||P_3$ must all be valid sponge inputs.

3.3 Constructing a reseedable pseudo-random number generator

To define a PRNG formally, we need to specify a seed-complete encoding function $e(h)$ that maps the sequence h of feed and fetch requests onto a string of bits, as in Section 2. The output of $e(h)$ is then used as input to the sponge function. In practice, the idea is not to call the sponge function with the whole $e(h)$ every time a fetch is requested. Instead, the construction uses the sponge function in a cascaded way, reusing the state as explained in Section 3.2. To allow the state of the sponge function to be reused as described above, $e(h)$ must be such that if $h' = h||\texttt{fetch}(l)||\texttt{feed}(\sigma)$, then $p(e(h))||0^{r(\lceil l/r \rceil - 1)}$ is a prefix of $e(h')$.

We now explain how to link a mode to a practical implementation. To make the description easier, we describe a mode with two restrictions. We later discuss how to implement a more elaborate mode without these restrictions. The first restriction is on the length of the seed requests. For a fixed integer k, we require that the length of the seeding material σ in any feed request $\texttt{feed}(\sigma)$ is such that $|p(\sigma)| = kr$. In other words, after padding, the seeding material covers exactly k blocks of r bits. The second restriction is that the first request must be `feed`.

The mode is stateful, and its state is composed of $m \in \mathbb{N}$, the number of bits fetched since the last feed. We start with a new execution of a sponge function, and we set $m = 0$. Depending on the type of requests, the following operations are done on the sponge function on the one hand and on the encoding function $e(h)$ on the other. We denote by e a string that reflects $e(h)$ as the requests are appended to the history h.

- If the request is $\texttt{fetch}(l)$, the following is done.
  - The implementation produces l output bits by squeezing them from the sponge function. Formally, e will be adapted during the next feed request.
  - The value of m is adapted: $m \leftarrow m + l$.
- If the request is $\texttt{feed}(\sigma)$, the following is done.
  - Formally, this feed request triggers a query to the sponge function with e as input. If it is not the first request, e is up-to-date only up to the last feed request. So, the effect of the fetch requests since the last feed request must be incorporated into e, as if e was previously absorbed. First, e becomes $p(e)$ to simulate the padding when switching to the squeezing phase after the previous feed request. Then $\lceil m/r \rceil - 1$ blocks of r zeroes are appended to e to account for the extra calls to the f function during the subsequent fetch requests. Now m is reset: $m \leftarrow 0$. (This part affects only e formally; nothing needs to be done in the implementation.)
  - Then, the implementation absorbs σ. Formally, this is reflected by appending σ to e.
  - Finally, the implementation switches the sponge function to the squeezing phase. This means that the absorbed data must be padded and the permutation f is applied to the state. (Formally, this does not change e, as the padding is by definition performed when switching to the squeezing phase.)

To show that the encoding function is seed-complete, let us demonstrate how to find the seeding material from it. If $e(h)$ is empty, no feed request has been done and the seeding material is the empty string. If $e(h)$ is not empty, it necessarily ends with the fixed amount of seeding material from the last feed request, which we extract. Before that, there can be one or more blocks of r bits equal to zero. This can only come from blocks that simulate fetch requests, as the padding function p would necessarily create a non-zero block. So, we can skip backwards consecutive blocks of zeroes, until the beginning of $e(h)$ is reached or a non-zero block is encountered. In this last case, we can extract the seeding material from the k blocks of r bits and move backwards by the same amount. Finally, we repeat this process until the beginning of $e(h)$ is reached.

The construction, described directly on top of the permutation f, is given in Algorithm 1. For completeness, we also give in Algorithm 2 an implementation of the squeezing phase of a sponge function, although it follows in a straightforward way from the definition [3]. The cost of a feed request is always k calls to the permutation f. Consecutive fetch requests totalling m bits of output cost $\lceil m/r \rceil - 1$ calls to f. So a $\texttt{fetch}(l)$ with $l \leq r$ just after a feed request is free.

Algorithm 1 Direct implementation of the PRNG using the permutation f

$s = 0^{r+c}$
$m = 0$
while requests are received **do**
  if the request is `fetch`(σ) with $|p(\sigma)| = kr$ **then**
    $P_1 || \ldots || P_k = p(\sigma)$
    for $i = 1$ to k **do**
      $s = s \oplus (P_i || 0^c)$
      $s = f(s)$
    end for
    $m = 0$
  end if
  if the request is `fetch`(l) **then**
    Squeeze l bits from the sponge function (see Algorithm 2)
  end if
end while

Algorithm 2 Implementation of the squeezing of l bits from the sponge function

Let a be the number of available bits, i.e., $a = r$ if $m = 0$ or $a = (-m \bmod r)$ otherwise
while $l > 0$ **do**
  if $a = 0$ (we need to squeeze the sponge further) **then**
    $s = f(s)$
    $a = r$
  end if
  Output $l' = \min(a, l)$ bits by taking bits $r - a$ to $r - a + l' - 1$ of the state
  Subtract l' from a and from l, and add l' to m
end while

The restriction of fixed-size feed requests is not essential and can be removed. The description of the mode would be only a bit more complex, but would distract the reader from the aspects of this construction that tightly integrate to a sponge function and its underlying function f. In fact, the restriction of fixed-size feed requests makes it easy to ensure and to show that the encoding function is seed-complete. To allow for variable length seeding materials and retain seed-completeness, some form of padding within the encoding function must be introduced to make sure that the boundaries of the seeding material can be identified. Furthermore, one may have to add a way to distinguish blocks of zero-valued seeding material from zero blocks due to fetch requests. This can be done, e.g., by putting a bit 1 in every block that contains seeding material.

The restriction of the first request being a feed request can be removed, even though it makes little sense generating pseudo-random bits without first feeding seeding material. If the first request is a fetch, the implementation immediately pads the (empty string) input, switches the sponge function to the squeezing phase and produces output bits by squeezing. Formally, in the next feed request, this must be accounted for in e by setting e to $p(\text{empty string}) || 0^{r(\lceil m/r \rceil - 1)}$.

4 Security

Hash functions are often designed in two steps. In the first step, one chooses a mode of operation that relies on a cryptographic primitive with fixed input size (e.g., a compression function or a permutation) and builds a function that can process a message of arbitrary size. If the security of the mode of operation can be proven, it then guarantees that any potential flaw can only come from the underlying cryptographic primitive, and thereby reduces the scope of cryptanalysis.

We proceed similarly to assess the security of the PRNG, in two steps. First, we look at the security of the construction against generic attacks, i.e., against attacks that do not use the specific properties of f. We do this in the following subsections. Then, the security of the PRNG depends on the actual function f and we give an example in Section 5.

4.1 Indifferentiability

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 [11]. The system can use an underlying cryptographic primitive (e.g., a compression function or a permutation) as a public subsystem. For instance, many hash function constructions have been proven to be indifferentiable from a random oracle when using an ideal compression function or a random permutation as public subsystem (e.g., [8]).

By using indifferentiability, one can build a construction that does not have any generic flaw, i.e., any undesired property or attack that does not rely on the specific properties of the underlying primitive.

Theorem 1. *The pseudo-random number generator $\mathcal{P}[\mathcal{F}]$ that uses a permutation $\mathcal{F}$ is (t_D, t_S, N, ϵ)-indifferentiable from an ideal PRNG, for any t_D, $t_S = O(N^2)$, $N < 2^c$ and any ϵ with $\epsilon > N^2/2^{c+1}$ when $1 \ll N$.*

Proof. The proof follows immediately from [4, Theorem 2], where the (t_D, t_S, N, ϵ)-indifferentiability is proven between the sponge construction and a random oracle. In [4, Theorem 2], the adversary has access to two interfaces: one to the permutation $\mathcal{F}$ or its simulator, and one to input a message $m \in \mathbf{Z}_2^*$. In the context of this theorem, the same settings apply, except that the adversary does not have a direct access to the latter interface but only through the encoding function $e(h)$. The same restriction applies both on the side of the sponge construction and on the side of the random oracle. Since the adversary has no better access than in [4, Theorem 2], her probability of success cannot be higher. □

Distinguishing the sponge-based PRNG calling a random permutation from an ideal PRNG defined in Section 2 takes about $2^{c/2}$ operations. In other words, the former is as secure as the latter if c is large enough.

4.2 Resistance against state recovery

Indifferentiability provides a proof of resistance against all possible generic attacks on the construction. However, in practice, we can also look at the resistance of the construction against generic attacks with a specific goal. In this case, the resistance cannot be lower than $2^{c/2}$ but may be higher.

The main purpose of the PRNG is to avoid that an adversary, who has seen some of the generated bits, can predict other values. A way to predict other output bits is to recover the state of the PRNG by observing the generated pseudo-random bits. In fact, since we use a permutation, the adversary can equivalently recover the state at any time during a fetch request. She can also determine the state before or after a feed request if she can guess the seeding material input during that request.

Let the state of a sponge function be denoted as (a, x), where a is the outer part (i.e., the r-bit part output during the squeezing phase) and x represents inner part (i.e., the remaining c bits). Let $(a_0, a_1, \ldots, a_\ell)$ be a sequence of known output blocks. The goal of the adversary is to find a value x_0 such that $f(a_{i-1}, x_{i-1}) = (a_i, x_i)$ for $1 \leq i \leq \ell$ and some values x_i. Notice that once x_0 is fixed, the values x_i, $1 \leq i \leq \ell$ follow immediately. Furthermore, since f is a permutation, the adversary can choose to first determine x_i for some index i and then compute all the other $x_{j \neq i}$ from $(a_i, x_i$ by applying f and f^{-1}.

An instance of the *passive state recovery problem* is given by a vector (a_0, a_1, …, a_ℓ) of r-bit values. We focus on the case where such a sequence of values was actually observed, so that we are sure there is at least one solution. Also, we assume that there is only one solution, i.e., one value x_0. This is likely if $\ell r > c$, and the probability that more than one solution exists decreases exponentially with $\ell r - c$. The adversary wants to determine unseen output blocks, so she wants to have only one solution anyway and will ask for more output blocks to remove any ambiguity.

The adversary can query the permutation f with values (a, x) and get $f(a, x)$ or its inverse to get $f^{-1}(a, x)$. If f is a random permutation, we wish to compute an upper bound on the success probability after N queries.

Theorem 2. *Given an instance of the passive state recovery problem $A = (a_0$, a_1, …, $a_\ell)$ and knowing that there is one and only one solution x_0, the success probability after N queries is at most $N2^{-c}m(A)$, with $m(A)$ the multiplicity defined as*

$$
\begin{aligned}
m(A) &= \max\{m_{\mathrm{f}}(A), m_{\mathrm{b}}(A)\}, \textit{ with} \\
m_{\mathrm{f}}(A) &= \max_{a \in \mathbf{Z}_2^r} |\{i \ : \ 0 \leq i < \ell \wedge a_i = a\}|, \textit{ and} \\
m_{\mathrm{b}}(A) &= \max_{a \in \mathbf{Z}_2^r} |\{i \ : \ 1 \leq i \leq \ell \wedge a_i = a\}|.
\end{aligned}
$$

Proof. Let $F_1(A)$ be the set of permutations f such that there is only one solution to the state recovery problem with instance A. For a given value (a, x), within $F_1(A)$, the inner part of $f(a, x)$ (or $f^{-1}(a, x)$) can be symmetrically chosen among the 2^c possible values as the problem instance does not express any

constraints on the inner parts. In other words, if x is such that the outer part of $f(a, x)$ is b, then for any $x' \neq x$ there exists another permutation $f' \in F_1(A)$ where x' is such that the outer part of $f'(a, x')$ is b too. Such symmetries exist also for multiple inner values, independently of each other, as long as the corresponding outer values are different. E.g., if $a_1 \neq a_2$ and (x_1, x_2) is such that the outer parts of $f(a_i, x_i)$ are b_i for $i = 1, 2$, then for any $(x'_1, x'_2) \neq (x_1, x_2)$ there exists another permutation $f' \in F_1(A)$ where (x'_1, x'_2) verifies the same equality.

Let us first consider that $\ell = 1$. In this case, $m(A) = 1$.

Let $F_1(A, x_0, x_1)$ be the subset of $F_1(A)$ where the value x_0 is the solution and $f(a_0, x_0) = (a_1, x_1)$. The sets $F_1(A, x_0, x_1)$ partition the set $F_1(A)$ into 2^{2c} subsets of equal size identified by x_0 and x_1, or in other words, x_0 and x_1 cut the set in an orthogonal way.

The goal of the adversary is to determine in which subset $F_1(A, x_0, x_1)$ the permutation f is. To do so, she is going to make queries of the form (a_0, x_0) and check if the outer part of $f(a_0, x_0)$ is a_1 (called *forward queries*), or she can make queries to the inverse permutation and check if $f^{-1}(a_1, x_1)$ gives a_0 as outer part (called *backward queries*). As the subsets $F_1(A, x_0, x_1)$ cut $F_1(A)$ orthogonally in x_0 and x_1, forward queries help determine whether x_0 is the solution but without reducing the set of possible values for x_1, and vice-versa for backward queries. So, after N_f forward queries and N_b backward queries, the probability that one of them gives the solution is $1 - (1 - N_\mathrm{f}/2^c)(1 - N_\mathrm{b}/2^c) \leq N/2^c$, where the probability is taken over all permutations f drawn uniformly from $F_1(A)$.

Let us now consider the general case where $\ell > 1$. The reasoning can be generalized in a straightforward way if all the a_i are different, but some adaptations have to be made to take into account the values appearing multiple times. Given a set of indexes $\{i_1, \ldots, i_m\}$ such that $a_{i_1} = a_{i_2} = \cdots = a_{i_m}$, there may or may not be constraints on the possible values that the corresponding inner values $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ can take. For instance, if $a_{i_1-1} \neq a_{i_2-1}$ or if $a_{i_1+1} \neq a_{i_2+1}$, then necessarily $x_{i_1} \neq x_{i_2}$. In another example, A can be periodic, allowing the x_i values to be equal.

Let $i(j, k)$ be a partition of the indexes 0 to ℓ such that $a_{i(j,k)} = a_{i(j',k')}$ iff $j = j'$, i.e., the j index identifies the subsets and the k index the indices within that subset. Let $F_1(A, x_0, x_1, \ldots, x_\ell)$ be the subset of $F_1(A)$ such that $(x_0, x_1, \ldots, x_\ell)$ is the solution. Here, the set $F_1(A)$ is again cut into subsets of equal size if we use the n vectors $(x_{i(j,1)}, \ldots, x_{i(j,m_j)})$ as identifiers, and each of these vectors cut $F_1(A)$ in an orthogonal way. (In general, however, the values x corresponding to identical values a do not cut $F_1(A)$ in an orthogonal way.)

The adversary can make a forward query to check whether $f(a_{i(j,k)}, x_{i(j,k)})$ gives $a_{i(j,k)+1}$ as outer value. Using the same query, she can also check whether $f(a_{i(j,k')}, x_{i(j,k)})$ yields $a_{i(j,k')+1}$ for any other k' (as long as $i(j, k') < \ell$). The same reasoning goes for backward queries: does $f^{-1}(a_{i(j,k')}, x_{i(j,k)})$ yield $a_{i(j,k')-1}$ for any k' (as long as $i(j, k') > 0$). So, a forward (resp. backward) query can count as up to $m_\mathrm{f}(A)$ (resp. $m_\mathrm{b}(A)$) chances to hit the correct outer value. After N queries, the probability that one of them gives the solution is at most

$m(A)N/2^c$, where the probability is taken over all permutations f drawn uniformly from $F_1(A)$. □

The previous theorem also imposes an upper bound on the success probability of preimage attacks, generically against a sponge function. This follows from the fact that finding a preimage implies that the state can be recovered.

This theorem covers the case of a passive adversary who observes output blocks. Now, the PRNG implementation could allow seeding material to be provided from outside, hence allowing an active adversary to absorb blocks of his choice. This case is covered in the next theorem. We assume that the adversary controls the blocks b_i that are injected at each iteration, i.e., the PRNG computes $f(a_i \oplus b_i, x_i) = (a_{i+1}, x_{i+1})$ and the adversary observes a_{i+1}. Now an instance of the problem is also determined by the injected blocks $B = (b_0, b_1, \ldots, b_\ell)$.

Theorem 3. *Given an instance of the active state recovery problem $A = (a_0, a_1, \ldots, a_\ell)$, $B = (b_0, b_1, \ldots, b_\ell)$ and knowing that there is one and only one solution x_0, the success probability after N queries is at most $N2^{-c}\ell$.*

Proof. The reasoning is the same as in Theorem 2, except that the queries are slightly different. In a forward query, the adversary checks if the outer part of $f(a_i \oplus b_i, x_i)$ is a_{i+1}. In a backward query, she checks if the outer part of $f^{-1}(a_i, x_i)$ is $a_{i-1} \oplus b_{i-1}$. Another difference is that now the forward multiplicity to be considered is

$$m_{\mathrm{f}}(A, B) = \max_{a \oplus b \in \mathbf{Z}_2^r} |\{i \ : \ 0 \leq i < \ell \wedge a_i \oplus b_i = a \oplus b\}|,$$

as one forward query can be used to check inner values at up to $m_{\mathrm{f}}(A, B)$ indexes at once. Furthermore, the adversary can influence the multiplicity, e.g., by making sure $a_i \oplus b_i$ is always the same value. So $m(A) \leq \ell$ and the success probability after N queries is at most $N2^{-c}\ell$. □

An active attacker can use $\ell = 2^{c/2}$ output blocks and the complexity of her attack is going to be $N = 2^{c/2}$, a result in agreement with the indifferentiability result of Theorem 1. However, here we can distinguish between the *data complexity*, i.e., the available number of output data of the PRNG and the *time complexity*, the number of queries to f, of the attack. If the implementation of a PRNG limits the number of output blocks to some value $\ell_{\max} < 2^{c/2}$, the time complexity of a generic attack is bounded by $N = 2^c/\ell_{\max} > 2^{c/2}$.

4.3 Forward security

Our construction does not inherently provide forward security, but it can be explicitly triggered by using the following technique. One can fetch $r' \leq r$ bits out of the current PRNG and feed them immediately afterwards. This way, the r' bits of the outer part of the state will be set to zero, making this process an irreversible step. By repeating this process $\lceil c/r' \rceil$ times, the adversary has to guess at least c bits when evaluating the state backwards. This process can be activated, for instance, at regular intervals.

5 A concrete example with KECCAK

KECCAK is a family of sponge functions submitted to the SHA-3 contest organized by NIST [13,6,7]. The family uses seven permutations ranging from a width of 25 bits to a width of 1600 bits. While the SHA-3 proposal uses KECCAK-f[1600] only, other members of the family with a smaller width can be interesting in the context of a PRNG in an embedded device. For instance, KECCAK[$r = 96, c = 104$] and KECCAK[$r = 64, c = 136$] both use KECCAK-f[200] as underlying permutation. This permutation is suitable for devices with scarse resources as the state can be stored in only 25 bytes. In hardware it can be built in a very compact core and in software it can be implemented with bitwise Boolean instructions and rotations within bytes only. These sponge functions can produce 96 and 64 pseudo-random bits, resp., per call to KECCAK-f[200].

In terms of security, KECCAK follows what is called the hermetic sponge strategy [7,5]. This means that the KECCAK-f permutations are designed with the target that they cannot be distinguished from a randomly-chosen permutation. Biased output bits on one of the KECCAK members, for instance, would imply a distinguisher on the underlying permutation KECCAK-f and would therefore contradict the design strategy.

Against passive state recovery attacks in the generic case, Theorem 2 proves a resistance of $2^c/m(A)$. If a sequence of ℓr output bits is known, the expected value of $m(A)$ is close to 1 unless $\ell > 2^{r/2}$. One can limit to $r2^{r/2}$ the number of output bits between times where the state has gained at least c bits of fresh seeding material. This way, KECCAK[$r = 96, c = 104$] and KECCAK[$r = 64, c = 136$] provides a resistance of about 2^{104} and 2^{136}, resp., against state recovery, at least as long as no distinguisher on KECCAK-f[200] is found.

If the PRNG allows the user to provide seeding material, active state recovery attacks must also be considered. Here, the implementation can limit, e.g., to $\ell_{\max} = 2^{24}$ or 2^{32} output blocks before the state has again been fed with c bits of fresh seeding material. In this case, KECCAK[$r = 64, c = 136$] provides a resistance of about 2^{112} and 2^{104}, respectively.

We have implemented our PRNG based on KECCAK[$r = 96, c = 104$] and KECCAK[$r = 64, c = 136$] and passed the statistical tests proposed by NIST [15]. The tests were performed on 200 sequences of 10^6 bits each. The sequences were generated by squeezing 2×10^8 bits after providing the empty string as input, namely $\lfloor \text{KECCAK}[r = 96, c = 104]() \rfloor_{2\times 10^8}$ and $\lfloor \text{KECCAK}[r = 64, c = 136]() \rfloor_{2\times 10^8}$.

6 Conclusions

We have presented a construction for building a reseedable pseudo-random number generator using a sponge function. This construction is efficient in terms of memory use and processing, and inherits the provable security properties of the sponge construction. We have provided bounds on generic state recovery attacks allowing the use of a small state. We have given a concrete example of such a PRNG based on KECCAK with a state of only 25 bytes that is particularly suitable for embedded devices.

References

1. 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.
2. M. Bellare and B. Yee, *Forward-security in private-key cryptography*, Cryptology ePrint Archive, Report 2001/035, 2001, `http://eprint.iacr.org/`.
3. 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`.
4. ———, *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.
5. ———, *Cryptographic sponges*, 2009, `http://sponge.noekeon.org/`.
6. ———, Keccak *specifications, version 2*, NIST SHA-3 Submission, September 2009, `http://keccak.noekeon.org/`.
7. ———, Keccak *sponge function family main document*, NIST SHA-3 Submission (updated), September 2009, `http://keccak.noekeon.org/`.
8. 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.
9. 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.
10. N. Ferguson and B. Schneier, *Practical cryptography*, John Wiley & Sons, 2003.
11. 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.
12. NIST, *Federal information processing standard 186-2, digital signature standard (DSS)*, May 1994.
13. ———, *Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family*, Federal Register Notices **72** (2007), no. 212, 62212–62220, `http://csrc.nist.gov/groups/ST/hash/index.html`.
14. ———, *NIST special publication 800-90, recommendation for random number generation using deterministic random bit generators (revised)*, March 2007.
15. ———, *NIST special publication 800-22, a statistical test suite for random and pseudorandom number generators for cryptographic applications (revision 1)*, August 2008.
16. J. Viega, *Practical random number generation in software*, ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference (Washington, DC, USA), IEEE Computer Society, 2003, p. 129.