Sponge Functions

Guido Bertoni[1], Joan Daemen[1], Michaël Peeters[2], and Gilles Van Assche[1]
gro.noekeon@noekeon.org

[1] STMicroelectronics
[2] NXP Semiconductors

Abstract. A good cryptographic hash function should behave like a random oracle: it should not have weaknesses that a random oracle does not have. Due to the existence of inner collisions, iterated hash functions can never satisfy this ideal. We propose a construction with a finite state called a *sponge* and show that a random sponge can only be distinguished from a random oracle due to inner collisions. We evaluate the strength of random sponges by computing the probability of success for a number of attacks as a function of their workload and show that these results shed a new light on the classical Merkle-Damgård construction. We propose to use random sponges of given parameters as a reference for specifying security claims for hash functions, but also MAC functions and some types of stream ciphers. The main goal of sponge functions is for designers to be able to formulate a compact security claim.

1 Introduction

When designing a hash function, it is important to know which security criteria the result must satisfy, and when publishing it, the paper should state the security criteria it claims to satisfy. The established security criteria for a cryptographic hash function are collision-resistance, pre-image resistance and 2nd pre-image resistance [15]. Often, designers claim lower bounds for the complexity of the three corresponding attacks. In many cases, however, no explicit claims are made and the hash function is supposed to offer a security level implied by the length of its digest. The problem with these criteria is that they do not express what we have come to expect of a cryptographic hash function. Some applications require that a hash function is correlation-free [1] or resists length-extension [18]. More recently, a series of attacks [9, 10, 5, 12] has shown that certain hash function constructions do not offer as much security as expected, leading to the introduction of yet other criteria, such as *chosen target forced prefix preimage resistance*. As was already predicted in [1], there is no reason to assume that no new criteria will appear, so the design of a hash function seems like a moving target.

Remarkably, a random oracle [3] is a theoretical construction that satisfies all known security criteria for hash functions and it seems hard to imagine that new security criteria will be introduced that a random oracle does not satisfy. Hence, we could replace all security criteria by a single one: *a good hash function behaves as a random oracle*. But what does this mean?

Informally speaking, a random oracle maps a variable-length input message to an infinite output string. It is completely random, i.e., the produced bits are uniformly and independently distributed. The only constraint is that identical input messages produce identical outputs. A hash function produces only a fixed number of output bits, say, n bits. So, a hash function should behave as a random oracle whose output is truncated to n bits. In general, it is easy to compute the resistance of a random oracle (truncated to n bits) to certain attacks. For instance, the expected number of calls to the oracle to generate a collision is of the order of $2^{n/2}$. To find a (second) pre-image, this number is 2^n. The hash function is then considered broken if someone finds an attack on the hash function with a complexity smaller than for a random oracle.

Most practical hash functions are iterated. They operate on a chaining value, which is iteratively modified by a function taking the message as an argument. This is a very convenient property, as the whole message can be hashed on the fly. For instance, a network application can hash the stream of data as it comes, without the need to store it into memory.

Iterated hash functions have *state collisions*, that is, collisions in the chaining value. The existence of state collisions yields properties that do not exist for random oracles. For instance, assume that M_1 and M_2 are two messages that form a state collision in an iterated hash function. Then, for any suffix N, the messages $M_1|N$ and $M_2|N$ will produce identical hash values. A random oracle does not have this property: even if M_1 and M_2 produce the same hash value (of finite length n), $M_1|N$ and $M_2|N$ produce hash values that are independent of the hash value obtained from M_1 and M_2. Note that the state collisions are not a problem per se, but rather the fact that they lead to the described externally visible behaviour.

In the light of state collisions, the claimed reference model cannot be a random oracle for iterated hash functions. It is an unreachable goal for an iterated hash function to be as strong as a random oracle. There are two ways to address this problem. First, one can abandon iterated hash functions and use non-streamable hash functions such as the zipper hash construction [14]. This may indeed solve the problem but may be unsuitable for many applications of hash functions since the entire message must be available in memory.

A second approach is to stick to iterated hash function constructions and learn to live with state collisions. This is the approach we follow in this paper. In Section 2, we define a new reference model called sponge functions. In Section 3, we show that a random sponge can only be distinguished from a random oracle due to inner collisions. In Section 4, we evaluate the strength of random sponges by computing the probability of success for a number of operations as a function of their workload. In Section 5, we evaluate the strength of random sponges when being used as a hash function and compare with classical hash function constructions as applied in MD4-like [17] designs. In Section 6, we discuss the issue of choosing the type and parameter values when a random sponge is used as reference for expressing security claims. In Section B, we explain how a sponge can be used as a reference for MAC function and as a stream cipher.

2 Definitions

2.1 Sponge function

A sponge has the same interface as a random oracle: it takes a variable-length input and produces an infinite-length output. Before defining it, let us list the necessary ingredients.

- Let $\mathcal{A}$ be a group, which we call the *alphabet group*; the set $\mathcal{A}$ will represent both input and output characters. The group operation is denoted by a $+$ and the neutral element by 0. Note that, in this context, a “character” can represent anything between a bit ($\mathcal{A} = \mathbf{Z}_2$), a trit ($\mathcal{A} = \mathbf{Z}_3$) and a block of n bits ($\mathcal{A} = \mathbf{Z}_2^n$).
- Let $\mathcal{C}$ be a finite set whose elements will represent the inner part of the state of a sponge.
- Let $0 \in \mathcal{C}$ be an arbitrary element of $\mathcal{C}$, which will be part of the initial value of the state of a sponge.

- Let $p(m)$ be a mapping from the set of messages m to strings of characters of $\mathcal{A}$. The mapping $p(m)$ must be injective, and must be such that $|p(m)| \geq 1$ and that the last character of $p(m)$ is never 0.

Note that the set of messages m can remain abstract, as the sponge will only process strings of characters. Consider the example of messages that are binary strings and $\mathcal{A} = \mathbf{Z}_2^n$. In this case, the mapping $p(m)$ can be reversible padding: a single bit 1 is appended followed by the minimal number of bits 0 such that the length of the result is a multiple of n.

Definition. A *sponge function* takes as input a variable-length string p of characters of $\mathcal{A}$ and produces an infinite output string z of characters of $\mathcal{A}$. It is determined by a transformation f of $\mathcal{A} \times \mathcal{C}$. The input of the sponge function is p, a string of length $|p| \geq 1$ that does not end with 0; the individual characters of p are denoted $p_i \in \mathcal{A}$, for $0 \leq i < |p|$.

The sponge function has an internal state $S = (S_\mathcal{A}, S_\mathcal{C}) \in \mathcal{A} \times \mathcal{C}$, whose initial value is $(0, 0)$, where $0 \in \mathcal{A}$ is the neutral element of $\mathcal{A}$ and $0 \in \mathcal{C}$ is as described above. To evaluate the sponge function, one proceeds in two phases:

- **Absorbing:** For each input character p_i, the state is updated as
$$S \leftarrow f(S_\mathcal{A} + p_i, S_\mathcal{C}).$$
- **Squeezing:** The infinite-length output z is produced one character $z_j \in \mathcal{A}$ at a time by evaluating
$$z_j = S_\mathcal{A},$$
and updating the state as
$$S \leftarrow f(S).$$

Definition. The *rate* of a sponge is

$$r = \log_2 A, \text{ with } A = |\mathcal{A}|.$$

Definition. The *capacity* of a sponge is

$$c = \log_2 C, \text{ with } C = |\mathcal{C}|.$$

Definition. A *binary sponge* is a sponge with rate equal to 1 and an integer capacity.

A useful notation for the sponge function consists in defining the S_f function. For a given input string p, $S_f[p]$ denotes the value of the state obtained after absorbing p. If $S = S_f[p]$, we call p a *path* to S (under f). The $S_f = (S_{\mathcal{A},f}, S_{\mathcal{C},f})$ function is defined by the following recursion:

$$S_f[\text{empty string}] = (0, 0),$$
$$S_f[x|a] = f(S_f[x] + a) \text{ for any string } x \text{ and any character } a,$$

where the symbol $|$ denotes the concatenation and $S + a$ is defined as $S + a = (S_\mathcal{A} + a, S_\mathcal{C})$. In general, the j-th character of the output is

$$z_j = S_{\mathcal{A},f}[p|0^j],\ j \geq 0,$$

where 0^j denotes a string with j consecutive characters equal to 0.

The S_f function can be used to express the state that the sponge traverses both as it absorbs an input p and as it is being squeezed. The traversed states are $S_f[p']$ for any p' prefix of $p|0^\infty$, including the empty string.

This allows us to explain the rationale behind the $p(m)$ function. First, the fact that the mapping is injective and $p(m)$ never ends with 0 ensures that we cannot have two messages m_1, m_2 that will yield the same input to the S_f function when the sponge is squeezed, i.e.,

$$(m_1, j) \neq (m_2, k) \ \Rightarrow \ p(m_1)|0^j \neq p(m_2)|0^k.$$

Second, the requirement on the length, namely, $|p(m)| \geq 1$, ensures that the f function is evaluated at least once. Otherwise, if the input is the empty string, we know that the first output character $S_{\mathcal{A},f}[\text{empty string}] = 0$ independently of f, which is not very satisfactory.

Note that as opposed to most hash functions, a sponge function generates infinite output strings like a random oracle. This makes it suited to also serve as reference for stream ciphers and so-called mask generation functions [13, 8].

2.2 State collisions and inner collisions

Definition. A *state collision* is a pair of different paths $p \neq q$ to the same state: $S_f[p] = S_f[q]$.

Depending on where the state collision occurs, it models different effects of the finite internal state. State collisions obtained during the absorbing part may lead to identical hash function values: $S_f[p] = S_f[q]$ implies that the squeezing part will give the same output values $S_f[p|0^j] = S_f[q|0^j]$ for all j. State collisions can also model cycles in the output sequence: if for some p and d we have $S_f[p] = S_f[p|0^d]$, the output sequence displays periodicity.

Definition. An *inner collision* is a pair of two different paths $p \neq q$ to the same inner state: $S_{\mathcal{C},f}[p] = S_{\mathcal{C},f}[q]$.

Clearly, a state collision on $p \neq q$ implies an inner collision on $p \neq q$. The converse is not true. However, it is very easy to produce a state collision from an inner collision. Given $p \neq q$ such that $S_{\mathcal{C},f}[p] = S_{\mathcal{C},f}[q]$, one can produce a state collision on $p|a \neq q|b$ for any $a, b \in \mathcal{A}$ that satisfy $S_{\mathcal{A},f}[p] + a = S_{\mathcal{A},f}[q] + b$.

2.3 Random sponges

For a given choice of the group $\mathcal{A}$, the set $\mathcal{C}$ and the initial value $(0, 0)$, the mapping f entirely determines the sponge function. There are thus $(AC)^{AC}$ possible such sponge functions.

Definition. A *random transformative sponge* (or *T-sponge* for short) with given $\mathcal{A}$, $\mathcal{C}$ and $0 \in \mathcal{C}$ is a sponge function drawn randomly and uniformly from the set of the $(AC)^{AC}$ sponge functions.

For reasons that become clear in the sequel, we need to distinguish the subset of $(AC)!$ sponge functions for which f is a permutation.

Definition. A *random permutive sponge* (or *P-sponge* for short) with given $\mathcal{A}$, $\mathcal{C}$ and $0 \in \mathcal{C}$ is a sponge function drawn randomly and uniformly from the set of the $(AC)!$ sponge functions for which f is a permutation.

We will use the term random sponge when we make statements that are true for both a random T-sponge and a random P-sponge.

3 Distinguishing a random sponge

We now show that if a random sponge is used in a black-box setting, an adversary can only distinguish it from a random oracle by the presence or absence of inner collisions.

The setting is as follows. The adversary has access to a black box, which is either a random sponge (RS) (with a given $\mathcal{A}$, $\mathcal{C}$ and $0 \in \mathcal{C}$) or a random oracle (RO). The a priori probability of the black box being either a RO or a RS is $\frac{1}{2}$. The adversary queries the black box (BB) as many times as she wishes, even adaptively, by sequentially asking for a set of messages $m^{(1)} \ldots m^{(q)}$ the first $|z^{(i)}|$ characters of output:

$$\begin{aligned} z^{(1)} &= \mathrm{BB}(p^{(1)}), \\ z^{(2)} &= \mathrm{BB}(p^{(2)}), \\ &\ldots \\ z^{(q)} &= \mathrm{BB}(p^{(q)}), \end{aligned}$$

with $p^{(i)} = p(m^{(i)})$. She then has to guess whether the black box is a random oracle or a random sponge.

Let us precisely relate the queries and the absence of inner collision. For a given set of queries, a random sponge traverses some states when it absorbs the input strings and when it is then being squeezed. There may be states that are equally traversed for different queries, e.g., if $p^{(i)}$ and $p^{(j)}|0^{|z^{(j)}|-1}$ have a common prefix. We denote the set of paths to states traversed during the distinguishing experiment by $\mathcal{P}$. We have:

$$\mathcal{P} = \left\{ x \text{ is a prefix of } p^{(i)}|0^{|z^{(i)}|-1} \text{ for some } 1 \le i \le q \right\}.$$

In the context of a given set of queries, no inner collision means that

$$\nexists p \neq q \in \mathcal{P} \ : \ S_{\mathcal{C},f}[p] = S_{\mathcal{C},f}[q].$$

The following theorem on the output of a random sponge in the case that there are no inner collisions holds both for random T-sponges and random P-sponges:

Theorem 1. *The output characters returned by a random sponge to a sequence of queries are uniformly and independently distributed if no inner collisions occur during the queries.*

Proof. Consider the output character $z_j^{(i)}$ of the i-th query: $S_{A,f}[x]$ with $x = p^{(i)}|0^j$ and let $\mathcal{P}^x$ be the set of paths to the states traversed in the queries 1 to $i-1$ and in the current query for the previous output characters. We denote the set of states and inner states corresponding to $\mathcal{P}^x$ by $\mathcal{S}^x$ and $\mathcal{S}_C^x$ respectively.

The requirement that no inner collision takes place during the generation of the output character $S_{A,f}[x]$ restricts the value of the inner state $S_{C,f}[x]$ to be different from all values in $\mathcal{S}_C^x$.

For a random T-sponge the value of $S_f[x]$ must be in $\mathcal{A} \times (\mathcal{C} \setminus \mathcal{S}_C^x)$ due to this requirement. By construction these values are equiprobable. For a random P-sponge the invertibility of f imposes that $S_f[x]$ must be different from all states traversed already (except $(0,0)$), so here the set $S_f[x]$ is chosen from is $(\mathcal{A} \times (\mathcal{C} \setminus \mathcal{S}_C^x)) \setminus \mathcal{S}^x$. Using $\mathcal{S}^x \subset \mathcal{A} \times \mathcal{S}_C^x$ this can be simplified to $\mathcal{A} \times (\mathcal{C} \setminus \mathcal{S}_C^x)$. Hence in both cases all values in $\mathcal{A}$ are equiprobable for $S_{\mathcal{A},f}[x]$ and independent of the states previously traversed. □

In the distinguishing experiment, the adversary may distinguish the random sponge by the consequences of a state collision. Specifically, a state collision at the end of the absorbing phase or during the squeezing phase leads to collisions in the output strings. As a state collision implies an inner collision, we can upper bound the success probability of a distinguishing attack by the probability an inner collision occurs in a given query.

The success probability depends on the guessing rule used by the adversary. Let R_{RS} be the set of observations for which she guesses that the black box is a random sponge. The probability of success reads

$$P(\text{success}) = \frac{1}{2} + \frac{1}{2} \sum_{r \in R_{\text{RS}}} P(r \text{ observed}|\text{RS}) - P(r \text{ observed}|\text{RO}).$$

To maximize the success probability, the adversary has to choose

$$R_{\text{RS}} = \{r \ : \ P(r \text{ observed}|\text{RS}) \geq P(r \text{ observed}|\text{RO})\}.$$

The only thing the adversary can do is to base his decision on the detection of an inner collision. We know that $P(\text{IC}|\text{RO}) = 0$ by definition and $P(\text{no IC}|\text{RS}) = 1 - P(\text{IC}|\text{RS}) < 1$ so that $P(\text{no IC}|\text{RS}) < P(\text{no IC}|\text{RO})$, and the optimal detection rule comes down to $R_{\text{RS}} = \{\text{IC detected}\}$. We have thus proven the following theorem.

Theorem 2. *The probability $P(\textit{success})$ of correctly distinguishing a random sponge from a random oracle is upper bounded as*

$$P(\textit{success}) \leq \frac{1}{2} + \frac{1}{2} P(\textit{IC}|\textit{RS}).$$

The value of $P(\text{IC}|\text{RS})$ depends on A, C and $N = |\mathcal{P}| \leq \sum_i |p^{(i)}| + |z^{(i)}|$, the total number of input and output characters in the query. As the distinguishing experiment can be emulated by using the experiment in Section A.2, the success probability for inner collisions given in Section 4 forms an upper bound for the distinguishing attack.

4 Intrinsic strength of random sponges

Indistinguishability in the black box model is not sufficient. When being used as a hash function, a sponge must offer (collision, (2nd-)preimage, etc.) resistance against adversaries who can see the inner state. More particularly, the function must not have any externally visible weaknesses even with respect to an adversary who can see the inner state.

With the purpose to quantify the resistance of random sponges against such adversaries, we compute the success probabilities of four operations that are not by themselves attacks but that can be used in the context of attacks and as such provide upper bounds to the resistance a sponge function can offer:

- Inner collision: finding two different paths $p \neq q$ to the same inner state: $S_{\mathcal{C},f}[p] = S_{\mathcal{C},f}[q]$.
- Path to an inner state: given $S_{\mathcal{C}}$, finding a path p to $S_{\mathcal{C}}$, i.e., $S_{\mathcal{C},f}[p] = S_{\mathcal{C}}$.
- Output cycles: finding an input string p and an integer d such that $S_f[p] = S_f[p|0^d]$.
- Binding an output string to a state: given a string $t = t_0, t_1, \dots t_m$, finding a state value S such that the sponge generates t as output, i.e., $S_{\mathcal{A}} = t_0$, $f_{\mathcal{A}}(S) = t_1$, $f_{\mathcal{A}}(f(S)) = t_2 \dots$, $f_{\mathcal{A}}(f^{m-1}(S)) = t_m$. Here we can distinguish two cases:

- Short output string ($mr < c$): the number of possible output strings of $m+1$ characters is below the number of possible inner states. It is likely that an inner state value can be found, but the expected number of solutions is $\approx 2^{c-mr}$.
- Long output string ($mr > c$): the number of possible output strings of $m+1$ characters is above the number of possible inner states. For a randomly chosen output string, the probability that an inner state value may be found is $\approx 2^{c-mr}$. If one is found, it is likely that the inner state value is unique.

Note that the applicability of these operations is due to the fact that a sponge operates on a finite state and that subsequently they also apply to iterated constructions such as iterated hash functions and certain types of MAC functions and stream ciphers. They do not apply to a random oracle.

We adopt the following model. In the beginning, the adversary has no information about f. The only way she can gain information on f is to make calls to f (and f^{-1}). We obtain expressions for the optimal probability of success $P(\text{success})$ as function of N, where N is the number of calls the adversary can make to f in the case of a T-sponge and the total number of calls she can make to f and f^{-1} in the case of a P-sponge. This probability is equal to the number of transformations (or permutations) f for which the attack has succeeded, divided by the total number of transformations (or permutations) of given dimensions. So a success probability of 1 % means that for 99 % of the possible choices of f the attack does not work.

The expressions for $P(\text{success})$ for the different operations are of the form $1 - \exp(f(N))$ with $f(N)$ a polynomial in N of degree one or two. To simplify notation, we define the cost function $c_p(N)$ of an attack by $c_p(\text{success}) = -\log(1 - P(\text{success}))$. This gives:

$$P(\text{success}) = 1 - \exp\left(-c_p(\text{success})\right) .$$

For values of N such that $c_p(\text{success}) \ll 1$ we can use the $\log(1+\epsilon)$ approximation: $\log(1+\epsilon) \approx \epsilon$ when $\epsilon \ll 1$, yielding:

$$P(\text{success}) \approx c_p(\text{success}) .$$

For the derivation of the success probabilities and the optimum attack strategies, we refer to Appendix A. Table 1 lists the resulting cost functions for the four operations and the two types of random sponges for large values of C. This is justified as small values of C lead to weak sponge functions.

Case	Inner collision	path finding	output cycle	output binding	
				$rm > c$	$rm < c$
T-sponge	$\frac{N(N+1)}{2C}$	$\frac{N}{C}$	$\frac{N(N+1)}{2AC}$	$\frac{A-1}{A}\frac{N}{C}$	$\frac{A-1}{A}\frac{N}{A^m}$
P-sponge	$\frac{N(N+1)}{2C} - \frac{N(N-1)}{2AC}$	$\frac{N(N+4)}{4C} - \frac{N^2}{4AC}$	$\frac{N}{AC}$	$\frac{A-1}{A}\frac{N}{C}$	$\frac{A-1}{A}\frac{N}{A^m}$

Table 1. Cost functions for different operations.

When we consider values of N that are much larger than 1, we can neglect the linear terms in the cost functions that are quadratic. Moreover we can express the cost functions in terms of the capacity $c = \log_2(C)$ and rate $r = \log_2(A)$ and let $y = \log_2(N)$. Table 2 gives the resulting cost functions for binary sponges ($r = 1$) and high rate ($r \gg 1$) sponges. The

Case	Inner	path	output	output binding	
	collision	finding	cycle	$rm > c$	$rm < c$
High rate T-sponge	$2^{2y-(c+1)}$	2^{y-c}	$2^{2y-(c+r+1)}$	2^{y-c}	2^{y-rm}
Binary T-sponge	$2^{2y-(c+1)}$	2^{y-c}	$2^{2y-(c+2)}$	$2^{y-(c+1)}$	2^{y-m}
High rate P-sponge	$2^{2y-(c+1)}$	$2^{2y-(c+2)}$	$2^{y-(c+r)}$	2^{y-c}	2^{y-rm}
Binary P-sponge	$2^{2y-(c+2)}$	$2^{2y-(c+3)}$	$2^{y-(c+1)}$	$2^{y-(c+1)}$	2^{y-m}

Table 2. Cost functions for different operations with $y = \log_2(N)$.

work factor W, the expected number of calls N for the attack to succeed, is given by:

$$W = \sum_{N=1}^{\infty} N(P(N) - P(N-1)) \ .$$

If we approximate $P(N)$ by a continuous function and fill in the cost function, this becomes

$$W = \int_0^{\infty} x \frac{dP}{dx} dx = \int_0^{\infty} x \frac{dc_p(x)}{dx} exp(-c_p(x)) dx \ .$$

Filling in the cost functions listed in Table 2 leads to integrals that can be readily solved. For the linear cost functions, i.e., $c_p(y) = 2^{y-x}$, we obtain $W = 2^x$. For the quadratic cost functions, i.e., $c_p(y) = 2^{2y-x}$, we obtain $W = \sqrt{\pi} 2^{x/2} \approx 2^{1+x/2}$.

Clearly, the most important parameter is the capacity c. The impact of the rate r on the success probabilities is rather limited, with the exception of the detection of output cycles. The difference in resistance between a random T-sponge and a random P-sponge is mainly in path finding and in the length of output cycles. In a random T-sponge, finding a path has expected workload 2^c, while in a random P-sponge, this is only about $2^{2+c/2}$. On the other hand, a random T-sponge is expected to end up in a cycle after about $2^{(c+r+3)/2}$ characters while for a random P-sponge this is 2^{c+r-1}.

5 Sponge as a hash function

We will now consider a number of classical hash function attacks and show how the operations discussed in the previous section limit the resistance of a sponge against these attacks. For simplicity, we consider the case of a high rate r and use y to indicate the workload $N = 2^y$ of the attack. We each time compare with the behaviour of a random oracle where its output is truncated to n bits. It is important to distinguish between n, the digest length in bits, and c, the capacity.

5.1 Output collisions

If we have an inner collision p, q, we can have a state collision with $p|a, q|b$, for any a and b that verify $S_{\mathcal{A},f}[p] + a = S_{\mathcal{A},f}[q] + b$. Then, any pair of inputs $p|a|m, q|b|m$ leads to an output collision, independent of the digest length n. In a random sponge, the expected workload to generate an inner collision is of the order $2^{(c+3)/2}$. In a random oracle truncated to n bits, the expected workload to generate an output collision is of the order $2^{(n+3)/2}$. So, a random sponge truncated to n bits with $n < c$ offers a similar level of resistance against output collisions than a random oracle truncated to n bits. If $n > c$, the best strategy to generate an output collision is to use an inner collision; if $n < c$, going via an inner collision does not lead to a smaller expected workload.

As for multicollisions [9], an 2^s-fold multicollision in a random sponge can be realized by the chaining of s inner collisions and hence has expected workload $s2^{(c+3)/2}$. For a truncated random oracle this complexity is of the order $2^{n(2^s-1)/2^s}$. So taking $c > 2n$, a random sponge is not weaker than a random oracle in this respect.

5.2 2nd Pre-image

Assume we are looking for a 2nd pre-image for a message p of length ℓ. In a sponge, we have a 2nd pre-image if we can find a 2nd path to the inner state $T = S_{\mathcal{C},f}[p']$ with p' the prefix of p where only the last character $p_{\ell-1}$ is removed. Given this path q, we have $S_f[q|x] = S_f[p]$ with $x = -S_{\mathcal{A},f}[q] + S_{\mathcal{A},f}[p'] + p_{\ell-1}$. We have computed the cost function for this problem in Appendix A.3 for a random T-sponge and we found an expected workload of the order $2^c/\ell$ if $\ell < 2^{c/2}$. Note that its expected workload must be at least that of generating an inner collision as a 2nd pre-image implies an inner collision.

In a truncated random oracle the expected workload is of the order 2^n and is independent of ℓ. Hence if we impose a limit to the length $\ell_{\max}$, a truncated random T-sponge offers a similar level of resistance against 2nd pre-images as a truncated random oracle if $n < c - \log_2(\ell_{\max})$.

It is now interesting to take a look at the 2nd pre-image attack presented in [10] and the herding attack presented in [12] that both apply to iterated hash functions. If we apply these attacks to a random T-sponge with $c = n$ we obtain expected attack complexities lower than those obtained in [10] and [12]. The finite state of the iterated hash function makes that generating pre-images becomes easier as the first pre-image becomes longer. Including length-coding in the message padding somewhat improves the resistance, but not as expected. However, having an inner state that is twice as large as the digest, i.e., $c > 2n$ is a more fundamental solution to these problems.

For a random P-sponge, the expected workload is between $2^{(c+4)/2}$ if $\ell \ll 2^{c/2}$ and a minimum of $2^{(c+3)/2}$ due to the fact that a 2nd pre-image implies an inner collision. So for small values of ℓ, the workload is close to that of finding a path to an inner state; for values of ℓ near $2^{c/2}$, the workload comes close to that of generating an inner collision, but stays smaller.

In general, a truncated random sponge offers a similar level of resistance against 2nd pre-images as a truncated random oracle if $c > 2n$ as a 2nd preimage implies an inner collision and the expected workload of generating an inner collision is $2^{(c+3)/2}$.

5.3 Pre-image

In a sponge, a pre-image can be obtained by binding an output string to a state and subsequently finding a path to that state. Note that $n = (m + 1)r$.

For a P-sponge we bind the output to a state S. Then we compute $T = f^{-1}(S)$ and subsequently we find a path p to $T_{\mathcal{C}}$. This gives a path to S given by the found path to T, namely $p|(-S_{\mathcal{A},f}[p] + T_{\mathcal{A}})$. The expected workload for finding a pre-image for a truncated random P-sponge in this way is hence $2^{n-r} + 2^{c/2}$ if $n < c$. If $n > c$ it may be that the output has no pre-image. If it has one, the expected workload is $2^{c-1} + 2^{c/2}$. The expected workload to find a pre-image in a truncated random oracle is 2^n. It follows that a truncated random P-sponge offers a similar resistance against pre-images as a truncated random oracle if $n < c/2$.

For a T-sponge, after having bound the output to a state S, we cannot compute a state $T = f^{-1}(S)$. Therefore we need to bind the output to a state T directly. Instead of guessing the inner part of the state corresponding with the first output symbol t_0, we need to guess a state T such that $f_{\mathcal{A}}(T) = t_0$. This multiplies the number of trials by 2^r and the expected workload now becomes 2^n for $n < c + r$ and 2^{c+r} for $n > c + r$. The expected workload for finding a pre-image is hence $2^n + 2^{c-1}$ for $n < c+r$ and $2^{r+c} + 2^{c-1}$ for $n > c+r$. A truncated random T-sponge offers a similar resistance against pre-images as a truncated random oracle if $n < c$.

5.4 Length extension

Length extension is the property that given a digest $h(p)$ of an input p, but not the input itself, one can compute the digest of an input $p|p'$ with known p'. In a sponge, it is possible to do this if one can determine from the output the state $S_f[p]$ with $p = p(m)$ for a message m. One can then compute $S_f[p|p']$ and generate the output by squeezing this. The length extension only works if the state value bound to the output is equal to $S_f[p]$ and not some other state value that gives rise to the same output. If the output is longer than $c+r$ it is very likely that there is only a single corresponding state value. Otherwise the expected number of solutions is 2^{c+r-n} and length extension is only successful if the right solution is taken. For length extension it makes no sense to compare the security level with that of a random oracle, as a random oracle does not exhibit the length extension weakness at all.

5.5 Correlation immunity

Correlation immunity is the absence of large correlation between input and output of a hash function. Clearly, such measurable correlation would enable to distinguish the sponge function from a random oracle. As we have shown in Section 3 that a random sponge can only be distinguished on the basis of the presence or absence of inner collisions, large correlations will not appear in a random sponge as long as $y < c/2$. A similar reasoning applies for large differential properties between input and output.

5.6 Discussion

In all attacks discussed, it is clear the smaller the capacity, the more vulnerable the sponge function becomes with respect to the attacks exploiting the fact that the sponge function operates on a finite state. Let us now take a look at the current mainstream hash function design practice with this in mind.

Most existing hash functions make use of a so-called compression function consisting of a block cipher structure with a feedforward loop added to it. In this compression function the block cipher *encrypts* the chaining value with a message block as key. Prior to hashing, the message is padded with the message length encoded in the padding and the digest is the final value of the chaining value. This is known as the Merkle-Damgård construction and it has provable collision resistance if the compression function is collision-resistant [7, 16].

The Merkle-Damgård construction limits the value of the capacity that can be claimed to that of the digest length n. In a Merkle-Damgård hash function a collision in the output is an inner collision and length-extension is trivial due to the fact that the digest reveals the complete inner state. So while Merkle-Damgård imposes $c = n$, our analysis shows that for a given digest length n, it is best to have a capacity c that is as large as possible.

A possible direction for Merkle-Damgård hash functions is to design compression functions operating on a larger state and just truncate the final chaining value to the desired digest length. The disadvantage of this final truncation is that the reduction proof for collision-resistance is no longer valid. This does not seem such a disaster as there is no evidence that a collision-resistant fixed-length compression function would be easier to design than a collision-resistant hash function in the first place. While current mainstream hash functions appear to have an intrinsic digest length, doing truncation allows to use the same hash function for all digest length smaller than a maximum. By adopting a squeezing phase rather than just taking the digest as a part of the chaining value, any digest length could be supported. Clearly, the resistance of the hash function is limited by the size of the inner state.

Finally, note that for a random sponge the resistance against attacks does not lie in the resistance against attacks of an underlying primitive, but rather in the absence of specific properties in the used primitive. Our analysis shows that the vast majority of existing transformations (or even permutations) will give a sponge with excellent resistance as long as the capacity is large enough.

6 Security claims

6.1 Expressing a security claim

The random sponge can be used as a reference model for the security claim of a hash function design. To do so, the following parameters should be given:

- the capacity c of the sponge;
- the rate r of the sponge;
- whether the sponge is a random T-sponge or a random P-sponge;
- an optional limitation on the input length (e.g., an upper bound on the number of input bits);
- an optional limitation on the output length (e.g., a range of output lengths).

Then, the security claim is that the designed hash function should not exhibit externally visible weaknesses that the reference model does not have. By an externally visible weakness, we mean that the weakness has to be expressed in terms of input and output strings only. A property is not an attack if it needs to refer to the inside of the construction. (As such, the four operations discussed in Section 4 are not attacks by themselves.)

6.2 Choosing a reference

When a designer decides to express the security properties of his design with respect to a random sponge, he must choose between a T-sponge and a P-sponge and decide values for its capacity and rate. For a given capacity and rate, a random T-sponge almost systematically offers a higher or equal security level than a random P-sponge. The exception is the length of output cycles. One may conclude that for hash functions, a random T-sponge is a better model, leading to a more demanding security claim.

However, when we look at the practice of hash function design, almost all hash functions are designed to be all-purpose. This is especially the case for standard hash functions. The same hash function should be usable in wide range of applications and it should satisfy all security criteria simultaneously. If one expresses the security claim of such a hash function with respect to a random sponge, be it a T-sponge or P-sponge, the value of the capacity used in the claim shall be high enough to offer a sufficient resistance against collisions. In a sponge this is limited by the resistance against inner collisions, for which the expected complexity is of the order $2^{c/2}$, both for a random T-sponge and for a random P-sponge. This imposes the same lower bound on c, both for a P-sponge and for a T-sponge: c should be chosen sufficiently large so that generating inner collisions will not become even remotely feasible in the timeframe that the hash function will be used. So even if a random P-sponge is used as a reference, its weaker resistance against 2nd preimages than a T-sponge will not be within reach as long as generating inner collisions is out of reach.

Nowadays, a capacity of $c = 256$ seems to offer already a comfortable security margin. By further taking $c = 512$, one can say that when truncated to $n = 256$ bits, the sponge function offers the same resistance as a random oracle with respect to the known attacks that are also applicable to random oracles. The value of the rate of the reference sponge is not so important. In our opinion it would be best to choose among $r = 1$ and r equal to the length of the input blocks.

While for Merkle-Damgård hash functions a random T-sponge seems to be the right reference, there are several designs where a random P-sponge is more appropriate. This includes designs such as PANAMA [6] and more recently RADIOGATÚN [4] and Grindahl [11]. Basically, these functions operate by the repeated iteration of a single invertible round function on a large state. As the round function is a permutation, one can use a meet-in-the-middle approach to find a path to an inner state. This makes random P-sponges the natural reference in security claims of this kind of functions. Typically, the capacity in the claim will be much smaller than the size of the state, making up of the partial control an attacker has on the evolution of the internal state.

6.3 Flat sponge claim

If we examine our analysis for sponge functions, we see that all attacks that exploit the finite state of the random sponge have a success probability (resp. expected workload) that stays below (resp. above) that of finding inner collisions. To further simplify the choice of parameters for the reference model, one could take this fact to flatten the differences between T-sponges and P-sponges and formulate an even simpler claim.

The *flat sponge claim* is the following. Given the capacity c (and possibly a limitation on the input and/or output sizes), the success probablity of any attack should be smaller than or equal to the maximum of that for a random oracle and $1 - \exp\left(2^{2y-(c+1)}\right)$, with $N = 2^y$ the number of calls to the round function (or its inverse).

7 Sponges, MAC functions and stream ciphers

From Appendix B it appears that a sponge function where the first part of the input is kept secret and used as a key offers a high level of security if the capacity is large enough. This makes random sponges also suitable to be used as reference for MAC functions and stream ciphers.

On the other hand, when we have a function that is as secure as a random sponge with given capacity, we can turn it into a secure MAC function and a stream cipher by simply taking a secret key as the first part of the input. For existing hash functions with the Merkle-Damgård structure this is not the case as they have weaknesses that prevent these constructions to be secure. These weaknesses are compensated by having more complicated constructions such as HMAC [2] and MGF1 defined in PKCS #1 [13].

8 Conclusions

We have introduced sponge functions and have shown that random sponges are well suited to serve as reference for expressing security claims for hash functions, MAC functions and stream ciphers. They are only distinguishable from random oracles by the detection of inner collisions and the probability of inner collisions can be made arbitrarily small by increasing a security parameter, called the capacity. We have evaluated the strength of random sponges with respect to a series of attacks and show that their strength can be augmented by increasing their capacity. Thanks to the sponge, designers can make compact security claims.

References

1. R. Anderson, *The classification of hash functions*, Proceedings of the IMA Conference in Cryptography and Coding, 1993, 1993.
2. M. Bellare, R. Canetti, and H. Krawczyk, *Keying hash functions for message authentication*, Advances in Cryptology – Crypto '96 (N. Koblitz, ed.), LNCS, no. 1109, Springer-Verlag, 1996, pp. 1–15.
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. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, RadioGatún*, a belt-and-mill hash function*, Second Cryptographic Hash Workshop, Santa Barbara, August 2006, `http://radiogatun.noekeon.org`.
5. 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.
6. J. Daemen and C. S. K. Clapp, *Fast hashing and stream encryption with PANAMA*, Fast Software Encryption 1998 (S. Vaudenay, ed.), LNCS, no. 1372, Springer-Verlag, 1998, pp. 60–74.
7. I. Damgård, *A design principle for hash functions,*, Advances in Cryptology – Crypto '89 (G. Brassard, ed.), LNCS, no. 435, Springer-Verlag, 1989, pp. 416–427.
8. IEEE, *P1363-2000, standard specifications for public key cryptography*, 2000.
9. A. Joux, *Multicollisions in iterated hash functions. application to cascaded constructions*, Advances in Cryptology – Crypto 2004 (M. Franklin, ed.), LNCS, no. 3152, Springer-Verlag, 2004, pp. 306–316.
10. J. Kelsey and B. Schneier, *Second preimages on n-bit hash functions for much less than* 2^n *work*, Advances in Cryptology – Eurocrypt'2005 (R. Cramer, ed.), LNCS, no. 3494, Springer-Verlag, 2005, pp. 474–490.
11. L. Knudsen, C. Rechberger, and S. Thomsen, *Grindahl - a family of hash functions*, Fast Software Encryption 2007 (A. Biryukov, ed.), LNCS, Springer-Verlag, 2007, to appear.
12. T. Kohno and J. Kelsey, *Herding hash functions and the nostradamus attack*, Advances in Cryptology – Eurocrypt'2006 (S. Vaudenay, ed.), LNCS, no. 4004, Springer-Verlag, 2006, pp. 222–232.
13. RSA Laboratories, *PKCS # 1 v2.1 RSA Cryptography Standard*, 2002.
14. M. Liskov, *Constructing secure hash functions from weak compression functions: The case for non-streamable hash functions*.
15. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.
16. R. Merkle, *One way hash functions and des,*, Advances in Cryptology – Crypto '89 (G. Brassard, ed.), LNCS, no. 435, Springer-Verlag, 1989, pp. 428–446.
17. R. Rivest, *The MD4 message digest algorithm*, Advances in Cryptology – Crypto'90 (A. Menezes and S. Vanstone, eds.), LNCS, no. 537, Springer-Verlag, 1991, pp. 303–311.
18. Wikipedia, *Cryptographic hash function*, 2007, `http://en.wikipedia.org/wiki/Cryptographic_hash_function`.

A Computations

In this appendix we compute the success probabilities of the four operations introduced in Section 4.

A.1 Graph representation

One can associate a graph with a sponge function with AC nodes and AC edges: the *sponge graph*. The nodes are the state values and for every couple (S, T) with $T = f(S)$ there is a directed edge from S to T. From each node starts exactly one edge. For a P-sponge, in each node arrives exactly one edge. The nodes can be partitioned by the value of the inner state and we call the subset of all nodes with the same inner state value a *supernode*. Edges between nodes are therefore also edges between supernodes. There are C supernodes, one for each inner state value. The A nodes within a supernode are identified by their $S_{\mathcal{A}}$ value, which we will call their *index*.

One can absorb an input string p by following edges starting from supernode 0, the *root*. First we draw an edge from $(p_0, 0)$. This edge arrives in node with index $S_{\mathcal{A},f}[p_0]$ of supernode $S_{\mathcal{C},f}[p_0]$. Then we draw an edge from the node within that supernode with index $S_{\mathcal{A},f}[p_0]+p_1$ and the node where it arrives is $S_f[p_0p_1]$. For p_i, we draw an edge from the node with index $S_{\mathcal{A},f}[p_0p_1 \dots p_{i-1}] + p_i$ within supernode $S_{\mathcal{C},f}[p_0p_1 \dots p_{i-1}]$. It follows that the graphic representation of a path to an inner state is a sequence of directed edges from the root to the corresponding supernode. Given the graphic representation, one can reconstruct the value of p. The i-th character of the path p is determined by the edges arriving at and starting from the i-th supernode on the path: it is the index of the node where the outgoing edge starts minus the index of the node where the incoming edge arrives. The first symbol of the path p_0 corresponds with the root where there is no incoming edge and it is just equal to the index of the node where the first edge starts.

In the following, we represent the information the adversary has learned in the experiment in a graph that represents the part of the sponge graph known to her. We call this the *adversary graph*.

In the beginning, the adversary has no information about f and her graph has no edges. The only way she can gain information on f is to make calls to f (and f^{-1}). Without loss of generality, we assume the adversary makes no queries corresponding with known edges. Hence, a call to f corresponds to adding to the adversary graph an edge starting from a given node and a call to f^{-1} with adding an edge arriving in a given node.

In the adversary graph, we say that a supernode $T_{\mathcal{C}}$ is reachable from a supernode $S_{\mathcal{C}}$ if there is a sequence of directed edges from $S_{\mathcal{C}}$ to $T_{\mathcal{C}}$ (in the right direction) or if $T = S$. We call the supernodes that are reachable from the root, *rooted supernodes* and denote their set by $\mathcal{R}$, with $R = |\mathcal{R}|$. We also call all nodes in a rooted supernode rooted.

A.2 Inner collisions

The adversary has an inner collision if she finds two paths from the root to some supernode. We consider the i-th call of the adversary and express the probability that it leads to an inner collision on the condition that no inner collisions were discovered yet. As can be seen in Figure 1 this implies that the new edge must connect a rooted supernode to a supernode from which a rooted supernode can be reached. We call a supernode from which a rooted supernode can be reached in the adversary graph an $\mathcal{R}$-reaching supernode and their set by $\mathcal{V}$ with $V = |\mathcal{V}|$. Clearly, $\mathcal{R} \subseteq \mathcal{V}$. Initially, $\mathcal{V} = \mathcal{R} = \{0\}$ and $R = V = 1$. Right before adding the i-th edge, the graph contains $i - 1$ edges and $R \leq V \leq i$.

Random T-sponge The adversary can only add edges starting from chosen nodes. If the new edge starts from a rooted node, the probability of success is V/C. Moreover it adds one to R and hence also to V. If the new edge starts from a non-rooted node, the probability of success is 0. It leaves R invariant and may add one to V if it arrives in a node in $\mathcal{V}$. It follows that the success probability of future edges is optimized by always adding edges starting from rooted nodes. The exact shape of the rooted tree is not important. So applying this strategy, right before adding the i-th edge, we have $R = V = i$ yielding:

$$P(\text{no IC}) = \prod_{i=1}^{N} (1 - \frac{i}{C}).$$

If $N \ll C$, we can use the $\log(1+\epsilon)$ approximation: $\log(1+\epsilon) \approx \epsilon$ when $\epsilon \ll 1$. This gives:

$$P(\text{IC}) \approx 1 - \exp\left(-\sum_{i=1}^{N} \frac{i}{C}\right) = 1 - \exp\left(-\frac{N(N+1)}{2C}\right) \ .$$

The expression we obtain for the success probabilities of the attacks in the following subsections have the same shape. Keeping in mind that cost function $c_p(N)$ of an attack is given by $c_p(\text{success}) = -\log(1 - P(\text{success}))$, we have

$$P(\text{success}) = 1 - \exp\left(-c_p(\text{success})\right) \ .$$

Note that when $c_p(\text{success}) \ll 1$, applying $\log(1+\epsilon)$ approximation yields $P(\text{success}) \approx c_p(\text{success})$. In this case, we have:

$$c_p(\text{IC}) \approx \frac{N(N+1)}{2C} \ .$$

Random P-sponge The adversary can add edges starting from chosen nodes and edges arriving in chosen nodes. An edge starting from a chosen node can only arrive in a node that has no incoming edge yet; an edge arriving in a chosen node can only start from an edge that has no outgoing edge yet.

If an edge is added starting from a chosen node that is rooted, the probability of success is $\frac{(A-1)V+1}{AC-i}$: the number of nodes in $\mathcal{V}$ with no incoming edge divided by the total number of nodes with no incoming edge. If an edge is added arriving in a chosen node in $\mathcal{V}$, the probability of success is similarly $\frac{(A-1)R+1}{AC-i}$.

The higher the values of R and V, the better the future probabilities of success, so one may do other things to augment R or V faster. An edge starting from a chosen node that is not rooted cannot lead to an inner collision. It leaves R invariant and may add one to V. But an edge arriving in a chosen node in $\mathcal{V}$ adds one to V with certainty, so this always leads to better success probabilities. Similarly, an edge arriving in a chosen node that is not in $\mathcal{V}$ cannot lead to an inner collision but it may adds one to R. An edge starting from a chosen node that is rooted adds one to R with certainty, so this always always leads to better success probabilities.

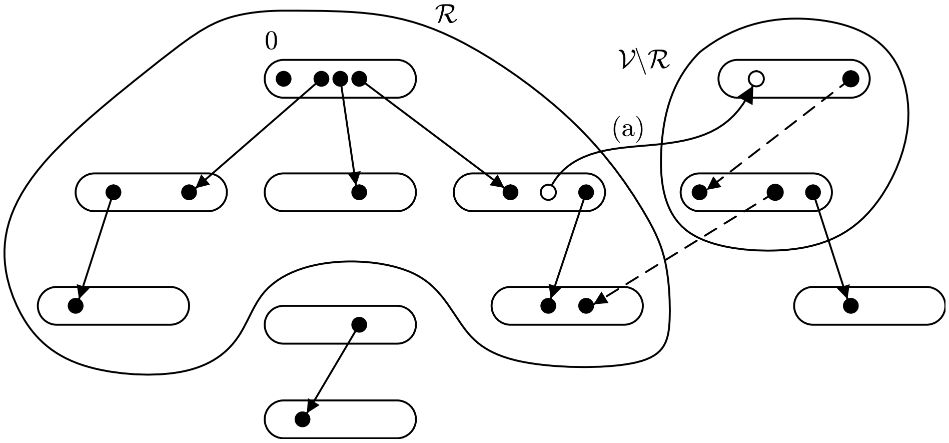

Fig. 1. Adding an edge (a) resulting in an inner collision. Edge (a) must start in $\mathcal{R}$ and arrive in $\mathcal{V}$.

At any time, $R \leq V \leq i$. Globally, the optimal strategy is one in which the probability of success of the i-th call is $\frac{(A-1)i+1}{AC-i}$. When adding an edge arriving in a chosen node in $\mathcal{V}$ that does not lead to an inner collision, R is not affected and hence it leads to $R < i$, while in the optimal strategy $R = i$. It follows that in the optimal strategy only a single edge arriving in a chosen node in $\mathcal{V}$ may be added and all other edges are just edges added to rooted nodes. We obtain:

$$P(\text{no IC}) = \prod_{i=1}^{N} \left(1 - \frac{(A-1)i+1}{AC-i}\right) = \prod_{i=1}^{N} \frac{1 - \frac{i}{C} - \frac{1}{AC}}{1 - \frac{i}{AC}}.$$

Using the $\log(1+\epsilon)$ approximation this gives:

$$c_p(\text{IC}) \approx \sum_{i=1}^{N} -\frac{i-1}{AC} + \frac{i}{C} = \frac{N(N+1)}{2C} - \frac{N(N-1)}{2AC}\ .$$

A.3 Finding a path to an inner state

Given a target inner state $T_{\mathcal{C}}$, the adversary must find a path p such that $S_{\mathcal{C},f}[p] = T_{\mathcal{C}}$. We consider the i-th call of the adversary and express the probability that it leads to a path on the condition that no path was found yet. As it can be seen in Figure 2 this implies that the new edge must connect a rooted supernode to a supernode from which $T_{\mathcal{C}}$ can be reached. We call a supernode (and its nodes) from which the target can be reached a target-reaching supernode (and nodes) and their set by $\mathcal{V}$ with $V = |\mathcal{V}|$. Initially, $\mathcal{V} = \{T_{\mathcal{C}}\}$, $\mathcal{R} = \{0\}$ and $R = V = 1$. Right before the i-th call, the graph contains $i-1$ edges and $R \leq i$, $V \leq i$ and $R + V \leq i + 1$.

Random T-sponges The adversary can only add edges starting from chosen nodes. If an edge is added starting from a chosen node that is rooted, the probability of success is V/C. Otherwise, the probability of success is 0, it leaves R invariant and adds one to V with probability V/C. It follows that to optimize the probability of success it is best to systematically add edges starting from chosen nodes that are rooted. So applying this strategy, right before the i-th call, we have $R = i$ and $V = 1$ yielding:

$$P(\text{no path}) = \prod_{i=1}^{N}(1 - \frac{1}{C}) \ .$$

Using the $\log(1 + \epsilon)$ approximation for $C \gg 1$, this yields:

$$c_p(\text{path}) \approx \frac{N}{C} \ .$$

We will now discuss a variant of the operation: finding a second path to an inner state if there is already a path of length ℓ. This is relevant when generating 2nd pre-images when being used as a hash function. We consider the probability to find a 2nd path after adding N edges, also counting the ℓ edges corresponding with the absorbing of message p. After adding these ℓ edges, $\mathcal{R}$ and $\mathcal{V}$ each contain the set of ℓ supernodes on the path from the root to T. For $N > \ell$ this gives

$$P(\text{no 2nd path}) = \prod_{i=\ell}^{N}(\ell - \frac{1}{C}) \ ,$$

and subsequently, if $\ell \ll C$

$$c_p(\text{2nd path}) \approx \frac{\ell N}{C} \ .$$

Random P-sponges The adversary can add edges starting from chosen nodes and edges arriving in chosen nodes. An edge starting from a chosen node can only arrive in a node that has no incoming edge yet, an edge arriving in a chosen node can only start from an edge that has no outgoing edge yet.

If the edge starts from a chosen node that is rooted, the probability of success is $\frac{(A-1)V+1}{AC-i}$, if the supernodes of $\mathcal{V}$ with the edges form a tree. This is the number of nodes in $\mathcal{V}$ with no incoming edges divided by the total number of nodes with no incoming edges. This adds 1 to R if there is no inner collision and leaves V invariant.

If an edge is added arriving in a chosen target-reaching node, the probability of success is similarly $\frac{(A-1)R+1}{AC-i}$, in the assumption that there are no inner collisions. This adds 1 to V if the new edge does no start from a target-reaching node and leaves R invariant. We will assume that there are no inner collisions and that the supernodes of $\mathcal{V}$ form a tree and later see whether this assumption is justified. In any case, the absence of inner collisions results in the largest success probabilities and hence upper bounds the success probability.

The higher the values of R and V, the better the future probabilities of success. It follows that adding an edge arriving in target-reaching nodes augments the probability of success when later adding edges starting from rooted nodes and vice versa.

An edge starting from a chosen node that is not rooted cannot lead to a path to $T_{\mathcal{C}}$. It leaves R invariant and may add one to V with small probability. With the eye on increasing the success probability of future calls, adding an edge starting from a rooted node is always better. Similarly, an edge arriving in a chosen node that is not in $\mathcal{V}$ cannot lead to a path to $T_{\mathcal{C}}$. It adds one to R with small probability and adding an edge arriving in a target-reaching node is always better.

We introduce a variable ρ_i that is 1 if the i-th edge added is one starting from a chosen node that is rooted and -1 otherwise and denote the values of R and V right before adding the i-th edge by R_i and V_i. Then the probability that the i-th edge added does not result in a path becomes

$$1 - \frac{(A-1)\left(\frac{1+\rho_i}{2}V_i + \frac{1-\rho_i}{2}R_i\right) + 1}{AC - i} = \frac{1 - \frac{i+1}{AC} - \frac{A-1}{2AC}\left(V_i + R_i - \rho_i(R_i - V_i)\right)}{1 - \frac{i}{AC}}$$

Using the $\log(1+\epsilon)$ approximation we obtain:

$$c_p(\text{path}) \approx \sum_{i=1}^{N}\left(\frac{1}{AC} + \frac{A-1}{2AC}\left(V_i + R_i - \rho_i(R_i - V_i)\right)\right)$$

We have $V_i + R_i \leq i + 1$, where equality applies if there are no inner collisions in $\mathcal{R}$ and if the supernodes of $\mathcal{V}$ form a tree. We assume $V_i + R_i = i + 1$ and later verify whether this assumption was justified. Moreover, we have $R_i - V_i = \sum_{j=1}^{i-1} \rho_i$. This gives:

$$c_p(\text{path}) \approx \frac{N}{AC} + \frac{A-1}{4AC}\left(N^2 + 3N + 1 - \sum_{i=1}^{N}\sum_{j=1}^{i-1} 2\rho_i\rho_j\right)$$

We can now work out the last term using:

$$\sum_{i=1}^{N}\sum_{j=1}^{i-1} 2\rho_i\rho_j = \sum_{i=1}^{N}\sum_{j=1}^{N} \rho_i\rho_j - \sum_{i=1}^{N} \rho_i\rho_i = (R_{N+1} - V_{N+1})^2 - N$$

Filling this in gives:

$$c_p(\text{path}) \approx \frac{N(N+4) - (R_{N+1} - V_{N+1})^2}{4C} - \frac{N^2 - (R_{N+1} - V_{N+1})^2}{4AC}$$

This is maximized if $R_{N+1} = V_{N+1}$ for N even and if $|R_{N+1} - V_{N+1}| = 1$ for N odd, i.e., if $\mathcal{R}$ and $\mathcal{V}$ have the same number of nodes just before the path is found. As it is not known in advance when the path will be found, the best strategy is to add edges starting from chosen nodes in $\mathcal{R}$ and edges arriving in chosen nodes in $\mathcal{R}$ in an alternating fashion, guaranteeing $(R_N - V_N)^2 \leq 1$. For even N this gives:

$$c_p(\text{path}) \approx \frac{N(N+4)}{4C} - \frac{N^2}{4AC} \; .$$

the probability of success becomes significant when N is of the order of $2\sqrt{C}$ and hence when R and V are of the order $\sqrt{C}$. This implies that for these values of N there may be inner collisions but their small number compared to R make that their presence does not affect the success probability significantly.

A.4 Detecting cycles in the output

The goal is to detect cycles in outputs corresponding to valid input strings. The adversary can take an input string p and absorb it, resulting in a node $S_f[p]$. From this node, the output characters $S_{\mathcal{A},f}[p|0^j]$ are generated by following a *chain* of nodes connected by edges, i.e., $S_f[p|0^j] = f(S_f[p|0^{j-1}]$, where we define a chain as a sequence of nodes connected by directed

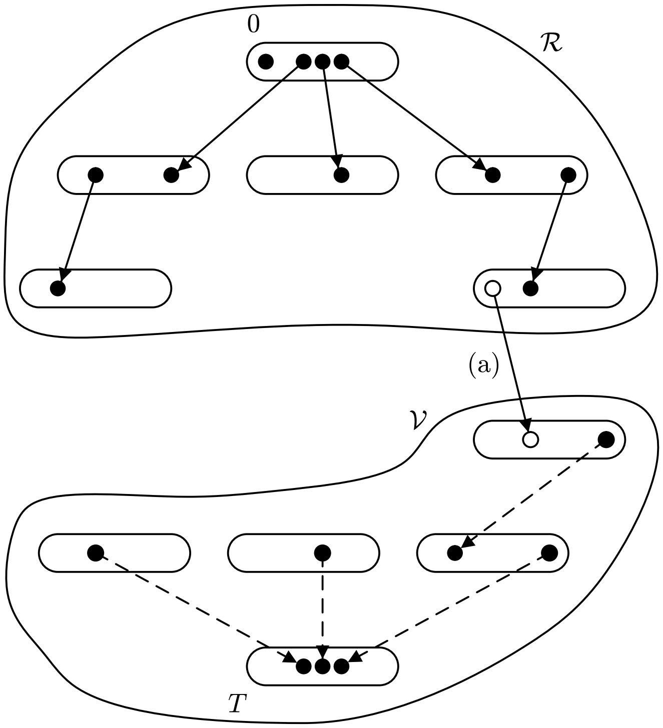

Fig. 2. Adding an edge (a) resulting in a path. Edge (a) must start in $\mathcal{R}$ and arrive in $\mathcal{V}$.

edges. The first node in the chain is the node $U = S_f[p'] + p_{|p|-1}$ with p' equal to p with the last character $p_{|p|-1}$ removed.

The adversary finds a cycle by creating nodes in such a chain by adding edges to the last node of the chain until the new edge arrives in a node in the chain. The shortest valid input strings have length 1 containing a non-zero character. Before adding the i-th edge, the chain contains i nodes.

Random T-sponges The probability that the new edge arrives in one of the nodes of the chain is $i/(AC)$. Using the $\log(1+\epsilon)$ approximation, this results in:

$$c_p(\text{output cycle}) \approx \frac{N(N+1)}{2AC} \ .$$

Random P-sponges At any moment, there is only a single node in the chain that has no incoming edge, the first one. The probability that the new edge arrives in a node in the chain is hence $1/(AC)$. This results in:

$$c_p(\text{output cycle}) \approx \frac{N}{AC} \ .$$

A.5 Binding an output string to a state

The goal is to find a state S such that $S_{\mathcal{A}} = t_0$, $f_{\mathcal{A}}(S) = t_1$, $f_{\mathcal{A}}(f(S)) = t_2$, $f_{\mathcal{A}}(f^{m-1}(S)) = t_m$. As $S_{\mathcal{A}} = t_0$, only the inner part $S_{\mathcal{C}}$ is unknown.

The probability, over the transformations (or permutations) f and over $S_{\mathcal{C}} \in \mathcal{C}$, that the following condition is verified:

$$f_{\mathcal{A}}(f^{i-1}(t_0, S_{\mathcal{C}})) = t_i, \forall i \in \{1 \dots m\}, \tag{1}$$

depends not only on the length of t, $|t| = m+1$, but also on the structure of t. First of all, a string showing perodicity d has slightly more probability, as cycles help to satisfy the condition (1).

In a random T-sponge the values of t_i affect the success probability due to the existence of cycles. The probability that a value S is in a cycle of length d is approximately $1/AC$ (if $d \ll \sqrt{AC}$) and the probability that this cycle leads to the correct output sequence is $1/A^{d-1}$.

In a random P-sponge the values of t_i affect the success probability in the following way. When several t_i have the same value, either the corresponding $S_{\mathcal{C}}$ values are different or there is a cycle. This effect is strongest if A is small, e.g. in binary P-sponges. In the absence of a cycle, where one would expect a probability for the i-th check to be successful equal to $1/A$, this probability is only $\frac{C-q}{AC-i}$ with q the number of t_j values with $j < i$ that have the same value as t_i. The effect therefore grows in importance if the t_i are more biased. Clearly, for large values of C, this can be neglected and we can approximate the success probability by A^{-1}. The probability that a value S leads to a cycle of period d is $1/AC$ (see Section A.4) and the probability that this cycle leads to the correct output sequence is approximately $1/A^{d-1}$.

Both for a random T-sponge and a random P-sponge, the probability that a cycle occurs for a candidate inner state is $1/AC$ and for a given transformation (or permutation) the probability that there is not a single state leading to the cycle of period d is approximately $1 - 1/A$. The expected number of such state values is hence so small that this effect can be neglected.

The adversary can make random guesses of $S_{\mathcal{C}}$ until he finds one such that $f_{\mathcal{A}}(t_0, S_{\mathcal{C}}) = t_1$. From there, he can evaluate $f_{\mathcal{A}}(f(t_0, S_{\mathcal{C}}))$ and check if it is equal to t_2. If so, he continues, possibly until he reaches the last character of the sequence; if not, he starts again from a new guess for $S_{\mathcal{C}}$. At each step, in the absence of a cycle and neglecting biases, the adversary has a probability of A^{-1} to get the correct next character. Once an incorrect character is encountered, the adversary starts with the next guess for $S_{\mathcal{C}}$. Clearly, the average number of calls to f to eliminate a guess is $\frac{A}{A-1}$.

If $A^m < C$ the probability for a guess to be successful is:

$$P(\text{success with guess}) = 1 - (1 - A^{-m}) \ .$$

Taking into account the number of calls to f for a guess, we obtain the following cost function, both for a random T-sponge as for a random P-sponge.

$$c_p(\text{state binding}) \approx \frac{A-1}{A} N / A^m \ .$$

When $mr > c$ the expected number for N is larger than the number of inner state values and an inner state value that leads to the given output sequence only exists for a fraction of the possible transformations (or permutations). The case $mr > c$ implies that the adversary has to try a large fraction of the values $S_{\mathcal{C}} \in \mathcal{C}$. By construction, he cannot look for more than C values of $S_{\mathcal{C}}$. If the sponge transformation (or permutation f) is fixed but a priori unknown to the adversary, we can distinguish two situations:

1. The adversary is certain that the string t has been observed from the output of the sponge S_f.
2. The adversary does not know if the string t has been observed from the output of the sponge S_f.

In the first case, he will surely find a satisfactory $S_{\mathcal{C}}$. If $mr \gg c$, it is highly probable that only one such candidate exists. In the second case, he may not necessarily find a satisfactory $S_{\mathcal{C}}$, and his chances decrease as m increases. For $mr > c$, the cost function becomes:

$$c_p(\text{success with guess}) =\approx \frac{A-1}{A}\frac{N}{C} \ .$$

B Other applications of sponges

We can use a sponge as a MAC function by just taking as input the concatenation of a secret key and the message, taking the truncated output as the MAC and hiding the inner state from the adversary. Similarly, a sponge can be used as a stream cipher. The input is the concatenation of a secret key and a diversifier, the output is the keystream and the inner state is hidden from the adversary. We denote the length of the key (in characters) by k. In both cases, the first part of the input to the sponge is the secret key. We call this a *keyed sponge.*

The keyed sponge model does not allow related-key attacks. To extend the model to also allow related-key attacks, one can have inputs to the sponge consisting of two parts: the input string an a k-character key offset. The keyed sponge first adds the key offset characterwise to the key, concatenates the input string to the result and absorbs that into the sponge.

In describing attacks on a keyed sponge, the adversary can make two types of queries. The first type are calls to f and in the case of a P-sponge also f^{-1}. The total number of such calls is denoted by $N = 2^y$, that represents what is usually called the computational complexity of the attack. The second type are queries to the keyed sponge similar to those in Section 3. The sum of the number of input characters and output characters (and key offset characters) is denoted by $M = 2^q$. M represents what is usually called the data complexity of the attack: the amount of data computed with the key.

We can have several scenarios. The most powerful is a distinguishing attack. In this case the adversary can make queries to a black box that is either a random oracle or a keyed sponge as in Section 3. What is different from the setting in Section 3 is that additionally the adversary can make calls to f (and f^{-1}). The adversary can distinguish in three ways:

– State binding: the adversary asks the black box for an output sequence longer than $c + r$ characters and binds a state to the output sequence using the white box. For this attack to work, we need $M > c + r$ and for a random sponge it has $c_p(\text{success}) \approx 2^{y-c}$.
– Key guessing: the adversary asks the black box for output sequences with a total of more than k characters. She then uses the white box to generate the same output sequences where the value of the key is guessed. For this attack to work, we need $M \approx k$ and for a random sponge it hash $c_p(\text{success}) \approx 2^{y-k}$.
– Observing inner collisions: the adversary asks the black box for output sequences until he observes an inner collision. This attack has $c_p(\text{success}) \approx 2^{2q-(c+1)}$.

Note that this analysis is valid both for the fixed-key and the related-key case. There appear to be no other methods to distinguish the keyed sponge from a random oracle.

Another scenario is stream prediction when being used as a stream cipher, or MAC forgery when being used as a MAC. In both cases, the adversary shall predict the output of the keyed sponge to an input using only its outputs to a set of other inputs. Here again we see the three approaches. After key guessing the adversary can produce the output corresponding to any input. After state binding, the adversary can generate the output to any string p that has a prefix in the input strings for which the adversary knows the output string. In case of P-sponge, the adversary can even generate the output to any string p. If the adversary observes an inner collision in the black box, MAC forgery is easy. If two input strings $K|p$ and $K|q$ collide and the adversary knows the output corresponding to input string $K|p|m$, he also knows the output corresponding to input string $K|q|m$. When being used as a stream cipher, it depends on the range of allowed IV values. If an inner collision occurs before the last character of the IV has been absorbed, the same technique can be applied as in the case of MAC forgery. If a cycle is detected in an output sequence (special case of inner collision) the complete output sequence is known based on a single period. Finally, in the case of a MAC function one can simply guess the MAC value, hence the resistance against MAC forgery is also limited by the length of the MAC n: the adversary can correctly guess the MAC to a message with probability 2^{-n}.