Cryptographic sponge functions

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

http://sponge.noekeon.org/

Version **0.1**
January 14, 2011

[1]STMicroelectronics
[2]NXP Semiconductors

Contents

Chapter 1

Introduction

In the context of cryptography, sponge functions provide a particular way to generalize hash functions to more general functions whose output length is arbitrary. A sponge function instantiates the sponge construction, which is a simple iterated construction building a variable-length input variable-length output function based on a fixed length permutation (or transformation). With this interface, a sponge function can also be used as a stream cipher, hence covering a wide range of functionality with hash functions and stream ciphers as particular points.

From a theoretical point of view, sponge functions model in a very simple way the finite memory any concrete construction has access to. A random sponge function is as strong as a random oracle, except for the effects induced by the finite memory. This model can thus be used as an alternative to the random oracle model for expressing security claims.

From a more practical point of view, the sponge construction and its sister construction, called the duplex construction, can be used to implement a large spectrum of the symmetric cryptography functionality. This includes hashing, reseedable pseudo random bit sequence generation, key derivation, encryption, message authentication code (MAC) computation and authenticated encryption. This provides users with a lot of functionality from a single fixed permutation, hence making the implementation easier. The designers of cryptographic primitives may also find it advantageous to develop a strong permutation without worrying about other components such as the key schedule of a block cipher.

1.1 Roots

The idea of developing sponge functions came during the design of RadioGatún [10]. This cryptographic hash function has a variable-length input and a variable-length output. When we proposed RadioGatún, we faced the problem that we had to express a claim of cryptographic security. For a hash function with fixed output length n, one usually implicitly or explicitly claims its security to be as good as a random oracle whose output is truncated to n bits. This implies the resistance to the traditional hash function attacks, such as $2^{n/2}$ for collision and 2^n for (second) pre-image attacks.

For cryptographic primitives with variable-length output, such as RadioGatún, expressing the required resistance with respect to the output length makes little sense as this would imply that it should be possible to increase the security level indefinitely by just taking longer outputs. Rather than claiming resistance levels against the traditional hash function attacks, we decided to express the security claim as what an ideal function could achieve. In the paper [10] we proposed for that purpose something we called an *ideal mangling function*. However, after publication we noticed that this was not ideal and we decided to dig more

deeply into this subject. Our goal was to specify a function that behaves like a random oracle, with the sole exception that it would have inner collisions. This search led to so-called random sponge functions. The results of this initial search was presented at the Dagstuhl seminar on Symmetric Cryptography in January 2007, and soon after the final definition of sponge functions was given at the Ecrypt Hash Workshop in Barcelona [11].

1.2 The sponge construction

The sponge construction is a simple iterated construction for building a function 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.

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.

1.3 Sponge as a reference of security claims

One could exhaustively list all the properties that a hash function should resist to and assign them resistance levels. Alternatively, claiming the security of a concrete function with regard to a model means comparing the success probability of an attack on the concrete function against that on the model. This allows compact security claims, which address all the possible properties at once, including future requirements not foreseen in an exhaustive list.

In fixed digest-length hash functions, the required resistance against attacks is expressed relative to the digest length. Until recently one has always found it reasonable to expect a hash function to be as strong as a random oracle with respect to the classical attacks. However, this changed after the publication of the generic attacks listed in Section 6.6.1.

An iterated function uses a finite memory to store its state and processes the input, block per block. At any point in time, the state of the iterated function summarizes the input blocks received so far. Because it contains a finite number of bits, collisions can happen in this state. Random oracles, on the other hand, do not have collisions in their "state" as such a concept does not exist. This is the main reason for which random oracles cannot be used directly to express security claims of functions with variable-length output: they would simply never exhibit any effects of the finite memory any concrete iterated function has.

Random sponges functions, on the other hand, provide an alternative to the random oracle model for expressing security claims. A random sponge is an instance of the sponge construction with f chosen randomly from the set of transformations (or of permutations) over b bits. We have shown that a random sponge function is as strong as a random oracle,

except for the effects induced by the finite memory. A random sponge can serve as a reference model for expressing compact security claims for iterated hash functions and stream ciphers.

When using a random sponge as a security model, one considers the success of a particular attack. Such a success probability depends not only on the nature of the attack considered but also on the chosen parameters of the random sponge, i.e., its capacity, bitrate and whether it calls a random permutation or a random transformation. The flat sponge claim is a simplification in the sense that we consider only the worst-case success probability, determined by the RO differentiability bound, which depends solely on the capacity of the random sponge. Hence, it flattens the claimed success probabilities of all attacks using a single parameter: the claimed capacity c_{claim}.

1.4 Sponge as a design tool

As said, our initial goal was to define a reference for security properties of hash function designs. Despite our original intention we realized that the sponge construction could also lead to practical hash function designs. An important aspect is that the design can be based on a permutation, as opposed to a compression function or a block cipher. Designing a suitable permutation is easier than designing a suitable compression function or block cipher. This is rather good news in itself: all the symmetric cryptographic primitives can be based on a fixed-length permutation. A permutation has a single input and therefore treats all the input bits on an equal footing. This is a welcome simplification compared to modes making use of a block cipher or a tweakable block cipher.

Generic attacks are attacks that do not exploit the properties of the concrete primitive but only the properties of the construction. The indifferentiability framework provides us with a way to upper bound the success probability of generic attacks, and we used it to show that sponge functions are actually resistant to such attacks below a complexity of $2^{c/2}$. In fact, these results show that any attack against a sponge function implies that the permutation it uses can be distinguished from a typical randomly-chosen permutation. This naturally leads to the following design strategy, which we called the hermetic sponge strategy: adopting the sponge construction and building an underlying permutation f that should not have any structural distinguishers.

In this approach, one designs a permutation f on $b = r + c$ bits and uses it in the sponge construction to build the sponge function F. In addition, one makes a flat sponge claim on F with a claimed capacity equal to the capacity used in the sponge construction, namely $c_{\text{claim}} = c$. In other words, the claim states that the best attacks on F must be generic attacks. Hence, $c_{\text{claim}} = c$ means that any attack on F with expected complexity below $2^{c/2}$ implies a structural distinguisher on f, and the design of the permutation must therefore avoid such distinguishers.

In the hermetic sponge strategy, the capacity determines the claimed level of security, and one can trade claimed security for speed by increasing the capacity c and decreasing the bitrate r accordingly, or vice-versa.

1.5 Sponge as a versatile cryptographic primitive

With its arbitrarily long input and output sizes, the sponge construction allows building various primitives such as a hash function, a stream cipher or a MAC. In some applications the input is short (e.g., a key and a nonce) while the output is long (e.g., a key stream). In

other applications, the opposite occurs, where the input is long (e.g., a message to hash) and the output is short (e.g., a digest or a MAC).

Another set of usage modes takes advantage of the duplex construction, a construction that is closely related to the sponge construction and whose security can be shown to be equivalent. The duplex construction allows the alternation of input and output blocks at the same rate as the sponge construction, like a full-duplex communication. This allows one to implement an efficient reseedable pseudo random bit sequence generation and an authenticated encryption scheme requiring only one call to f per input block.

1.6 Structure of this document

The structure of this document is as follows. First, Chapter 2 provides the definitions of the two central constructions in this document and some auxiliary functions. Then, Chapter 3 explains modes of use of the sponge construction while Chapter 4 presents modes built on top of the duplex construction. We investigate generic algorithms in Chapter 5 and give formal security proofs in Chapter 6. This is followed by Chapter 7, which presents the use of random sponges as a security reference. Finally, Chapter 8 presents a practical strategy for the design of sponge functions with an iterated permutation.

Chapter 2

Definitions

In this chapter we list our conventions and provide the definitions of the two central constructions in this document and some auxiliary functions that are useful in the description and analysis of these constructions

2.1 Conventions and notation

We denote the absolute value of a real number x is denoted by $|x|$.

We often use the approximation $\log(1+\epsilon) \approx \epsilon$ if when $\epsilon \ll 1$. We call this the $\log(1+\epsilon)$ approximation.

We denote the cardinality of a set S by $|S|$.

2.1.1 Bitstrings

We denote the length in bits of a bitstring M by $|M|$. A bitstring M can be considered as a sequence of blocks of some fixed length x, where the last block may be shorter. The number of blocks of M is denoted by $|M|_x$. The blocks of M are denoted by M_i and the index ranges from 0 to $|M|_x - 1$. We denote the empty string by empty string. It has length 0 and no bits. It can be seen either as a string with no blocks or with a single zero-length block. Unless explicitly stated otherwise, we will assume that it has 0 blocks.

We denote truncation of a bitstring M to its ℓ first bits by $\lfloor M \rfloor_\ell$. A bitstring consisting of n zeroes is denoted by 0^n and the concatenation of two strings M and N is denoted as $M||N$.

We denote the set of all bitstrings including the empty string by $\mathbb{Z}_2^*$ and excluding the empty string by $\mathbb{Z}_2^+$. The set of infinite-length bitstrings is denoted by $\mathbb{Z}_2^\infty$.

2.1.2 Padding rules

For the padding rule we use the following notation: the padding of a message M to a sequence of x-bit blocks is denoted by $M||\text{pad}[x](|M|)$. This notation highlights that we only consider padding rules that append a bitstring that is fully determined by the bitlength of M and the block length x. We may omit $[x]$, $(|M|)$ or both if their value is clear from the context. For injective padding rules, we use the term *unpadding* the retrieval from M from $P = M||\text{pad}[x](|M|)$. Note that for any injective padding rule there exist strings P for which this is not possible.

Definition 1. *A padding rule is* sponge-compliant *if it never results in the empty string and if it satisfies following criterion:*

$$\forall n \geq 0, \forall M, M' \in \mathbb{Z}_2^* : M \neq M' \;\Rightarrow\; M||\text{pad}[r](|M|) \neq M'||\text{pad}[r](|M'|)||0^{nr} \tag{2.1}$$

We define now the simplest padding rule that is sponge-compliant.

Definition 2. Simple padding, *denoted by* pad10*, *appends a single bit 1 followed by the minimum number of bits 0 such that the length of the result is a multiple of the block length.*

Simple padding appends at least 1 bit and at most the number of bits in a block. The simplest padding rule that allows securely using the same f with different rates (see Section 6.5) is the following.

Definition 3. Multi-rate padding, *denoted by* pad10*1, *appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length.*

Clearly, this padding rule is sponge-compliant as well as it is injective and cannot result in an empty string or a string with all-zero last block. Multi-rate padding appends at least 2 bits and at most the number of bits in a block plus one.

2.1.3 Random oracles, transformations and permutations

We denote a random oracle by $\mathcal{RO}$ and use the definition of [6].

Definition 4. *A random oracle $\mathcal{RO}$ takes as input binary strings of any length and returns for each input a random infinite string, i.e., it is a map from $\mathbb{Z}_2^*$ to $\mathbb{Z}_2^\infty$, chosen by selecting each bit of $\mathcal{RO}(M)$ uniformly and independently, for every M.*

We denote a call to $\mathcal{RO}$ where the output is truncated to its ℓ first bits by $Z = \mathcal{RO}(M, \ell)$. We also need the concept of a random (fixed-width) transformation.

Definition 5. *A* random transformation *with given width b is a transformation drawn randomly and uniformly from the set of all 2^{b2^b} b-bit transformations.*

Finally, we define a random (fixed-width) permutation.

Definition 6. *A* random permutation *with given width b is a permutation drawn randomly and uniformly from the set of all $2^b!$ b-bit permutations.*

2.2 The sponge construction

The sponge construction [11] builds a function $\text{SPONGE}[f, \text{pad}, r]$ with domain $\mathbb{Z}_2^*$ and co-domain $\mathbb{Z}_2^\infty$ using a fixed-length transformation or permutation f, a sponge-compliant padding rule "pad" and a parameter *bitrate* r.

A finite-length output can be obtained by truncating it to its ℓ first bits. We call an instance of the sponge construction a sponge function.

The transformation or permutation f operates on a fixed number of bits, the *width* b . The sponge construction has a state of b bits. First, all the bits of the state are initialized to zero. The input message is padded and cut into r-bits blocks. Then it proceeds in two phases: the *absorbing phase* followed by the *squeezing phase*. In these phases the first r bits of the state and the remaining $b - r$ bits of the state s are treated differently. We denote the former by the outer part $\bar{s}$ and the latter by the inner part or inner state $\hat{s}$. The length of the inner state is $b - r$ and is called the *capacity* c . The two phases are:

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

Squeezing phase The outer part of the state is iteratively returned as output blocks, interleaved with applications of the function f. The number of iterations is determined by the requested number of bits ℓ.

Finally the output is truncated to its first ℓ bits. The c-bit inner state is never directly affected by the input blocks and never output during the squeezing phase. The capacity c actually determines the attainable security level of the construction, as proven in Chapters 5 and 6. We use the term *random sponge* to denote a sponge function with f a random transformation or permutation.

The term *generic attack* is often used. For sponge functions we define it as follows:

Definition 7. *An attack on a sponge function is a* generic attack *if it does not exploit specific properties of f.*

The sponge construction is illustrated in Figure 2.1, and Algorithm 1 provides a formal definition.

In our original paper on sponge function [11] we treated a more general case with the outer part and message blocks being elements of an arbitrary group and the inner part elements of an arbitrary set. Because of its practical relevance, we abandon this generic representation to the more specific case where the state is a binary string of a given length b and the message blocks are r-bit strings.

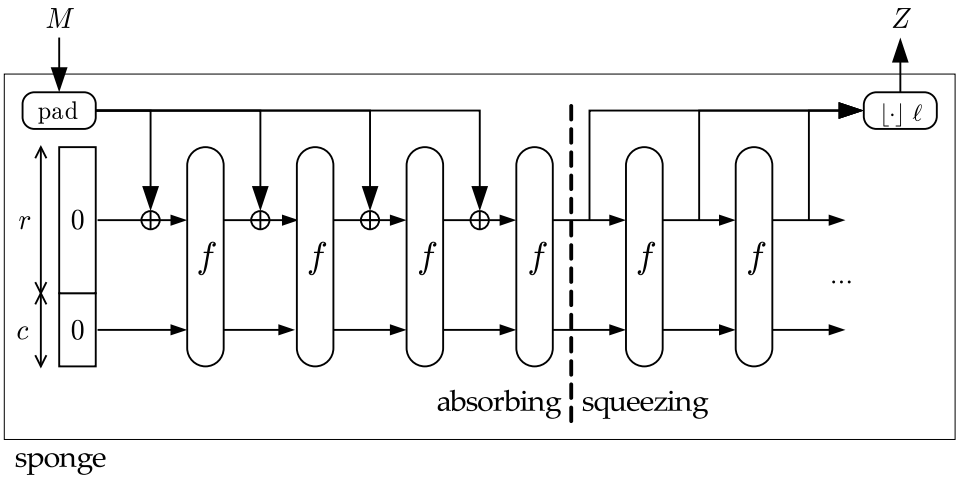

Figure 2.1: The sponge construction $Z = \text{SPONGE}[f, \text{pad}, r](M, \ell)$

2.3 The duplex construction

Like the sponge construction, the *duplex construction* $\text{DUPLEX}[f, \text{pad}, r]$ uses a fixed-length transformation or permutation f, a padding rule pad and a parameter bitrate r to build a cryptographic scheme [14]. Unlike a sponge function that is stateless in between calls, the duplex construction results in an *object* that accepts calls that take an input string and return an output string that depends on all inputs received so far. We call an instance of the duplex construction a *duplex object*, which we denote D in our descriptions. We prefix the calls made to a specific duplex object D by its name D and a dot.

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

Require: $r < b$

Interface: $Z = \text{sponge}(M, \ell)$ with $M \in \mathbb{Z}_2^*$, integer $\ell > 0$ and $Z \in \mathbb{Z}_2^\ell$
$P = M||\text{pad}[r](|M|)$
$s = 0^b$
for $i = 0$ to $|P|_r - 1$ **do**
  $s = s \oplus (P_i||0^{b-r})$
  $s = f(s)$
end for
$Z = \lfloor s \rfloor_r$
while $|Z|_r r < \ell$ **do**
  $s = f(s)$
  $Z = Z||\lfloor s \rfloor_r$
end while
return $\lfloor Z \rfloor_\ell$

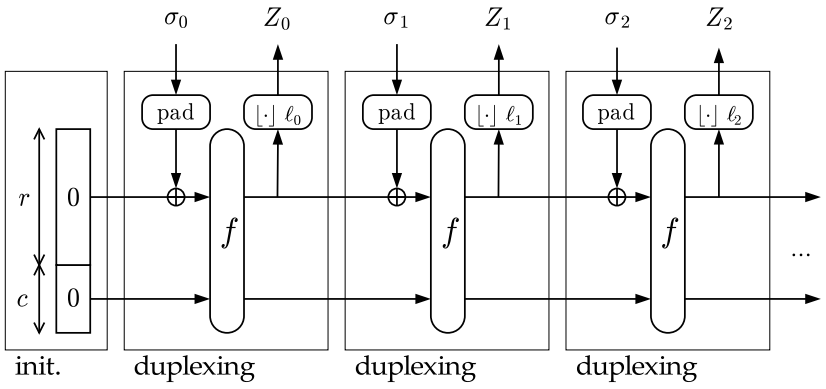

Figure 2.2: The duplex construction

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

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

$$\rho_{\max}(\text{pad}, r) = \min\{x : x + |\text{pad}[r](x)| > r\} - 1\ . \tag{2.2}$$

Clearly the maximum duplex rate is smaller than the bitrate and its value is maximized by taking a padding rule which adds as few bits as possible.

Upon receipt of a D.duplexing(σ, ℓ) call, the duplex object pads the input string σ and XORs it into the outer part of the state. Then it applies f to the state and returns the first ℓ bits of the outer part of the state at the output. We denote a call with σ the empty string by the term *blank call*, and a call with $\ell = 0$, i.e., without output a *mute call*. The duplex construction is illustrated in Figure 2.2, and Algorithm 2 provides a formal definition.

In Section 6.4 we prove that the output of a duplexing call is the output of a sponge function and that as such the duplex construction is as secure as the sponge construction

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

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

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

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

with the same parameters.

2.4 Auxiliary functions

In this section we define two auxiliary functions that simplify the expression of and reasoning about attacks on the sponge construction.

2.4.1 The absorbing function and path

The first auxiliary function is the absorbing function $\textsc{absorb}[f, r]$. It takes as input a string P with $|P|$ multiple of r and returns the value of the state obtained after absorbing P. The absorbing function is defined in Algorithm 3.

In our original paper on sponge function [11] we called this the S_f function.

Algorithm 3 The absorbing function $\textsc{absorb}[f, r]$

Require: $r < b$

Interface: $s = \text{absorb}(P)$ with $P \in \mathbb{Z}_{2^r}^*$ and $s \in \mathbb{Z}_2^b$
$s = 0^b$
for $i = 0$ to $|P|_r - 1$ **do**
  $s = s \oplus (P_i || 0^{b-r})$
  $s = f(s)$
end for
return s

Definition 8. *We call P a* path *to the state s if $s = \text{absorb}(P)$.*

Clearly absorb(empty string) $= 0^b$. In general, the j-th block of the output of a sponge function corresponding to an input M is equal to:

$$Z_j = \overline{\text{absorb}}(P || 0^{rj}),\ j \geq 0, \tag{2.3}$$

with $P = M || \text{pad}[r](|M|)$.

Alternatively, the absorbing function can be used to express the states that the sponge traverses both as it absorbs an input M and as it is being squeezed. The traversed states are absorb(P') for any P' prefix of $P||0^\infty$, with $P = M||\text{pad}[r](|M|)$, including the empty string.

2.4.2 The squeezing function

An auxiliary function that is in some way the dual of the absorbing function is the squeezing function SQUEEZE$[f, r]$. For a given state s, squeeze(s, ℓ) denotes the output truncated to ℓ bits of the sponge function with s the state at the beginning of the squeezing phase. The squeezing function is defined in Algorithm 4.

Algorithm 4 The squeezing function SQUEEZE$[f, r]$

Require: $r < b$

Interface: $Z =$ squeeze(s, ℓ) with $s \in \mathbb{Z}_2^b$, integer $\ell > 0$ and $Z \in \mathbb{Z}_2^\ell$
$Z = \lfloor s \rfloor_r$
while $|Z|_r r < \ell$ **do**
  $s = f(s)$
  $Z = Z || \lfloor s \rfloor_r$
end while
return $\lfloor Z \rfloor_\ell$

2.5 Primary attacks on a sponge function

In this section we present a number of attacks that apply to sponge functions due to their final state and hence do not apply to a random oracle. These attacks impose upper limits to the security that a sponge function can offer and are as such fundamental. For that reason we call them *primary attacks*. In Chapter 5 we will provide generic algorithms for these attacks. Note that in [11] and [15] the primary attacks were called critical operations.

The sponge construction can be defined as the subsequent application of a padding rule, an absorbing function and a squeezing function. For $Z =$ SPONGE$[f, \text{pad}, r](M, \ell)$, we have:

$$\begin{array}{rcl} P & = & M || \text{pad}[r](|M|) \\ s & = & \text{ABSORB}[f, r](P) \\ Z & = & \text{SQUEEZE}[f, r](s, \ell). \end{array} \tag{2.4}$$

It is in general hard to find a path P to a given state s and hard to find the state s for a given output Z. So, the two auxiliary functions of the sponge construction are hard to invert. Moreover, it is generically hard to find two different paths to the same state. The latter are called *state collisions* and can be fully defined in terms of the absorbing function.

Definition 9. *A* state collision *is a pair of different paths $P \neq Q$ to the same state:* absorb$(P) =$ absorb(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 outputs: absorb$(P) =$ absorb(Q) implies that the squeezing part will give the same output values absorb$(P||0^{rj}) =$ absorb$(Q||0^{rj})$ for all j. State collisions can also model cycles in the output sequence: if for some P and d we have absorb$(P) =$ absorb$(P||0^{dr})$, the output sequence displays periodicity.

Definition 10. *An* inner collision *is a pair of different paths $P \neq Q$ to the same inner state:* $\widehat{\text{absorb}}(P) = \widehat{\text{absorb}}(Q)$.

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 $\widehat{\text{absorb}}(P) = \widehat{\text{absorb}}(Q)$, one can produce a state collision on $P||A \neq Q||B$ for any $A, B \in \mathbb{Z}_2^r$ that satisfy $\overline{\text{absorb}}(P) \oplus A = \overline{\text{absorb}}(Q) \oplus B$.

In general it is hard to find a state s such that $\text{squeeze}(s, |Z|) = Z$ for long strings Z. Depending on the origin of Z and the goal of the adversary, we distinguish between two cases: output binding and state recovery.

In output binding the string Z is not necessarily the result of the squeezing of a state and so there may be no solution.

Definition 11. *Given an arbitrary string Z, output binding is finding a state s such that* $\text{squeeze}(s, |Z|) = Z$.

The expected number of states that squeeze to a given string Z is $2^{b-|Z|}$. If $|Z| > b$, the probability that such a state exists is $\approx 2^{b-|Z|}$.

In state recovery the string Z has been obtained by the squeezing of a state s. There may be other state values $s' \neq s$ that upon squeezing result in the provided string but the state s is considered as the only solution.

Definition 12. *State recovery is finding a state s, given a string Z with $Z = \text{squeeze}(s, |Z|)$.*

If $|Z| > b$, it is likely that there is only a single solution and that output binding results in recovery of the unique state that it was squeezed from. If $|Z| \leq b$ there are typically several states that squeeze to Z and output binding does not necessarily result in state recovery.

Chapter 3

Sponge applications

In this chapter we explain modes of use of the sponge construction giving rise to a wide range of cryptographic functions.

The modes presented in this chapter do not only apply to sponge functions, but to any function that has a security claim with a random sponge as security reference or a flat sponge claim. So whenever the text says "sponge function", it means "functions claimed to behave like a random oracle". We will use the symbol F to denote such a function. Some of the presented modes do not require F to support variable-length outputs and some of them do not require the support for long inputs.

3.1 Basic techniques

A sponge function only takes a single input M, that is a string of arbitrary length. Unlike some other constructions, a sponge function does not have a so called initial value (IV) that can be used as an additional input. The sponge function treats the input M as a white page and a mode of use may apply structure to this input. The different modes of sponge functions simply consist of ways to map different types of inputs such as keys, diversifiers, messages to the sponge input M and truncation of the output to the desired length. In this section we present a number of basic techniques that can be used to construct modes.

3.1.1 Domain separation

Thanks to the randomness and arbitrary-length input of random oracles, a single random oracle can be used to implement multiple random oracles using the mechanism of *domain separation* . It suffices to partition the domain in multiple cosets. One example of domain separation is partitioning the strings between those that start with 0 and those that start with 1. Given a single random oracle $\mathcal{RO}$, this allows defining two independent random oracles $\mathcal{RO}_0(M) \triangleq \mathcal{RO}(0||M)$ and $\mathcal{RO}_1(M) \triangleq \mathcal{RO}(1||M)$. This is a very powerful tool in building different types of functions using a random oracle. The mechanism of domain separation can likewise be applied to sponge functions.

In the remainder of this section we present a simple scheme that allows anyone to delimit his/her part of the domain and impose his/her own format within that domain. The idea is to apply domain separation by fixing the first part of the input to a namespace name. The owner of the namespace can then define the format of any input data, appended to the namespace name. We propose the namespace name to be a uniform resource identifier (URI) [32], similarly to what is done for XML [60]. The namespace name is encoded in UTF-8 [31]

as a sequence of bytes, followed by the byte 0^8:

$$F_{\text{NS}}[\text{ns}](\text{data}) \triangleq F(\text{UTF8}(\text{ns})||0^8||\,\text{encode}_{\text{ns}}(\text{data})),$$

where $\text{encode}_{\text{ns}}$ is a function defined by the owner of the namespace ns. This realizes domain separation: two inputs, formatted using different namespaced conventions, will thus always be different.

Using a specific namespace also implies how the output of the sponge function is used. The namespace owner can decide what is the output length, if not arbitrarily long, or in which way the desired output length is encoded.

3.1.2 Keying

One can turn a sponge function into a keyed function by including in the input a secret key K. In its most simple form, M consists of the concatenation of a key K and an input M', so either $M = K||M'$ or $M = M'||K$. Traditionally, such a function is called a pseudo-random function (PRF) $F_K(M')$. If the sponge function behaves like a random oracle, the PRF behaves as a random function to anyone not knowing the key K but having access to the sponge function. The key can be put before or after the message. Putting it before allows state precomputation (see Section 3.1.3) and results in better resistance against generic attacks. We refer to Section 5.11 for a more discussion on this.

3.1.3 State precomputation

A sponge function processes its input M in blocks of r bits. One may apply some form of padding in the formatting of the input to pre-compute state values. For example, if in a keyed sponge the key K is padded to a complete input block, one can compute the state value obtained after absorbing the key and store this. When evaluating the keyed sponge for this particular key K, one can start directly from the stored state value, saving a call to f. In this respect it is best to place the input parameters that change least often at the beginning. This is a technique that can be applied in modes where some input fields keep their value for many calls, such as the key or a namespace name.

3.2 Modes of use of sponge functions

In Table 3.1, we present a number of modes of use of a sponge function.

The first six modes in Table 3.1 do not require the support of a variable-length output and can hence be implemented with hash functions, in as far as they are claimed to behave as random oracles.

A sponge function can be used as an n-bit hash function by simple truncation of its output. If the hash function is to be used in the context of randomized hashing, a random value (i.e., the salt) can be prepended to the message. Domain separation using the same prepending idea applies if one needs to simulate independent hash function instances.

A slow n-bit one-way function can be built by appending the input with N zero bits taking for N a large number. Slow one-way functions are useful as so-called password-based key derivation functions, where the relative high computation time protects against password guessing. The function can be made arbitrarily slow by increasing N. Note that if f is a permutation increasing N does not result in entropy loss.

Functionality	Expression	Input	Output
n-bit hash function	$h = H(M)$	M	$\lfloor Z \rfloor_n$
n-bit randomized hash function	$h = H_R(M)$	$R\|\|M$	$\lfloor Z \rfloor_n$
n-bit hash function instance differentiation	$h = H_D(M)$	$D\|\|M$	$\lfloor Z \rfloor_n$
Slow n-bit one-way function	$h = H_{\text{slow}}(M)$	$M\|\|0^N$	$\lfloor Z \rfloor_n$
n-bit MAC function	$T = \text{MAC}(K, [\text{IV},]M)$	$K\|\|\text{IV}\|\|M$	$\lfloor Z \rfloor_n$
Random-access stream cipher (n-bit block)	$z_i = F(K, \text{IV}, i)$	$K\|\|\text{IV}\|\|i$	$\lfloor Z \rfloor_n$
Stream cipher	$Z = F(K, \text{IV})$	$K\|\|\text{IV}$	as is
Deterministic random bit generator (DRBG)	$z = \text{DRBG}(\text{seed})$	seed	as is
Mask generating and key derivation function	$\text{mask} = F(\text{seed}, \ell)$	seed	$\lfloor Z \rfloor_\ell$

Table 3.1: Examples of usage scenario's for a random oracle

A message authentication code (MAC) takes as input a key, an initial value (IV) and a message. It is basically just a PRF where the message is extended with an IV. The random-access stream cipher mode works similarly to the Salsa20 family of stream ciphers [9]: it takes as input a key, a nonce and a block index and produces a block of key stream.

A sponge function can also be used as a stream cipher. One can input the key and some initial value and then get key stream in the squeezing phase. Similarly, a simple pseudo-random bit generator can be constructed by absorbing the seed data and and then producing the desired number of bits. For having a mask generating function (also called key derivation function) one simple uses as input the seed and one truncates the output to the requested number of bits.

Our presentation of modes covers most common applications of sponge functions but is not meant to be exhaustive and other modes can be readily built. For example, if a randomized MAC function is required, it suffices to take as input the concatenation of a key K, a random salt R and the input M'.

3.3 Parallel and tree hashing

Tree hashing (see, e.g., [47, 58]) can be used to speed up the computation of a hash function by taking advantage of parallelism in modern architectures. It can be defined in terms of a sponge function as compression function F. In this section, we propose a tree hashing scheme.

In a nutshell, the construction works as follows. Consider a rooted tree, with internal nodes and leaves. We call the root of the tree its *final node*. The input message is cut into blocks, which are spread onto the leaves. To each leaf node one then applies F and truncates its output to C bits to form its *chaining value*. An internal node gathers the chaining values of its (ordered) sons, concatenates them and applies F again to result in its chaining value. This process is repeated recursively until the final node is reached. The output of the tree hash function is obtained by applying F to the final node resulting in an indefinite length output. The calls to F, for different nodes, process independent data and so can be parallelized.

Since the input message is arbitrarily long and a priori unknown, we have to define how the tree can grow or how a tree with a fixed number of nodes can accept a growing number of input blocks.

Note that rather than a plain sponge function, the compression function F may also take a key and/or and salt. If these parameters are at the beginning of the input M, one may apply

precomputation of the state once and for all nodes.

3.3.1 Specifications

Our tree hashing mode supports two options:

final node growing (FNG) The degree of the final node grows as a function of the input message length, and the number of leaves increases proportionally.

leaf interleaving (LI) The tree size and the number of leaves are determined by tree mode parameters and independent of the message length, but the message input blocks are interleaved onto the leaves.

The three hashing scheme takes two inputs: a message M that is a binary string and a set of tree parameters, collectively denoted A:

- the tree growing mode $G \in \{\text{LI}, \text{FNG}\}$;
- the height H of the tree, with $H > 0$;
- the degree D of the nodes;
- the leaf block size B.
- the chaining value size C.

When $G = \text{LI}$, the tree is a balanced rooted tree of height H: All internal nodes have degree D. When $G = \text{FNG}$, the degree of the final node depends on the input message length and all other internal nodes have degree D.

The tree has $H + 1$ levels of nodes indexed by k. At level 0 we have leaf nodes containing message bits. The nodes at the other levels $k > 1$ contain the concatenation of chaining values, where each chaining value is obtained by applying F to a node at level $k - 1$ and truncate its output to C bits. At level H there is only a single node, called the *final node*. The output of the hashing mode is obtained by applying F to this node.

The mode can now be fully defined by specifying how the nodes are formed. All nodes end with two node-type frame bits. The first of these bits indicates whether it is a leaf node (1) or not (0) and the second bit indicates whether it is a final node (1) or not (0). We explain now how the remaining parts of the nodes are formed.

We denote the number of B-bit blocks in the message by $|M|_B$ and index the message blocks from 0 to $|M|_B - 1$. Note that message block $|M|_B - 1$ may have less than B bits. We denote the number of leaf nodes by L. At each level we index the nodes starting from 0. The node with index i at level k with $0 < k < H$ contains the sequence of the D chaining values corresponding with the nodes at level $k - 1$ with indices iD to $i(D + 1) - 1$ respectively.

If $G = \text{LI}$, we have $L = D^H$ and at level k there are D^{H-k} nodes. The leaf with index i contains the sequence of message blocks $M_i, M_{i+L}, M_{i+2L}, \ldots$. The final node has the sequence of chaining values of the D nodes of level $H - 1$, followed by the coding of the tree parameters H, D, B, C each coded as two bytes, followed by a byte that codes $G = \text{LI}$.

If $G = \text{FNG}$, we have $L = RD^{H-1}$ with $R = \left\lceil \frac{|M|_B}{D^{H-1}} \right\rceil$ and at level k there are $D^{H-(k+1)}$ nodes. the leaf with index i has message block i if $i < |M|_B$ or no message block otherwise. The final node has the sequence of chaining values of the R nodes of level $H - 1$, followed by the coding of the tree parameters H, D, B, C each coded as two bytes, followed by a byte that codes $G = \text{FNG}$.

If the optimal number of independent processes is known, one can simply use the LI mode ($G = \text{LI}$) with $H = 1$ and D equal to or greater than this number of independent processes. Tree hashing in this case comes down to a simple parallel hashing, where the B-bit blocks of the input message are equally spread onto D different sponge functions. The D results are then combined at the final node to make the final output string.

In addition to the LI and FNG growing modes, one can make the tree grow by increasing its height H until the number of leaves L is large enough for $|M|$. Setting $G = \text{LI}$ in this case does not really interleave the input blocks, but fixes the tree. Knowing whether a node is going to be the final node (if H is large enough) or not becomes significant only at the end of the absorbing phase of a node. Once H is large enough, the implementation can then fix it and mark the candidate final node as final.

3.3.2 Soundness

In [13], we define a set of four conditions for a tree hashing mode to be *sound*. Here soundness is defined in the scope of the indifferentiability framework [45]. For a sound tree hashing mode, its $\mathcal{RO}$ differentiating advantage is upper bounded by $q^2/2^{C+1}$ with q the number of queries to the underlying hash function and C the length of the chaining values.

Our mode satisfies the four following conditions, hence is sound. For the terminology, please refer to [13].

- The mode is tree-decodable. The final node allows to determine the value of all tree parameters, whose knowledge is sufficient to decode all nodes.
- The mode is message-complete, as each message bit is assigned to a leaf node. The length of the message can be determined from the length of the leaf nodes.
- The mode is parameter-complete as it codes the value of all tree parameters in the final node.
- The mode enforces domain separation between final and inner nodes.

From the soundness of the construction, the $\mathcal{RO}$ differentiating advantage of this scheme is $N^2 2^{1-C}$ with C the length of the chaining values and N the number of calls to the underlying function F. If F has a claimed security level indicated by a capacity c, the optimum choice is to take the value of C equal to this capacity c, resulting in a total claimed $\mathcal{RO}$ differentiating advantage of $N^2 2^{-c}$.

Chapter 4

Duplex applications

In this chapter we present modes built on top of the duplex construction. We present an efficient authenticated encryption mode, a reseedable pseudo random bit sequence generator (PRG) and a hash function construction called *overwrite mode*.

4.1 Authenticated encryption

Authenticated encryption (AE) has been extensively studied in the last ten years. Block cipher modes clearly are a popular way to provide simultaneously both integrity and confidentiality. Many block cipher modes have been proposed, e.g., [5, 33, 37, 55, 51, 7, 41, 44, 56, 52] and most of these come with a security proof against generic attacks. Interestingly, there have also been attempts at designing dedicated hybrid primitives offering efficient simultaneous stream encryption and MAC computation [29, 61]. However, these primitives were shown to be weak [48, 53, 64]. The mode we present in this section shares with these hybrid primitives that it offers efficient simultaneous stream encryption and MAC computation. It shares with the block cipher modes that it has provable security against generic attacks. However, it is the first such construction that requires a permutation rather than a block cipher. An important efficiency parameter of an AE mode is the number of calls to the block cipher or to the permutation per block. While encryption or authentication alone require one call per block, some AE modes only require one call per block for both functions. The duplex construction naturally provides a good basis for building such an AE mode.

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

4.1.1 Modeling authenticated encryption

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

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

We assume the wrapping and unwrapping operations as such to be deterministic. Hence two inputs (A, B) and (A', B') that are equal will under the same key K give rise to the same

output (C, T). If this determinism is a problem, it can be tackled by expanding A with a nonce.

4.1.2 Security requirements

For a key K chosen secretly and uniformly over $|K|$ bits, an authenticated encryption scheme that satisfies the following security requirements would be very useful:

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

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

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

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

4.1.3 An ideal system

We can define a reference system that satisfies these requirements using a pair of *random oracles* $(\mathcal{RO}_1, \mathcal{RO}_2)$, with encryption and tag computation implemented as follows:

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

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

Key stream sequences give no information on tags and vice versa as they are obtained by calls to different random oracles. Additionally, as the key is only used as input to random oracles, the key recovery infeasibility requirement is satisfied. The two random oracles $\mathcal{RO}_1$ and $\mathcal{RO}_2$ can be implemented from a single random oracle $\mathcal{RO}$ using domain separation.

The simplest way to build an actual system that behaves as the reference system described above would be to replace the random oracle $\mathcal{RO}$ by a sponge function. However, such a solution requires two sponge function executions: one for the generation of the key stream and one for the generation of the tag. We aim for a solution that requires only a single call to f per input block. To achieve this, we define a mode on top of the duplex construction.

4.1.4 The authenticated encryption mode SpongeWrap

We propose an authenticated encryption mode SpongeWrap that realizes a generalization of the authenticated encryption process defined in Section 4.1.1. Similarly to the duplex construction, we call an instance of the authenticated encryption mode a SpongeWrap object.

Upon initialization of a SpongeWrap object, it loads the key K. From then on one can send requests to it for wrapping and/or unwrapping data. The key stream blocks used for encryption and the tags depend on the key K and the data sent in all previous requests. The process defined in Section 4.1.1 can be implemented with the SpongeWrap mode using only a single wrap or unwrap request.

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

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

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

Before being forwarded to D, every key, header, data or cryptogram block is extended with a so-called *frame bit*. Note that if A, B or C are the empty string, they are treated as having a single block consisting of the empty string. The rate ρ of the SpongeWrap mode determines the size of the blocks and hence the maximum number of bits processed per call to f. Its upper bound is $\rho_{\max}(\text{pad}, r) - 1$ due to the inclusion of one frame bit per block. A formal definition of SpongeWrap is given in Algorithm 5.

4.1.5 Security

Theorem 1. *The authenticated encryption mode SpongeWrap$[f, \text{pad}, r, \rho]$ defined in Algorithm 5 satisfies the security requirements of key recovery, tag forgery and plaintext recovery described in Section 4.1.2 if sponge$[f, \text{pad}, r]$ is secure.*

Proof. This modes follows the ideal construction of Section 4.1.3, with two differences: first, the random oracle is replaced by a sponge function (via the duplexing-sponge lemma) and second, we allow the key stream to depend on previous blocks. For the former, the security of the SpongeWrap mode thus depends on the security of the underlying sponge function. The introduction of the dependency on previous blocks does not reduce the security of the ideal construction but is required to match the interface of the duplex construction. Hence, the security of the SpongeWrap mode reduces to the ability to have injective encoding functions s_k and h_t.

The frame bit used in Algorithm 5 serves two purposes:

Domain separation The duplex (or equivalently, sponge) inputs to generate key stream blocks and those to generate tag blocks are in separate domains. Every duplex re-

Algorithm 5 The authenticated encryption mode SpongeWrap$[f, \text{pad}, r, \rho]$.

Require: $\rho \leq \rho_{\max}(\text{pad}, r) - 1$
Require: $D = \text{duplex}[f, \text{pad}, r]$
This algorithm treats A, B or C instances equal to the empty string as a single (empty) block

Interface: W.initialize(K) with $K \in \mathbb{Z}_2^+$
D.initialize()
for $i = 0$ to $|K|_\rho - 2$ **do**
  D.duplexing$(K_i||1, 0)$
end for
D.duplexing$(K_{|K|_\rho - 1}||0, 0)$

Interface: $(C, T) = W$.wrap(A, B, ℓ) with $A, B\mathbb{Z}_2^*$, integer $\ell > 0$, $C \in \mathbb{Z}_2^{|B|}$ and $T \in \mathbb{Z}_2^\ell$
for $i = 0$ to $|A|_\rho - 2$ **do**
  D.duplexing$(A_i||0, 0)$
end for
$Z = D$.duplexing$(A_{|A|_\rho - 1}||1, |B_0|)$
$C = B_0 \oplus Z$
for $i = 0$ to $|B|_\rho - 2$ **do**
  $Z = D$.duplexing$(B_i||1, |B_{i+1}|)$
  $C = C||(B_{i+1} \oplus Z)$
end for
$Z = D$.duplexing$(B_{|B|_\rho - 1}||0, \rho)$
while $|Z| < \ell$ **do**
  $Z = Z||D$.duplexing$(0, \rho)$
end while
$T = \lfloor Z \rfloor_\ell$
return (C, T)

Interface: $B = W$.unwrap(A, C, T) with $A, C \in \mathbb{Z}_2^*$, $T \in \mathbb{Z}_2^+$, $B \in \mathbb{Z}_2^{|C|} \cup \{\text{error}\}$
for $i = 0$ to $|A|_\rho - 2$ **do**
  D.duplexing$(A_i||0, 0)$
end for
$Z = D$.duplexing$(A_{|A|_\rho - 1}||1, |C_0|)$
$B_0 = C_0 \oplus Z$
for $i = 0$ to $|C|_\rho - 2$ **do**
  $Z = D$.duplexing$(B_i||1, |C_{i+1}|)$
  $B_{i+1} = C_{i+1} \oplus Z$
end for
$Z = D$.duplexing$(B_{|C|_\rho - 1}||0, \rho)$
while $|Z| < \ell$ **do**
  $Z = Z||D$.duplexing$(0, \rho)$
end while
if $T = \lfloor Z \rfloor_\ell$ **then**
  return $B_0||B_1||\ldots B_w$
else
  return Error
end if

sponse that is used to encipher the next block has as input a string ending with a frame bit 1, whereas every duplex response that is used to form a tag has as input a string ending with a frame bit 0.

Decodability The key, header and body blocks can be recovered from the duplex input sequence. This implies that two different sequences K, $A^{(0)}$, $B^{(0)}$, $A^{(1)}$, $B^{(1)}$, ... and K', $A'^{(0)}$, $B'^{(0)}$, $A'^{(1)}$, $B'^{(1)}$, ... cannot lead to two equal duplex input sequences. This follows from the sequences of blocks representing K, A and B that can be delimited using frame bits. Namely, the last key block can be identified by a frame bit 0. Then, for each $W.\text{wrap}(A, B, \ell)$ call, the last block of A is identified by a frame bit 1, and the last block of B by a frame bit 0. Finally, the duplexing inputs containing only the frame bit 0 can only be used for producing the tag T, as the $\{A_i\}_{i<v}$ blocks of the next $W.\text{wrap}(A, B, \ell)$ call cannot be empty.

The theorem follows from the following properties:

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

□

4.1.6 Advantages and limitations

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

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

As compared to some block cipher based authenticated encryption modes, it has some limitations. First, the mode as such is serial and cannot be parallelized at algorithmic level. Some block cipher based modes do actually allow parallelization, for instance, the offset codebook (OCB) mode [54]. Yet, SpongeWrap can support parallel streams in a fashion similar to tree hashing, but with some overhead.

Second, if a system does not impose the nonce requirement on A, an attacker may send two requests (A, B) and (A, B') with $B \neq B'$. In this case, the first differing blocks of B and B', say B_i and B'_i, will be enciphered with the same key stream, making their bitwise XOR available to the attacker. Some block cipher based modes are *misuse resistant*, i.e., they are designed in such a way that in case the nonce requirement is not fulfilled, the only information an attacker can find out is whether B and B' are equal or not [56]. Yet, many applications already provide a nonce, such as a packet number or a key ID, and can put it in A.

4.1.7 An application: key wrapping

Key wrapping is the process of ensuring the secrecy and integrity of cryptographic keys in transport or storage, e.g., [49, 27]. A *payload key* is wrapped with a *key-encrypting key* (KEK). We can use the SpongeWrap mode with K equal to the KEK and let the data body be the payload key value. In a sound key management system every key has a unique identifier. It is sufficient to include the identifier of the payload key in the header A and two different payload keys will never be enciphered with the same key stream. When wrapping a private key, the corresponding public key or a digest computed from it can serve as identifier.

4.2 Reseedable pseudo random bit sequence generation

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 bit sequence generator (PRG) 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 PRG is similar to a stream cipher.

Finally, some applications require a pseudo-random bit sequence generator to support forward security: The compromise of the current state does not enable the attacker to determine the previously generated pseudo-random bits [8, 23].

The state of the PRG 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 PRG. Hence, we need a way to gather and combine seeding material coming from several sources into the state of the PRG. Loading different seeds into the PRG shall result in different output sequences. The latter implies that different seeds result in different state values. In this respect, a PRG is similar to a cryptographic hash function that should be collision-resistant.

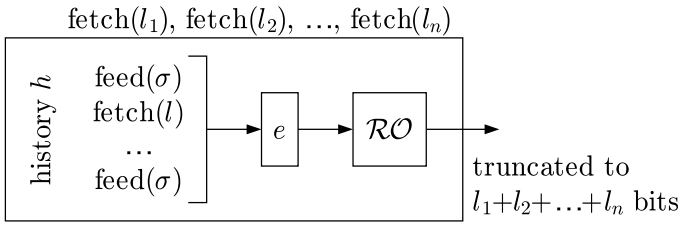

Figure 4.1: Response of an ideal PRG to fetch requests

It is convenient for a pseudo-random bit sequence generator to be reseedable, i.e., to allow the insertion of additional seeding material after pseudo-random bits have been generated. Instead of throwing away the current state of the PRG, reseeding combines the current state of the generator with the new seeding material. From a user's point of view, a reseedable PRG can be seen as a black box with an interface to request pseudo-random bits and an interface to provide fresh seeds. In the sequel we will use PRG to indicate a reseedable pseudo-random bit sequence generator.

4.2.1 Modeling an ideal PRG

We define a PRG 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 PRG;
- *fetch* request, `fetch`(ℓ), instructs the PRG to return ℓ bits.

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

Informally, the requirements for a PRG 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 [6]. For a reseedable PRG we cannot just use a random oracle as it has a different interface. However, we define an ideal PRG 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 x to $x + \ell - 1$ of its response to the requester, with x 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 4.1. 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 PRG 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 PRG returns independent and a priori uniformly distributed bits.

We thus propose the following definition of an ideal PRG.

Definition 13. *An* ideal PRG *is a history-keeping mode calling a random oracle with an encoding function $e(h)$ that is seed-complete.*

Often one sees the security requirement called *forward security*, also called forward secrecy. This requires that the compromise of the current state does not enable the attacker to determine the previously generated pseudo-random bits [8, 23]. Note that our ideal PRG does not satisfy this requirement.

4.2.2 SpongePRG: a PRG mode

The simplest way to build an actual system that behaves as the reference system described above would be to replace the random oracle $\mathcal{RO}$ by a sponge function. At first sight, this is not practical as it needs to store all past queries and hence requires ever growing amounts of memory. However, making use of a duplex function allows removing these obstacles.

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

In [16] a reseedable PRG was defined based on the sponge construction that implements the required functionality. In this section we present a PRG built on top of the duplex construction, called SpongePRG. This mode is similar to the one proposed in [16] in that it minimizes the number of calls to f.

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

If f is a permutation, the operation of a SpongePRG object is invertible and revealing the state allows the attacker to backtrack the generation back to the most recent unknown seed fed into it. Still, forward security can be explicitly enforced by means of a P.forget() request. The effect of a P.forget() request is the resetting to zero of the first ρ bits of the state and a subsequent application of f. This is done $\lfloor c/\rho \rfloor$ times. Guessing the state before this operation given the state afterwards requires guessing at least c bits and hence is infeasible for reasonable values of c.

The SpongePRG mode is defined in Algorithm 6. Note that the buffers do not require separate storage but can be implemented merely as pointers to the state: The input buffer requires a pointer to the state indicating from where on new bits must be XORed into the state, while the output buffer pointer points in the state where the next output bit must be taken. The storage is thus limited to the b-bit state and two integers.

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

Algorithm 6 Pseudo random bit sequence generator mode SpongePRG$[f, \text{pad}, r, \rho]$

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

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

Interface: P.feed(σ) with $\sigma \in \mathbb{Z}_2^+$
$M = B_{\text{in}} || \sigma$
for $i = 0$ to $|M|_\rho - 2$ **do**
  D.duplexing$(M_i, 0)$
end for
$B_{\text{in}} = M_{|M|_\rho - 1}$
$B_{\text{out}} =$ empty string

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

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

4.2.3 Advantages and limitations

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. If f is a permutation, we can keep the state size small thanks to two reasons. First, the use of a permutation preserves the entropy of the state. Second, we have strong upper bounds on the success probability of generic attacks against keyed sponge instances (see Section 5.11).

Making sure that the seeding material provides enough entropy is out of scope of this document. This aspect has been studied in the literature, e.g., [28, 59] 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. We don't see this as a big disadvantage for the following two reasons. First, regular reseeding with sufficient entropy already prevents the attacker from going backwards. Second, an embedded security device such as a smartcard in which such a PRG would be used is designed to protect the secrecy of keys and therefore reading out the state is expected to be difficult.

4.3 The mode Overwrite

In [35] sponge-like constructions were proposed and cryptanalyzed. In some of these constructions, absorbing is done by overwriting part of the state by the message block rather than XORing it in. A concrete function that follows such a construction is the hash function Grindahl [43].

These overwrite functions have the advantage over sponge functions that between calls to f, only c bits must be kept instead of b. This may not be useful when hashing a message in a continuous fashion, as b bits must be processed by f anyway. However, when hashing a partial message, then putting it aside to continue later on, having to store only c bits may be useful on some platforms.

It turns out that an overwrite function can be built using the duplex construction. If the first ρ bits of the state are known to be Z, overwriting them with a message block P_i is equivalent to XORing in $Z \oplus P_i$. This idea is also used in the forget call of the SpongePRG mode and is formally implemented in Algorithm 7. In practice, of course, the implementation can just overwrite the first ρ bits of the state by a message block. As a matter of fact, Algorithm 7 can be rewritten to call f directly, similar to the sponge construction. We leave this as an exercise for the reader.

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

Theorem 2. *The construction* Overwrite$[f, \text{pad}, r, \rho]$ *is as secure as* sponge$[f, \text{pad}, r]$.

Proof. The construction Overwrite$[f, \text{pad}, r, \rho]$ is defined in terms of calls to duplex$[f, \text{pad}, r]$. From the sponge-duplexing lemma, the output of such a call is the output to sponge$[f, \text{pad}, r]$ for a specific input. Hence, the theorem comes down to showing that the input M to Overwrite can be recovered from the inputs to the duplexing calls.

The coding using the frame bits in Algorithm 7 allows, for any input sequence of D, finding the last block $(P_w \oplus Z)$ and the length of the original input M. To recover the message M from the input sequence, one can start with the first block. Since $Z = 0^\rho$ in the first block, the first block in the D.duplexing() call allows recovering the first block of M. Then, this block allows determining the output Z that was XORed into the next block, and so on. □

We have thus proven that the security of Overwrite is equivalent to that of the sponge construction with the same parameter, but at a cost of 2 bits of bitrate (or equivalently, of capacity): one for the padding rule (assuming pad10^* is used) and one for the frame bit.

Algorithm 7 The construction $\textsc{Overwrite}[f, \text{pad}, r, \rho]$

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

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

Chapter 5

Generic attacks

5.1 Introduction

In this chapter we investigate generic algorithms for the primary attacks on sponge functions. The vulnerability of the sponge construction with respect to these attacks is due to its finite state and hence they do not apply to a random oracle. The success probabilities of primary attacks impose upper limits to the resistance the sponge construction can offer. We also discuss the implications of these algorithms on the security of sponge functions in the context of hashing.

5.2 Graphical representation of a sponge function

One can associate to a sponge function a graph with $2^b = 2^{r+c}$ nodes and 2^b 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. If f is a permutation, 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 2^c supernodes, one for each inner state value. The 2^r nodes within a supernode are identified by the outer part $\overline{s}$ of their state.

One can absorb an input string P by following edges starting from supernode 0, the *root*. First we draw an edge from $P_0||0^c$. This edge arrives in node with outer part $\overline{\text{absorb}}(P_0)$ of supernode $\widehat{\text{absorb}}(P_0)$. Then we draw an edge from the node within that supernode with outer part $\overline{\text{absorb}}(P_0) \oplus P_1$ and the node where it arrives is $\text{absorb}(P_0||P_1)$. For P_i, we draw an edge from the node with outer part $\overline{\text{absorb}}(P_0||P_1 \ldots ||P_{i-1}) \oplus P_i$ within supernode $\widehat{\text{absorb}}(P_0||P_1 \ldots ||P_{i-1})$. 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 block of the path P is determined by the edges arriving at and starting from the i-th supernode on the path: it is the outer part of the node where the outgoing edge starts XOR the outer part 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 outer part of the node where the first edge starts.

5.3 The model of the adversary

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} in the case it is a permutation).

This corresponds to the real-world situation where an adversary has a specification of f and where the most efficient way to compute $f(x)$ (or $f^{-1}(x)$) for a given x is executing a program that computes f (or $f^{-1}(x)$). One may remark that one could precompute and store a large table $(x, f(x))$ couples, but in our model the calls to f for the precomputation is are included in the complexity.

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 graph has no edges. 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 $\widehat{t}$ is reachable from a supernode $\widehat{s}$ if there is a sequence of directed edges from $\widehat{s}$ to $\widehat{t}$ (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.

5.3.1 The cost function

We obtain expressions for the optimal probability of success $\Pr(\text{success})$ as function of N, where N is the number of calls the adversary can make to f in the case it is a transformation and the total number of calls she can make to f and f^{-1} in the case it is a permutation. 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 $\Pr(\text{success})$ for the different primary attacks turn out to be of the form $1 - e^{v(N)}$ with $v(N)$ a polynomial in N of degree one or two. To simplify notation, we define the cost function $c_p(N)$ of an attack by $c_p(\text{success}) = -\log(1 - \Pr(\text{success}))$. This gives:

$$\Pr(\text{success}) = 1 - e^{-c_p(\text{success})} .$$

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

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

5.4 Generating 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 5.1, 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 $R \leq V \leq i$.

5.4.1 With f a random transformation

The adversary can only add edges starting from chosen nodes. If the new edge starts from a rooted node, the probability of success is $V/2^c$. 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:

$$\Pr(\text{no IC}) = \prod_{i=1}^{N}(1 - \frac{i}{2^c}).$$

If $N \ll 2^c$, we can use the $\log(1+\epsilon)$ approximation, yielding:

$$\Pr(\text{IC}) \approx 1 - e^{-\sum_{i=1}^{N} \frac{i}{2^c}} = 1 - e^{-\frac{N(N+1)}{2^{c+1}}} .$$

The cost function is therefore:

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

5.4.2 With f a random permutation

If f is a permutation, the adversary can add edges starting from chosen nodes and additionally edges arriving in chosen nodes. Moreover, 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 the number of nodes in $\mathcal{V}$ with no incoming edge divided by the total number of nodes with no incoming edge:

$$\frac{(2^r - 1)V + 1}{2^{r+c} - i} .$$

If an edge is added arriving in a chosen node in $\mathcal{V}$, the probability of success is similarly

$$\frac{(2^r - 1)R + 1}{2^{r+c} - i} .$$

The higher the values of R and V, the better the probabilities of success in subsequent queries, so one could make other kinds of queries 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.

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

$$\frac{(2^r - 1)i + 1}{2^{r+c} - i} .$$

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

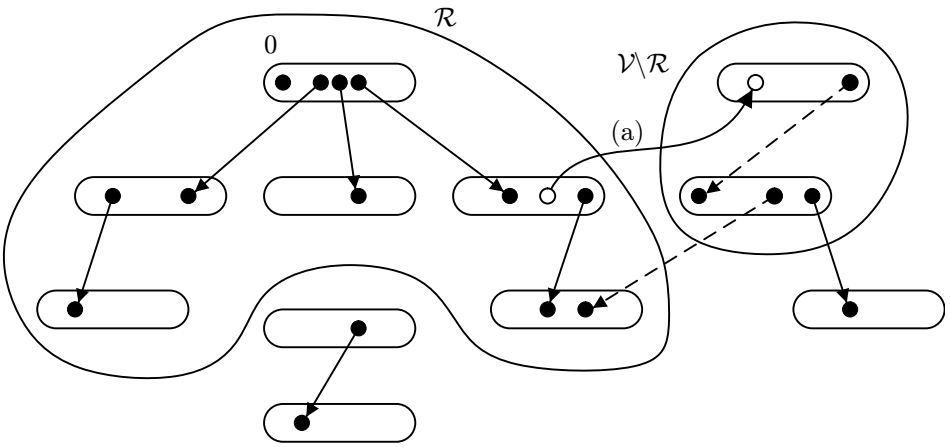

Figure 5.1: Adding an edge (a) resulting in an inner collision. Edge (a) must start in $\mathcal{R}$ and arrive in $\mathcal{V}$.

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:

$$\Pr(\text{no IC}) = \prod_{i=1}^{N} \left(1 - \frac{(2^r-1)i+1}{2^{r+c}-i}\right) = \prod_{i=1}^{N} \frac{1 - \frac{i}{2^c} - \frac{1}{2^{r+c}}}{1 - \frac{i}{2^{r+c}}}.$$

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

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

5.5 Finding a path to an inner state

Given a target inner state $\widehat{t}$, the adversary must find a path p such that $\widehat{\text{absorb}}(p) = \widehat{t}$. 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 5.2 this implies that the new edge must connect a rooted supernode to a supernode from which $\widehat{t}$ 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, $\mathcal{V} = \{\widehat{t}\}$, $\mathcal{R} = \{0\}$ and $R = V = 1$. Right before the i-th call, the graph contains $i-1$ edges and $R \leq i$, $V \leq i$ and $R + V \leq i + 1$.

5.5.1 With f a random transformation

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 $V/2^c$. Otherwise, the probability of success is 0, it leaves R invariant and adds one to V with probability $V/2^c$. 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:

$$\Pr(\text{no path}) = \prod_{i=1}^{N} (1 - \frac{1}{2^c}) .$$

Using the $\log(1+\epsilon)$ approximation for $2^c \gg 1$, this yields:

$$c_p(\text{path}) \approx \frac{N}{2^c} .$$

We will now discuss a variant of the attack: 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

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

and subsequently, if $\ell \ll 2^c$

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

5.5.2 With f a random permutation

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 and if the supernodes of $\mathcal{V}$ with the edges form a tree, the probability of success is the number of nodes in $\mathcal{V}$ with no incoming edges divided by the total number of nodes with no incoming edges, i.e.,

$$\frac{(2^r-1)V+1}{2^{r+c}-i} .$$

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

$$\frac{(2^r-1)R+1}{2^{r+c}-i} ,$$

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 check whether this assumption is justified.

The higher the values of R and V, the better the probabilities of success of subsequent queries. 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 $\widehat{t}$. 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 $\widehat{t}$. 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 δ_i that is 1 if the i-th edge added starts 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

$$1 - \frac{(2^r - 1)\left(\frac{1+\delta_i}{2} V_i + \frac{1-\delta_i}{2} R_i\right) + 1}{2^{r+c} - i} = \frac{1 - \frac{i+1}{2^{r+c}} - \frac{2^r-1}{2^{r+c+1}} \left(V_i + R_i - \delta_i (R_i - V_i)\right)}{1 - \frac{i}{2^{r+c}}}$$

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

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

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 $R_i - V_i = \sum_{j=1}^{i-1} \delta_i$. This gives:

$$c_p(\text{path}) \approx \frac{N}{2^{r+c}} + \frac{2^r - 1}{2^{r+c+2}} \left(N^2 + 3N + 1 - \sum_{i=1}^{N} \sum_{j=1}^{i-1} 2\delta_i \delta_j \right)$$

We can now work out the last term using:

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

Filling this in gives:

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

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:

$$c_p(\text{path}) \approx \frac{N(N+4)}{2^{c+2}} - \frac{N^2}{2^{r+c+2}} \,.$$

the probability of success becomes significant when N is of the order of $2\sqrt{2^c}$ and hence when R and V are of the order $\sqrt{2^c}$. 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.

5.6 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 absorb(P). From this node, the output blocks $\overline{\text{absorb}}(P||0^{jr})$ are generated by following a *chain* of nodes connected by edges, i.e., absorb$(P||0^{jr}) = f(\text{absorb}(P||0^{(j-1)r}))$, where we define a chain as a sequence of nodes connected by directed edges. The first node in the chain is the node $u = \text{absorb}(P') \oplus (P_{|P|_r - 1}||0^r)$ with P' equal to P with its last block $P_{|P|_r-1}$ removed.

The adversary finds a cycle by extending at the end by adding edges until the new edge arrives in a node in the chain. The shortest valid input strings consist of a single non-zero block. Before adding the i-th edge, the chain contains i nodes.

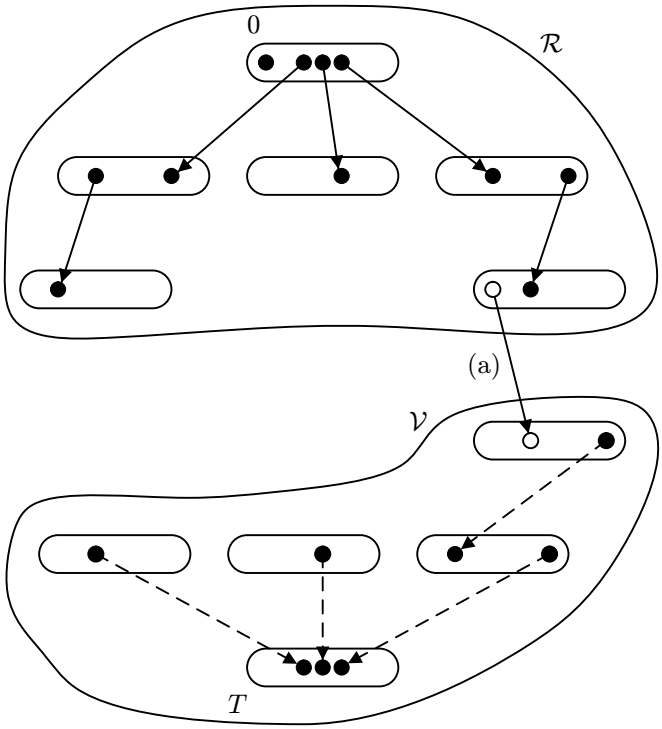

Figure 5.2: Adding an edge (a) resulting in a path. Edge (a) must start in $\mathcal{R}$ and arrive in $\mathcal{V}$.

5.6.1 With f a random transformation

The probability that the new edge arrives in one of the nodes of the chain is $i/2^{r+c}$. Using the $\log(1+\epsilon)$ approximation, this results in:

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

5.6.2 With f a random permutation

At any moment, there is only a single node in the chain that has no incoming edge, the node u. The probability that the new edge arrives in a node in the chain is hence $1/(2^{r+c})$. This results in:

$$c_p(\text{output cycle}) \approx \frac{N}{2^{r+c}} .$$

5.7 State recovery

State recovery consists of finding a state s given a string $Z = \text{squeeze}(s, |Z|)$.

5.7.1 With f a random transformation

The adversary can make guesses a for $\widehat{s}$ and use queries to f to verify their correctness. The probability of success after n guesses $n2^{-c}$. She can verify the correctness of a guess in the following way. She sends a query $f(a||Z_0)$ and check whether the outer part of the result b equals Z_1. If so, it can query $f(b)$ and verify whether the outer part of the result c equals Z_2

and so on. The expected number of queries for a wrong guess is:

$$1 + 2^{-r} + 2^{-2r} + \ldots \approx \frac{1}{1 - 2^{-r}}$$

So the expected success probability after N queries is

$$N \frac{1 - 2^{-r}}{2^c} .$$

5.7.2 With f a random permutation

In this section we will assume that $|Z|$ is a multiple of the bitrate. We will denote the solution s by s_0, $f(s_0)$ by s_1 and $f(s_i)$ by s_{i+1}.

When f is a permutation, the adversary can choose to first determine $\widehat{s_i}$ for some index i and then compute $\widehat{s_0}$ from s_i by repeatedly applying f^{-1}. This has an impact on the success probability.

5.7.2.1 Passive adversary

We first define the *forward and backward block partitions* of a string Z with $|Z| = mr$ and the corresponding *forward multiplicity* and *backward multiplicity*. The forward block partition $B_{\mathrm{f}}(Z)$ is a partition of the block indices i of Z with $0 \leq i < |Z|_r - 1$, grouped by equal values Z_i. We denote the subsets of $B_{\mathrm{f}}(Z)$ by $B_{(j)}$ and their corresponding Z_i values by $Z_{(j)}$. So we have $\forall i \in B_{(j)} : Z_i = Z_{(j)}$ and $\forall i \notin B_{(j)} : Z_i \neq Z_{(j)}$. Note that in the forward block partition, the index of the last block is excluded. The forward multiplicity of a string Z, denoted by $m_{\mathrm{f}}(Z, r)$, is equal to the cardinality of the largest subset of $B_{\mathrm{f}}(Z)$. In other words, it is the number of occurrences of the block value Z_i that occurs most often in Z.

Note that if r is large, for a random string with $|Z|_r < 2^{r/2}$ the forward multiplicity is typically 1, i.e., all blocks Z_i of Z are different. If $r = 1$, the blocks are bits and the forward multiplicity is at least equal to $(|Z| - 1)/2$.

The backward block partition $B_{\mathrm{b}}(Z)$ and backward multiplicity $m_{\mathrm{b}}(Z, r)$ of a string Z are defined in a similar way, with the only difference that the first block of Z is excluded, instead of the last one. So it is a partition of the indices i with $0 < i \leq |Z|_r - 1$.

Finally, we define the *multiplicity* of a string as the maximum of both multiplicities:

$$m(Z, r) = \max\{m_{\mathrm{f}}(Z, r), m_{\mathrm{b}}(Z, r)\}$$

We prove a bound for the case that there exists only a single solution, i.e., one value s_0 such that $\mathrm{squeeze}(s_0, |Z|) = Z$. This is likely if $|Z| > b$ and the probability that more than one solution exists decreases exponentially with $|Z| - b$.

Theorem 3 ([16]). *Given $Z = \mathrm{squeeze}(s, |Z|)$, the success probability of finding s after N queries is upper bound by $m(Z, r)\frac{N}{2^c}$, if there is only a single such value s.*

Proof. Let $F_1(Z)$ be the set of permutations f such that there is only one solution to the state recovery problem with instance Z. For a given value s, within $F_1(Z)$, the inner part of $f(s)$ (or $f^{-1}(s)$) 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 $\widehat{s_0}$ is such that $\overline{f}(Z_0||\widehat{s_0}) = Z_1$, then for any $\widehat{s_0'} \neq \widehat{s_0}$ there exists another permutation $f' \in F_1(Z)$ such that $\overline{f'}(Z_0||\widehat{s_0'}) = Z_1$ 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 $Z_1 \neq Z_2$ and

$(\widehat{s_1}, \widehat{s_2})$ is such that the outer parts of $f(s_i)$ are Z_i for $i = 1, 2$, then for any $(\widehat{s'_1}, \widehat{s'_2}) \neq (\widehat{s_1}, \widehat{s_2})$ there exists another permutation $f' \in F_1(Z)$ where $(\widehat{s'_1}, \widehat{s'_2})$ verifies the same equality.

Let us first consider the case $|Z|_r = 2$. Clearly, $m(Z, r) = 1$.

Let $F_1(Z, a_0, a_1)$ be the subset of $F_1(Z)$ where the value a_0 is the solution for $\widehat{s_0}$ and $f(Z_0||a_0) = (Z_1||a_1)$. The sets $F_1(Z, a_0, a_1)$ partition the set $F_1(Z)$ into 2^{2c} subsets of equal size identified by a_0 and a_1, or in other words, a_0 and a_1 cut the set in an orthogonal way.

The goal of the adversary is to determine in which subset $F_1(Z, a_0, a_1)$ the permutation f is. To do so, she can make two types of queries:

- Forward queries: she queries $f(Z_0||a)$ for a guess a and checks whether the outer part of the response is Z_1
- Backward queries: she queries $f^{-1}(Z_1||a)$ for a guess a and checks whether the outer part of the response is Z_0.

As the subsets $F_1(Z, a_0, a_1)$ cut $F_1(Z)$ orthogonally in a_0 and a_1, forward queries help determine whether a_0 is the solution but without reducing the set of possible values for a_1, and vice-versa for backward queries. So, after N_f forward queries and N_b backward queries, the success probability is

$$1 - \left(1 - N_\mathrm{f} 2^{-c}\right)\left(1 - N_\mathrm{b} 2^{-c}\right) \leq N 2^{-c},$$

where the probability is taken over all permutations f drawn uniformly from $F_1(Z)$.

Let us now consider the general case where $|Z|_r \geq 2$. The reasoning can be generalized in a straightforward way if all the Z_i are different, or more exactly, if $m(Z)r = 1$. Otherwise, some adaptations have to be made to take into account the values Z_i appearing multiple times. Given the set of indices $i, k \ldots$ in a subset $B_{(j)}$, there may or may not be constraints on the possible values that the corresponding inner values $\widehat{s_i}$ can take. For instance, if $Z_{i-1} \neq Z_{k-1}$ or if $Z_{i+1} \neq Z_{i_k+1}$, then necessarily $\widehat{s_i} \neq \widehat{s_k}$. In another example, Z can be periodic, allowing the s_i values to be equal.

The adversary can make a guess for the inner value $\widehat{s_i}$ for all $i \in B_{(j)}$ in a single query in the following way. She makes for a guess a a forward query to check whether $f(Z_{(j)})||a)$ gives as outer value Z_{i+1} for any $i \in T_{(j)}$. The same reasoning can be applied for backward queries. The adversary now makes a backward query to check whether $f^{-1}(Z_{(j)})||a)$ gives as outer value Z_{i-1} for any $i \in B_{(j)}$. So, a forward (resp. backward) query can count as up to $m_\mathrm{f}(Z, r)$ (resp. $m_\mathrm{b}(Z, r)$) chances to hit the correct outer value. If $f(Z_{(j)})||a)$ gives as outer value Z_{i+1} for some $i \in T_{(j)}$, the adversary can check whether this is the correct value by making an additional query $f(f(Z_{(j)})||a))$ and checking whether it gives as outer value Z_{i+2} and so on. In our upper bound we will ignore these additional queries. If r is large, there is typically just an additional query per $2^r/m_\mathrm{f}(Z, r)$ guesses and the bound remains tight. If $r = 1$, they however represent an important factor, resulting in a looser bound.

Let $F_1(Z, a_0, a_1, \ldots, a_{|Z|_r-1})$ be the subset of $F_1(Z)$ for which $(a_0, a_1, \ldots, a_{|Z|_r-1})$ is the solution for $(\widehat{s_0}, \widehat{s_1}, \ldots, \widehat{s_{|Z|_r-1}})$. In general, the elements $\{a_0, a_1, \ldots, a_{|Z|_r-1}\}$ do not cut $F_1(Z)$ in an orthogonal way. Consider now as elements the $|B|$ vectors $A_{(j)}$ with each such vector containing the $|B_{(j)}|$ elements a_i with $i \in T_{(j)}$. The vectors $A_{(j)}$ cut $F_1(Z)$ in an orthogonal way, as they constraint f on different outer values.

So, after n guesses, the probability that one of them gives the solution is at most $m(Z, r)2^{-c}n$, where the probability is taken over all permutations f drawn uniformly from $F_1(Z)$. The bound in the theorem follows from the fact that the average number of queries to be made for each wrong guess is not below $1/(1 - 2^{-r})$. □

5.7.2.2 Active adversary

In some modes of use such as those based on the duplex construction, an adversary may be able to absorb input blocks of choice. This case is covered in the next theorem. We assume that the adversary can choose the blocks P_i that are injected at each iteration, i.e., the mode computes $f(s_i \oplus (P_i||0^c)) = s_{i+1}$ and the adversary observes $Z_{i+1} = \overline{s_{i+1}}$. Now an instance of the problem is also determined by the injected blocks $P = (P_0, P_1, \ldots, P_m)$ (the value of the last block P_m is actually irrelevant, it is just there to simplify notation).

Theorem 4 ([16]). *Given an instance of the active state recovery problem Z, P and knowing that there is one and only one solution $\widehat{s_0}$, the success probability after N queries is at most*

$$\max\{m_{\mathrm{f}}(Z \oplus P, r), m_{\mathrm{b}}(Z, r)\}\frac{N}{2^c}\,.$$

Proof. The reasoning is the same as in Theorem 3, except that the queries are slightly different:

- In a forward query, the adversary checks for a guess a whether $\overline{f}((Z_i \oplus P_i)||a) = Z_{i+1}$.
- In a backward query, she checks for a guess a whether $\overline{f^{-1}}(Z_{i+1}||a) = Z_i \oplus P_i$

Clearly, the forward multiplicity of $Z \oplus P$ must be considered rather than that of Z as one forward query can be used to check inner values at up to $m_{\mathrm{f}}(Z \oplus P, r)$ indices at once. Note that the adversary can maximize the forward multiplicity to $|Z|_r - 1$ by choosing the blocks P_i such that $Z_i \oplus P_i$ always has the same value, resulting in a success probability after N queries of $N2^{-c}(|Z|_r - 1)$. □

5.7.3 With f a random transformation, revisited

In some attacks it may be sufficient for the adversary to recover the value of the state $s_{|Z|_r}$ of the sponge function after it has generated Z rather than before it. In that case, the adversary can, as in the case of f a random permutation, guess the value of any intermediate state s_i and compute $s_{|Z|_r}$ from that one by applying f. Clearly, then she can apply the same techniques as in the case of f a random permutation leading to similar success probabilities. The only difference is that only the forward multiplicity can be exploited.

5.8 Output binding

The goal is to find for a given string Z a state s that satisfies squeeze$(s, |Z|) = Z$. We only consider strings Z that consist of more than r bits. The success probability over the transformations (or permutations) f and over $\widehat{s} \in \mathbb{Z}_2^c$, that the following condition is verified:

$$\overline{f^i(Z_0||\widehat{s})} = Z_i, \forall i \in \{1 \ldots m\}, \tag{5.1}$$

depends not only on the length of Z, but also on its structure.

The adversary can make random guesses a until she finds one such that $\overline{f(Z_0||a)} = Z_1$. From there, she can evaluate $\overline{f^2(Z_0||a)}$ and check if it is equal to Z_2. If so, she continues, possibly until she reaches the last block of the sequence. If not, she starts again from a new guess a. At each step, in the absence of a cycle and neglecting biases, the adversary has a probability of 2^{r-1} to get the correct next block. Once an incorrect block is encountered, the

f	Inner collision	path finding	output cycle	state recovery	output binding
transformation	$\frac{N(N+1)}{2^{c+1}}$	$\frac{N}{2^c}$	$\frac{N(N+1)}{2^{r+c+1}}$	$\frac{N}{2^c}$	$\frac{1-2^{-r}}{2^{\lvert Z\rvert-r}}N$
permutation	$\frac{N(N+1)}{2^{c+1}} - \frac{N(N-1)}{2^{r+c+1}}$	$\frac{N(N+4)}{2^{c+2}} - \frac{N^2}{2^{r+c+2}}$	$\frac{N}{2^{r+c}}$	$\frac{\lvert Z\rvert_r-1}{2^c}N$	$\frac{1-2^{-r}}{2^{\lvert Z\rvert-r}}N$

Table 5.1: Cost functions for the primary attacks.

adversary starts with the next guess a. Clearly, the average number of calls to f to eliminate a guess is very close to $\frac{1}{1-2^{-r}}$.

If $|Z| < b$ the probability for a guess to be successful is:

$$\Pr(\text{success with guess}) = 2^{r-|Z|} .$$

Taking into account the number of calls to f for a guess, we obtain the following cost function, both for f a random transformation or permutation:

$$c_p(\text{output binding}) \approx \frac{2^r - 1}{2^{|Z|}} N .$$

When $|Z| > b$ the expected number for N is larger than the number of inner state values. It implies that the adversary has to try a large fraction of the values $\widehat{s} \in \mathbb{Z}_2^c$. By construction, she cannot look for more than 2^c values of $\widehat{a}$ and there is not necessarily a solution. An inner state value $\widehat{s}$ that leads to the given output sequence only exists for a fraction of the possible transformations (or permutations). The probability that such an inner state value exists for $|Z| > b$ is $\frac{2^{r+c}}{2^{|Z|}}$.

5.9 Summary of success probabilities

Table 5.1 lists the resulting cost functions for the primary attacks and for f a random transformation and a random permutation for large values of 2^c. This is justified as small values of 2^c lead to weak sponge functions. Note that for state recovery the probability of success is displayed rather than its cost function.

When we consider values of N that are much larger than 1, we can neglect the linear terms in those cost functions that are quadratic. This results in Table 5.2. Note that here also for state recovery the probability of success is displayed rather than its cost function.

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

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

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

$$W = \int_0^{\infty} N \frac{dP}{dN} dN = \int_0^{\infty} N \frac{dc_p(N)}{dN} e^{-c_p(N)} dN .$$

Filling in the simplified cost functions listed in Table 5.2 leads to integrals that can be readily solved. For the linear cost functions, i.e., $c_p(N) = 2^{-x}N$, we obtain $W = 2^x$. For the quadratic cost functions, i.e., $c_p(N) = 2^{-x}N^2$, we obtain $W = \sqrt{\pi}2^{x/2} \approx 2^{1+x/2}$.

f	rate rate	Inner collision	path finding	output cycle	state recovery	output binding
transformation	$r \ggg 1$	$2^{-(c+1)}N^2$	$2^{-c}N$	$2^{-(c+r+1)}N^2$	$2^{-c}N$	$2^{-(\|Z\|-r)}N$
transformation	$r = 1$	$2^{-(c+1)}N^2$	$2^{-c}N$	$2^{-(c+2)}N^2$	$2^{-c}N$	$2^{-\|Z\|}N$
permutation	$r \ggg 1$	$2^{-(c+1)}N^2$	$2^{-(c+2)}N^2$	$2^{-(c+r)}N$	$(\|Z\|_r - 1)2^{-c}N$	$2^{-(\|Z\|-r)}N$
permutation	$r = 1$	$2^{-(c+2)}N^2$	$2^{-(c+3)}N^2$	$2^{-(c+1)}N$	$(\|Z\| - b)2^{-c}N$	$2^{-\|Z\|}N$

Table 5.2: Simplified cost functions for the primary attacks.

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 differences in resistance between the case of a permutation f and a transformation f are mainly in path finding and in the length of output cycles. If f is a random transformation, finding a path has expected workload 2^c, while for f a random permutation this is only about $2^{2+c/2}$. On the other hand, a sponge function is expected to end up in a cycle after about $2^{(c+r+3)/2}$ blocks if f is a random transformation while this is 2^{c+r-1} if f is a random permutation.

5.10 Sponge functions used as a hash function

We will now consider a number of classical hash function attacks and show how the primary attacks limit the resistance of a sponge function against these attacks. For simplicity, we consider the case of a high rate r. 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.

5.10.1 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 $\overline{\text{absorb}}(P) \oplus A = \overline{\text{absorb}}(Q) \oplus B$. 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 [36], 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.

5.10.2 Second pre-image

Assume we are looking for a second pre-image for a message M and let P be this message after padding. In a sponge function, we have a second pre-image if we can find a second

path to the inner state $t = \widehat{\text{absorb}}(P')$ with P' the prefix of P where only the last block $P_{|P|_r-1}$ is removed. Given this path A, we have $\text{absorb}(Q||X) = \text{absorb}(P)$ with $X = \overline{\text{absorb}}(Q) \oplus \overline{\text{absorb}}(P') \oplus P_{|P|_r-1}$. We have computed the cost function for this problem in Section 5.5 for f a random transformation and we found an expected workload of the order $2^c/|P|_r$ if $|P|_r < 2^{c/2}$. Note that its expected workload must be at least that of generating an inner collision as a second pre-image implies an inner collision.

In a truncated random oracle the expected workload is of the order 2^n and is independent of $|P|_r$. Hence if we impose a limit to the number of blocks $\max |P|_r$, a sponge function with f a random transformation offers a similar level of resistance against 2nd pre-images as a truncated random oracle if $n < c - \log_2(\max |P|_r)$.

It is now interesting to take a look at the 2nd pre-image attack presented in [38] and the herding attack presented in [40] that both apply to iterated hash functions. If we apply these attacks to a sponge function with a random transformation f with $c = n$ we obtain expected attack complexities lower than those obtained in [38] and [40]. 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.

If f is a random permutation the expected workload is between $2^{(c+4)/2}$ if $|P|_r \ll 2^{c/2}$ 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 $|P|_r$, the workload is close to that of finding a path to an inner state. For values of $|P|_r$ near $2^{c/2}$, the workload comes close to that of generating an inner collision, but stays higher.

An interesting observation related to second pre-images was made by Gligoroski, Ødegård and Jensen in [34]. If an adversary can construct a non-empty path P to the inner state 0^c (the inner value of the root state), then she can construct an infinite number of second pre-images for any message M. We now explain how, for simplicity making abstraction of the padding.

Assume we have a path P to the inner state value of the root, i.e., $\widehat{\text{absorb}}(P) = 0^c$. We construct P' as follows:

$$P' = P \oplus (\overline{\text{absorb}}(P)||0^{|P|-r}) \,. \tag{5.2}$$

It is easy to verify that $\text{absorb}(P||P') = \text{absorb}(P)$ and so $\widehat{\text{absorb}}(P||P') = 0^c$. This can be generalized to $\text{absorb}(P||P'^*) = \text{absorb}(P)$ and so $\widehat{\text{absorb}}(P||P'^*) = 0^c$. So a single non-empty path P to 0^c allows constructing an infinite class of paths to 0^c. If we now take an arbitrary message M of at least one block, then any string $P||P'^*||M'$ with $M' = M \oplus (\overline{\text{absorb}}(P)||0^{|M|-r})$ is a path to $\text{absorb}(M)$.

This remarkable property is a consequence of the fact that the adversary can use an inner collision that it can chain with itself: both P and empty string are paths to the inner state 0^c. At first sight this may seem like a worrying observation. However, exploiting this requires finding first a path to a given inner state and this has expected workload $2^{(c+4)/2}$, higher than that of generating inner collisions.

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

5.10.3 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.

If f is a random permutation we bind the digest to a state s. Then we compute $t = f^{-1}(s)$ and subsequently we find a path P to $\widehat{t}$. This gives a path to s given by the found path to t, namely $P||(\overline{\text{absorb}}(P) \oplus \bar{t})$. The expected workload for finding a pre-image for a truncated sponge function with a random permutation f in this way is hence $2^{n-r} + 2^{c/2}$ if $n < b$. If $n > b$ 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 sponge function with a random permutation f offers a similar resistance against pre-images as a truncated random oracle if $n < c/2$.

If f is a random transformation, 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 block Z_0, we need to guess a state t such that $\overline{f(t)} = T_0$. This multiplies the number of trials by 2^r and the expected workload now becomes 2^n for $n < b$ and 2^b for $n > b$. The expected workload for finding a pre-image is hence $2^n + 2^{c-1}$ for $n < b$ and $2^b + 2^{c-1}$ for $n > b$. A truncated sponge function with f a random transformation offers a similar resistance against pre-images as a truncated random oracle if $n < c$.

5.10.4 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 function, it is possible to do this if one can recover the state absorb(P) with P a padded message from the output. One can then compute absorb$(P||P')$ and generate the output by squeezing this. The length extension only works if the state value bound to the output is equal to absorb(P) and not some other state value that gives rise to the same output. If the output is longer than b it is very likely that there is only a single corresponding state value. Otherwise the expected number of solutions is 2^{b-n} and length extension is only successful if the correct 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.

5.10.5 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 will show in Section 6.2 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 $N < 2^{c/2}$. A similar reasoning applies for large differential propagation probabilities between input and output.

5.11 Keyed modes

In describing attacks on a keyed sponge function, the adversary can make two types of queries. The first type are calls to f, and if f is a permutation also f^{-1}. We denote the total number of such calls by N, representing what is usually called the time (or computational) complexity of the attack. The second type are queries to the keyed sponge function. The sum of the total number of input blocks (and key offset blocks) and output blocks of queries to the keyed sponge function is denoted by M. M represents what is usually called the data

complexity of the attack: the amount of data computed with the key. There are several scenarios for keyed modes. Here we present two of them to illustrate how the primary attacks can be used.

If keys have a fixed length $|K|$, then exhaustive key search based on an output bits of at least $|K|$ bits requires guessing about $2^{|K|-1}$ key values. This is possible for any keyed function, even if it based on a random oracle. Attacks with an expected workload above $2^{|K|-1}$ are therefore not a threat.

5.11.1 Predicting the output of a stream cipher

The main security feature of a stream cipher is that an adversary who does not know the key K, but who may have observed part of an key stream, cannot predict key streams for that key it has not observed. Consider a stream cipher that takes as a key K and an initial value IV and where the key stream is obtained by applying a sponge function to their concatenation: $Z = F(K||IV)$.

A way to use primary attacks for predicting the output of a stream cipher based on a keyed sponge is state recovery. Once the adversary has recovered the state from a first part of Z for a given IV, she can squeeze that state for regenerating the remaining part of Z for that IV using that state. Note that for generating the trailing part of the output sequence Z, it is sufficient to recover the state at the end of the known part of Z (see Section 5.7.3). In the case of a large bitrate, the expected workload of this is 2^c calls to f, so as long as $|K| < c$ this poses no threat.

If f is a permutation, recovery of the state at some point in squeezing phase allows the adversary to compute the state of the keyed sponge that it has right after absorbing the key K and IV by applying f^{-1} repeatedly. If K and IV are in different r-bit blocks, she can even compute the state right after absorbing K, allowing her to reconstruct the output sequence Z for any IV. If the key K is in a single r-bit block, the value of this state allows the adversary to compute the key value. This is also the case if K and IV are together in a single r-bit block.

Due to their short input, the primary attacks generation of inner collisions and path finding are typically not useful when attacking stream ciphers. However, if a cycle is detected in part of an output sequence Z, the adversary can predict the full sequence Z.

5.11.2 MAC function

The main security feature of a MAC function is that an adversary who does not know the key K, but who may have observed tags for a number of messages, cannot predict tags for any other message with success probability above 2^{-n} if n is the tag length. Consider a MAC function that takes as a key K and an message M and where the tag is obtained by applying a sponge function to their concatenation and truncating its output to n bits: $t = \lfloor F(K||M) \rfloor_n$. We limit ourselves to the case that n is smaller than the capacity.

As in the case of stream ciphers, the adversary can attempt state recovery using tags. In total she needs at least b bits of output to fully determine the state. With the given construction, if the adversary can choose the messages, the description of the active adversary of Section 5.7.2.2 applies. If she can get the tags of $m + 1$ chosen messages, she can construct a string Z and a string P that consist of m n-bit blocks for which $Z_i \oplus P_i$ have the same chosen value. If n is larger than the bitrate r, using this for state recovery has expected workload about $2^c/(m-1)$ queries to f. If n is smaller than the bitrate, this becomes $2^{b-n}/(m-1)$.

If f is a random transformation, the state recovered is the state of the sponge function after absorbing some message M. The adversary can now reconstruct tags of all messages with this message as prefix.

If f is a random permutation and the key K and the message blocks M_i are absorbed in separate r-bit blocks, the adversary can recover the state of the sponge just after absorbing the key. She can use this state value to reconstruct the tag of any message of choice. If the key fits in a single r-bit block, the adversary can even recover the key. Note however that for forging tags, the knowledge of the state of the sponge function after absorbing the key is sufficient.

In any case, the success probability is $2^c/(m-1)$. For all key lengths such that $|K| < c - \log_2(m)$ with m the maximum number of messages that can be MACed with the same key, these attacks pose no threat.

The adversary can attempt to generate inner collisions in the keyed sponge function. The expected data complexity of this is $2^{c/2}$ blocks. Once an inner collision is observed, MAC forgery is easy. If two messages M and M' have the same tag value, any message $M'||A$ has the same tag value as $M||A$. This attack poses no threat as long as $m \lll 2^{c/2}$

Note that one can also define a MAC function by taking as input the message followed by the key: $t = \lfloor F(M||K) \rfloor_n$. In this case, an adversary has the advantage that she can try to generate inner collisions *offline*, i.e., without having to query the keyed sponge function. Additionally, she can try to construct a path to a state that occurs in the absorbing of a target message, leading to a second message with the same MAC.

Chapter 6

Security proofs

In this chapter we prove the security of the sponge and duplex constructions against generic attacks. First we prove that the only feature that sets a random sponge apart from a random oracle is the existence of inner collisions. Then we prove an upper bound for the success probability of distinguishing a random sponge from a random oracle. This bound covers an adversary that has access to f and f^{-1} and allows replacing a random oracle by a random sponge in any application. This inevitably comes with a loss of security, that can however be made negligible by taking a sufficiently large capacity. In the subsequent sections we prove that the security of the duplex construction is equivalent to that of the sponge construction. This is followed by a proof that the security of a set of sponge functions making use of the same transformation or permutation f and padding rule is equivalent to that of the sponge function in that set with the smallest capacity. Finally, we discuss the implications of the proven bounds.

When we speak about probabilities in this chapter, these are taken over the space of all b-bit transformations or b-bit permutations.

6.1 Inner collisions as only source of non-uniformity

In this section we prove a fundamental property of the sponge construction: the existence of inner collisions is the only property that sets a random sponge apart from a random oracle. More particularly, if f is a random transformation or permutation, then the bits of the responses of the sponge construction to a sequence of queries for which there are no inner collisions, are uniformly and independently distributed.

6.1.1 The need for sponge-compliant padding

We now show how the fundamental property above implies that the padding rule must be sponge-compliant, as defined in Definition 1 in Section 2.1.2.

First, assume a padding rule that maps the empty string to itself. When presented the empty string as output, the sponge returns as first block of its output zero as according to Equation (2.3) it is equal to $\overline{\text{absorb}}(\text{empty string}) = 0^r$, and this for any choice of f. This is clearly not uniformly and independently distributed.

Second, assume we can find a pair of message M and M' and integers l and l' such that

$$M||\text{pad}[r](|M|)||0^{lr} = M'||\text{pad}[r](|M'|)||0^{l'r}\,, \tag{6.1}$$

and let $Z = \text{sponge}(M)$ and $Z' = \text{sponge}(M')$. Then according to Equation (2.3), we have $Z_l = Z'_{l'}$, for any choice of f. This is clearly not uniformly and independently distributed.

By truncating trailing zeroes at both sides, Equation (6.1) can be simplified to

$$M||\text{pad}[r](|M|) = M'||\text{pad}[r](|M'|)||0^{nr} ,$$

with $n = l' - l$. This readily translates in the condition expressed in Equation (2.1). In fact, sponge-compliance imposes that the mapping from (M, i), with i the index of the output block to the path to Z_i with $Z = \text{sponge}(M)$ is injective.

6.1.2 The proof

We denote a sequence of queries to a system $\mathcal{X}$ by Q and denote the sequence of responses to Q by $\mathcal{X}(Q)$. In this case, Q is a sequence of couples $(M^{(i)}, \ell_i)$, with $M^{(i)} \in \mathbb{Z}_2^*$ and ℓ_i a positive integer, and $\mathcal{X}(Q)$ is a sequence of couples $(M^{(i)}, Z^{(i)})$ with $Z^{(i)}$ the ℓ_i-bit response of the random sponge to query i.

For a given sequence of queries Q, the 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 $P^{(j)}||0^{(\ell_i-1)r}$ have a common prefix. We denote the set of paths to states traversed during the distinguishing experiment by $\mathcal{P}$. We have:

$$\mathcal{P} = \left\{ X \text{ is a prefix of } P^{(i)}||0^{(\ell_i-1)r} \text{ for some } 1 \leq i \leq q \right\},$$

with q denoting the number of queries. In the context of a given sequence of queries, absence of inner collisions means that

$$\forall X \neq X' \in \mathcal{P} \ : \ \widehat{\text{absorb}}(X) \neq \widehat{\text{absorb}}(X').$$

We can now prove the following theorem.

Theorem 5. *Let f be a random transformation or random permutation and* pad *a sponge-compliant padding rule. The bits of the outputs returned by* SPONGE$[f, \text{pad}, r]$ *to a sequence of queries are uniformly and independently distributed if no inner collisions occur during the queries.*

Proof. Consider the j-th output block $Z_j^{(i)}$ of the i-th query: $Z_j^{(i)} = \overline{\text{absorb}}(X)$ with $X = P^{(i)}||0^{jr}$. 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 blocks. We denote the set of states and inner states corresponding to $\mathcal{P}^x$ by $\mathcal{S}^x$ and $\widehat{\mathcal{S}^x}$ respectively.

The requirement that no inner collision takes place during the generation of the output block $\overline{\text{absorb}}(X)$ restricts the value of the inner state $\widehat{\text{absorb}}(X)$ to be different from all values in $\widehat{\mathcal{S}^x}$.

If f is a random transformation, the value of absorb(X) must be in $\mathbb{Z}_2^r \times (\mathbb{Z}_2^c \setminus \widehat{\mathcal{S}^x})$ due to this requirement. By construction these values are equiprobable. If f is a random permutation, the invertibility of f imposes that absorb(X) must be different from all states traversed already (except $(0^r, 0^c)$), so here absorb(X) is chosen from is $(\mathbb{Z}_2^r \times (\mathbb{Z}_2^c \setminus \widehat{\mathcal{S}^x})) \setminus \mathcal{S}^x$. Using $\mathcal{S}^x \subset \mathbb{Z}_2^r \times \widehat{\mathcal{S}^x}$ this can be simplified to $\mathbb{Z}_2^r \times (\mathbb{Z}_2^c \setminus \widehat{\mathcal{S}^x})$. Hence in both cases all possible values in $\mathbb{Z}_2^r$ are equiprobable for $\overline{\text{absorb}}(X)$ and independent of the states previously traversed. As all possible values for the output blocks are equiprobable, so are the individual bits. □

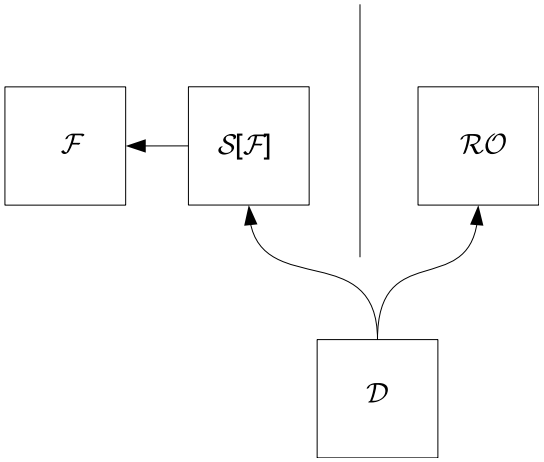

Figure 6.1: The distinguishing setting

6.2 Distinguishing a random sponge from a random oracle

In this section we prove the security of the sponge construction in a black-box setting. More particularly, we prove an upper bound on the success probability of distinguishing a random sponge from a random oracle for an adversary that does not have direct access to the random transformation or permutation f.

6.2.1 The adversary's setting

We consider an adversary that shall distinguish between two systems, as illustrated in Figure 6.1. The system at the left is the combination of the random transformation or permutation $\mathcal{F}$ and the sponge construction $\mathcal{S}$. The adversary may not make queries to $\mathcal{F}$ directly, but may send queries to $\mathcal{S}$, that in turn calls $\mathcal{F}$ to construct its responses. This is denoted by $\mathcal{S}[\mathcal{F}]$. We denote the interface to $\mathcal{S}[\mathcal{F}]$ by $\mathcal{H}$. The interface $\mathcal{H}$ takes as input a binary string $M \in \mathbb{Z}_2^*$ and an integer ℓ and returns a binary string $Z \in \mathbb{Z}_2^\ell$, the sponge output truncated to ℓ bits.

The system at the right consists of a random oracle $\mathcal{RO}$ providing the same interface as $\mathcal{S}[\mathcal{F}]$, the interface $\mathcal{H}$. When presented with an input (M, ℓ), this returns $\mathcal{RO}(M)$ truncated to ℓ bits.

We consider an adversary who is presented with a system $\mathcal{X}$ that is either $\mathcal{S}[\mathcal{F}]$ or $\mathcal{RO}$. The a priori probability of $\mathcal{X}$ being either $\mathcal{RO}$ or $\mathcal{S}[\mathcal{F}]$ is $\frac{1}{2}$. The adversary may send queries to the interface $\mathcal{H}$ of $\mathcal{X}$, even adaptively, by sequentially asking the first ℓ_i bits of output for a set of messages $M^{(1)} \dots M^{(q)}$. After sending all queries, she has to guess whether $\mathcal{X}$ is $\mathcal{RO}$ or $\mathcal{S}[\mathcal{F}]$ using the responses to the queries. We consider computationally unbounded adversaries that can optimally exploit the information present in the responses to queries and we try to upper bound the $\mathcal{RO}$ distinguishing advantage as a function of the total *cost* (or budget) of the queries.

6.2.2 The cost of queries

In our bounds we use a measure for the complexity of queries which is natural when applied to the sponge construction. We call this measure the *cost* and denote it by N. The cost N of a query to $\mathcal{X}$ is the total number of calls to $\mathcal{F}$ it would yield if $\mathcal{X} = \mathcal{S}[\mathcal{F}]$. The cost of a query is

fully determined the length of its input M and the requested output length ℓ. For example, if simple padding is used, a query contributes $\lfloor \frac{|M|+1}{r} \rfloor + \lceil \frac{\ell}{r} \rceil$ to the cost.

6.2.3 $\mathcal{RO}$ distinguishing advantage

The adversary is formalized as an algorithm $\mathcal{A}$ that returns 1 if she decides $\mathcal{X} = \mathcal{S}[\mathcal{F}]$ and 0 otherwise. The success probability of the adversary is given by

$$\frac{1}{2}\Pr(\mathcal{A}[\mathcal{S}[\mathcal{F}]] = 1) + \frac{1}{2}\Pr(\mathcal{A}[\mathcal{RO}] = 0) = \frac{1}{2} + \frac{1}{2}\left(\Pr(\mathcal{A}[\mathcal{S}[\mathcal{F}]] = 1) - \Pr(\mathcal{A}[\mathcal{RO}] = 1)\right) .$$

The succes probability is clearly determined by the rightmost expression. We denote this by the term *distinguishing advantage*:

$$\mathrm{Adv}(\mathcal{A}) = \left|\Pr(\mathcal{A}[\mathcal{S}[\mathcal{F}]] = 1) - \Pr(\mathcal{A}[\mathcal{RO}] = 1)\right| .$$

Without loss of generality, we take the absolute value to stick to the usual convention. The advantage of the adversary depends on the queries Q she sends and her guessing rule. For a given sequence of queries Q, let $\mathcal{R}(Q)_{\mathrm{RS}}$ be the set response sequences for which the adversary $\mathcal{A}$ guesses that $\mathcal{X}$ is $\mathcal{S}[\mathcal{F}]$. Then for Q, the probability that the adversary will return 1 if she addressing $\mathcal{S}[\mathcal{F}]$ is

$$\Pr(\mathcal{A}[\mathcal{S}[\mathcal{F}]] = 1) = \sum_{x \in \mathcal{R}(Q)_{\mathrm{RS}}} \Pr(\mathcal{S}[\mathcal{F}](Q) = x) .$$

And the probability that it will return 1 if it is addressing $\mathcal{RO}$ is

$$\Pr(\mathcal{A}[\mathcal{RO}] = 1) = \sum_{x \in \mathcal{R}(Q)_{\mathrm{RS}}} \Pr(\mathcal{RO}(Q) = x) .$$

It follows that the advantage as a function of Q is

$$\mathrm{Adv}(\mathcal{A}, Q) = \sum_{x \in \mathcal{R}(Q)_{\mathrm{RS}}} \left|\Pr(\mathcal{S}[\mathcal{F}](Q) = x) - \Pr(\mathcal{RO}(Q) = x)\right| .$$

This advantage is maximized by taking as guessing rule:

$$\mathcal{R}(Q)_{\mathrm{RS}} = \{x : \Pr(\mathcal{S}[\mathcal{F}](Q) = x) \geq \Pr(\mathcal{RO}(Q) = x)\} ,$$

yielding the following expression:

$$\mathrm{Adv}(\mathcal{A}, Q) = \frac{1}{2}\sum_{x} \left|\Pr(\mathcal{S}[\mathcal{F}](Q) = x) - \Pr(\mathcal{RO}(Q) = x)\right| . \tag{6.2}$$

We will now prove upper bounds for $\mathrm{Adv}(\mathcal{A})$ as a function of N, the cost of the queries. As we will show in Section 6.6.1, this upper bounds the success probabilities of generic attacks.

Theorem 6. *The $\mathcal{RO}$ distinguishing advantage of the sponge construction when calling a random transformation f is upper bounded by:*

$$1 - e^{-\frac{N(N+1)}{2^{c+1}}}$$

Theorem 7. *The $\mathcal{RO}$ distinguishing advantage of the sponge construction when calling a random permutation f is upper bounded by:*

$$1 - e^{-\frac{N(N+1)}{2^{c+1}} + \frac{N(N-1)}{2^{r+c+1}}}$$

Proof. Let $\Pr(\text{IC}|Q)$ denote the probability that a sequence of queries, when sent to $\mathcal{S}[\mathcal{F}]$ results in an inner collision.

As proven in Theorem 5, the bits of the responses of a random sponge to a sequence of queries are uniformly and independently distributed if no inner collision occurred during the queries. So for a sequence of responses x to Q that do not result in an inner collision, we have $\Pr(\mathcal{S}[\mathcal{F}](Q) = x|\text{no IC}) = \Pr(\mathcal{RO}(Q) = x)$. It follows that

$$\Pr(\mathcal{S}[\mathcal{F}](Q) = x) = \Pr(\mathcal{S}[\mathcal{F}](Q) = x|\text{IC}) \Pr(\text{IC}|Q) + \Pr(\mathcal{RO}(Q) = x)(1 - \Pr((\text{IC}|Q)).$$

Filling this in in Equation (6.2) yields:

$$\text{Adv}(\mathcal{A}, Q) = \frac{1}{2} \Pr(\text{IC}|Q) \sum_x |\Pr(\mathcal{S}[\mathcal{F}](Q) = x|\text{IC}) - \Pr(\mathcal{RO}(Q) = x)| \,.$$

As $\sum_x |\Pr(\mathcal{S}[\mathcal{F}](Q) = x|\text{IC}) - \Pr(\mathcal{RO}(Q) = x)| \leq 2$, we can upper bound the advantage by

$$\text{Adv}(\mathcal{A}, Q) \leq \Pr(\text{IC}|Q)) \,.$$

The right hand side of this equation is simply the success probability for generating an inner collisions in a sequence of queries Q. Filling in the success probabilities for generating inner collisions derived in Section 5.4 results in the two theorems. □

6.3 Differentiating a random sponge from a random oracle

Theorems 6 and 7 give a strong upper bound for the success probability of distinguishing the sponge construction calling a random transformation or permutation respectively. Unfortunately, this is with respect to an adversary that has no query access to f. This implies that the adversary does not have a specification of f and cannot access it directly. In any sponge function that is concrete and meant to be widely used, f must be publically specified. So f, and in case of a permutation also f^{-1}, can be queried by anyone having access to the specification or an implementation. So the distinguisher's setting of Section 6.2.1 and Figure 6.1 is of little use for the relevant use cases.

6.3.1 The indifferentiability framework

A solution to this problem is provided by the *indifferentiability framework* that was introduced by Maurer et al. in [45] as an extension of the classical notion of indistinguishability. It was applied to iterated hash functions by Coron et al. in [19]. We provide here an intuitive introduction and refer to the original papers for a more in-depth treatment and motivation.

What is actually required is a bound on the distinguishing advantage in a setting similar to that of Figure 6.1, where the adversary to the system at the left has additional query access to $\mathcal{F}$ (and in the case of a permutation also to $\mathcal{F}^{-1}$). At first sight such a system cannot be hard to distinguish from the random oracle $\mathcal{RO}$ at the right, merely due to the presence of the additional interface. An obvious solution to this problem would be to extend the system at the right with another component that has the same interface as $\mathcal{F}$. For the systems to be hard to distinguish, this component should simulate the behaviour of a random transformation (or permutation) of the same width as $\mathcal{F}$. For this reason it is called a *simulator*. There is an additional constraint. When making queries to the system at the left, the adversary can verify whether the responses to the queries are *sponge-consistent*. For each query to $\mathcal{S}[\mathcal{F}]$, it can emulate the sponge construction $\mathcal{S}$ itself and make queries to $\mathcal{F}$ directly. This should give the same results. For the right system to be hard to distinguish from the left system, it

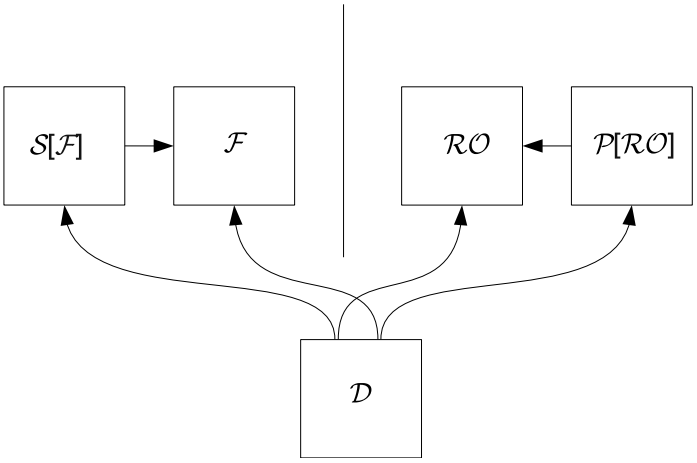

Figure 6.2: The differentiability setting

shall also behave sponge-consistent. For that reason, the simulator may have query access to the random oracle $\mathcal{RO}$ for satisfying sponge-consistency. So the simulator shall be an efficient algorithm with query access to $\mathcal{RO}$ and with the ability to generate random bits and store past queries it received.

The idea is now to construct a simulator $\mathcal{P}$ for which one can prove an upper bound on the advantage of distinguishing the left system from the right. Different simulators may result in different advantages and the goal of the designer is to bound this advantage as tightly as possible. We denote the advantage by the term $\mathcal{RO}$ *differentiating advantage* of the sponge construction when calling a random transformation (or permutation). As we will explain in Section 6.6, the upper bound on the $\mathcal{RO}$ differentiating advantage implies an upper bound for the success probability of any generic attack on the sponge construction equal to the succes probability for a random oracle plus the bound on the $\mathcal{RO}$ differentiating advantage. This is in fact the central idea of the indifferentiability framework.

In this remainder of this section we will prove upper bounds on the $\mathcal{RO}$-differentiating advantage of the sponge construction with f is a random tranformation and with f a random permutation.

6.3.2 The adversary's setting

The adversary shall distinguish between two systems that each have two components, as illustrated in Figure 6.2. The system at the left is the combination of the random transformation (or permutation) $\mathcal{F}$ and the sponge construction $\mathcal{S}[F]$. The adversary can make queries to both components separately, where the latter in turn calls the former to construct its responses. This is denoted by $\mathcal{S}[\mathcal{F}]$. The sponge construction $\mathcal{S}[\mathcal{F}]$ provides the interface $\mathcal{H}$ as specified in Section 6.2.1. If $\mathcal{F}$ is a random transformation it has a single interface $\mathcal{I}^1$ which takes as input an element s of $\mathbb{Z}_2^{r+c}$ and returns $t = \mathcal{F}(s)$, an element of the same set. If $\mathcal{F}$ is a random permutation, it has an additional interface $\mathcal{I}^{-1}$ that given input s returns $t = \mathcal{F}^{-1}(s)$. Note that the sponge construction only uses the interface $\mathcal{I}^1$.

The system at the right consists of a random oracle $\mathcal{RO}$ providing the interface $\mathcal{H}$ and a simulator $\mathcal{P}$. To construct its responses, the simulator can query $\mathcal{RO}$, denoted by $\mathcal{P}[\mathcal{RO}]$. Note that the simulator does not see the adversary's queries to the random oracle. We define two simulators, one for the case of a random transformation and another one for the case of a random permutation. The transformation simulator provides a single interface $\mathcal{I}^1$. The

permutation simulator provides both interfaces $\mathcal{I}^1$ and $\mathcal{I}^{-1}$.

Let $\mathcal{X}$ be either $(\mathcal{S}[\mathcal{F}], \mathcal{F})$ or $(\mathcal{RO}, \mathcal{P}[\mathcal{RO}])$. The sequence of queries Q to $\mathcal{X}$ consist of a sequence of queries to the interface $\mathcal{H}$, denoted Q^0 and a sequence of queries to the interface $\mathcal{I}^1$ (and $\mathcal{I}^{-1}$), denoted Q^1. Q^0 is a sequence of couples (M, ℓ), with $M \in \mathbb{Z}_2^*$ and ℓ a positive integer. Q^1 is a sequence of couples (s, b) with $s \in \mathbb{Z}_2^{r+c}$ and b either 1 or -1, indicating whether the interface $\mathcal{I}^1$ or $\mathcal{I}^{-1}$ is addressed. In the case that $\mathcal{F}$ is a transformation, b is restricted to 1.

The cost of queries to $\mathcal{H}$ is as defined in Section 6.2.2. The cost of a query to $\mathcal{I}^1$ or $\mathcal{I}^1$ is 1.

6.3.3 The simulators we use in our proofs

We define simulators for the case that $\mathcal{F}$ is a random transformation and for the case of a random permutation. In both cases, the simulator should behave as a deterministic function and give responses to queries Q^1 that in combination with the responses to queries Q^0 to the random oracle shall minimize the probability that the system $(\mathcal{RO}, \mathcal{P}[\mathcal{RO}])$ can be distinguished from a system $(\mathcal{S}[\mathcal{F}], \mathcal{F})$. In this section we informally explain how our simulators work.

A simulator keeps track of the queries it received and the responses it returned in a simulator graph, similar to the adversary graphs discussed in Section 5.3. Initially the simulator graph has no edges and for each new query to $\mathcal{I}^1(s)$ (or $\mathcal{I}^{-1}(s)$) it generates a response t and adds the edge (s, t) (or (t, s)). Note that using the responses of the simulator to its queries, the adversary can fully reconstruct the simulator graph.

In order to motivate the design of the simulators, we now discuss properties of this graph that it has at any moment during or after the queries, using an example depicted in Figure 6.3.

For a subset of the nodes in the simulator graph, the adversary knows a path. From Definition 8, it is clear that these are the nodes that have an incoming edge and are in a supernode that can be reached from supernode 0^c by following the directed edges from supernode to supernode. For this purpose, we define the set of *rooted* supernodes R as the subset of $\mathbb{Z}_2^c$ containing 0^c and all the supernodes accessible from it through the supernode graph. By extension, we say that a node $s = (\bar{s}, \widehat{s})$ is rooted if $\widehat{s} \in R$. So the adversary knows paths to all rooted nodes that have an incoming edge from another rooted node, plus the empty path of the $(0^r, 0^c)$ node. For each of these rooted nodes she can query the interface $\mathcal{H}$ of the system hoping to reveal an inconsistency, which is evidence that it is not $(\mathcal{S}'[\mathcal{F}], \mathcal{F})$. We call *sponge-consistent* the responses to a sequence of queries Q that do not result in such inconsistency.

Our simulators are built to guarantee sponge-consistent responses up to 2^c queries Q^1. We will now explain how they realize this. Whenever a simulator receives a query to $\mathcal{I}^1(s)$ with s rooted, it will result in an image t with known path. Therefore, the simulator constructs the outer part of t to be sponge-consistent by querying $\mathcal{RO}$ using the path to t (except for the all-zero path). When the simulator receives a query to $\mathcal{I}^1(s)$ with s not rooted, no path to the image t is known and it chooses t randomly from all the nodes (with no incoming edge, if $\mathcal{F}$ is a random permutation).

Moreover, the simulators are designed so that a call to $\mathcal{I}^1(s)$ results only in the path of a single node becoming known, that of $t = \mathcal{I}(s)$ if s is rooted. To achieve that, when selecting $\widehat{t}$ for a rooted node s, they exclude the supernodes with outgoing edges (cases *a* and *c* in Figure 6.3). And finally, they avoid the occurrence of nodes with multiple paths. For that, when selecting $\widehat{t}$ for a rooted node s, they exclude the rooted supernodes (case *b* in Figure 6.3) and those with outgoing edges (case *c* in Figure 6.3). The permutation simulator avoids paths of nodes becoming known as a result of a call to $\mathcal{I}^{-1}(s)$ altogether by excluding rooted supernodes when selecting $\widehat{t}$.

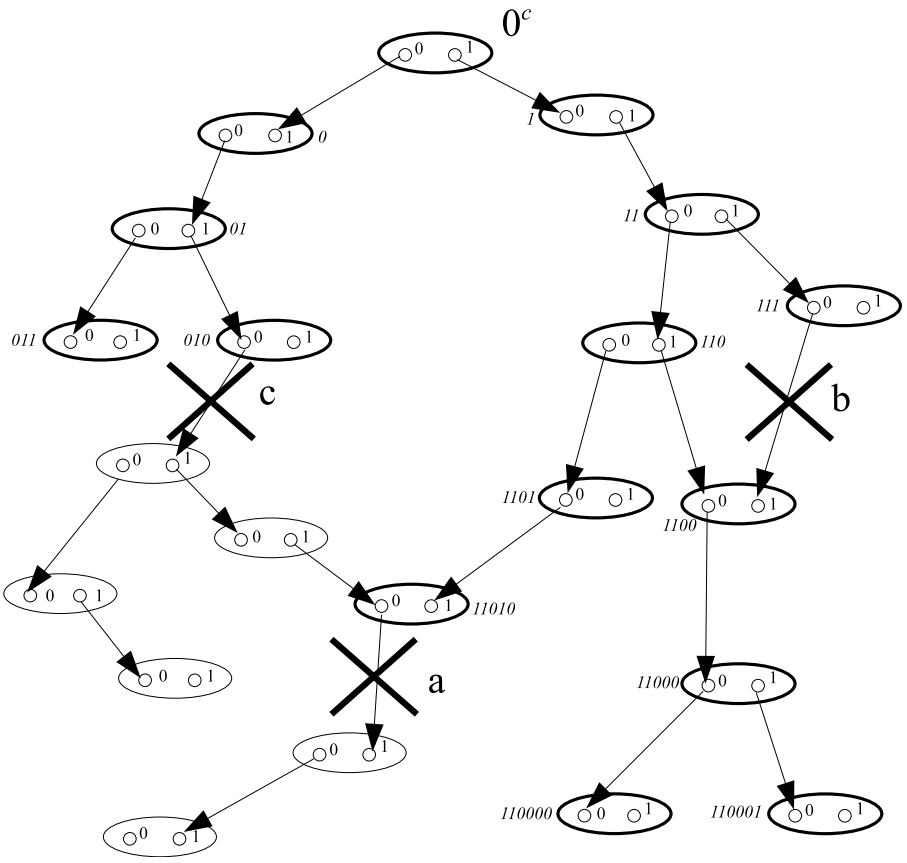

Figure 6.3: Example of simulator graph. The rooted supernodes are in bold. Paths are indicated in italic next to the nodes having a path.

Let O be the set of supernodes with an outgoing edge. When the simulator receives a query to $\mathcal{I}^1(s)$ with s a rooted node and all supernodes are rooted or have an outgoing edge, i.e., if $R \cup O = \mathbb{Z}_2^c$, it can no longer ensure sponge-consistency and we call the simulator *saturated*. As every query to the simulator adds at most one edge and that hence $R \cup O$ can be extended by at most 1 per query, this cannot happen before 2^c queries.

6.3.4 When being used with a random transformation

The simulator for the case that $\mathcal{F}$ is a random transformation is given in Algorithm 8. We prove upper bounds for the $\mathcal{RO}$ differentiating advantage by means of a series of lemmas and a final theorem.

Algorithm 8 The transformation simulator $\mathcal{P}[\mathcal{RO}]$

1: **Interface** $\mathcal{I}^1$, taking node s as input
2: **if** node s has no outgoing edge **then**
3: **if** node s is rooted AND $R \cup O \neq \mathbb{Z}_2^c$ (no saturation) **then**
4: Construct path to t: find path to s, append $\bar{s}$ and call the result P
5: Write P as $P = P'0^{rj}$ where P' does not end with 0^r
6: **if** P' can be unpadded into M **then**
7: Assign to $\bar{t}$ the value of block Z_j with $Z = \mathcal{RO}(M)$
8: **else**
9: Choose $\bar{t}$ randomly and uniformly
10: **end if**
11: Choose $\widehat{t}$ randomly and uniformly from $\mathbb{Z}_2^c \setminus (R \cup O)$
12: Let $t = \bar{t}||\widehat{t}$
13: **else**
14: Choose t randomly and uniformly from all nodes
15: **end if**
16: Add an edge from s to t
17: **end if**
18: **return** the node t at the end of the outgoing edge from s

Lemma 1. *To every node in the simulator graph there is at most one path, unless the simulator is saturated.*

Proof. First, we show that the rooted supernodes in the supernode graph form a tree. When no edges exist, this is indeed the case. The only way to create a new rooted node is by calling $\mathcal{I}^1(s)$ with s rooted. Assuming the simulator is not saturated, this happens only in first part of Algorithm 8 (lines 4–12), if s is rooted and has no outgoing edge. The new edge only adds a single supernode to R as the simulator selects it from the supernodes with no outgoing edges. Moreover, the new edge cannot arrive in a rooted supernode (because the simulator selects $\widehat{t}$ from $\mathbb{Z}_2^c \setminus R$) or in a supernode from which a rooted supernode can be reached (because the simulator select $\widehat{t}$ from the supernodes with no outgoing edges).

Then, for two connected supernodes $(\widehat{s}, \widehat{t})$, there exists only one edge in the simulator graph of the form $(\bar{s}||\widehat{s}, \bar{t}||\widehat{t})$. This is because the simulator chooses a distinct inner part for each new rooted node (unless it is saturated).

Finally, each r-bit block of the path is uniquely determined by the transitions on the outer part of the nodes. □

For a given sequence of queries Q and their responses $\mathcal{X}(Q)$, we define the *sponge consistency* as the property that the responses to Q^0 are equal to those that one would obtain by applying the sponge construction from the responses to Q^1 (when the queries Q^1 suffice to perform this calculation), i.e., that $\mathcal{X}(Q^0) = \mathcal{S}'[\mathcal{X}(Q^1)](Q^0)$. By construction, the queries, and their responses, made to the system $(\mathcal{S}'[\mathcal{F}], \mathcal{F})$ are sponge-consistent. For the sponge-consistency of the queries, and their responses, made to $(\mathcal{RO}, \mathcal{P}[\mathcal{RO}])$, we refer to the following lemma.

Lemma 2. *Given queries to the simulator $\mathcal{P}[\mathcal{RO}]$ described in Algorithm 8 and to $\mathcal{RO}$, it returns sponge-consistent responses, unless the simulator is saturated.*

Proof. The adversary can check for sponge-consistency by querying $\mathcal{H}$ for every node s in the simulator graph to which it knows the path P. The all-zero path does not correspond to a block that can be output by the sponge construction, so without loss of generality we assume that $P \neq 0^{rj}$.

Given the path P to the node s, its outer part must be equal to Z_j with $Z = \mathcal{RO}(M)$, where $M||\text{pad}(|M|) = P'$ and P' is a valid sponge input given by $P = P'0^{rj}$. As Lemma 1 says, there is only a single path to any rooted node in the simulator graph, and thus the simulator guarantees this equality for the response t to every query to $\mathcal{I}^1(s)$ with s a rooted node, as long as it is not saturated.

We also need to show that no path is assigned to a node unless its outer part is chosen by the lines 6–9 of Algorithm 8. Indeed, the supernode $\widehat{t}$ (at line 11) is the only supernode that becomes rooted due to the query. This is because the simulator excludes supernodes with outgoing edges in the selection of $\widehat{t}$ (as long as the simulator is not saturated).

It follows that the simulator guarantees sponge-consistency for all queries Q up to saturation. □

Lemma 3. *Any sequence of queries Q^0 up to cost 2^c can be converted to a sequence of queries Q^1 where Q^1 gives at least the same amount of information to the adversary and has no higher cost than Q^0.*

Proof. A query in Q^0 consists of an input M and a length ℓ. Let $P = M||\text{pad}(|M|)0^{r\lceil \frac{\ell}{r} \rceil}$. We can now convert this query into $|P|_r$ queries to $\mathcal{I}^1$. Let $s_0 = 0^r||0^c$ and $s_{i+1} = \mathcal{I}^1((\overline{s_i} \oplus P_i)||\widehat{s_i})$ for $0 \leq i < |P|_r$ be the responses to the new queries. As Lemma 2 says that all queries up to cost 2^c are sponge-consistent, the output to the original (M, ℓ) query consists of the concatenation of the outer parts of $s_{|P|_r}$ to $s_{|P|_r + \lceil \frac{\ell}{r} \rceil - 1}$ truncated to ℓ bits. By the definition of the cost of queries, the original query in Q^0 has cost $|P|_r + \lceil \frac{\ell}{r} \rceil - 1$ and it results in $|P|_r + \lceil \frac{\ell}{r} \rceil - 1$ queries in Q^1, each one with cost 1.

This process can be repeated for all queries in Q^0 resulting in a sequence of queries Q^1 with the same cost. If there are queries in Q^0 with inputs having common prefixes, these can give rise to the same queries in Q^1 resulting in a reduction in cost. □

Lemma 4. *The advantage of an adversary in distinguishing between $\mathcal{F}$ and $\mathcal{P}[\mathcal{RO}]$ with the responses to a sequence of $N < 2^c$ queries Q^1 is upper bounded by:*

$$f_{\text{T}}(N) = 1 - \prod_{i=1}^{N} \left(1 - \frac{i}{2^c}\right) .$$

Proof. The response sequence x to a sequence of N different queries is a sequence of N values in $\mathbb{Z}_2^{r+c}$. We can provide an upper bound of the advantage by computing the probability

distributions of the outcomes of the queries to $\mathcal{F}$ on the one hand and to $\mathcal{P}[\mathcal{RO}]$ on the other. We have

$$\mathrm{Adv}(\mathcal{A}) \leq \frac{1}{2}\sum_{x} |\Pr(x|\mathcal{F}) - \Pr(x|\mathcal{P}[\mathcal{RO}])|, \tag{6.3}$$

where the righthand side of this equation is known as the variational distance.

Since $\mathcal{F}$ is a transformation over $\mathbb{Z}_2^{r+c}$ chosen randomly and uniformly, the responses to the different queries are independent and uniformally distributed over $\mathbb{Z}_2^{r+c}$. It follows that all $(2^{r+c})^N$ possible outcomes are all equiprobable.

By inspecting Algorithm 8, the simulator always returns uniform values for the outer part of the image. For the inner part, the simulator chooses it non-uniformly only if the preimage s is rooted. To obtain the greatest possible variational distance, the optimum strategy consists in creating N rooted nodes. As a response to the first query, it may return all values but 0^r. At each subsequent query, one value of $\mathbb{Z}_2^c$ is added to R, and thus for each query, the simulator returns a inner part value different from 0^r and all previous ones. Note that by restricting $N < 2^c$ the simulator will not be saturated. Using this strategy gives us an upper bound on the variational distance. So for the simulator, there are $(2^r)^N(2^c-1)_{(N)}$ (where $a_{(n)}$ denotes $a!/(a-n)!$) possible responses with different inner parts, each with equal probability $((2^r)^N(2^c-1)_{(N)})^{-1}$, and the $(2^r)^N((2^c)^N - (2^c-1)_{(N)})$ others have probability 0. This gives:

$$\mathrm{Adv}(\mathcal{A}) \leq 1 - \frac{(2^c-1)_{(N)}}{(2^c)^N} = 1 - \prod_{i=1}^{N}\left(1 - \frac{i}{2^c}\right) . \tag{6.4}$$

□

We have now all ingredients to prove the following theorem.

Theorem 8. *The $\mathcal{RO}$ differentiating advantage of the sponge construction calling a random transformation is upper bound by:*

$$1 - \prod_{i=1}^{N}\left(1 - \frac{i}{2^c}\right) ,$$

with N the cost of the queries.

Proof. As discussed in Lemma 3 we can construct from a set of query sequences Q^0, Q^1 an equivalent sequence of queries $Q^{1\prime} \circ Q^1$ with no higher cost and giving at least the same information. So, without loss of generality, we only need to consider adversaries using queries $\overline{Q}^1 = Q^{1\prime} \circ Q^1$ and their response $\mathcal{X}(\overline{Q}^1)$ and no queries Q^0.

For any fixed query $\overline{Q}^1$, we look at the problem of distinguishing the random variable $\mathcal{F}(\overline{Q}^1)$ from the random variable $\mathcal{P}[\mathcal{RO}](\overline{Q}^1)$. For a sequence of queries $\overline{Q}^1$ with cost N, Lemma 4 upper bounds the advantage of such an adversary to the expression in the theorem.

The simulator is efficient and has running time $t_S = O(N^2)$: for each query to the simulator with s rooted, it must find the path to s and send a query to the random oracle of cost equal to the length of the path to s. The length of the path to s is upper bounded by N, the total number of rooted supernodes in the simulator graph. □

If N is significantly smaller than 2^c, we can use the $\log(1+\epsilon)$ approximation to simplify the expression for the upper bound:

$$1 - e^{-\frac{N(N+1)}{2^{c+1}}} < \frac{N(N+1)}{2^{c+1}}. \tag{6.5}$$

Note that this is equal to the probability of success of generating an inner collision in a sequence of queries of total cost N, as derived in Section 5.4.1. It follows that this bound is as tight as possible.

6.3.5 When being used with a random permutation

The simulator for the case that $\mathcal{F}$ is a random permutation is given in Algorithm 9. We now can prove upper bounds for the $\mathcal{RO}$ differentiating advantage using a series of similar lemmas.

Algorithm 9 The permutation simulator $\mathcal{P}[\mathcal{RO}]$

Interface $\mathcal{I}^1$, taking node s as input
if node s has no outgoing edge **then**
  if node s is rooted AND $R \cup O \neq \mathbb{Z}_2^c$ (no saturation) **then**
    Construct path to t: find path to s, append $\bar{s}$ and call the result P
    Write P as $P = P'0^{rj}$ where P' does not end with 0^r
    if P' can be unpadded into M **then**
      Assign to $\bar{t}$ the value Z_j with $Z = \mathcal{RO}(M)$
    else
      Choose $\bar{t}$ randomly and uniformly
    end if
    Choose $\widehat{t}$ randomly and uniformly from $\mathbb{Z}_2^c \setminus (R \cup O)$ and such that $\bar{t}||\widehat{t}$ has no incoming edge yet
    Let $t = \bar{t}||\widehat{t}$
  else
    Choose t randomly and uniformly from all nodes that have no incoming edge yet
  end if
  Add an edge from s to t
end if
return the node t at the end of the outgoing edge from s

Interface $\mathcal{I}^{-1}$, taking node s as input
if node s has no incoming edge **then**
  Choose $\bar{t}$ randomly and uniformly
  Choose $\widehat{t}$ randomly and uniformly from $\mathbb{Z}_2^c \setminus R$ and such that $(\bar{t}, \widehat{t})$ has no outgoing edge yet
  Let $t = \bar{t}||\widehat{t}$
  Add an edge from t to s
end if
return the node t at the beginning of the incoming edge into s

The proofs of Lemma 1 and Lemma 2 are valid for the permutation simulator with respect to all calls to $\mathcal{I}^1$ but do naturally not consider calls to $\mathcal{I}^{-1}$. The proofs can simply be extended to the permutation simulator case by noting that the $\mathcal{I}^{-1}$ interface of the simulator excludes rooted nodes in the selection of the response, implying that a call to $\mathcal{I}^{-1}$ cannot lead to new rooted nodes and hence also not to new paths. The proof of Lemma 3 is valid for the permutation simulator as it is. Finally, the output produced by the interfaces $\mathcal{I}^1$ and $\mathcal{I}^{-1}$ are consistent, i.e., if $\mathcal{I}^1(s) = t$ then $\mathcal{I}^{-1}(t) = s$ and vice-versa.

Instead of Lemma 4 we now have the following lemma.

Lemma 5. *The advantage of an adversary in distinguishing $\mathcal{F}$ and $\mathcal{P}[\mathcal{RO}]$ with the responses to a sequence of $N < 2^c$ queries Q^1 is upper bounded by:*

$$1 - \prod_{i=0}^{N-1} \left(\frac{1 - \frac{i+1}{2^c}}{1 - \frac{i}{2^{r+c}}} \right) .$$

Proof. The proof is similar to that of Lemma 4. Since $\mathcal{F}$ is a permutation over $\mathbb{Z}_2^{r+c}$ chosen randomly and uniformly, the only limitation is that for the i-th query, the image (or pre-image) shall not be equal to any of the found images (or pre-image), resulting in $(2^{r+c}) - i$ possibilities. This leads to $(2^{r+c})_{(N)}$ possible outcomes each with probability $((2^{r+c})_{(N)})^{-1}$ and $(2^{r+c})^N - (2^{r+c})_{(N)}$ outcomes with probability 0.

From inspecting Algorithm 9 if follows that the adversary obtains the greatest possible variational distance when he creates N rooted nodes. This leads to the same distribution as for the transformation simulator. The possible outcomes of the permutation simulator are a subset of the possible outcomes for $\mathcal{F}$. This gives:

$$\mathrm{Adv}(\mathcal{A}) \leq 1 - \frac{(2^r)^N (2^c - 1)_{(N)}}{(2^{r+c})_{(N)}} = 1 - \prod_{i=0}^{N-1} \left(\frac{1 - \frac{i+1}{2^c}}{1 - \frac{i}{2^{r+c}}} \right) . \tag{6.6}$$

□

These lemmas and proofs result in the following theorem, where the proof is similar to that of Theorem 8.

Theorem 9. *The $\mathcal{RO}$ differentiating advantage of the sponge construction calling a random permutation is upper bound by:*

$$1 - \prod_{i=0}^{N-1} \left(\frac{1 - \frac{i+1}{2^c}}{1 - \frac{i}{2^{r+c}}} \right) . \tag{6.7}$$

with N the cost of the queries.

If N is significantly smaller than 2^c, we can use the $\log(1+\epsilon)$ approximation to simplify the expression for the upper bound:

$$1 - e^{-\frac{N(N+1)}{2^{c+1}} - \frac{N(N-1)}{2^{r+c+1}}} < \frac{N(N+1)}{2^{c+1}} - \frac{N(N-1)}{2^{r+c+1}}. \tag{6.8}$$

Note that this is equal to the probability of success of generating an inner collision in a sequence of queries of total cost N, as derived in Section 5.4.2. It follows that this bound is as tight as possible.

Remarkably, using a random permutation results in a better bound than using a random transformation. By assigning distinct inner part values of rooted nodes, the simulators tend to generate an output distribution which is closer to that of a permutation than to that of a transformation.

6.4 Equivalence of the sponge and duplex constructions

In this section we prove a fundamental property of the duplex construction: the output of a call to a duplex object can be obtained by evaluating a sponge function with the same parameters to the input constructed from all previous inputs to the duplex object. The corollary of this is that the duplex construction inherits the security properties from the sponge construction.

The following lemma links the security of the duplex construction $\textsc{duplex}[f, \mathrm{pad}, r]$ to that of the sponge construction $\textsc{sponge}[f, \mathrm{pad}, r]$. Generating the output of a D.duplexing() call using a sponge function is illustrated in Figure 6.4.

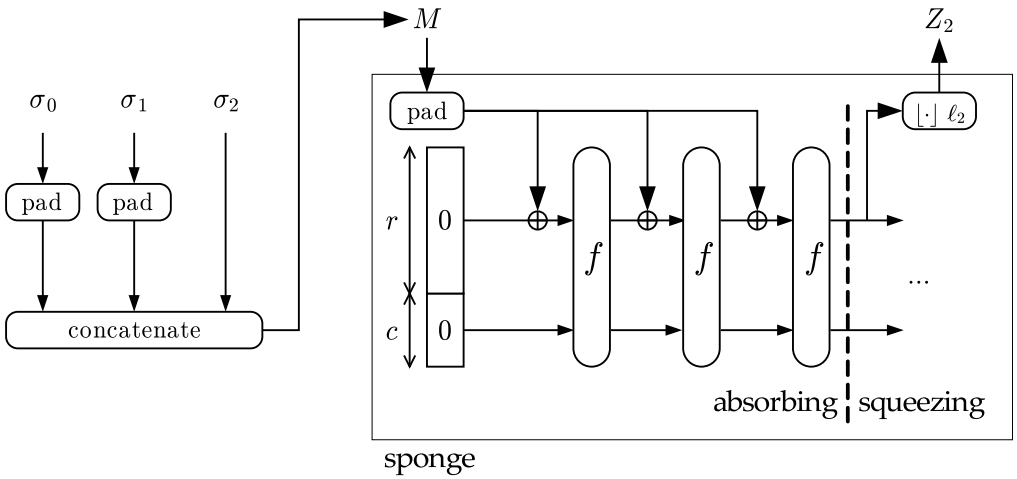

Figure 6.4: Generating the output of a duplexing call with a sponge

Lemma 6. [Duplexing-sponge lemma] *If we denote the input to the i-th call to a duplex object by* (σ_i, ℓ_i) *and the corresponding output by* Z_i *we have:*

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

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

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

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

Now assume that after the call $D.\text{duplexing}(\sigma_{i-1}, \ell_{i-1})$ the duplex object has the same state as the sponge function after absorbing $\sigma_0||\text{pad}_0||\sigma_1||\text{pad}_1||\ldots||\sigma_{i-1}||\text{pad}_{i-1}$. During the call $D.\text{duplexing}(\sigma_i, \ell_i)$, the block $\sigma_i||\text{pad}_i$ is XORed into the first r bits of the state and subsequently f is applied to the state. It follows that the state of the duplex object D after the call $D.\text{duplexing}(\sigma_i, \ell_i)$ is equal to the state of the sponge function after absorbing $\sigma_0||\text{pad}_0||\sigma_1||\text{pad}_1||\ldots\sigma_i||\text{pad}_i$. As the output just consists of the first ℓ_i bits of the state, this proves Lemma 6. □

The duplexing-sponge lemma states that the output of a duplexing call is the output of a sponge function with an input $\sigma_0||\text{pad}_0||\sigma_1||\text{pad}_1||\ldots||\sigma_i||\text{pad}_i$ and from this input the exact sequence $\sigma_0, \sigma_1, \ldots, \sigma_i$ can be recovered. As such, the duplex construction is as secure as the sponge construction with the same parameters. In particular, it inherits its upper bound on the $\mathcal{RO}$ differentiating advantage, where the input to the random oracle is the sequence of inputs to the duplexing calls since the initialization [12].

6.5 Optimum security of multi-rate sponge functions

The upper bound on the $\mathcal{RO}$ differentiating advantage of Section 6.3 covers the case of a single sponge function instance with a random transformation or permutation f with given width, padding rule and bitrate value. In this section we prove a bound on the $\mathcal{RO}$ differentiating advantage of any set of sponge functions sharing the same random f and padding rule, but with different bitrate (and so also capacity) values.

Clearly the achievable upper bound is at most that of the sponge function in the set with the smallest capacity, as an adversay can always just try to differentiate the weakest member of the set from a random oracle. In this section we will prove that this upper bound can be achieved, on the condition that the padding rule satisfies an additional requirement.

When considering the joint security of multiple sponge instances calling the same function f, simple padding is no sufficient. We will provide a proof for the simplest padding rule for which this is possible: the multi-rate padding (as defined in Section 2.1.2.

Theorem 10. *Given a random permutation (or transformation) f, differentiating the array of sponge functions $\text{SPONGE}[f, \text{pad10}^*1, r]$ with $0 < r \leq r_{\max}$ from an array of independent random oracles $(\mathcal{RO}_r)$ has the same advantage as differentiating $\text{SPONGE}[f, \text{pad10}^*, r_{\max}]$ from a random oracle.*

Proof. We can implement the array of sponge functions $\text{SPONGE}[f, \text{pad10}^*1, r]$ using a single sponge function $\text{sponge}_{\max} = \text{SPONGE}[f, \text{pad10}^*, r_{\max}]$, a bitrate-dependent input pre-processing function $I[r, r_{\max}]$ and a bitrate-dependent output post-processing function $O[r, r_{\max}]$. So we have:

$$\text{SPONGE}[f, \text{pad10}^*1, r] = O[r, r_{\max}] \circ \text{SPONGE}[f, \text{pad10}^*, r_{\max}] \circ I[r, r_{\max}],$$

The input pre-processing function $M' = I[r, r_{\max}](M)$ consists of the following steps:

1. Construct Q by padding M with multi-rate padding: $Q = M||\text{pad10}^*1[r](|M|)$
2. Construct Q' by splitting Q in r-bit blocks, extending each block with $0^{r_{\max}-r}$ and concatenating the blocks again.
3. Construct M' by unpadding Q' according to the padding rule pad10^*.

Note that the third step removes the trailing $r_{\max} - r$ bits with value 0 and the bit with value 1 just before that. It follows that the length of M' modulo $r_{\max}$ is $r - 1$, hence this pre-processing implements domain separation between the different r values for a given value of $r_{\max}$. Moreover, it is straightforward to extract M from $I[r, r_{\max}](M)$ and hence the pre-processing function is injective:

$$\forall (M_1, r_1) \neq (M_2, r_2) \Rightarrow I[r_1, r_{\max}](M_1) \neq I[r_2, r_{\max}](M_2). \tag{6.9}$$

The output post-processing function $Z = O[r, r_{\max}](Z')$ consists of splitting Z' in $r_{\max}$-bit blocks Z'_i, truncating each block to its first r bits $Z_i = \lfloor Z'_i \rfloor_r$ and concatenating the blocks again: $Z = Z_0||Z_1||\ldots$

We will now show that $\text{sponge}_{\max}$ loaded with $M' = I[r, r_{\max}](M)$ and $\text{SPONGE}[f, \text{pad10}^*1, r]$ loaded with M have the same state at the end of the absorbing phase. For this we will consider the sponge representation of Equation (2.4). Let $P = M||\text{pad10}^*1[r](|M|_r)$. The state of $\text{SPONGE}[f, \text{pad10}^*1, r]$ after absorbing P is given by $s = \text{ABSORB}[f, r](P)$. In this absorbing function, the r-bit blocks of P are XORed to the state, alternated with calls to f. Let $P' = M'||\text{pad10}^*[r_{\max}](|M'|_{r_{\max}})$. The state of $\text{sponge}_{\max}$ after absorbing P' is given by $s' = \text{ABSORB}[f, r_{\max}](P')$. In this absorbing function, the $r_{\max}$-bit blocks of P are XORed to the state, alternated with calls to f. It follows that $s = s'$ if the following three conditions are satisfied:

- P and P' have the same number of blocks,
- for each block the first r bits of P'_i are equal to those of P_i, and
- the last $r_{\max} - r$ bits of P'_i are zero.

Clearly, Q at the output of the first step of $I[r, r_{\max}]$ is equal to P. Moreover, Q' at the output of its second step is equal to P' as the unpadding in the third step of $I[r, r_{\max}]$ and the padding in $\text{sponge}_{\max}$ compensate each other. As each $r_{\max}$ blocks Q'_i consists of the Q_i followed by $0^{r_{\max}-r}$, the three conditions are satisfied.

If we now consider the output of $\text{sponge}_{\max}$ and $\text{SPONGE}[f, \text{pad10}^*1, r]$, for each iteration in the squeezing phase the former returns the first $r_{\max}$ bits of the state while the latter returns the first r bits of the state. Applying the output processing function $O[r, r_{\max}]$ to the output of $\text{sponge}_{\max}$ results in equality.

Assume now an attack that can differentiate the set of sponge functions $\text{SPONGE}[f, \text{pad10}^*1, r]$ from a set of random oracles with an advantage ϵ. Then this can be converted into an attack on $\text{sponge}_{\max}$ with the same advantage. Namely, the response $Z^{(i)}$ to a query $M^{(i)}$ to $\text{SPONGE}[f, \text{pad}_1, r]$ can be obtained from $\text{sponge}_{\max}$ by querying it with $I[r, r_{\max}](M^{(i)})$ and applying $O[r, r_{\max}]$ to its response $Z^{(i)}$. □

Note that for the proof to work it is crucial that the inner part (i.e., the c bits unaffected by the input or hidden from the output) of the sponge function instance with the smallest capacity is inside the inner parts of all other sponge function instances. This is realized in the sponge construction as the inner part of the state is systematically its last c bits.

So if several sponge construction instances are considered together, only the smallest capacity counts. When considering a sponge construction instance, one may wonder whether the mere existence of a sponge function instance with a smaller capacity has an impact on the security of that sponge instance. This is naturally not the case, as an adversary has access to f and can simulate any construction imaginable on top of f. What matters is that the value N used in the expression for the workload shall include all calls to f and f^{-1} of which results are used.

6.6 Implications of the bound on the $\mathcal{RO}$ differentiating advantage

It was suggested by Maurer et al. in [45], and later also formally proven by Andreeva et al. in [2], that the success probability of any attack on a construction (calling a random component) is upper bounded by the sum of the success probability of the same attack on a random oracle plus the $\mathcal{RO}$-differentiating advantage of the construction. Intuitively it is easy to see why. By contradiction, a generic attack on the construction with a larger success probability than that sum would constitute a method for differentiating that construction from a random oracle with an advantage above the upper bound.

Consider for example a random sponge with capacity c used for hashing by truncating its output to n bits. Consider the success probability of generating pre-images. The success probability of generating pre-images for a random oracle truncated to n bits is upper bounded by $q2^{-n}$ with q the number of messsages tried. The $\mathcal{RO}$ differentiating advantage of the sponge construction is upper bounded by $N^2 2^{-(c+1)}$ with N the number of calls to f (or f^{-1}). It follows that the success probability of a generic attack for generating pre-images in a sponge function is upper bounded by $q2^{-n} + N^2 2^{-(c+1)}$. If we assume that messages consist of a fixed number of blocks and trying a message has a fixed cost 2^a with a a small number, we have $q = N2^a$. The success probability now becomes $N2^{-n+a} + N^2 2^{-(c+1)}$. If $c > 2n$, the

second term never becomes larger than the first and the success probability is close to that for a random oracle. It follows that a random sponge based hash function offers a similar level to pre-image attacks as a random oracle if its capacity is at least twice its output length. Making the same exercise for the resistance against collisions results in the condition $c > n$.

These and other easy to characterize resistance levels make random sponges a good reference for expressing security claims. This is explained in more depth in Chapter 7.

6.6.1 Immunity to generic attacks

In the differentiating setting, f is assumed to be a random permutation or transformation. In any actual sponge functions, f will be a fixed and publically specified function. In practice, the queries to f and f^{-1} in the attack models correspond with computations of f and f^{-1} and N represents a computational cost. This is true if f does not have specific properties that may be exploited in attacks. Per definition, the bound on the $\mathcal{RO}$ differentiating advantage implies strict upper bounds for the success probability, and hence a provable lower bound for the expected workload of any *generic* attack, i.e., that does not exploit particular properties of f.

In the last few years a number of generic attacks against iterated hash functions have been published that demonstrated unexpected weaknesses:

- multicollisions [36],
- second pre-images on n-bit hash functions for much less than 2^n work [38],
- herding hash functions and the Nostradamus attack [40].

Clearly these attacks are covered by the bound on the $\mathcal{RO}$ differentiating advantage and for the sponge construction the workload of these attacks cannot be below $\sqrt{\pi}2^{c/2}$. As a matter of fact, all these attacks imply the generation of inner collisions and hence they pose no threat if generating inner collisions is difficult.

6.6.2 Randomized hashing

Interesting in this context is the application of randomized hashing [50]. Here a signing device randomizes the message prior to hashing with a random value that is unpredictable by the adversary. This increases the expected workload of generating a signature that is valid for two different messages from generating two colliding messages to that of generating a second pre-image for a message already signed. Now, if we keep in mind that for the sponge construction there are no generic attacks with expected workload of order below $2^{c/2}$, we can conclude the following. A lower bound for the expected complexity for generating a collision is $\min(2^{n/2}, 2^{c/2})$ and for generating a second pre-image $\min(2^n, 2^{c/2})$. Hence, if $c > 2n$, randomization increases the strength against signature forgery due to generic attacks against the hash function from $2^{n/2}$ to 2^n. If the capacity is between n and $2n$, the increase is from $2^{n/2}$ to $2^{c/2}$. If $c < n$, randomized hashing does not significantly increase the security level.

6.6.3 Security of keyed sponge functions

As discussed in Chapter 3 and Chapter 4 the sponge and duplex constructions can be used in keyed modes. In this section we explain the resistance of these modes against generic attacks.

With a random oracle, one can construct a pseudo-random function (PRF) $F_K(M)$ by prepending the message M with a key K, i.e., $F_K(M) = \mathcal{RO}(K||M)$. In such a case, the function behaves as a random function to anyone not knowing the key K but having access to the same random oracle. Note that the same reasoning is valid if K is appended to the message.

More specifically, let us consider the following distinguishing experiment where an adversary must distinguish between two systems. At the left we have a system consisting of $F_K(M) = \mathcal{RO}_1(K||M)$ and the random oracle instance $\mathcal{RO}_1$ used by the PRF. The adversary has query access to both of them. At the right we have a system consisting of a random oracle instance $\mathcal{RO}_2$ and also $\mathcal{RO}_1$ and the adversary has also query access to both of them. The adversary is presented with a system $\mathcal{X}$ that is one of these two systems with and must decide whether it is $(F_K, \mathcal{RO}_1)$ or $(\mathcal{RO}_2, \mathcal{RO}_1)$.

The only statistical difference between the two systems comes from the identity between $F_K(M)$ and $\mathcal{RO}_1(K||M)$, whereas $\mathcal{RO}_2(M)$ and $\mathcal{RO}_1(K||M)$ give independent results. Therefore, being able to detect such statistical difference means that the key K has been recovered. For a key K containing independent and uniform random bits, the distinguishing advantage expressed in terms of the number of queries q is upper bound by $q2^{-|K|}$.

As a consequence of the bound on the $\mathcal{RO}$ differentiating advantage, the same construction can be used with a sponge function. Now consider an adversary that must distinguish between the following two systems. A system at the left consisting of a keyed sponge construction calling a function f and that function f, either a random transformation or a random permutation. The system at the right consists of a random oracle and the function f. In both subsystems, the adversary can query both subsystems. Thanks to the bound on the $\mathcal{RO}$-differentiating advantage, the distinguishing advantage of the adversary, and hence the success probability of any attack on the keyed sponge construction, is upper bound by $q2^{-|K|} + N^2 2^{-(c+1)}$.

By assuming that queries are limited in length, we can bound q in terms of N by $q = N2^a$ with a a small integer, resulting a bound $N2^{-(|K|-a)} + N^2 2^{-(c+1)}$. As long as $N < 2^{c+1+a-|K|}$ the second term can be neglected. In the worst case, the key is found after $N = 2^{|K|}$ queries. Filling this in yields $2^{|K|} < 2^{c+1+a-|K|}$, resulting in the following upper bound for the key length, and so the attainable generic security level:

$$|K| < \frac{c+1+a}{2} .$$

Chapter 7

Random sponges as a security reference

When designing a cryptographic primitive, it is important to know which security criteria the result must satisfy, and when publishing it, its specifications should come with security criteria it claims to satisfy. Consider the case of cryptographic hash functions as an example. The traditional security criteria for a cryptographic hash function are collision resistance, pre-image resistance and 2nd pre-image resistance [46]. 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 [62]. More recently, a series of attacks [36, 38, 19, 40] 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 [6] 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. The output of a hash function has a fixed length, 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 compression function taking a message block 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. In other words, 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 [42]. 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 followed in all practical hash function proposals, including in our research.

Note that for stream ciphers and MAC functions a keyed random oracle would also be the ideal reference model. And for the same reason, the existence of state collisions, it would present an unattainable goal. In a MAC function the finite state also implies the existence of state collisions resulting in the same phenomenon as observed for hash functions. In a stream cipher the state collisions result in cycles in the key stream sequence, while the output of a random oracle is not cyclic.

7.1 A random sponge as a reference model

We have proven in Theorem 5 that a random sponge only differs from a random oracle by the mere existence of inner collisions. Moreover, we have proven a tight upper bound to the $\mathcal{RO}$ differentiating advantage of random sponges. This allows to provide tight upper bounds for the success probability for generic attacks. Therefore, we think random sponges are excellent candidates for serving as a security reference model for hash functions, stream ciphers, MAC functions and sponge functions.

7.1.1 Expressing a security claim

One can use random sponge can be used as a reference model for the security claim of a cryptographic primitive. To do so, the following parameters of the reference sponge should be chosen:

- the capacity c;
- the rate r;
- whether f is a random transformation or a random permutation;
- 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 cryptographic primitive 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. In this context, efficient primary attacks do not qualify as attacks by themselves but can inevitably be used to construct externally visible attacks.

7.1.2 Choosing the parameters

When a designer decides to express the security properties of his design with respect to a random sponge, he must decide whether he takes for f a random permutation or a random transformation and decide values for its capacity and rate. For a given capacity and rate, choosing a random transformation almost systematically offers a higher or equal security level than choosing a random permutation. The exception is the length of output cycles. One may conclude that for hash functions, taking for f a random transformation 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 with f a random permutation or a random transformation, the value of the capacity used in the claim shall be high enough to offer a sufficient resistance against collisions. In a random sponge this is limited by the resistance against inner collisions, for which the expected complexity is of the order $2^{c/2}$, both for f a random transformation or a random permutation. Both for f a random transformation or a random permutation, this imposes the same lower bound on c: 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 the weaker resistance against 2nd preimages due to f being a random permutation rather than a random transformation 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 random sponge offers a resistance level similar to 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 r equal to the length of the input blocks.

7.2 The flat sponge claim

If we consider our bound on the $\mathcal{RO}$ differentiating advantage for the sponge construction, we see that it is mainly determined by the capacity c and that r only has a small impact. To further simplify the choice of parameters for the reference model, we propose to formulate an even simpler claim making abstraction of whether f is a random transformation or permutation.

For this purpose we define the flat sponge claim.

Definition 14. *Given a capacity c_{claim}, the success probablity of any attack should be not higher than the sum of that for a random oracle and $1 - \exp\left(N^2 2^{-(c_{claim}+1)}\right)$, with the workload of the attack having the computational equivalent of N calls to f (or its inverse).*

Of course, one is free to amend this by imposing additional limitations, e.g. on the input and/or output lengths and the total cost.

Chapter 8

Sponge functions with an iterated permutation

In this chapter we present a practical strategy for the design of sponge functions that are efficient and secure. Instead of a collision-resistant compression function (Merkle-Damgård) or a random-looking compression function or ideal block cipher (as in [19]), our design strategy takes the design of a random-looking permutation. As a good block cipher should behave as a set of (independent and) random-looking permutations, hash function design can now benefit from insights gained in block cipher design. However, as opposed to a block cipher, a permutation has no key schedule and has not the concerns that come with it such as its computational overhead and possible related-key weaknesses. This makes in our opinion the sponge construction a very interesting alternative to the constructions based on a compression function. We build f as an iterated permutation.

In this chapter, we also discuss a number of properties of an iterated permutation that are particularly relevant when being used in a sponge construction.

8.1 The philosophy

8.1.1 The hermetic sponge strategy

In our design approach, we make a flat sponge claim with the same capacity as used in the sponge construction. This implies that for the claim to stand, the transformation or permutation f must be constructed such that it does not allow mounting attacks that have a higher success probability than generic attacks for the same workload. We call the design philosophy of adopting a sponge construction using a permutation that should not have exploitable properties the *hermetic sponge strategy*.

Thanks to the bound on the $\mathcal{RO}$ differentiating advantage an attack on $\text{SPONGE}[f, \text{pad}, r]$ with expected success probability higher than that of a generic attack implies a distinguisher for f. However, a distinguisher for f does not necessarily imply an exploitable weakness in $\text{SPONGE}[f, \text{pad}, r]$.

8.1.2 The impossibility of implementing a random oracle

Informally, a distinguisher for a particular f is the demonstration of any property that sets it significantly apart from a randomly chosen function (permutation or transformation). Remarkably, it is impossible to construct such a function that is efficient and has a reasonably

sized description or code. It is not hard to see why: any practical b-bit permutation (or transformation) has a compact description and implementation not shared by a randomly chosen permutation (or transformation) with its $\log_2 2^b! \approx (b-1)2^b$ (or $b2^b$) bits of entropy.

This is better known as the random oracle implementation impossibility. A formal proof for it was first given in [18] and later an alternative proof appeared in [45]. In their proofs, the authors construct a signature scheme that is secure when calling a random oracle but is insecure when calling a function F taking the place of the random oracle, where the function F has a limited (polynomial) running time and can be expressed as a Turing program of limited size. This argument is valid for any cryptographic function, and so includes any concrete sponge function. Now, looking more closely at the signature schemes used in [18] and [45], it turns out that they are especially designed to fail in the case of a concrete function. We find it hard to see how this property in a protocol designed to be robust may lead to its collapse of security. The proofs certainly have their importance in the more philosophical approach to cryptography, but we don't believe they prevent the design of cryptographic primitives that provide excellent security in well-engineered examples. Therefore, we propose addressing the random oracle implementation impossibility by just making an exception in the security claim.

8.1.3 The choice between a permutation and a transformation

As can be read in Chapter 5, the expected workload of the best generic attack for finding a second preimage of a message M when using a transformation is of the order $2^c/|M|_r$. When using a permutation this is only of order $2^{c/2}$. In that respect, a transformation has preference over a permutation. This argument makes sense when developing a hash function dedicated to offering resistance against second preimage attacks. Indeed, using a transformation allows going for a smaller value of c providing the same level of security against generic attacks.

When developing a general-purpose sponge function however, the choice of c is governed by the security level against the *most powerful* attack the function must resist. These are attacks that exploit inner collisions in the sponge function. The resistance against such attacks that a sponge function can offer is the same for a transformation or a permutation and of the order $2^{c/2}$.

8.1.4 The choice of an iterated permutation

Clearly, using a random transformation instead of a random permutation does not offer less resistance against the primary attacks, with the exception of detecting cycles (see Section 5.6) and the latter is only relevant if very long outputs are generated. Hence, why choose for a permutation rather than a transformation?

We believe a suitable permutation can be constructed as a fixed-key block cipher: as a sequence of simple and similar rounds.

The alternative would be to build a suitable transformation. In [25] an upper bound on the $\mathcal{RO}$-differentiating advantage was proven for a compression function consisting of a random permutation f with part of its input fixed and truncated output. However, this would result in an overall higher $\mathcal{RO}$-differentiating advantage for the same width of f. A variant of this method would be to employ a block cipher, fix its plaintext input and let the input of the transformation correspond with the *key* input of the block cipher. However, this would involve the definition of a key schedule and in our opinion results in less computational and memory usage efficiency and a more difficult analysis.

We propose to design iterated permutations for use in sponge functions in the same way as modern block ciphers: iterate a simple nonlinear round function enough times until the

resulting permutation has no properties that can be exploited in attacks. The remainder of this chapter deals with such properties and attacks. First, as an iterated permutation can be seen a block cipher with a fixed and known key, it should be impossible to construct for the full-round versions distinguishers like the known-key distinguishers for reduced-round versions of DES and AES given in [39]. This includes differentials with high differential probability (DP), high input-output correlations, distinguishers based on integral cryptanalysis or deviations in algebraic expressions of the output in terms of the input. We call this kind of distinguishers *structural*, to set them apart from trivial distinguishers that are of no use in attacks such as checking that $f(a) = b$ for some known input-output couple (a, b) or the observation that f has a compact description.

In the remainder of this chapter we will discuss some important structural distinguishers for iterated permutations, identify the properties that are relevant in the primary attacks and finally those for providing resistance to the classical hash function attacks.

8.2 Some structural distinguishers

In this section we discuss structural ways to distinguish an iterated permutation from a random permutation: differentials with high differential probability (DP), high input-output correlation, non-random properties in the algebraic expressions of the input in terms of the output (or vice versa) and the difficulty of solving a particular problem: the constrained-input constrained-output problem.

8.2.1 Differential cryptanalysis

A (XOR) *differential* over a function α consists of an input difference a' and an output difference b' and is denoted by a couple (a', b'). A pair *in* a differential is a pair $\{a, a \oplus a'\}$ such that $\alpha(a \oplus a') \oplus \alpha(a) = b'$. In general, one can define differentials and (ordered) pairs for any Abelian group operation of the domain and codomain of α. A pair in a differential is then defined as $\{a + a', a\}$ such that $\alpha(a + a') = \alpha(a) \odot b'$, where $+$ corresponds to the group operation of the domain of α and $\odot$ of its codomain. In the following we will however assume that both group operations are the bitwise XOR, or equivalently, addition in $\mathbb{Z}_2^b$.

The cardinality of a differential (a', b') is the number of pairs it contains and its differential probability (DP) is the cardinality divided by the total number of pairs with given input difference. We define the (restriction) weight of a differential $\mathrm{w_r}(a', b')$ as minus the binary logarithm of its DP, hence we have $\mathrm{DP}(a', b') = 2^{-\mathrm{w_r}(a', b')}$. The set of values a with a a member of a pair in a differential (a', b') can be expressed by a number of conditions on the bits of a. Hence a differential imposes a number of conditions on the absolute value at its input. Often these conditions can be expressed as $\mathrm{w_r}(a', b')$ independent binary equations.

It is well known (see, e.g., [22]) that the cardinality of non-trivial (i.e., with $a' \neq 0 \neq b'$) differentials in a random permutation operating on $\mathbb{Z}_2^n$ with n not very small has a Poisson distribution with $\lambda = 1/2$ [22]. Hence the cardinality of non-trivial differentials of an iterated permutation used in a sponge construction shall obey this distribution.

Let us now have a look at how differentials over iterated mappings are structured. A *differential trail* Q over an iterated mapping f of n_r rounds R_i consists of a sequence of $n_\mathrm{r} + 1$ differences $(q_0, q_1, \ldots, q_{n_\mathrm{r}})$. Now let $f_i = \mathrm{R}_{i-1} \circ \mathrm{R}_{i-2} \circ \ldots \mathrm{R}_0$, i.e., f_i consists of the first i rounds of α. A pair in a trail is a couple $\{a, a \oplus a'_0\}$ such that for all i with $0 < i \leq n_\mathrm{r}$:

$$f_i(a \oplus q_0) \oplus f_i(a) = q_i \,.$$

Note that a trail can be considered as a sequence of n_r round differentials (q_{i-1}, q_i) over each R_i. The cardinality of a trail is the number of pairs it contains and its DP is the cardinality divided by the total number of pairs with given input difference. We define the (restriction) weight of a differential trail $w_r(Q)$ as the sum of the weights of its round differentials.

The cardinality of a differential (a', b') over f is the sum of the cardinalities of all trails Q within that differential, i.e., with $q_0 = a'$ and $q_{n_r} = b'$. From this, the condition on the values of the cardinality of differentials of f implies that there shall be no trails with *high* cardinality and there shall not be differentials containing *many* trails with non-zero cardinality.

Let us take a look at the cardinality of trails. First of all, note that $DP(Q) = 2^{-w_r(Q)}$ is not necessarily true, although it usually is a good approximation when $w_r(Q) < b - 4$. The cardinality of the trail is then given by $2^{b-1} \times DP(Q)$. Now, when $w_r(Q) > b - 1$, we cannot have $DP(Q) = 2^{-w_r(Q)}$ as the number of pairs is an integer. A trail with $w_r(Q) > b - 1$ has typically no pairs, maybe one pair and very maybe a few pairs. If all trails over an iterated permutation have weight significantly above b, most trails with non-zero cardinality will only have a single pair. In other words, trails containing more than a single pair will be rare. In those circumstances, finding a trail with non-zero cardinality is practically equivalent to finding a pair in it. This makes such trails of very small value in cryptanalysis.

If there are no trails with low weight, it remains to be verified that there are no systematic clustering of non-zero cardinality trails in differentials. A similar phenomenon is that of *truncated differentials*. These are differentials where the input and output differences are not fully determined. A first type of truncated differentials are especially a concern in ciphers where the round function treats the state bits in sets, e.g., bytes. In that case, a typical truncated differential only specifies which bytes in the input and/or output differences are passive (equal to zero) and which ones are active (different from zero). The central point of these truncated differentials is that they also consist of truncated trails and that it may be possible to construct truncated trails with high cardinality. Similar to ordinary differential trails, truncated trails also impose conditions on the bits of the intermediate computation values of a, and the number of such conditions can again be quantified by defining a weight function.

A second type of truncated differentials are those where part of the output is truncated. Instead of considering the output difference over the complete output of f, one considers it over a subset of (say, n of) its output bits (e.g., the inner part $\widehat{f}$). For a random b-bit to n-bit function, the cardinality of non-trivial differentials has a normal distribution with mean 2^{b-n-1} and variance 2^{b-n-1} [22]. Again, this implies that there shall be no trails of the truncated function f with low weight and there shall be no clustering of trails.

Given a trail for f, one can construct a corresponding trail for the truncated version of f. This requires exploiting the properties of the round function of f. In general, the trail for the truncated version will have a weight that is equal to or lower than the original trail. How much lower depends on the round function of f. Typically, the trail in f determines the full differences up to the last few rounds. In the last few rounds the difference values in some bit positions may become unconstrained resulting in a decrease of the number of conditions.

8.2.2 Linear cryptanalysis

A (XOR) *correlation* over a function α, defined by a linear mask v at the input and a linear mask u at the output is denoted by a couple (v, u). It has a correlation value denoted by $C(v, u)$ equal to the correlation between the Boolean functions $v^T a = \sum v_i a_i$ and $u^T b = \sum u_i b_i$ with $b = \alpha(a)$ and the summations taken over GF(2). This correlation is a real number in the

interval $[-1, 1]$. We define the (correlation) weight of a correlation by:

$$\mathrm{w_c}(v, u) = -\log_2(C^2(v, u)) \ .$$

In general, one can define correlations for any Abelian group operation of the domain and codomain of α, where $C(v, u)$ is a complex number in the closed unit disk [4]. In the following we will however assume that both group operations are the bitwise XOR, or equivalently, addition in $\mathbb{Z}_2^b$. We only give an introduction here, for more background, we refer to [21].

Correlations in a permutation operating on $\mathbb{Z}_2^b$ are integer multiples of 2^{2-b}. The distribution of non-trivial correlations (i.e., with $u \neq 0 \neq v$) in a random permutation operating on $\mathbb{Z}_2^b$ with b not very small has as envelope a normal distribution with mean 0 and variance 2^{-b} [22]. Hence non-trivial correlations of an iterated permutation used in a sponge construction shall obey this distribution.

Let us now have a look at how correlations over iterated mappings can be decomposed into *linear trails*. A *linear trail* Q over an iterated mapping f of n_r rounds R_i consists of a sequence of $n_\mathrm{r} + 1$ masks $(q_0, q_1, \ldots, q_{n_\mathrm{r}})$. A linear trail can be considered as a sequence of n_r round correlations (q_i, q_{i+1}) over each R_i and its *correlation contribution* $C(Q)$ consists of the product of the correlations of its round correlations: $C(Q) = \prod_i C(q_i, q_{i+1})$. It follows that $C(Q)$ is a real number in the interval $[-1, 1]$. We define the correlation weight of a linear trail by

$$\mathrm{w_c}(Q) = -\log_2(C^2(Q)) = \sum_i \mathrm{w_c}(q_i, q_{i+1}) \ .$$

A correlation $C(v, u)$ over f is now given by the sum of the correlation contributions of all linear trails Q within that correlation, i.e., with $q_0 = v$ and $q_{n_\mathrm{r}} = u$. From this, the condition on the values of the correlations of f implies that there shall be no trails with *high* correlation contribution (so low weight) and there shall not be correlations containing *many* trails with high correlation contributions.

8.2.3 Algebraic expressions

In this section we discuss distinguishers exploiting particular properties of algebraic expressions of iterated mappings, more particularly those of the algebraic normal form (ANF) considered over $\mathrm{GF}(2)$. In a mapping operating on b bits, one may define a grouping of bits in d-bit blocks for any d dividing b and consider the ANF over $\mathrm{GF}(2^d)$. The derivations are very similar, the only difference is that the coefficients are in $\mathrm{GF}(2^d)$ rather than $\mathrm{GF}(2)$ and that the maximum degree of individual variables is $2^d - 1$ rather than 1.

Let $g : \mathrm{GF}(2)^b \to \mathrm{GF}(2)$ be a mapping from b input bits to one output bit. The ANF is the polynomial

$$g(x_0, \ldots, x_{b-1}) = \sum_{e \in \mathrm{GF}(2)^b} G(e) x^e, \text{ with } x^e = \prod_{i=0}^{b-1} x_i^{e_i} \text{ and } G(e) \in \mathrm{GF}(2).$$

Given the truth table of $g(x)$, one can compute the ANF of g with complexity of $O(b2^b)$ as in Algorithm 10.

When g is a (uniformly-chosen) random function, each monomial x^e is present with probability one half, or equivalently, $G(e)$ behaves as a uniform random variable over $\{0, 1\}$ [30]. A transformation $f : \mathrm{GF}(2)^b \to \mathrm{GF}(2)^b$ can be seen as a tuple of b binary functions $f = (f_i)$. For a (uniformly-chosen) random transformation, each $F_i(e)$ behaves as a uniform and independent random variable over $\{0, 1\}$.

Algorithm 10 Computation of the ANF of $g(x)$

Input $g(x)$ for all $x \in \mathrm{GF}(2)^b$
Output $G(e)$ for all $e \in \mathrm{GF}(2)^b$
Define $G[t] = G(e)$, for $t \in \mathbb{N}$, when $t = \sum_i e_i 2^i$
Start with $G(e) \leftarrow g(e)$ for all $e \in \mathrm{GF}(2)^b$
for $i = 0$ to $b - 1$ **do**
  for $j = 0$ to $2^{b-i-1} - 1$ **do**
    for $k = 0$ to $2^i - 1$ **do**
      $G[2^{i+1}j + 2^i + k] \leftarrow G[2^{i+1}j + 2^i + k] + G[2^{i+1}j + k]$
    end for
  end for
end for

If f is a random permutation over b bits, each $F_i(e)$ is not necessarily an independent uniform variable. For instance, the monomial of maximal degree $x_0 x_1 \dots x_{b-1}$ cannot appear since the bits of a permutation are balanced when x is varied over the whole range $\mathrm{GF}(2)^b$.

If b is small, the ANF of the permutation f can be computed explicitly by varying the b bits of input and applying Algorithm 10. A statistical test on the ANF of the output bit functions can be performed and if an abnormal deviation is found, the permutation f can be distinguished from a random permutation. Examples of statistical tests on the ANF can be found in [30].

If b is large, only a fraction of the input bits can be varied, the others being set to some fixed value. All the output bits can be statistically tested, though. This can be seen as a sampling from the actual, full b-bit, ANF. For instance, let $\tilde{f}$ be obtained by varying only the first $n < b$ inputs of f and fixing the others to zero:

$$\tilde{f}(x_0, \dots, x_{n-1}) = f(x_0, \dots, x_{n-1}, 0, \dots, 0).$$

Then, it is easy to see that any monomial x^e in the ANF of $\tilde{f}$ also appears in the ANF of f, and vice-versa, whenever $i \geq n \Rightarrow e_i = 0$.

A powerful type of attack that exploits algebraic expressions with a low degree are *cube attacks*, recently introduced in [24]. Cube attacks recover secret bits from polynomials that take as input both secret and tweakable public variables. Later *cube testers* were introduced in [3], that detect nonrandom behaviour rather than perform key extraction and can attack cryptographic schemes described by polynomials of relatively high degree. Cube testers are very well suited for building structural distinguishers.

8.2.4 The constrained-input constrained-output (CICO) problem

In this section we define and discuss a problem related to f whose difficulty is crucial if it is used in a sponge construction: the constrained-input constrained-output (CICO) problem. Let:

- $\mathcal{X} \subseteq \mathbb{Z}_2^b$: a set of possible inputs.
- $\mathcal{Y} \subseteq \mathbb{Z}_2^b$: a set of possible outputs.

Solving the CICO problem consists in finding a couple (x, y) with $y = f(x)$, $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.

The sets $\mathcal{X}$ and $\mathcal{Y}$ can be expressed by a number of equations in the bits of x and y respectively. In the simplest variant, the value of a subset of the bits of x (or y) are fixed. A

similarly simple case is when they are determined by a set of linear conditions on the bits of x (or y).

We define the weight of $\mathcal{X}$ as

$$\mathrm{w}(\mathcal{X}) = b - \log_2 |\mathcal{X}|,$$

and $\mathrm{w}(\mathcal{Y})$ likewise. When the conditions $y = f(x)$, $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are considered as independent, the expected number of solutions is $2^{b-(\mathrm{w}(\mathcal{X})+\mathrm{w}(\mathcal{Y}))}$. Note that there may be no solutions, and this is even likely if $\mathrm{w}(\mathcal{X}) + \mathrm{w}(\mathcal{Y}) > b$.

The expected workload of solving a CICO problem depends on b, $\mathrm{w}(\mathcal{X})$ and $\mathrm{w}(\mathcal{Y})$ but also on the nature of the constraints and the nature of f. If we make abstraction of the difficulty of finding members of $\mathcal{X}$ or $\mathcal{Y}$, generic attacks impose upper bounds to the expected complexity of solving the CICO problem:

- If finding x values in $\mathcal{X}$ is easy,
  - Trying values $x \in \mathcal{X}$ until one is found with $f(x) \in \mathcal{Y}$ is expected to take $2^{\mathrm{w}(\mathcal{Y})}$ calls to f.
  - Trying all values $x \in \mathcal{X}$ takes $2^{b-\mathrm{w}(\mathcal{X})}$ calls to f. If there is a solution, it will be found.
- If finding y values in $\mathcal{Y}$ is easy,
  - Trying values $y \in \mathcal{Y}$ until one is found with $f^{-1}(y) \in \mathcal{X}$ is expected to take $2^{\mathrm{w}(\mathcal{X})}$ calls to f^{-1}.
  - Trying all values $y \in \mathcal{Y}$ takes $2^{b-\mathrm{w}(\mathcal{Y})}$ calls to f^{-1}. If there is a solution, it will be found.

When $\mathrm{w}(\mathcal{X})$ or $\mathrm{w}(\mathcal{Y})$ is small or close to b, this problem may be generically easy, provided there is a solution.

Often a CICO problem can be easily expressed as a set of algebraic equations in a set of unknowns and one may apply algebraic techniques for solving these equations such as Gröbner bases [20].

8.2.5 Multi-block CICO problems

The CICO problem can be extended from a single iteration of f to multiple iterations in a natural way. We distinguish two cases: one for the absorbing phase and another one for the squeezing phase.

An e-block absorbing CICO problem for a function f is defined by two sets $\mathcal{X}$ and $\mathcal{Y}$ and consists of finding a solution $(x_0, x_1, x_2, \ldots x_e)$ such that

$$\begin{aligned} & x_0 \in \mathcal{X} \,, \\ & x_e \in \mathcal{Y} \,, \\ \text{for } 0 < i < e: \quad & \widehat{x_i} = 0^c \,, \\ & y_1 = f(x_0) \,, \\ \text{for } 1 < i < e: \quad & y_i = f(y_{i-1} \oplus x_{i-1}) \,, \\ & x_e = f(y_{e-1} \oplus x_{e-1}) \,. \end{aligned}$$

A priori, this problem is expected to have solutions if $\mathrm{w}(\mathcal{X}) + \mathrm{w}(\mathcal{Y}) \leq c + er$.

An e-block squeezing CICO problem for a function f is defined by $e+1$ sets $\mathcal{X}_0$ to $\mathcal{X}_e$ and consists of finding a solution x_0 such that:

$$\begin{aligned} &\text{for } 0 \leq i \leq e: \quad x_i \in \mathcal{X}_i \,, \\ &\text{for } 0 < i \leq e: \quad x_i = f(x_{i-1}) \,. \end{aligned}$$

A priori, this problem is expected to have solutions if $\sum_i \mathrm{w}(\mathcal{X}_i) < b$. If it is known that there is a solution, it is likely that this solution is unique if $\sum_i \mathrm{w}(\mathcal{X}_i) > b$.

Note that if $e = 1$ both problems reduce to the simple CICO problem.

8.2.6 Cycle structure

Consider the infinite sequence $a, f(a), f(f(a)), ...$ with f a permutation over a finite domain and a an element of that set. This sequence is periodic and the set of different elements in this sequence is called a *cycle* of f. In this way, a permutation partitions its domain into a number of cycles.

Statistics of random permutations have been well studied, see [63] for an introduction and references. The cycle partition of a permutation used in a sponge construction shall again respect the distributions. For example, in a random permutation over $\mathbb{Z}_2^b$:

- The expected number of cycles is $b \ln 2$.
- The expected number of fixed points (cycles of length 1) is 1.
- The number of cycles of length at most m is about $\ln m$.
- The expected length of the longest cycle is about $G \times 2^b$, where G is the Golomb-Dickman constant ($G \approx 0.624$).

8.3 The usability of structural distinguishers

A structural distinguisher or a non-generic attack for a sponge function implies a structural distinguisher for the underlying function f. However, a structural distinguisher for f does not necessarily imply a structural distinguisher for a sponge function calling f. There are two aspects.

First, there is the aspect of applicability. For example, a structural distinguisher that imposes values to bits in the inner part of the input to f are hard to exploit, as an adversary cannot directly access these bits. Applicability must be studied on a case-by-case basis for each structural distinguisher.

The second aspect is the distinguishing advantage, which can bring qualitative arguments to the (non-)usability of structural distinguishers. Informally speaking, this is the advantage of an adversary trying to distinguish f from a random permutation using the particular distinguisher. A structural distinguisher with a distinguishing advantage that is small compared to the $\mathcal{RO}$-differentiating advantage will not increase the success probability of any attack noticeably. As a matter of fact, there are structural distinguishers with a distinguishing advantage that is zero up to some number N of queries to f. If $N > 2^{b/2}$, such a structural distinguisher cannot possibly jeopardize the security of a sponge function making use of f, whatever its capacity. Actually, the maximum capacity value is $b-1$ and for this capacity value the security of the sponge function collapses anyway above $2^{b/2}$ queries due to the existence of inner collisions.

8.4 Conducting primary attacks using structural distinguishers

8.4.1 Inner collisions

Assume we want to generate an inner collision with two single-block inputs. This requires finding states a and a^* such that

$$\widehat{f(a)} \oplus \widehat{f(a^*)} = 0^c \text{ with } \widehat{a} = \widehat{a^*} = 0^c \,.$$

This can be rephrased as finding a pair $\{a, a^*\}$ with $\widehat{a} = \widehat{a^*} = 0^c$ in the differential $(a \oplus a^*, 0^c)$ of $\widehat{f}$. Requiring $\widehat{a} = \widehat{a^*} = 0^c$ is needed to obtain valid paths from the root state to iteration of f where the differential occurs. In general, it is required to know a path to the inner state $\widehat{a} = \widehat{a^*} = \widehat{\text{absorb}}(P)$; the case $\widehat{a} = \widehat{a^*} = 0^c$ is just a special case of that as $0^c = \widehat{\text{absorb}}(\text{empty string})$.

8.4.1.1 Exploiting a differential trail

Assume f is an iterated function and we have a trail Q in $\widehat{f}$ with initial difference a' and final difference b' such that $\widehat{a'} = \widehat{b'} = 0^c$. This implies that for a pair (a, a^*) in this trail, the intermediate values of a satisfy $\mathrm{w_r}(Q)$ conditions. If $\mathrm{w_r}(Q)$ is smaller than b, the expected number of pairs of such a trail is $2^{b-\mathrm{w_r}(Q)}$.

Let us now assume that given a trail and the value of $\widehat{a}$, it is easy to find pairs $\{a, a \oplus a'\}$ in it with given $\widehat{a}$. We consider two cases:

- $\mathrm{w_r}(Q) < r$: it is likely that the trail contains pairs with $\widehat{a} = 0^c$ and an inner collision can be found readily. The paths consist of the first r bits of the members of the found pair, $a \neq a^*$.
- $\mathrm{w_r}(Q) \geq r$: the probability that the trail contains a pair with $\widehat{a} = 0^c$ is $2^{r-\mathrm{w_r}(Q)}$.

If several trails are available, one can extend this attack by trying it for different trails until a pair in one of them is found with $\widehat{a} = 0^c$. If the weight of trails over f is lower bounded by $w_{\min}$, the expected workload of this method is higher than $2^{w_{\min}-r}$. With this method, differential trails do not lead to a attacks faster than generic attacks if $w_{\min} > c/2 + r = b - c/2$.

One can extend this attack by allowing more than a single block in the input. In a first variant, an initial block in the input is used to vary the inner part of the state and are equal for both members of the pair that will be found. Given a trail in the second block, the problem is now to find an initial block that, once absorbed, leads to an inner state at the input of the trail, for which the trail in the second block contains a pair. In other words, that leads to an inner state that satisfies a number of equations due to the trail in the second block. The equations in the second block define a set $\mathcal{Y}$ for the output of the first block with $\mathrm{w}(\mathcal{Y}) \approx \mathrm{w_r}(Q) - r$: the conditions imposed by the trail in the second block on the inner part of the state at its input. Moreover, the fact that the inner part of the input to f in the first iteration is fixed to zero defines a set $\mathcal{X}$ with $\mathrm{w}(\mathcal{X}) = c$. Hence, even if a pair can be found that is in the trail, a CICO problem must be solved with $\mathrm{w}(\mathcal{X}) = c$ and $\mathrm{w}(\mathcal{Y}) \approx \mathrm{w_r}(Q) - r$ for determining the first block of the inputs.

Note that if there are no trails with weight below b, the expected number of pairs per trail is smaller than 1 and trails containing more than a single pair will be rare. In this case, even if a trail with non-zero cardinality can be found, the generation of an inner collision implies solving a CICO problem for the first block with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$.

One can input pairs that consist of multiple input blocks where there is a difference in more than a single input block. Here, chained trails may be exploited in subsequent iterations of f. However, even assuming that the transfer of equations through f due to a trail and conditions at the output is easy, one ends up in the same situation with a number of conditions on the bits of the inner part of the state at the beginning of the first input differential. And again, if there are no trails with weight below b, the generation of an inner collision implies solving a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$.

If $c > b/2$, typically a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$ will have no solution. In that case one must consider multiple blocks and the problem to solve becomes a multi-block absorbing CICO problem. The required number of rounds e for there to be a solution is $\lceil c/r \rceil$.

8.4.1.2 Exploiting a differential

In the search for inner collisions, all pairs $(a, a \oplus a')$ with $\widehat{a} = 0^c$ in a differential $(a', 0^c)$ with $\widehat{a'} = 0^c$ over $\widehat{f}$ are useful, and not only the pairs of a single trail. So it seems like a good idea to consider differentials instead of trails. However, where for a given trail it may be easy to determine the pairs it contains, this is not true in general for a differential. Still, an $\widehat{f}$-differential may give an advantage with respect to a trail if it contains more than a single trail with low weight. On the other hand, the conditions to be pairs in a set of trails tend to become more complicated as the number of trails grows. This makes algebraic manipulation more and more difficult as the number of trails to consider grows.

If there are no trails over $\widehat{f}$ with weight below b, the set of pairs in a differential is expected to be a set that has no simple algebraic characterization and we expect the most efficient way to determine pairs in a differential is to try different outputs of f with the required difference and computing the corresponding inputs.

8.4.1.3 Truncated trails and differentials

As for ordinary differential trails, the conditions imposed by a truncated trail can be transferred to the input and for finding a collision a CICO problem needs to be solved. Here the factor $\mathrm{w}(\mathcal{Y})$ is determined by the weight of the truncated trail. Similarly, truncated trails can be combined to truncated differentials and here the same difficulties can be expected as when combining ordinary trails

8.4.2 Path to an inner state

If $c \geq b/2$, this is simply a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$ and solving it results in a single-block path to an inner state. If $c < b/2$, an e-block path to the inner state can be found by solving a multi-block absorbing CICO problem with $e = \lceil r/c \rceil$.

8.4.3 Detecting a cycle

This is strongly linked to the cycle structure of f. If f is assumed to behave as a random permutation, the overwhelming majority of states will generate very long cycles. Short cycles typically do exist, but due to the sheer number of states, the probability that these will be observed is extremely low.

8.4.4 State recovery

If the capacity is smaller than the bitrate, it is highly probable that a sequence of two output blocks fully determines the inner state. In that case, finding the inner state is a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = r$.

If the capacity is larger than the bitrate, one needs more than two output blocks to uniquely determine the inner state. Finding the state consists in solving a multi-block squeezing CICO problem with $\mathrm{w}(\mathcal{X}_i) = r$. The required number of rounds e to uniquely determine the state is $\lceil b/r \rceil$.

8.5 Classical hash function criteria

In this section we discuss the properties of an iterated permutation that are relevant in the classical hash function criteria.

8.5.1 Collision resistance

We assume that the sponge function output is truncated to its first n bits and we try to generate two outputs that are the same for two different inputs. We can distinguish two ways to achieve this: with or without an inner collision. While the effort for generating an inner collision is independent of the length of the output to consider, this is not the case in general for generating output collisions. If n is smaller than the capacity, the generic attack to generate an output collision directly has a smaller workload than generating an inner collision. Otherwise, generating an inner collision and using this to construct a state collision is expected to be more efficient.

We refer to Section 8.4.1 for a treatment on inner collisions. With some small adaptations, that explanation also applies to the case of directly generating output collisions. The only difference is that for the last iteration of the trail, instead of considering differentials $(a', 0^c)$ over $\widehat{f}$, one needs to consider differentials $(a', 0^n)$ over $\lfloor f \rfloor_n$. When exploiting a trail, and in the absence of high-probability trails, this reduces to solving a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$ to find a suitable first block.

8.5.2 Preimage resistance

We distinguish three cases:

- $n > b$: in this case the output fully determines the state just prior to squeezing. Generating a preimage implies the recovery of this state and subsequently finding a path to the recovered state. As explained in Sections 8.4.2 and 8.4.4, this comes down to solving two CICO problems.

- $r < n \leq b$: Here a sequence of input block can in theory be found by solving a problem that can be seen as a combination of a multi-round squeezing CICO problem and a multi-round absorbing CICO problem.

- $n \leq r$: A single-block preimage can be found by solving a single-block CICO problem with $\mathrm{w}(\mathcal{X}) = c$ and $\mathrm{w}(\mathcal{Y}) = n$.

8.5.3 Second preimage resistance

There are two possible strategies for producing a second preimage. In a first strategy, the adversary can try to find a second path to one of the inner states traversed when absorbing the first message. Finding a second preimage then reduces to finding a path to a given inner state [11], which is discussed in Section 8.4.2. As a by-product, this strategy exhibits an inner collision.

In a second strategy, the adversary can ignore the inner states traversed when absorbing the first message and instead take into account only the given output. In this case, the first preimage is of no use and the problem is equivalent to finding a (first) preimage as discussed in the two last bullets of Section 8.5.2.

8.5.4 Length extension

Length extension consists in, given $h(M)$ for an unknown input M, being able to predict the value of $h(M||X)$ for some string X. For a sponge function, length extension is successful if one can find the inner state at the end of the squeezing of M. This comes down to state recovery, discussed in Section 8.4.4. Note that the state is probably only uniquely determined if $n \geq b$. Otherwise, the expected number of solutions of output binding is 2^{b-n}. In that case, the probability of success of length extension is $\max(2^{n-b}, 2^{-n})$.

In principle, if the permutation f has high input-output correlations (v, u) with $\widehat{v} = \widehat{u} = 0^c$, this could be exploited to improve the probability of guessing right when doing length extension by a single block.

8.5.5 Output subset properties

One can define an m-bit hash function based on a sponge function by, instead of taking the m first bits of its output, just specify m bit positions in the output and consider the corresponding m bits as the output. Such a hash function shall not be weaker than a hash function where the m bits are just taken as the first m bits of the sponge output stream. If the m bits are from the same output block, there is little difference between the two functions. If the m bits are taken from different output blocks, the CICO problems implied by attacking the function tend to become more complicated and are expected to be harder to solve.

8.6 Keyed modes

Distinguishing the output of a keyed sponge function from a random oracle can be done by finding the key, or by detecting properties in the output that would not be present for a random oracle. Examples of such properties are the detection of large DP values or high correlations over f. If the key is shorter than the bitrate, finding it given the output corresponding to a single input is a CICO problem. If the key is longer, this becomes a multi-round absorbing CICO problem. If more than a single input-output pair is available, this is no longer the case. In general, an adversary can even request outputs corresponding with adaptively chosen inputs.

When we use a keyed mode for MAC computation, the length of the key is typically smaller than the bitrate and the output is limited to (less than) a single output block. For this case, breaking the MAC function can be considered as solving the following generic problem for f.

An adversary can query f for inputs P with $P = K||X||0^c$ and

- K: a secret key,
- X: a value of $r - |K|$ bits chosen by the adversary,

and is given the first n bits of $f(P)$, with $n \leq r$. The goal of the adversary is predict the output of $\lfloor f(P) \rfloor_n$ for non-queried values of X with a success probability higher than 2^{-n}.

Bibliography

[1] R. Anderson, *The classification of hash functions*, Proceedings of the IMA Conference in Cryptography and Coding, 1993, 1993.

[2] E. Andreeva, B. Mennink, and B. Preneel, *Security reductions of the second round SHA-3 candidates*, Cryptology ePrint Archive, Report 2010/381, 2010, http://eprint.iacr.org/.

[3] J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir, *Cube testers and key recovery attacks on reduced-round MD6 and Trivium*, in Dunkelman [26], pp. 1–22.

[4] T. Baignères, J. Stern, and S. Vaudenay, *Linear cryptanalysis of non binary ciphers*, Selected Areas in Cryptography (C. M. Adams, A. Miri, and M. J. Wiener, eds.), Lecture Notes in Computer Science, vol. 4876, Springer, 2007, pp. 184–211.

[5] M. Bellare and C. Namprempre, *Authenticated encryption: Relations among notions and analysis of the generic composition paradigm*, Asiacrypt (T. Okamoto, ed.), Lecture Notes in Computer Science, vol. 1976, Springer, 2000, pp. 531–545.

[6] 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.

[7] M. Bellare, P. Rogaway, and D. Wagner, *The EAX mode of operation*, in Roy and Meier [57], pp. 389–407.

[8] M. Bellare and B. Yee, *Forward-security in private-key cryptography*, Cryptology ePrint Archive, Report 2001/035, 2001, http://eprint.iacr.org/.

[9] D. J. Bernstein, *The Salsa20 family of stream ciphers*, 2007, Document ID: 31364286077dcdff8e4509f9ff3139ad, http://cr.yp.to/papers.html#salsafamily.

[10] 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/.

[11] ———, *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.

[12] ———, *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.

[13] ———, *Sufficient conditions for sound tree and sequential hashing modes*, Cryptology ePrint Archive, Report 2009/210, 2009, http://eprint.iacr.org/.

[14] ———, *Duplexing the sponge: single-pass authenticated encryption and other applications*, Second SHA-3 candidate conference, August 2010.

[15] ———, Keccak *sponge function family main document*, NIST SHA-3 Submission (updated), June 2010, `http://keccak.noekeon.org/`.

[16] ———, *Sponge-based pseudo-random number generators*, CHES (S. Mangard and F.-X. Standaert, eds.), Lecture Notes in Computer Science, vol. 6225, Springer, 2010, pp. 33–47.

[17] A. Biryukov (ed.), *Fast software encryption, 14th international workshop, FSE 2007, Luxembourg, Luxembourg, march 26-28, 2007, revised selected papers*, Lecture Notes in Computer Science, vol. 4593, Springer, 2007.

[18] R. Canetti, O. Goldreich, and S. Halevi, *The random oracle methodology, revisited*, Proceedings of the 30th Annual ACM Symposium on the Theory of Computing, ACM Press, 1998, pp. 209–218.

[19] 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.

[20] D. A. Cox, J. B. Little, and D. O'Shea, *Ideals, varieties, and algorithms*, third ed., Springer, 2007.

[21] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis, PhD thesis*, K.U.Leuven, 1995.

[22] J. Daemen and V. Rijmen, *Probability distributions of correlation and differentials in block ciphers*, Journal of Mathematical Cryptology **1** (2007), no. 3, 221–242.

[23] 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.

[24] I. Dinur and A. Shamir, *Cube attacks on tweakable black box polynomials*, Cryptology ePrint Archive, Report 2008/385, 2008, `http://eprint.iacr.org/`.

[25] Y. Dodis, L. Reyzin, R. Rivest, and E. Shen, *Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to MD6*, in Dunkelman [26], pp. 104–121.

[26] O. Dunkelman (ed.), *Fast software encryption, 16th international workshop, fse 2009, leuven, belgium, february 22-25, 2009, revised selected papers*, Lecture Notes in Computer Science, vol. 5665, Springer, 2009.

[27] M. Dworkin, *Request for review of key wrap algorithms*, Cryptology ePrint Archive, Report 2004/340, 2004, `http://eprint.iacr.org/`.

[28] N. Ferguson and B. Schneier, *Practical cryptography*, John Wiley & Sons, 2003.

[29] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno, *Helix: Fast encryption and authentication in a single cryptographic primitive*, Fast Software Encryption (T. Johansson, ed.), Lecture Notes in Computer Science, vol. 2887, Springer, 2003, pp. 330–346.

[30] E. Filiol, *A new statistical testing for symmetric ciphers and hash functions*, Proc. Information and Communications Security 2002, volume 2513 of LNCS, Springer, 2002, pp. 342–353.

[31] IETF (Internet Engineering Task Force), *RFC 3629: UTF-8, a transformation format of ISO 10646*, 2003, `http://www.ietf.org/rfc/rfc3629.txt`.

[32] _____, *RFC 3986: Uniform resource identifier (URI): Generic syntax*, 2005, `http://www.ietf.org/rfc/rfc3986.txt`.

[33] V. D. Gligor and P. Donescu, *Fast encryption and authentication: XCBC encryption and XECB authentication modes*, Fast Software Encryption 2001 (M. Matsui, ed.), Lecture Notes in Computer Science, vol. 2355, Springer, 2001, pp. 92–108.

[34] D. Gligoroski, R. Ødegård, and R. Jensen, *Observation: An explicit form for a class of second preimages for any message M for the SHA-3 candidate Keccak*, Available online, 2010, http://cio.nist.gov/esd/emaildir/lists/hash-forum/msg02057.html.

[35] M. Gorski, S. Lucks, and T. Peyrin, *Slide attacks on a class of hash functions*, Asiacrypt (J. Pieprzyk, ed.), Lecture Notes in Computer Science, vol. 5350, Springer, 2008, pp. 143–160.

[36] 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.

[37] C. S. Jutla, *Encryption modes with almost free message integrity*, Advances in Cryptology – Eurocrypt 2001 (B. Pfitzmann, ed.), Lecture Notes in Computer Science, vol. 2045, Springer, 2001, pp. 529–544.

[38] 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.

[39] L. R. Knudsen and V. Rijmen, *Known-key distinguishers for some block ciphers*, Advances in Cryptology – Asiacrypt 2007, 2007, pp. 315–324.

[40] 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.

[41] T. Kohno, J. Viega, and D. Whiting, *CWC: A high-performance conventional authenticated encryption mode*, in Roy and Meier [57], pp. 408–426.

[42] M. Liskov, *Constructing secure hash functions from weak compression functions: The case for non-streamable hash functions*.

[43] L.Knudsen, C. Rechberger, and S. Thomsen, *The Grindahl hash functions*, in Biryukov [17], pp. 39–57.

[44] S. Lucks, *Two-pass authenticated encryption faster than generic composition*, Fast Software Encryption (H. Gilbert and H. Handschuh, eds.), Lecture Notes in Computer Science, vol. 3557, Springer, 2005, pp. 284–298.

[45] 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.

[46] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.

[47] R. C. Merkle, *Secrecy, authentication, and public key systems, PhD thesis*, UMI Research Press, 1982.

[48] F. Muller, *Differential attacks against the Helix stream cipher*, in Roy and Meier [57], pp. 94–108.

[49] NIST, *AES key wrap specification*, November 2001.

[50] ———, *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`.

[51] ———, *NIST special publication 800-38C, recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality*, July 2007.

[52] ———, *NIST special publication 800-38D, recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC*, November 2007.

[53] S. Paul and B. Preneel, *Solving systems of differential equations of addition*, ACISP (C. Boyd and J. M. González Nieto, eds.), Lecture Notes in Computer Science, vol. 3574, Springer, 2005, pp. 75–88.

[54] P. Rogaway, M. Bellare, and J. Black, *OCB: A block-cipher mode of operation for efficient authenticated encryption*, ACM Trans. Inf. Syst. Secur. **6** (2003), no. 3, 365–403.

[55] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, *OCB: A block-cipher mode of operation for efficient authenticated encryption*, CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security (New York, NY, USA), ACM, 2001, pp. 196–205.

[56] P. Rogaway and T. Shrimpton, *A provable-security treatment of the key-wrap problem*, Eurocrypt (S. Vaudenay, ed.), Lecture Notes in Computer Science, vol. 4004, Springer, 2006, pp. 373–390.

[57] B. K. Roy and W. Meier (eds.), *Fast software encryption, 11th international workshop, FSE 2004, Delhi, India, February 5-7, 2004, revised papers*, Lecture Notes in Computer Science, vol. 3017, Springer, 2004.

[58] P. Sarkar and P. J. Schellenberg, *A parallelizable design principle for cryptographic hash functions*, Cryptology ePrint Archive, Report 2002/031, 2002, `http://eprint.iacr.org/`.

[59] 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.

[60] W3C, *Namespaces in XML 1.0 (second edition)*, 2006, `http://www.w3.org/TR/2006/REC-xml-names-20060816`.

[61] D. Whiting, B. Schneier, S. Lucks, and F. Muller, *Fast encryption and authentication in a single cryptographic primitive*, ECRYPT Stream Cipher Project Report 2005/027, 2005, `http://www.ecrypt.eu.org/stream/phelixp2.html`.

[62] Wikipedia, *Cryptographic hash function*, 2008, `http://en.wikipedia.org/wiki/Cryptographic_hash_function`.

[63] ———, *Random permutation statistics*, 2008, `http://en.wikipedia.org/wiki/Random_permutation_statistics`.

[64] H. Wu and B. Preneel, *Differential-linear attacks against the stream cipher Phelix*, in Biryukov [17], pp. 87–100.