Generic Security of the SAFE API and Its Applications

Dmitry Khovratovich[1], Mario Marhuenda Beltrán[2], and Bart Mennink[2]

[1] Ethereum Foundation, Luxembourg
khovratovich@gmail.com
[2] Radboud University, Nijmegen, The Netherlands
m.marhuenda@cs.ru.nl, b.mennink@cs.ru.nl

Abstract. We provide security foundations for SAFE, a recently introduced API framework for sponge-based hash functions tailored to prime-field-based protocols. SAFE aims to provide a robust and foolproof interface, has been implemented in the Neptune hash framework and some zero-knowledge proof projects, but despite its usability and applicability it currently lacks any security proof. Such a proof would not be straightforward as SAFE abuses the inner part of the sponge and fills it with protocol-specific data.
In this work we identify the SAFECore as versatile variant sponge construction underlying SAFE, we prove indifferentiability of SAFECore for all (binary and prime) fields up to around $|\mathbb{F}_p|^{c/2}$ queries, where $\mathbb{F}_p$ is the underlying field and c the capacity, and we apply this security result to various use cases. We show that the SAFE-based protocols of plain hashing, authenticated encryption, verifiable computation, non-interactive proofs, and commitment schemes are secure against a wide class of adversaries, including those dealing with multiple invocations of a sponge in a single application. Our results pave the way of using SAFE with the full taxonomy of hash functions, including SNARK-, lattice-, and x86-friendly hashes.

Keywords: SAFE, sponge, API, field elements, indifferentiability

1 Introduction

The sponge construction is a permutation-based mode for cryptographic hashing. It was first introduced by Bertoni et al. [13], and it quickly gained in popularity, in particular in light of the SHA-3 competition [33], which was won by the Keccak sponge function [11]. The sponge operates on a b-bit state, which is split into a c-bit inner part, where c is called the "capacity", and an r-bit outer part, where r is called the "rate". On input of a message, the sponge first injectively pads this message and splits it into r-bit chunks. These chunks are then absorbed one by one by adding them to the outer part of the state, where each addition is interleaved with an evaluation of a b-bit permutation of the state. After the message is absorbed, digests are squeezed r bits at a time by extracting them from the outer part.

Under the assumption that the permutation is random, Bertoni et al. [13] proved that the sponge behaves like a random oracle up to around $2^{c/2}$ queries in the indifferentiability framework [29]. Naito and Ohta proved a similar result for a slightly more general setting where the initial message block can be $r + c/2$ bits, and the squeezing is performed with around $r + c/2 - \log_2(c)$ bits at a time [32].

These are powerful results: they imply that the sponge construction behaves like a random oracle and can replace it as such in many applications, as long as less than $2^{c/2}$ evaluations of the permutation are made. They imply that finding collisions, preimages, or second preimages is not easier than finding them for a random oracle (up to this bound),[3] but they can also be used in keyed applications [11]. Improved but comparable results for keyed applications are derived by using the sponge's sibling, namely the duplex construction [9,10,18,19, 31]. A thorough account of the duplex can be found in the work by Mennink [30].

1.1 Field-Based Sponges and SAFE API

Both the sponge and the duplex specification, however, see their inputs and outputs as raw bits, and leave application-specific encoding to the users. The exact encoding, as long as it is injective and reasonably simple, does not pose a performance problem for regular hash functions such as SHA-2/3 as they are usually not a bottleneck in applications. The situation is drastically different in protocols that operate on prime field elements rather than on bits, and particularly in those that deal with verifiable computation of hash functions – e.g. private cryptocurrencies and mixers [2, 25], recursive proof systems [15, 26], and zero-knowledge virtual machine (ZKVM) computations [38]. The infamous example of Zcash's transaction requiring 40 seconds to be generated triggered the design of field-oriented hash functions Poseidon [22], Rescue [4], MiMC [3], and Reinforced Concrete [21]. With many of these functions designed in the sponge framework, it became crucial to utilize as much throughput of the sponge as possible, ideally removing all possible overhead such as padding. Indeed, a sponge-based hash function with a rate of $r = 2$ elements spends one permutation to hash a pair of unpadded field elements, but two permutation calls if the input is padded. For obvious reasons, a straightforward removal of padding or building a hash function directly from the inner permutation rather than the sponge framework has led to terrible bugs [1] and, in general case, to bad practices. Extensive use of a sponge function may also incur domain-separation issues or even cross-oracle collisions, i.e. collisions between the implementations of two different random oracles in the same protocol.

Another problem, not specific to sponge functions, arises in the context of the interactive protocols and the Fiat-Shamir heuristic [20] to make them non-interactive. Researchers have found several critical bugs in the implementations of Fiat-Shamir [8,17,24], which are partly attributed to the fact that the protocol state is stateful and interactive whereas the older hash functions from the SHA

[3] For preimage resistance, an improved result is derived, cf. [27].

family are not interactive and few implementations are stateful. It is natural to implement Fiat-Shamir via sponges, but no concrete design has been proposed so far.

To salvage this issue, Aumasson et al. [5] proposed SAFE (Sponge API for Field Elements), a generic API for sponge functions specifically tailored towards its use on field elements. They also provided a production-ready reference implementation. SAFE has already been implemented in Filecoin's Neptune hash framework and has been integrated in other zero-knowledge proof projects [28, 36]. A sponge call in SAFE takes as input an input-output (IO) pattern IO, that among others contains the particular order of the absorb calls and squeeze calls, and optionally a domain separator D. The IO and D are then hashed onto $c/2$ elements of the inner part of the state (using a general collision resistant hash function like SHA-3). Then, it operates a sponge in an online mode, where data is absorbed as it comes and squeezed as it is needed, provided the absorbing and squeezing happen in accordance with the IO pattern IO. At first sight, this IO pattern seems to limit the generality as it encodes the upcoming hash *in advance*, but this is not a problem for most applications of SAFE: e.g. Merkle trees, interactive protocols, and verifiable encryption of cryptocurrency transactions all know how much data and in which order should be hashed. On the other hand, the usage of the IO avoids the need to use padding, eliminates misuse patterns by limiting the set of callable operations, and contributes to avoiding collisions between instantiations of different oracles. As such, SAFE forms a versatile API for many protocols that use hash functions under the hood, and would like to do it securely.

One may wonder the choice of using two different hash function, a regular one for absorbing the IO pattern and the domain separator, and a field-oriented one for the online mode. However, there are multiple reasons to do so. First, since IO and D are usually protocol constants, it makes sense to precompute the initial state of SAFE. If IO and D were simply absorbed into the sponge, the size of the precomputation would be b field elements. By hashing them into the inner part, we reduce this to $c/2$ field elements. Second, since D is optional and of variable length, it would require an extra padding (and a specification of it) to make sure it does not overlap with the future absorptions. This extra padding can be costly if working over large fields. Third, the hashing of IO and D is currently independent of the sponge used within SAFE. This allows us to reuse hashed states across different sponges while still providing the same security. This also emphasizes that the processing of IO is different from processing the data as those are inputs that are generated by different roles: protocol designer and protocol party, respectively.

1.2 Generic and Improved Security of SAFE API

It is clear that SAFE API is a versatile API with many potential applications. For example, it has already found employment in the Filecoin's Neptune[4] hash

[4] https://github.com/filecoin-project/neptune/tree/master/src/sponge.

framework. However, despite all its utility, a rigorous analysis of SAFE API, providing a tight bound on its security, is missing. It is possible to argue security of SAFE in the random oracle model using the indifferentiability result of Naito and Ohta [32], but the resulting bound is not quite as good. Most importantly, Naito and Ohta apply an injective padding to the message, which is absent in SAFE. In addition, as in SAFE the encoding of the IO and D are hashed into only $c/2$ field elements of the inner part, an adaptation of the security proof of Naito and Ohta to this setting would give $c/4$ field element security at best. For authenticated encryption applications, i.e. where absorbing rounds and squeezing rounds are interleaved, an additional point of concern is raised as one uses the same IO pattern for different message lengths.

These issues leave us with an undesired situation: (i) the security proof is not rigorous, and (ii) even if it were correct, only security up to $|\mathbb{F}_p|^{c/4}$ evaluations is guaranteed due to possible hash function collisions.

In order to both derive a rigorous analysis and to improve this $|\mathbb{F}_p|^{c/4}$ bound, we first describe a variant sponge construction, called `SAFECore`, on top of a cryptographic hash function `H` and a permutation `P`. It gets as input an IO pattern $IO \in (\mathbb{N}_+)^*$, optionally a domain separator $D \in \{0,1\}^*$, and a message M of appropriate length. The IO pattern is required to be of even length, and alternatingly describes the number of field elements absorbed and squeezed. Note that this definition is slightly different from the IO pattern in the SAFE API, but the changes are only cosmetic and are made to make the security proof easier to construct and process. Besides, a translation between the two is clear. The `SAFECore` construction then operates by first hashing the IO pattern and the domain separator onto the *entire inner part* (as opposed to half of it) using hash function `H`, and processing a sponge as usual using permutation `P`. The message is required to be of appropriate length as dictated by the IO pattern IO, and the number of squeezed blocks will be determined by the *next* element in IO. We stress that this seems like a restriction compared to the original SAFE API, but this restriction is solely to make the proof convenient; after all, `SAFECore` will be used as *building block* to argue security of the SAFE API. A detailed description of `SAFECore` is given in Section 3, including our security result (proof in Section 4) guaranteeing indifferentiability up to $|\mathbb{F}_p|^{c/2}$ queries. Here, we stress that we have improved security compared to what was suggested for the SAFE API based on the work of Naito and Ohta. This is because of our observation that one can hash the IO pattern and the domain separator onto the *entire* inner part, without any risk and thus with a free security improvement from $|\mathbb{F}_p|^{c/4}$ to $|\mathbb{F}_p|^{c/2}$. The observation is comparable to the truncated permutation without initial value construction of Grassi and Mennink [23] with significant difference that their construction only makes one permutation call on input of a partially random partially chosen state, instead of a full-fledged sponge.

Our proof is field-agnostic, which extends the domain of SAFE from the originally envisioned 256-bit fields to both bigger and smaller ones: processing 380-bit curve coordinates [14], 64-bit hashes for verifiable computation based

on FRI commitments [34], 12-bit hashes for aggregating post-quantum signatures [35], and other lattice-based scenarios.

1.3 Applications

We stress that `SAFECore` is not made to be ran only in isolation, it rather serves as building block to argue security of the much more versatile and user-friendly SAFE API. In more detail, the specification of the SAFE API [5] (discussed in Section 5) is very general, but to assure generality in a foolproof interface, it follows strict rules with respect to the IO pattern IO and the upcoming absorbing and squeezing evaluations. `SAFECore`, in turn, is defined in such a way that it is possible to describe any correct application of the SAFE API in terms of the `SAFECore` construction. This immediately implies generic security of the application in light of our indifferentiability result of Theorem 2.

To exemplify this, we discuss in Sections 5.1–5.4 various applications of SAFE based on those given by the designers [5]: plain hashing, commitment schemes, interactive protocols, and authenticated encryption. For each of these applications, we describe *exactly* how they can be built by using `SAFECore` internally, and we derive generic security in the appropriate model for the application. We demonstrate that all applications achieve 128-bit security in their respective model, provided $c \geq \mu$ elements and provided they output μ elements, where μ is the number of field elements that correspond to 256 bits (or a little less). For example, for prime fields stemming from elliptic curve groups, we typically set $\mu = 1$. Likewise, for 64-bit Goldilocks prime field [34], we would have $\mu = 4$.

Our results, in fact, imply something stronger. Each particular application defines for the underlying protocols which kind of adversaries it protects against. As SAFE requires the protocol that uses it to specify the length of input and output messages, in many real-world scenarios the application does not bother with collisions or preimages that violate the specification. We call this setting *single-oracle security*. A less frequent case is when the application needs protection against inputs of other lengths too. This case may arise when a protocol employs different random oracles that take different inputs. We call this scenario *cross-oracle security*. We show that when all the oracles are implemented with `SAFECore`, and the adversary has only limited control on how these oracles are initialized using the IO pattern and the domain separation, then the security still holds up to 128-bit security.

Theorem 1. *(Informal) Let $\mathcal{P}$ be a cryptographic protocol that employs random oracles $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k$ and is secure in the random oracle model against adversaries that make up to 2^λ queries to the oracles. Then, the implementation of this protocol with oracle $\mathcal{R}_i$ instantiated with the SAFE API using a field of size at least $2^{2\lambda}$ and a domain separator D_i (pairwise distinct) is secure against adversaries that make up to 2^λ queries to underlying hash* `H` *and permutation* `P`*.*

A more detailed statement can be found in Section 5.

1.4 Outline

Section 2 introduces the notation we will use and the necessary context, such as the sponge construction and the indifferentiability framework. Section 3 describes the `SAFECore` construction in detail and its generic security result (Theorem 2). In Section 4 we give a formal proof of the security result. We discuss the SAFE API in detail in Section 5, where we also demonstrate how the security of `SAFECore` implies the security of any proper evaluation of the SAFE API. In Sections 5.1–5.4 this fundamental observation is applied to various use cases of the SAFE API in order to derive simple and meaningful security claims. The work is concluded in Section 6.

2 Preliminaries

2.1 Notation

We use machine typographic fonts to denote functions (e.g. $\mathtt{A}, \mathtt{a}$), upper case bold to denote sets (e.g. $\mathbf{A}, \mathbf{B}$), and case sans-serif to denote variables (e.g. a, b). To denote the set of natural numbers, we use $\mathbb{N}$. We use $\emptyset$ to denote the empty set and $\mathbf{S}^* = \cup_{i=0}^{\infty} \mathbf{S}^i$ for a set $\mathbf{S}$. Given $x \in \mathbf{S}^*$, there exists a unique n so that $x \in \mathbf{S}^n$, we denote it by $\mathtt{len}(x) = n$. Abusing notation, we denote the empty string by $\emptyset$ as well. For a finite set $\mathbf{S}$, we say that $x \xleftarrow{\$} \mathbf{S}$ when x is sampled uniformly from $\mathbf{S}$. Throughout, we will use r to denote the rate and c to denote the capacity. For an explanation of their meaning see Section 3.1.

Given a tuple $x = (x_1, x_2, \ldots) \in \mathbf{S}^*$, we also denote it $x = x_1 \| x_2 \| \ldots$ and we use both notations interchangeably. We denote $x[1:k] = (x_1, \ldots, x_k)$. We denote by $\mathtt{left}_r : \mathbf{S}^* \to \mathbf{S}^r$ and $\mathtt{right}_c : \mathbf{S}^* \to \mathbf{S}^c$ the functions defined by $\mathtt{left}_r(m_r \| m_{\text{rest}}) = m_r$ and $\mathtt{right}_c(m_{\text{rest}} \| m_c) = m_c$.

Given $M \in \mathbf{S}^*$, we denote $\mathtt{cut}_r(M) = (M_1, \ldots, M_\ell)$, where:

$$
\begin{aligned}
M_1 &= M[1:r]\,, \\
M_2 &= M[r+1:2r]\,, \\
M_3 &= M[2r+1:3r]\,, \\
&\vdots \\
M_\ell &= M[(\ell-1)r+1 : \mathtt{len}(M)] \| 0^{-M \bmod r}\,,
\end{aligned}
$$

where 0^ℓ denotes the all 0's string of bits of length ℓ. We denote by $\mathtt{pad}_r(\cdot)$ an injective padding, e.g. an injective function $\mathbf{S}^* \to (\mathbf{S}^r)^*$. A usual padding is the 10-padding, which works by appending one 1 and filling the rest with 0's, in the case elements $\mathbf{S}$ can be represented by a string of bits, e.g. when $\mathbf{S}$ is a finite field.

We use `RO` to denote a random oracle [6].

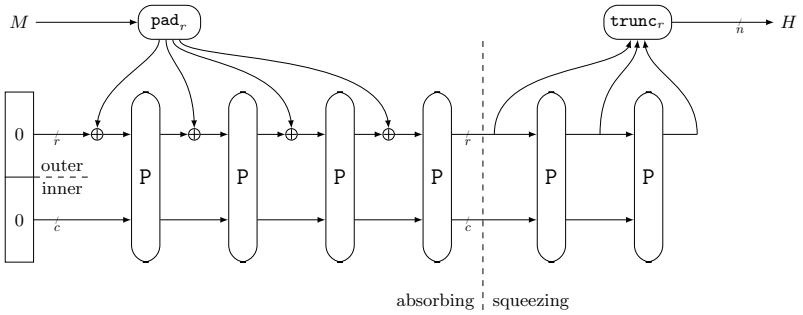

Fig. 1: Sponge construction, with a requested output of n bits.

2.2 Security Model

In this paper we use the indifferentiability framework, first introduced by Maurer et al. [29] and refined to the context of hash functions by Coron et al. [16]. We introduce the indifferentiability framework below. We will use it to analyze the `SAFECore` construction in Section 3.

Consider a construction $\mathtt{C}$, relying on an ideal primitive $\mathtt{P}$: $\mathtt{C}^{\mathtt{P}} : \mathbf{S}^* \to \mathbf{S}^*$. Then consider a simulator, $\mathtt{S}$, with the same interface as $\mathtt{P}$. Finally, we consider a distinguisher $\mathtt{D}$, which is an algorithm having access to either $(\mathtt{RO}, \mathtt{S}^{\mathtt{RO}})$ or $(\mathtt{C}^{\mathtt{P}}, \mathtt{P})$. In the first case we say that $\mathtt{D}$ is in the ideal world, denoted by W_I, whereas in the second case, it is said to be in the real world, denoted by W_R. The goal of $\mathtt{D}$ is to determine in which world it was placed. If $\mathtt{D}$ determines it is in W_R, it outputs 0, and 1 otherwise.

The advantage of $\mathtt{D}$ is defined as:

$$\mathbf{Adv}^{\text{iff}}_{\mathtt{C},\mathtt{S}}(\mathtt{D}) = \left|\Pr\left[\mathtt{D}^{W_I} \Rightarrow 1\right] - \Pr\left[\mathtt{D}^{W_R} \Rightarrow 1\right]\right| . \tag{1}$$

2.3 Sponge Construction

In this section, we give a description of the standard sponge construction operating on bits. Let $b, r, c \in \mathbb{N}$ such that $b = r + c$. Let $\mathtt{P} : \{0,1\}^b \to \{0,1\}^b$ be a permutation.

First, the sponge gets an input, $(M, n) \in \{0,1\}^* \times \mathbb{N}_+$. It *absorbs* the message M and then it *squeezes* n bits as output. A formal description is given in Figure 1 and Algorithm 1.

2.4 Limitations in Application

The sponge construction is a powerful versatile tool. In particular, it can be used to argue security of the duplex construction and security of keyed applications of the sponge (e.g. the keyed sponge [12]) or keyed applications of the duplex (e.g. SpongeWrap [10]) follow from the indifferentiability of the sponge construction.

Algorithm 1 Sponge construction

Data: input $(M, n) \in \{0,1\}^* \times \mathbb{N}_+$
Result: output $Z \in \{0,1\}^\infty$

$S = 0^b$ ▷ State of the sponge construction
$Z = \emptyset$ ▷ Output string
$(M_1, \ldots, M_\ell) = \mathtt{pad}_r(M)$
for $1 \leq i \leq \ell$ **do** ▷ Absorb
    $S \leftarrow \mathtt{P}(S \oplus (M_i \| 0^c))$
end for
for $1 \leq i \leq \lceil \frac{n}{r} \rceil$ **do** ▷ Squeeze
    $Z \leftarrow Z \| \mathtt{left}_r(S)$
    $S \leftarrow \mathtt{P}(S)$
end for
return $Z[1:n]$ ▷ $Z[1:n]$ means the first n bits

On the downside, however, the sponge requires an injective padding. A typical choice for this is the 10-padding, which on input of a message $M \in \{0,1\}^*$ appends a single 1 and a sufficient number of 0's such that the resulting string is in $(\{0,1\}^r)^*$. Although in most use cases this is fine, it is problematic if the sponge is not applied on raw bits but rather on (large) field elements where we take a low value for r. For example, if one uses a permutation on top of two field elements, one simply takes $c = r = 1$, and padding *always* incurs an extra permutation call.

We stress that one cannot simply discard the 10-padding. The reason for this is that ending with a 0^r-block could be problematic. Consider, for the sake of example, a simplified setting where the sponge is evaluated for two padded messages, $M \in \{0,1\}^{3r}$ with a requested digest of $2r$ bits and $M' = M \| 0^r \in \{0,1\}^{4r}$ with a requested digest of r bits. In this case, we will necessarily have

$$\mathtt{Sponge}(M, 2r)[r+1:2r] = \mathtt{Sponge}(M', r)[1:r]\,,$$

which would happen for a `RO` with negligible probability.

We stress that it *is* possible to have padded messages ending with a 0^r-block, but only in very restricted settings, where in particular overlapping squeeze/absorb evaluations are avoided. This case is, however, not supported by the current sponge indifferentiability proofs [13, 32].

3 `SAFECore` construction

In this section, we will describe the `SAFECore` construction, which will be a building block that we will use in Section 5 to argue security of the full SAFE API. We first describe the construction in Section 3.1, we give an extensive example use case in Section 3.2, and we discuss the security of `SAFECore` in Section 3.3.

3.1 Construction

In this section we give a description of the SAFECore construction. Consider a finite field $\mathbb{F}_p$. Let $b, r, c \in \mathbb{N}$ such that $b = r + c$. Let $\mathtt{P} : \mathbb{F}_p^b \to \mathbb{F}_p^b$ be a permutation, and let $\mathtt{H} : (\mathbb{N}_+)^* \times \{0,1\}^* \to \mathbb{F}_p^c$ be a hash function. Given $X = (X_r, X_c) \in \mathbb{F}_p^b$, we reuse the previous notation: $\mathtt{left}_r(X) = X_r$ and $\mathtt{right}_c(X) = X_c$.

SAFECore takes an input $(IO, D, M) \in (\mathbb{N}_+)^* \times \{0,1\}^* \times (\mathbb{F}_p)^*$. Here, IO is the input-output (IO) pattern, D, an optional domain separator that will mostly be of use in the applications in Section 5, and message, M, which is expected to obey to IO in a certain way. To be precise, IO is a tuple of even length, that we decompose as $IO = (I_1, O_1, \ldots, I_\ell, O_\ell)$, where the I_i correspond to the number of elements of $\mathbb{F}_p$ absorbed and the O_i correspond to the number of elements of $\mathbb{F}_p$ squeezed. Looking ahead, the SAFE API alternates absorbing phases with squeezing phases as prescribed by IO. As SAFECore will be used as building block, it is more restricted. To be precise, in SAFECore the message M is restricted to the condition that its length $\mathtt{len}(M)$ should be equal to $I_1 + I_2 + \cdots + I_k$ for some $k \leq \ell$. In this case, the number of squeezed elements will be O_k.

Formally, we define the set of acceptable inputs:

$$\mathbf{I} = \left\{ (IO, D, M) \in (\mathbb{N}_+)^* \times \{0,1\}^* \times (\mathbb{F}_p)^* \;\middle|\; \begin{array}{l} IO = (I_1, O_1, \ldots, I_\ell, O_\ell)\,, \\ \\ \exists\, k \text{ such that } \mathtt{len}(M) = \displaystyle\sum_{j=1}^{k} I_j \end{array} \right\}. \tag{2}$$

For any $(IO, D, M) \in \mathbf{I}$, we define $\mathsf{absrnds}(IO, M)$ as the unique number k such that $\mathtt{len}(M) = \sum_{j=1}^{k} I_j$.

On input of a tuple $(IO, D, M) \in \mathbf{I}$, SAFECore evaluates H on input of (IO, D) to obtain a value $H \in \mathbb{F}_p^c$, which it uses to initialize the inner part of the sponge. Then, a variant of the sponge is used to absorb M *in accordance with the IO pattern* IO. For this, a specific padding function SAFECorePad (Algorithm 2), will be employed. SAFECorePad properly pads each absorption round (noting that I_j is expressed in terms of elements and not in terms of r-element blocks) and for blank evaluations of in-between squeezing rounds. Then, at the end, it squeezes $O_{\mathsf{absrnds}(IO,M)}$ elements in $\mathbb{F}_p$. We stress that this last step is *not* in accordance with how the SAFE API works, recalling that it alternates absorbing and squeezing phases, but after all, SAFECore is defined more restrictively as being an easy-to-analyze building block for the SAFE API. A full description of the SAFECore construction is given in Figure 2 and Algorithm 3.

3.2 Example

Consider an instantiation of SAFECore with parameters $c = 2, r = 2$. A typical IO pattern could be $IO = (8, 6, 5, 3, 4, 7)$. In the SAFE API (that we will discuss

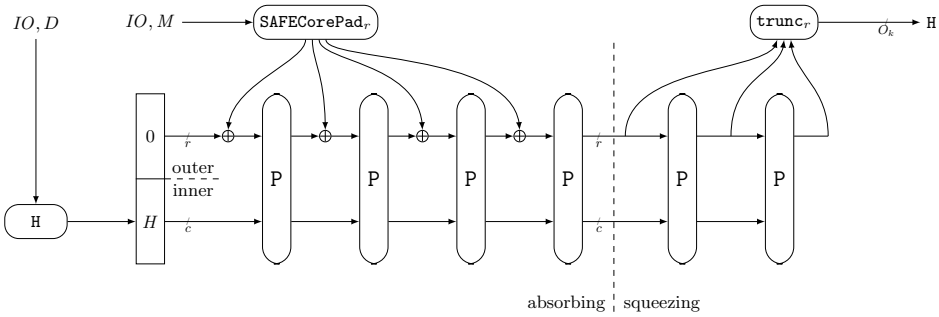

Fig. 2: `SAFECore` construction, where the input message M is of length $I_1+\cdots+I_k$ elements and the digest consists of O_k elements. The function `SAFECorePad` is described in Algorithm 3.

Algorithm 2 Description of `SAFECorePad`

Data: input $(IO, M) \in \mathbf{I}$
Result: output $M' \in (\mathbb{F}_p^r)^*$

$k = \texttt{absrnds}(IO, M)$
$M' = \emptyset$ ▷ Output string
for $1 \leq i \leq k-1$ **do**
    $M' \leftarrow M' \| M[I_1 + \cdots + I_{i-1} + 1 : I_1 + \cdots + I_i] \| 0^{-I_i \bmod r}$
    $M' \leftarrow M' \| 0^{r\lceil O_i/r \rceil}$
end for
$M' \leftarrow M' \| M[I_1 + \cdots + I_{k-1} + 1 : I_1 + \cdots + I_k] \| 0^{-I_k \bmod r}$
return M'

in Section 5), this pattern means that we start with absorbing 8 elements in $\mathbb{F}_p$ (which happens in 4 rounds, as $r = 2$), followed by squeezing 6 elements in $\mathbb{F}_p$ (which happens in 3 rounds), followed by absorbing 5 elements (which happens in 3 rounds), and so on. However, `SAFECore` is more restrictive than that, in order to be able to have an easy-to-analyze building block for the SAFE API. Concretely, for the example IO, there are three permissible message lengths:

- 8 elements from $\mathbb{F}_p$, in which case the output consists of 6 elements from $\mathbb{F}_p$;
- 13 elements from $\mathbb{F}_p$, in which case the output consists of 3 elements from $\mathbb{F}_p$;
- 17 elements from $\mathbb{F}_p$, in which case the output consists of 7 elements from $\mathbb{F}_p$.

We remark that, although IO puts restrictions on the length of M, one may allow arbitrary-length squeezing at the end. We have not included this option in our formalization in order to stay close to the SAFE API, but the security analysis in our work would allow this.

Algorithm 3 Description of SAFECore

Data: input $(IO, D, M) \in \mathbf{I}$
Result: output $Z \in (\mathbb{F}_p)^*$

$S = 0^r \| \mathtt{H}(IO, D)$ ▷ State of the SAFECore construction
$Z = \emptyset$ ▷ Output string
$M' = \mathtt{SAFECorePad}(IO, M)$ ▷ See Algorithm 2
for $1 \le i \le \mathtt{len}(M')/r$ **do** ▷ Absorb
    $S \leftarrow \mathtt{P}(S \oplus (M'[r \cdot (i-1)+1 : r \cdot i] \| 0^c))$
end for
$k = \mathtt{absrnds}(IO, M)$
for $1 \le i \le \lceil O_k / r \rceil$ **do** ▷ Squeeze
    $Z \leftarrow Z \| \mathtt{left}_r(S)$
    $S \leftarrow \mathtt{P}(S)$
end for
return $Z[1 : O_k]$ ▷ $Z[1 : O_k]$ means the first O_k elements

3.3 Security of SAFECore Construction

When D is in the real world, we count the cost of queries by how many times H and P are called, where duplicate queries are only counted once. For our running example of Section 3.2, if one makes three evaluations of SAFECore as suggested, where the three message inputs are prefixes of each other, the total cost is the total number of unique permutation evaluations, which happens to be as much as the cost of the longest query of the three.

When D is in the ideal world, we likewise count the cost of queries by how many times H and P would have been called, had the same query been made in the real world.

We now state the main security result.

Theorem 2 (Security of SAFECore). *Let* C *be the* SAFECore *construction based on a random oracle* H *and random permutation* P*. There exists a simulator* S*, such that for any distinguisher* D *making at most* $Q_{\mathtt{H}}$ *unique hash queries and* $Q_{\mathtt{P}}$ *unique primitive queries:*

$$\boldsymbol{Adv}^{\mathrm{iff}}_{\mathtt{C},\mathtt{S}}(\mathtt{D}) \le \frac{3 \cdot \binom{Q_{\mathtt{H}}}{2} + 2 \cdot \binom{Q_{\mathtt{P}}}{2} + 4 \cdot Q_{\mathtt{P}} \cdot Q_{\mathtt{H}}}{|\mathbb{F}_p|^c} + \frac{3 \cdot \binom{Q_{\mathtt{P}}}{2}}{|\mathbb{F}_p|^b}. \tag{3}$$

The proof is given in Section 4.

4 Proof of Theorem 2

Let C be the SAFECore construction based on a random oracle H and random permutation P. Our goal is to construct a simulator S such that for any distinguisher D, the following distance is "small", in a precise way:

$$\mathbf{Adv}^{\mathrm{iff}}_{\mathtt{C},\mathtt{S}}(\mathtt{D}) = \left| \Pr\left[\mathtt{D}^{\mathtt{RO},\mathtt{S}^{\mathtt{RO}}} \Rightarrow 1\right] - \Pr\left[\mathtt{D}^{\mathtt{C}^{\mathtt{H},\mathtt{P}},\mathtt{H},\mathtt{P}} \Rightarrow 1\right] \right|. \tag{4}$$

Here, $\mathtt{S}$ simulates both the hash function $\mathtt{H}$ and the construction $\mathtt{P}$. The world $(\mathtt{RO}, \mathtt{S}^{\mathtt{RO}})$ is called the ideal world and $(\mathtt{C}^{\mathtt{H},\mathtt{P}}, \mathtt{H}, \mathtt{P})$ is called the real world. These worlds are depicted in Figure 3.

First, in Section 4.1, we will describe our simulator. In Section 4.2 we will describe an intermediate world and apply the triangle inequality to derive two easier-to-bound distances from (4). These two distances are then bounded in Sections 4.4 and 4.5, using bad events introduced in Section 4.3. The proof is inspired by that of Naito and Ohta [32], but in addition taking into account the hashing functionality and its related bad events.

4.1 Simulator

We first define $\mathbf{I}_{\text{ext}}$ (read $\mathbf{I}$ extended):

$$\mathbf{I}_{\text{ext}} = \bigcup_{(IO,D,M)\in\mathbf{I}} \left\{ (IO, D, M'\|M'') \;\middle|\; \begin{array}{l} M' = \mathtt{SAFECorePad}(IO, M)\,, \\ M'' \in \{\emptyset, 0^r, \ldots, 0^{r(\lceil O_k/r\rceil - 1)}\}\,, \\ \text{where } k = \mathtt{absrnds}(IO, M) \end{array} \right\},$$

where the function $\mathtt{SAFECorePad}$ is defined in Algorithm 3. Intuitively $\mathbf{I}_{\text{ext}}$ covers all tuples for which the simulator knows that, if it receives an input value X to $\mathtt{S}_{\mathtt{P}}$ "completing" $(IO, D, M) \in \mathbf{I}_{\text{ext}}$, it will have to output a value consistent with the random oracle. However, it will also need to know *which* indices of the random oracle output it has to select. Therefore, for any $(IO, D, M) \in \mathbf{I}_{\text{ext}}$, we define $\mathtt{0elts}(IO, M)$ as the total number of elements (i.e. the length of $\mathtt{len}(M'')$) attached to M'.

The simulator can be queried through three interfaces: $\mathtt{S}_{\mathtt{H}}$, $\mathtt{S}_{\mathtt{P}}$, and $\mathtt{S}_{\mathtt{P}}^{-1}$. It maintains tables $\mathbf{C}_{\mathtt{H}}$ and $\mathbf{C}_{\mathtt{P}}$ recording the query-response pairs of each query: any input-output tuple $\mathtt{S}_{\mathtt{H}}(IO, D) \mapsto H$ is stored as (IO, D, H) in $\mathbf{C}_{\mathtt{H}}$, and any input-output tuple $\mathtt{S}_{\mathtt{P}}(X) \mapsto Y$ or $\mathtt{S}_{\mathtt{P}}^{-1}(Y) \mapsto X$ is stored as (X, Y) in $\mathbf{C}_{\mathtt{P}}$. Furthermore, we define:

$$\begin{aligned}
\mathbf{D}_{\mathtt{H}} &= \{(IO, D) \in (\mathbb{N}_+)^* \times \{0,1\}^* \mid \exists H \in \mathbb{F}_p^c \text{ s.t. } (IO, D, H) \in \mathbf{C}_{\mathtt{H}}\}\,, \\
\mathbf{R}_{\mathtt{H}} &= \{H \in \mathbb{F}_p^c \mid \exists (IO, D) \in (\mathbb{N}_+)^* \times \{0,1\}^* \text{ s.t. } (IO, D, H) \in \mathbf{C}_{\mathtt{H}}\}\,, \\
\mathbf{D}_{\mathtt{P}} &= \{X \in \mathbb{F}_p^b \mid \exists Y \in \mathbb{F}_p^b \text{ s.t. } (X, Y) \in \mathbf{C}_{\mathtt{P}}\}\,, \\
\mathbf{R}_{\mathtt{P}} &= \{Y \in \mathbb{F}_p^b \mid \exists X \in \mathbb{F}_p^b \text{ s.t. } (X, Y) \in \mathbf{C}_{\mathtt{P}}\}\,.
\end{aligned}$$

The simulator maintains a graph that it uses to avoid discrepancies that $\mathtt{D}$ might detect. We adopt the graph representation from Bertoni et al. [13].

The nodes are elements of $\mathbb{F}_p^b$. Two nodes $X, Y \in \mathbb{F}_p^b$ are joined by an edge if $\exists M \in \mathbb{F}_p^r$ such that $(X \oplus (M\|0^c), Y) \in \mathbf{C}_{\mathtt{P}}$. Then M is the label of the edge joining X and Y, which we denote as $X \xrightarrow{M} Y$. We write $X \to Y$ to denote that X and Y are linked through a 0-string label. We say that X is a root node if there exists $(IO, D, H) \in \mathbf{C}_{\mathtt{H}}$ so that $X = 0^r\|H$. For simplicity, we denote $X \xrightarrow{M_1} Y \xrightarrow{M_2} Z$ by $X \xrightarrow{M_1\|M_2} Z$. The graph is initialized by the simulator as

Algorithm 4 Simulator $\mathtt{S}$

Function $\mathtt{S}_{\mathtt{H}}$:
Data: input $(IO, D) \in (\mathbb{N}_+)^* \times \{0,1\}^*$
Result: output $H \in \mathbb{F}_p^c$

$H \xleftarrow{\$} \mathbb{F}_p^c$
$\mathbf{C}_{\mathtt{H}} \leftarrow \mathbf{C}_{\mathtt{H}} \cup \{(IO, D, H)\}$
return H

Function $\mathtt{S}_{\mathtt{P}}$:
Data: input $X \in \mathbb{F}_p^b$
Result: output $Y \in \mathbb{F}_p^b$

if $\exists (IO, D, M\|u) \in \mathbf{I}_{\text{ext}}, H \in \mathbb{F}_p^c : (IO, D, H) \in \mathbf{C}_{\mathtt{H}} \wedge \left(0^r \| H \overset{M}{\Longrightarrow} X \oplus (u\|0^c)\right)$ **then**
  $\alpha \leftarrow \mathtt{0elts}(IO, M\|u)$
  $M' \leftarrow \mathtt{left}_{\mathtt{len}(M\|u)-\alpha}(M\|u)$
  $Y_r \leftarrow \mathtt{RO}(IO, D, M')[\alpha+1 : \alpha+r]$
  $Y_c \xleftarrow{\$} \mathbb{F}_p^c$
  $Y \leftarrow Y_r \| Y_c$
else
  $Y \xleftarrow{\$} \mathbb{F}_p^b$
end if
$\mathbf{C}_{\mathtt{P}} \leftarrow \mathbf{C}_{\mathtt{P}} \cup \{(X, Y)\}$
return Y

Function $\mathtt{S}_{\mathtt{P}}^{-1}$:
Data: input $Y \in \mathbb{F}_p^b$
Result: output $X \in \mathbb{F}_p^b$

$X \xleftarrow{\$} \mathbb{F}_p^b$
$\mathbf{C}_{\mathtt{P}} \leftarrow \mathbf{C}_{\mathtt{P}} \cup \{(X, Y)\}$
return Y

being empty, then it is updated lazily in the following way: When a query is made, it is added to the table, and the proper edges and labels are added to the graph.

The three simulator interfaces are formally described in Algorithm 4. Here, we recall that the distinguisher does not make redundant queries.

4.2 Intermediate World

We will use an intermediate world, which we denote W_S. This world behaves like the real world, with the exception that the ideal primitives, i.e. $\mathtt{H}$ and $\mathtt{P}$, are replaced by the simulator interfaces. The world is depicted in Figure 3.

By the triangle inequality, we have:

$$(4) \leq \left|\mathbf{Pr}[\mathtt{D}^{W_I} \Rightarrow 1] - \mathbf{Pr}[\mathtt{D}^{W_S} \Rightarrow 1]\right| \tag{5}$$
$$+ \left|\mathbf{Pr}[\mathtt{D}^{W_S} \Rightarrow 1] - \mathbf{Pr}[\mathtt{D}^{W_R} \Rightarrow 1]\right|. \tag{6}$$

Distance (5) is bounded in Section 4.4 and distance (6) is bounded in Section 4.5. Before doing so, we define bad events in Section 4.3.

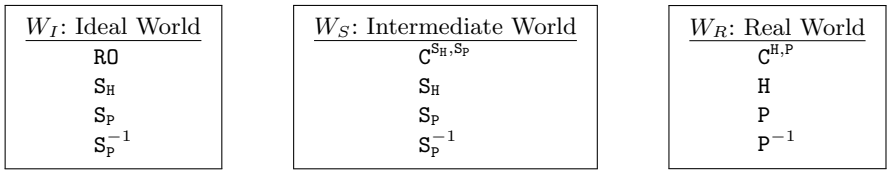

Fig. 3: Worlds involved in the security proof.

4.3 Bad events

When the distinguisher makes a query, the simulator will try to maintain consistency with the ideal world. However, it is possible that an earlier response is such that the simulator cannot guarantee consistency anymore. To capture these cases, we will define additional bad events. Note that the distinguisher can make Q queries, $Q_\mathtt{H}$ of which to the hash interface and $Q_\mathtt{P}$ of which to the permutation interface. Consider $i \in \{1, \ldots, Q\}$. We define the following bad events:

- $\mathbf{CollH}_i$: the i-th query is a query (IO, D, H) to $\mathtt{S}_\mathtt{H}$ and there exists $(IO', D', H') \in \mathbf{C}_\mathtt{H}$ such that $(IO, D) \neq (IO', D')$ and $H = H'$.
- $\mathbf{CollP}_i$: the i-th query is a query (X, Y) to $\mathtt{S}_\mathtt{P}$ or $\mathtt{S}_\mathtt{P}^{-1}$ and there exists $(X', Y') \in \mathbf{C}_\mathtt{P}$ such that either[5]
  - $X \neq X'$ and $Y = Y'$, or
  - $Y \neq Y'$ and $X = X'$.
- $\mathbf{ConnectP}_i$: either
  - the i-th query is a query (X, Y) to $\mathtt{S}_\mathtt{P}$ and there exists $(X', Y') \in \mathbf{C}_\mathtt{P}$ such that $\mathtt{right}_c(Y) = \mathtt{right}_c(X')$, or
  - the i-th query is a query (X, Y) to $\mathtt{S}_\mathtt{P}^{-1}$ and there exists $(X', Y') \in \mathbf{C}_\mathtt{P}$ such that $\mathtt{right}_c(X) = \mathtt{right}_c(Y')$.
- $\mathbf{ConnectPH}_i$: either
  - the i-th query is a query (X, Y) to $\mathtt{S}_\mathtt{P}$ and there exists $(IO, D, H) \in \mathbf{C}_\mathtt{H}$ such that $\mathtt{right}_c(Y) = H$, or
  - the i-th query is a query (X, Y) to $\mathtt{S}_\mathtt{P}^{-1}$ and there exists $(IO, D, H) \in \mathbf{C}_\mathtt{H}$ such that $\mathtt{right}_c(X) = H$, or
  - the i-th query is a query (IO, D, H) to $\mathtt{S}_\mathtt{H}$ and there exists $(X, Y) \in \mathbf{C}_\mathtt{P}$ such that $H = \mathtt{right}_c(X)$ or $H = \mathtt{right}_c(Y)$.

We furthermore define:

$$\mathbf{Bad}_i = \mathbf{CollH}_i \vee \mathbf{CollP}_i \vee \mathbf{ConnectP}_i \vee \mathbf{ConnectPH}_i\,.$$

[5] Here, we remark that the distinguisher never makes a redundant query, so it can never set the former condition in an inverse query or the latter condition in a forward query.

For each of the bad events $\mathbf{Event}_i \in \{\mathbf{Bad}_i, \mathbf{CollH}_i, \mathbf{CollP}_i, \mathbf{ConnectP}_i, \mathbf{ConnectPH}_i\}$, we write:

$$\mathbf{Event} = \bigcup_{i=1}^{Q} \mathbf{Event}_i \,.$$

Bad event **CollH** avoids hash collisions, which are problematic as they would allow different IO patterns and domain separators leading to the same root in the graph. Bad event **CollP** avoids collisions in the permutation interface. Bad event **ConnectP** avoids the case that a permutation query accidentally extends a path in the graph. Finally, bad event **ConnectPH** avoids accidentally making a non-rooted path rooted and avoids accidental collisions at the H-value.

For each of these events, if relevant, we add a superscript (like $\mathbf{Bad}^{(1)}$, $\mathbf{Bad}^{(2)}$, or $\mathbf{Bad}^{(3)}$) to indicate to which of the games (see Figure 3) it applies.

The bad events are quite straightforward to bound, and we can obtain the following lemma. In this lemma, we consider both the general bad event **Bad** as the isolated bad event **CollP**, as both results are needed separately.

Lemma 1. *For any distinguisher* $\mathtt{D}$ *making at most* $Q_{\mathtt{H}}$ *unique hash queries and* $Q_{\mathtt{P}}$ *unique primitive queries, the following holds for* $j = 1, 2$*:*

$$\mathbf{Pr}[\mathbf{CollP}^{(j)}] \leq \frac{\binom{Q_{\mathtt{P}}}{2}}{|\mathbb{F}_p|^b} \,, \tag{7}$$

$$\mathbf{Pr}[\mathbf{Bad}^{(j)}] \leq \frac{\binom{Q_{\mathtt{H}}}{2} + \binom{Q_{\mathtt{P}}}{2} + 2 \cdot Q_{\mathtt{P}} \cdot Q_{\mathtt{H}}}{|\mathbb{F}_p|^c} + \frac{\binom{Q_{\mathtt{P}}}{2}}{|\mathbb{F}_p|^b} \,, \tag{8}$$

and the following holds for $j = 1, 2, 3$*:*

$$\mathbf{Pr}[\mathbf{CollH}^{(j)}] \leq \frac{\binom{Q_{\mathtt{H}}}{2}}{|\mathbb{F}_p|^c} \,. \tag{9}$$

Proof. The bad events can in fact be easily bounded in isolation:

$$\mathbf{Pr}[\mathbf{Bad}] \leq \mathbf{Pr}[\mathbf{CollH}] + \mathbf{Pr}[\mathbf{CollP}] + \mathbf{Pr}[\mathbf{ConnectP}] + \mathbf{Pr}[\mathbf{ConnectPH}] \,.$$

For each of these four events, $\mathbf{Event} \in \{\mathbf{CollH}, \mathbf{CollP}, \mathbf{ConnectP}, \mathbf{ConnectPH}\}$, we observe that:

$$\mathbf{Pr}[\mathbf{Event}] \leq \sum_{i=1}^{Q} \mathbf{Pr}[\mathbf{Event}_i \mid \neg\mathbf{Event}_{i-1}] \leq \sum_{i=1}^{Q} \mathbf{Pr}[\mathbf{Event}_i] \,.$$

We will now consider the events separately, where the reasoning for **CollH** holds for $j = 1, 2, 3$ and the reasoning of the other events for $j = 1, 2$. In the rest of this proof we omit the superscript.

CollH. Note that this bad event only involves hash queries, so w.l.o.g. i runs from 1 to $Q_{\mathtt{H}}$. At the point of the i-th query, there are at most $i-1$ tuples in $\mathbf{C}_{\mathtt{H}}$. As the response H of the i-th query is uniformly randomly selected from $\mathbb{F}_p^c$, it sets the bad event with probability $(i-1)/|\mathbb{F}_p|^c$. We thus obtain that:

$$\mathbf{Pr}[\mathbf{CollH}] \leq \sum_{i=1}^{Q_{\mathtt{H}}} \frac{i-1}{|\mathbb{F}_p|^c} \leq \frac{\binom{Q_{\mathtt{H}}}{2}}{|\mathbb{F}_p|^c}\,.$$

CollP. Note that this bad event only involved primitive queries, so w.l.o.g. i runs from 1 to $Q_{\mathtt{P}}$. At the point of the i-th query, there are at most $i-1$ tuples in $\mathbf{C}_{\mathtt{P}}$. If the i-th query is a forward query, since the b elements of Y are uniformly randomly selected from $\mathbb{F}_p$, it sets the bad event with probability $(i-1)/|\mathbb{F}_p|^b$. The same holds in case the i-th query is an inverse query. As any query is either a forward or an inverse query (not both), we obtain that:

$$\mathbf{Pr}[\mathbf{CollP}] \leq \sum_{i=1}^{Q_{\mathtt{P}}} \frac{i-1}{|\mathbb{F}_p|^b} \leq \frac{\binom{Q_{\mathtt{P}}}{2}}{|\mathbb{F}_p|^b}\,.$$

ConnectP. Note that this bad event only involves primitive queries, so w.l.o.g. i runs from 1 to $Q_{\mathtt{P}}$. At the point of the i-th query, there are at most $i-1$ tuples in $\mathbf{C}_{\mathtt{P}}$. If the i-th query is a forward query, as the c inner elements of Y are uniformly randomly selected from $\mathbb{F}_p^c$, it sets the bad event with probability $(i-1)/|\mathbb{F}_p|^c$. The same holds in case the i-th query is an inverse query. As any query is either a forward or an inverse query (not both), we obtain that

$$\mathbf{Pr}[\mathbf{ConnectP}] \leq \sum_{i=1}^{Q_{\mathtt{P}}} \frac{i-1}{|\mathbb{F}_p|^c} \leq \frac{\binom{Q_{\mathtt{P}}}{2}}{|\mathbb{F}_p|^c}\,.$$

ConnectPH. Any query to $\mathtt{S}_{\mathtt{P}}/\mathtt{S}_{\mathtt{P}}^{-1}$ may set the bad event if its response (either Y in forward queries or X in inverse queries) has its c inner elements equal to H for an earlier query to $\mathtt{S}_{\mathtt{H}}$. Likewise, any query to $\mathtt{S}_{\mathtt{H}}$ may set the bad event if its response H equals the c inner elements of any X or Y for an earlier query to $\mathtt{S}_{\mathtt{P}}/\mathtt{S}_{\mathtt{P}}^{-1}$. As all fresh inner values and all fresh values H are uniformly randomly selected from $\mathbb{F}_p^c$, and there are at most $Q_{\mathtt{P}}$ queries to $\mathtt{S}_{\mathtt{P}}/\mathtt{S}_{\mathtt{P}}^{-1}$ and at most $Q_{\mathtt{H}}$ queries to $\mathtt{S}_{\mathtt{H}}$, and any pair sets bad with probability $2/|\mathbb{F}_p|^c$. We thus obtain that:

$$\mathbf{Pr}[\mathbf{ConnectPH}] \leq \frac{2 \cdot Q_{\mathtt{P}} \cdot Q_{\mathtt{H}}}{|\mathbb{F}_p|^c}\,.$$

Conclusion. The lemma immediately follows from adding the individual bad events. □

4.4 Bound of (5)

We will use the following lemma, which informally states that the simulator in game 2 operates consistently with the random oracle in game 1 as long as no bad event occurs.

Lemma 2. *Unless a bad event happens in game 1 or game 2, we always have the following result. For any rooted path in the simulator graph of the following form*

$$0^r \| H \xrightarrow{\texttt{SAFECorePad}(M)} Y_1 \to \cdots \to Y_\ell \,, \tag{10}$$

where $(IO, D, H) \in \mathbf{C}_{\mathtt{H}}$, $(IO, D, M) \in \mathbf{I}$, *and where* $\ell \leq \lceil O_k / r \rceil$ *for* $k = \texttt{absrnds}(IO, M)$,

$$\texttt{left}_r(Y_1) \| \cdots \| \texttt{left}_r(Y_\ell) = \texttt{RO}(IO, D, \texttt{SAFECorePad}(M))[1 : r \cdot \ell] \,. \tag{11}$$

Proof. We proceed by induction on the number of queries the distinguisher D makes. Clearly, $\mathbf{Bad}_1^{(j)}$ never happens. Assume that the lemma holds for any simulator performing $Q - 1$ queries. Consider distinguisher D making its Q-th query, where $\mathbf{Bad}_i^{(j)}$ has not occurred for $i < Q$. By hypothesis,

$$\texttt{left}_r(Y_1) \| \cdots \| \texttt{left}_r(Y_\ell) = \texttt{RO}(IO, D, \texttt{SAFECorePad}(M))[1 : r \cdot \ell] \,, \quad \ell < Q$$

for any path on the simulator's graph.

Assume $\mathbf{Bad}_Q^{(j)}$ does not occur in the Q-th query and suppose there is a path on the simulator's graph contradicting (10). In other words, there is a path:

$$0^r \| H \xrightarrow{\texttt{SAFECorePad}(M)} Y_1 \to \cdots \to Y_{\ell-1} \to Y_\ell \,,$$

where necessarily

$$\texttt{left}_r(Y_1) \| \cdots \| \texttt{left}_r(Y_{\ell-1}) = \texttt{RO}(IO, D, \texttt{SAFECorePad}(M))[1 : r \cdot (\ell - 1)]$$

but

$$\texttt{left}_r(Y_l) \neq \texttt{RO}(IO, D, \texttt{SAFECorePad}(M))[r \cdot (\ell - 1) + 1 : r \cdot \ell] \,.$$

By the construction of the simulator, we know there must be another path from $0^r \| H$ to Y_ℓ satisfying (10). This implies that in the simulator's graph there is a node with two out-going (or two in-going) edges, in which case $\mathbf{CollP}_Q$ must have occurred, there is a rooted node with an in-going edge, in which case $\mathbf{ConnectPH}_Q$ must have occurred, there is a cycle, in which case $\mathbf{ConnectP}_Q$ must have occurred, or the selection of (IO, D) was ambiguous in the first place, in which case $\mathbf{CollH}_Q$ must have occurred. Since by hypothesis, neither of those occurred, we conclude that the result holds. □

From Lemma 2, we can conclude that W_I and W_S are identical, i.e. their outputs are identically distributed, as long as **Bad** does not happen in either world. More formally, by the fundamental lemma of game playing [7] (or by [37]) we have:

$$\mathbf{Pr}[\mathtt{D}^{W_I} \Rightarrow 1 \mid \neg\mathbf{Bad}^{(1)}] = \mathbf{Pr}[\mathtt{D}^{W_S} \Rightarrow 1 \mid \neg\mathbf{Bad}^{(2)}]\,.$$

Similar to Naito and Ohta [32, Section 3.4], we obtain from (8) of Lemma 1:[6]

$$(5) \leq \mathbf{Pr}[\mathbf{Bad}^{(1)}] + \mathbf{Pr}[\mathbf{Bad}^{(2)}] \leq \frac{2 \cdot \binom{Q_\mathtt{H}}{2} + 2 \cdot \binom{Q_\mathtt{P}}{2} + 4 \cdot Q_\mathtt{P} \cdot Q_\mathtt{H}}{|\mathbb{F}_p|^c} + \frac{2 \cdot \binom{Q_\mathtt{P}}{2}}{|\mathbb{F}_p|^b}\,. \tag{12}$$

4.5 Bound of (6)

The intermediate world W_S and the real world W_R (see Figure 3) are identical, except for the fact that $\mathtt{P}/\mathtt{P}^{-1}$ is a permutation whereas $\mathtt{S}_\mathtt{P}/\mathtt{S}_\mathtt{P}^{-1}$ is a random function. First note that $\mathtt{S}_\mathtt{P}$ queries its oracle on input of a tuple $(IO, D, \mathtt{SAFECorePad}(M))$, which is always distinct for each evaluation. Thus, the outputs of $\mathtt{S}_\mathtt{P}/\mathtt{S}_\mathtt{P}^{-1}$ are always uniformly randomly drawn. In the real world, it may happen that $\mathtt{P}$ is evaluated twice for the same value for a different construction evaluation, while this would not happen in the intermediate world. However, this would only happen in case of event $\mathbf{CollH}^{(3)}$. Assuming that this never happens, the two oracles $\mathtt{P}/\mathtt{P}^{-1}$ and $\mathtt{S}_\mathtt{P}/\mathtt{S}_\mathtt{P}^{-1}$ are identical as long as the latter does not output colliding values, which would in turn trigger event $\mathbf{CollP}^{(2)}$. From (7) and (9) of Lemma 1:

$$(6) \leq \mathbf{Pr}[\mathbf{CollH}^{(3)}] + \mathbf{Pr}[\mathbf{CollP}^{(2)}] \leq \frac{\binom{Q_\mathtt{H}}{2}}{|\mathbb{F}_p|^c} + \frac{\binom{Q_\mathtt{P}}{2}}{|\mathbb{F}_p|^b}\,.$$

5 SAFE API

The SAFE API [5] considers a sponge with a state of $b = r + c$ field elements in $\mathbb{F}_p^b$, where r is the rate and c the capacity. The sponge operates on a permutation $\mathtt{P} : \mathbb{F}_p^b \to \mathbb{F}_p^b$. In addition, a hash function $\mathtt{H} : (\mathbb{N}_+)^* \times \{0,1\}^* \to \mathbb{F}_p^c$ is involved upon initialization. A sponge object exposes four operations:

- `START`. This operation officially marks the start of a sponge life. It receives as input an IO pattern, IO, and a domain separator D. The input, IO, prescribes exactly the sequence of future calls and their respective lengths in the form of a string of 32-bit words (the exact encoding is slightly different from that of Section 3, but the difference is irrelevant for the current discussion), and D is an arbitrary domain separator which could for instance be used to distinguish between different use cases. It feeds IO and D into the hash function to obtain a c-element tag $T = \mathtt{H}(IO, D)$. This tag is then used to initialize the inner part of the state.

[6] In their work, Naito and Ohta omitted a factor 2, which is included here. Our bound can also be derived from [37, Lemma 1].

– `ABSORB`. It receives as input a length L and an array $X[L]$ of L field elements, and absorbs them r elements at a time, interleaved with a call of `P`. The function also checks if the input matches the IO pattern.
– `SQUEEZE`. It receives as input a length L stating the requested number of blocks, and squeezes them r elements at a time, interleaved with a call of `P`. The function also checks if the input matches the IO pattern.
– `FINISH`. This operation officially marks the end of a sponge life. It receives no input and outputs 'OK' or 'NOK', depending on whether the sponge evaluation was correctly executed.

It is important to note that the functions `ABSORB` and `SQUEEZE` can be evaluated element-wise, and they only evaluate the permutation once they exhausted the entire outer part, i.e. once they absorbed/squeezed r elements. In addition, a transition from `ABSORB` to `SQUEEZE` is always made through a permutation evaluation, even if they did not exhaust the outer part. The other way around, this is not the case: one can e.g. squeeze r elements and then absorb r elements before the next permutation call is made. Details on this, and how it is implemented, can be found in [5].

Example 1. We will explain how the example of Section 3.2 would appear in the SAFE API, with parameters $c = 2, r = 2$. We have an IO pattern $IO = (8, 6, 5, 3, 4, 7)$, and any domain separator D. Let $M = M[1 : 17]$ be any input of the correct length. We describe two different ways to process this IO pattern, domain separator, and a message using the SAFE API in Algorithms 5 and 6. The two evaluations are, in fact, equivalent. For example, in Algorithm 5, line 3 incurs 4 evaluations of `P` (recall that $r = 2$), whereas in Algorithm 6, line 3 incurs 2 evaluations of `P` and line 4 incurs 2 evaluations of `P`. The two evaluations in Algorithms 5 and 6 succeed upon finishing; if there were a mismatch between the number of absorbed/squeezed elements and what was prescribed by the IO pattern, finish would fail.

Algorithm 5 Example evaluation of SAFE API

1: $Z = \emptyset$
2: `START`(IO, D)
3: `ABSORB`$(8, M[1 : 8])$
4: $Z \leftarrow Z \| $`SQUEEZE`$(6)$
5: `ABSORB`$(5, M[9 : 13])$
6: $Z \leftarrow Z \| $`SQUEEZE`$(3)$
7: `ABSORB`$(4, M[14 : 17])$
8: $Z \leftarrow Z \| $`SQUEEZE`$(7)$
9: **return** `FINISH`() ? Z : $\perp$

Algorithm 6 Example evaluation of SAFE API

1: $Z = \emptyset$
2: `START`(IO, D)
3: `ABSORB`$(5, M[1 : 5])$
4: `ABSORB`$(3, M[6 : 8])$
5: $Z \leftarrow Z \| $`SQUEEZE`$(3)$
6: $Z \leftarrow Z \| $`SQUEEZE`$(3)$
7: `ABSORB`$(4, M[9 : 12])$
8: `ABSORB`$(1, M[13])$
9: $Z \leftarrow Z \| $`SQUEEZE`$(3)$
10: `ABSORB`$(4, M[14 : 17])$
11: $Z \leftarrow Z \| $`SQUEEZE`$(3)$
12: $Z \leftarrow Z \| $`SQUEEZE`$(4)$
13: **return** `FINISH`() ? Z : $\perp$

By definition, these evaluations of the SAFE operations are covered almost exactly by `SAFECore`, with the crucial difference that SAFE for efficiency and implementation reasons allows element-wise data processing whereas in `SAFECore` all inputs are basically absorbed at once before the first squeezing starts. It turns out that this does not restrict the generality of `SAFECore`, and in particular, we can argue security of any use case of the SAFE API. For example, for Example 1, we have

$$Z \leftarrow \texttt{SAFECore}(IO, D, M[1:8]) \tag{13a}$$
$$\| \texttt{SAFECore}(IO, D, M[1:13]) \tag{13b}$$
$$\| \texttt{SAFECore}(IO, D, M[1:17])\,. \tag{13c}$$

Note that in `SAFECore`, the function `SAFECorePad` assures proper padding of M to account for squeezing rounds in (13b) and (13c). Because in Theorem 2 we proved that `SAFECore` is indifferentiable from a random oracle up to bound (4), we can obtain that the output string (13) is indistinguishable from random, provided $Q_{\mathtt{H}}, Q_{\mathtt{P}} \ll |\mathbb{F}_p|^{c/2}$.

This result can be straightforwardly generalized to the observation that all outputs of an evaluation of the SAFE API are indistinguishable from random, except in case two evaluations have a common prefix. To understand this, let us first consider the example case above, where we query the SAFE API on input of $IO = (8, 6, 5, 3, 4, 7)$, any domain separator D, and on two different messages $M = M[1:17]$ and $M' = M'[1:17]$ satisfying that $M[1:8] = M'[1:8]$. Then, in the evaluation of the SAFE API in Algorithm 5 or 6, the first 6 squeezed elements will be equal in the two evaluations, the remaining 10 elements may be either equal or independently distributed depending on the values $M[9:17]$ and $M'[9:17]$. This can in fact also be concluded from (13).

More formally, we say that two tuples (IO, D, M) and (IO', D', M') have a common prefix of k phases if

$$(IO, D, M[1 : I_1 + I_2 + \cdots + I_k]) = (IO', D', M'[1 : I_1 + I_2 + \cdots + I_k])$$

but

$$M[I_1 + I_2 + \cdots + I_k + 1 : I_1 + I_2 + \cdots + I_{k+1}] \neq M'[I_1 + I_2 + \cdots + I_k + 1 : I_1 + I_2 + \cdots + I_{k+1}]\,.$$

Then, in the SAFE API, the first $O_1 + O_2 + \cdots + O_k$ squeezed elements will be identical but the future squeezes will be mutually independent. Obviously, common digests for common prefixes is not a bug, but rather a feature that is also present in duplex constructions [9, 10, 18, 19, 31]. By using different IO patterns $IO \neq IO'$, different domain separators $D \neq D'$, or a nonce that initializes M, the problem is avoided all the way.

We can conclude the following for the SAFE API.

Corollary 1 (Security of SAFE API). *Under the assumption that* `H` *is a random oracle and* `P` *a random permutation, and as long as the total number of*

primitive evaluations $Q_{\mathtt{H}}, Q_{\mathtt{P}}$ are less than $|\mathbb{F}_p|^{c/2}$, outputs of SAFE are indistinguishable from random up to common prefix.

This corollary, in turn, has immediate consequences for many practical use cases of the SAFE API. In the remainder of this section, we discuss various examples in more detail. In each of these applications, $\mu \in \mathbb{N}$ is the number of field elements that correspond to 256 bits (or a little less), and we take $c \geq \mu$.

5.1 Fixed-Length Hashing

In order to hash an array of $\ell \in \mathbb{N}$ field elements $M = M[1:\ell] \in \mathbb{F}_p^\ell$ and obtain a digest of μ elements, one can evaluate the SAFE operations as follows. First, we fix IO pattern $IO = (\ell, \mu)$ and arbitrary domain separator D. Then, the hash digest is generated as follows:

1: $\mathtt{START}(IO, D)$
2: $\mathtt{ABSORB}(\ell, M[1:\ell])$
3: $Z \leftarrow \mathtt{SQUEEZE}(\mu)$
4: **return** $\mathtt{FINISH}()\ ?\ Z\ :\ \perp$

By definition, this is exactly the same as evaluating `SAFECore`:

$$Z \leftarrow \mathtt{SAFECore}(IO, D, M)\,, \tag{14}$$

where M is restricted to match the IO pattern IO and the length of Z is prescribed by IO as well. Note that, just like in the comparison of Algorithms 5 and 6, for hashing the consumer is allowed to absorb and squeeze element-wise, but it does not matter much. We obtain the following corollary.

Corollary 2. *Under the assumption that* `H` *is a random oracle and* `P` *a random permutation, above fixed-length hashing construction outputs Z that is indistinguishable from random as long as the total number of* `START` *calls and the total number of permutation calls do not exceed $|\mathbb{F}_p|^{c/2}$. In particular, for $c \geq \mu = \log_p 2^{256-\epsilon}$ the fixed-length hashing construction is preimage resistant against an adversary that makes at most $\min\{|\mathbb{F}_p|^{c/2}, |\mathbb{F}_p|^{\mu}\}$ queries and collision resistant against an adversary that makes at most $\min\{|\mathbb{F}_p|^{c/2}, |\mathbb{F}_p|^{\mu/2}\}$ queries implying security up to $128 - \epsilon/2$ bits.*

Merkle tree hashing is a subclass of this scenario.

5.2 Commitment Schemes

In order to commit to ℓ d-tuples of field elements $X_1, X_2, \ldots, X_\ell \in \mathbb{F}_p^d$ and randomness $R \in \mathbb{F}_p$ and obtain a digest of μ elements, one can evaluate the SAFE operations as follows. First, we fix IO pattern $IO = (\ell \cdot d + 1, \mu)$ and arbitrary domain separator D. Then, the commitment is generated as follows:

1: $\mathtt{START}(IO, D)$
2: $\mathtt{ABSORB}(\ell \cdot d + 1, X_1, ||X_2|| \ldots ||X_\ell||R)$
3: $Z \leftarrow \mathtt{SQUEEZE}(\mu)$

4: **return** `FINISH`() ? Z : $\perp$

By definition, this is exactly the same as evaluating `SAFECore`:

$$Z \leftarrow \texttt{SAFECore}(IO, D, X_1 \| X_2 \| \cdots \| X_\ell \| R)\,, \tag{15}$$

just like for the example of Section 5.1. In fact, the application is merely identical, but the security model is different. Here, we do not aim for collision or preimage resistance as in Corollary 2, but rather to binding and hiding. Moreover, as our adversary can freely choose IO and D, our security results applies not to a single invocation of a commitment scheme but also to protocols where *several commitment schemes are used in parallel*.

Corollary 3. *Under the assumption that* `H` *is a random oracle and* `P` *a random permutation, above commitment scheme construction outputs Z that is indistinguishable from random as long as the total number of* `START` *calls and the total number of permutation calls do not exceed $|\mathbb{F}_p|^{c/2}$. In particular, for $c \geq \mu = \log_{|\mathbb{F}|} 2^{256-\epsilon}$ the commitment scheme construction is computationally binding and hiding against an adversary that makes at most $\min\{|\mathbb{F}_p|^{c/2}, |\mathbb{F}_p|^{\mu}\}$ queries to* `H` *and* `P`*, implying security up to $128 - \epsilon/2$ bits.*

Note that the IO pattern will be the same for committing $\ell \cdot d$ 1-field elements. If this difference matters for an application, a domain separator should be used.

5.3 Multi-Round Interactive Protocols

A non-interactive argument of knowledge is often based on a multi-round interactive protocol, where a verifier is replaced by a hash function within the Fiat-Shamir paradigm. SAFE is suitable for implementing such a hash with minimum overhead. As an example, consider a 5-round protocol. Let $n \in \mathbb{N}$ be the length of the common input, and let $\lambda_1, \lambda_2, \lambda_3 \in \mathbb{N}$ be the lengths of proof elements:

- Prover and verifier agree on the common input $N \in \mathbb{F}_p^n$;
- Prover prepares and sends proof elements $\pi_1 \in \mathbb{F}_p^{\lambda_1}$ and $\pi_2 \in \mathbb{F}_p^{\lambda_2}$;
- Verifier responds with challenge $C_1 \in \mathbb{F}_p^{\mu}$;
- Prover prepares and sends proof element $\pi_3 \in \mathbb{F}_p^{\lambda_3}$;
- Verifier responds with challenges $C_2, C_3 \in \mathbb{F}_p^{\mu}$;
- Prover sends final proof π_4.

Here the prover sends a proof of knowledge in three steps while getting verifier's challenges in-between. To make the protocol non-interactive we apply the Fiat-Shamir transformation where the challenges are generated as follows. First, we fix IO pattern $IO = (n + \lambda_1 + \lambda_2, \mu, \lambda_3, 2\mu)$ and arbitrary domain separator D. Then, the challenges are generated as follows:

1: `START`(IO, D)
2: `ABSORB`$(n + \lambda_1 + \lambda_2, N \| \pi_1 \| \pi_2)$
3: $C_1 \leftarrow$ `SQUEEZE`(μ)

4: $\texttt{ABSORB}(\lambda_3, \pi_3)$
5: $C_2 \leftarrow \texttt{SQUEEZE}(\mu)$
6: $C_3 \leftarrow \texttt{SQUEEZE}(\mu)$
7: **return** $\texttt{FINISH}()\ ?\ (C_1, C_2, C_3)\ :\ \perp$

By definition, this is exactly the same as evaluating `SAFECore`:

$$C_1 \leftarrow \texttt{SAFECore}(IO, D, N \| \pi_1 \| \pi_2)\,, \tag{16a}$$
$$C_2 \| C_3 \leftarrow \texttt{SAFECore}(IO, D, N \| \pi_1 \| \pi_2 \| \pi_3)\,. \tag{16b}$$

We obtain, by security of `SAFECore`, that this non-interactive version of the protocol is as secure as the interactive one up to the security of `SAFECore`. We note that as our adversary is powerful enough to choose arbitrary IO and D, the security holds when *several such protocols co-exist in one application*, whether in parallel or recursively.

Corollary 4. *Suppose the multi-round interactive protocol construction is computationally sound against an adversary that makes up to 2^t calls to* `H` *and* `P` *assuming* `H` *is a random oracle and* `P` *a random permutation. Then the non-interactive protocol (above) outputs (C_1, C_2, C_3) that are indistinguishable from random as long as the total number of* `START` *calls and the total number of permutation calls do not exceed $|\mathbb{F}_p|^{c/2}$. In particular, for $c \geq \mu = \log_{|\mathbb{F}|} 2^{256-\epsilon}$ the non-interactive protocol construction is sound against an adversary that makes at most $\min\{2^t, |\mathbb{F}_p|^{c/2}, |\mathbb{F}_p|^{\mu}\}$ queries, implying security up to $128 - \epsilon/2$ bits.*

5.4 Authenticated Encryption

SAFE allows to perform authenticated encryption using the SpongeWrap mode [10], with subtle differences that the 1-padding (present in the original SpongeWrap) can be avoided by using the IO pattern de facto as prefix. Let k be the key length, n the nonce length, and t the tag length. In order to encrypt and authenticate ℓ blocks of data $M_1, M_2, \ldots, M_\ell$ each of length λ_i with key $K \in \mathbb{F}_p^k$ and nonce $N \in \mathbb{F}_p^n$ in order to obtain ciphertext blocks $C_1, C_2, \ldots, C_\ell$ and tag $T \in \mathbb{F}_p^\mu$, we proceed as follows. First, we fix IO pattern

$$IO = (k + n, \lambda_1, \lambda_1, \lambda_2, \lambda_2, \ldots, \lambda_\ell, \lambda_\ell, \mu)$$

and an arbitrary domain separator D. Then, the message is encrypted and authenticated as follows:

1: $\texttt{START}(IO, D)$
2: $\texttt{ABSORB}(k + n, K \| N)$
3: $Z_1 \leftarrow \texttt{SQUEEZE}(\lambda_1)$
4: $\texttt{ABSORB}(\lambda_1, M_1)$
5: $Z_2 \leftarrow \texttt{SQUEEZE}(\lambda_2)$
6: $\texttt{ABSORB}(\lambda_2, M_2)$
7: $\cdots$
8: $Z_\ell \leftarrow \texttt{SQUEEZE}(\lambda_\ell)$

9: $\texttt{ABSORB}(\lambda_\ell, M_\ell)$
10: $T \leftarrow \texttt{SQUEEZE}(\mu)$
11: $(C_1, C_2, \ldots, C_\ell) \leftarrow (Z_1 + M_1, Z_2 + M_2, \ldots, Z_\ell + M_\ell)$
12: **return** $\texttt{FINISH}()$? $(C_1, \ldots, C_\ell, T)$: $\perp$

By definition, this is exactly the same as evaluating `SAFECore`:

$$Z_1 \leftarrow \texttt{SAFECore}(IO, D, K\|N)\,, \tag{17a}$$
$$Z_2 \leftarrow \texttt{SAFECore}(IO, D, K\|N\|M_1)\,, \tag{17b}$$
$$\vdots$$
$$Z_\ell \leftarrow \texttt{SAFECore}(IO, D, K\|N\|M_1\|\cdots\|M_{\ell-1})\,, \tag{17c}$$
$$T \leftarrow \texttt{SAFECore}(IO, D, K\|N\|M_1\|\cdots\|M_\ell)\,, \tag{17d}$$

with the final output being $(Z_1 + M_1, Z_2 + M_2, \ldots, Z_\ell + M_\ell, T)$. We obtain, by security of `SAFECore`, that this authenticated encryption scheme is secure.

Corollary 5. *Under the assumption that* `H` *is a random oracle and* `P` *a random permutation, above authenticated encryption construction outputs Z that is indistinguishable from random as long as the total number of* `START` *calls and the total number of permutation calls do not exceed $|\mathbb{F}_p|^{c/2}$. In particular, for $c \geq \mu = \log_{|\mathbb{F}|} 2^{256-\epsilon}$ the authenticated encryption construction offers confidentiality and authenticity against an adversary that makes at most $\min\{|\mathbb{F}_p|^{c/2}, |\mathbb{F}_p|^\mu\}$ queries, implying security up to $128 - \epsilon/2$ bits.*

This construction is the most efficient when $\lambda_i \equiv 0 \mod r$, that is, all blocks fit the rate parameter of the sponge. This mode can be adapted to support associated data (authenticated but not encrypted), in the same vein as the SpongeWrap mode. Note that there is no padding overhead, nor we spend unneeded calls to the inner permutation.

6 Conclusion

We have formally proven the security of the SAFE API with applications to many use cases, from hashing to interactive protocols. A number of typical applications have been highlighted in Sections 5.1–5.4, but extensions to protocol composition, variable-length hashing, PRNGs, and support of multiple fields are possible. The most important observation is that it is possible to get rid of the padding schemes at the (arguably smaller) cost of pre-declaring the pattern of absorptions and squeezes. As the majority of applications of the SAFE API know this pattern in advance, we have placed no significant burden on the designers. Our results, perhaps surprisingly, demonstrated that SAFE API is better than it was originally considered. In particular, our results demonstrate that the full inner part of the sponge can be used to hash the IO pattern onto, without any security loss. This principle can be used in the future applications of sponges, which may put all the application/run metadata (properly processed) into the capacity, and then run the sponge in a simple but flexible and foolproof way.

Acknowledgements. Mario Marhuenda Beltrán and Bart Mennink are supported by the Netherlands Organisation for Scientific Research (NWO) under grant VI.Vidi.203.099.

References

1. Longsight faulty design (2018), https://github.com/zcash/zcash/issues/2233#issuecomment-416648993
2. Tornado Cash Privacy Solution Version 1.4 (2021), https://tornado.cash/Tornado.cash_whitepaper_v1.4.pdf
3. Albrecht, M.R., Grassi, L., Rechberger, C., Roy, A., Tiessen, T.: MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10031, pp. 191–219 (2016), https://doi.org/10.1007/978-3-662-53887-6_7
4. Aly, A., Ashur, T., Ben-Sasson, E., Dhooghe, S., Szepieniec, A.: Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols. IACR Trans. Symmetric Cryptol. 2020(3), 1–45 (2020), https://doi.org/10.13154/tosc.v2020.i3.1-45
5. Aumasson, J., Khovratovich, D., Quine, P.: SAFE (Sponge API for Field Elements) – A Toolbox for ZK Hash Applications (2022), https://safe-hash.dev/
6. Bellare, M., Rogaway, P.: Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In: Denning, D.E., Pyle, R., Ganesan, R., Sandhu, R.S., Ashby, V. (eds.) CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993. pp. 62–73. ACM (1993), https://doi.org/10.1145/168588.168596
7. Bellare, M., Rogaway, P.: Code-Based Game-Playing Proofs and the Security of Triple Encryption. Cryptology ePrint Archive, Paper 2004/331 (2004), https://eprint.iacr.org/2004/331, https://eprint.iacr.org/2004/331
8. Bernhard, D., Pereira, O., Warinschi, B.: How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In: Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18. pp. 626–643. Springer (2012)
9. Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R.: Farfalle: parallel permutation-based cryptography. IACR Trans. Symmetric Cryptol. 2017(4), 1–38 (2017), https://tosc.iacr.org/index.php/ToSC/article/view/801
10. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Duplexing the sponge: Single-pass authenticated encryption and other applications. In: Miri, A., Vaudenay, S. (eds.) Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7118, pp. 320–337. Springer (2011), https://doi.org/10.1007/978-3-642-28496-0_19
11. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak. In: International Conference on the Theory and Application of Cryptographic Techniques (2013)
12. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Sponge Functions (2007)

13. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Smart, N.P. (ed.) Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4965, pp. 181–197. Springer (2008), https://doi.org/10.1007/978-3-540-78967-3_11
14. Bowe, S.: BLS12-381: New zk-SNARK elliptic curve construction (2017), https://electriccoin.co/blog/new-snark-curve
15. Chiesa, A., Ojha, D., Spooner, N.: Fractal: Post-quantum and transparent recursive proofs from holography. In: Canteaut, A., Ishai, Y. (eds.) Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12105, pp. 769–793. Springer (2020), https://doi.org/10.1007/978-3-030-45721-1_27
16. Coron, J., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård Revisited: How to Construct a Hash Function. In: Shoup, V. (ed.) Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3621, pp. 430–448. Springer (2005), https://doi.org/10.1007/11535218_26
17. Cortier, V., Gaudry, P., Yang, Q.: How to fake zero-knowledge proofs, again. In: E-Vote-Id 2020-The International Conference for Electronic Voting (2020), available at https://hal.inria.fr/hal-02928953/document
18. Daemen, J., Mennink, B., Van Assche, G.: Full-State Keyed Duplex with Built-In Multi-user Support. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10625, pp. 606–637. Springer (2017), https://doi.org/10.1007/978-3-319-70697-9_21
19. Dobraunig, C., Mennink, B.: Leakage Resilience of the Duplex Construction. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11923, pp. 225–255. Springer (2019), https://doi.org/10.1007/978-3-030-34618-8_8
20. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Crypto. vol. 86, pp. 186–194. Springer (1986)
21. Grassi, L., Khovratovich, D., Lüftenegger, R., Rechberger, C., Schofnegger, M., Walch, R.: Reinforced Concrete: A Fast Hash Function for Verifiable Computation. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 1323–1335. CCS '22, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3548606.3560686
22. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: Bailey, M., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 519–535. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/grassi
23. Grassi, L., Mennink, B.: Security of Truncated Permutation Without Initial Value. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II.

Lecture Notes in Computer Science, vol. 13792, pp. 620–650. Springer (2022), https://doi.org/10.1007/978-3-031-22966-4_21
24. Haines, T., Lewis, S.J., Pereira, O., Teague, V.: How not to prove your election outcome. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 644–660. IEEE (2020), https://doi.org/10.1109/SP40000.2020.00048
25. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: ZCash protocol specification (2023), https://github.com/zcash/zips/blob/master/protocol/protocol.pdf
26. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 13510, pp. 359–388. Springer (2022), https://doi.org/10.1007/978-3-031-15985-5_13
27. Lefevre, C., Mennink, B.: Tight Preimage Resistance of the Sponge Construction. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 13510, pp. 185–204. Springer (2022), https://doi.org/10.1007/978-3-031-15985-5_7
28. Maller, M., Khovratovich, D.: Baloo: open source implementation (2022), https://github.com/mmaller/caulk-dev/tree/main/baloo
29. Maurer, U.M., Renner, R., Holenstein, C.: Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In: Naor, M. (ed.) Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2951, pp. 21–39. Springer (2004), https://doi.org/10.1007/978-3-540-24638-1_2
30. Mennink, B.: Understanding the Duplex and Its Security. IACR Cryptol. ePrint Arch. p. 1340 (2022), https://eprint.iacr.org/2022/1340
31. Mennink, B., Reyhanitabar, R., Vizár, D.: Security of Full-State Keyed Sponge and Duplex: Applications to Authenticated Encryption. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9453, pp. 465–489. Springer (2015), https://doi.org/10.1007/978-3-662-48800-3_19
32. Naito, Y., Ohta, K.: Improved Indifferentiable Security Analysis of PHOTON. In: Abdalla, M., Prisco, R.D. (eds.) Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8642, pp. 340–357. Springer (2014), https://doi.org/10.1007/978-3-319-10879-7_20
33. NIST: SHA-3 Competition. In: International Conference on the Theory and Application of Cryptographic Techniques (2007-2012)
34. Polygon Team: Introducing Plonky2 (2017), https://polygon.technology/blog/introducing-plonky2
35. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU. Submission to the NIST's post-quantum cryptography standardization process 36(5) (2018)

36. Setty, S.: Nova: open source implementation
37. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Paper 2004/332 (2004), https://eprint.iacr.org/2004/332, https://eprint.iacr.org/2004/332
38. Zhang, Y.: Introducing zkEVM (2022), https://scroll.io/blog/zkEVM