Generalized Indifferentiable Sponge and its Application to Polygon Miden VM

Tomer Ashur  
3MI Labs, Polygon Research  
Leuven, Belgium  
tomer@3milabs.tech

Amit Singh Bhati  
COSIC, KU Leuven  
Leuven, Belgium  
amitsingh.bhati@esat.kuleuven.be

ABSTRACT
Cryptographic hash functions are said to be the work-horses of modern cryptography. One of the strongest approaches to assess a cryptographic hash function’s security is indifferentiability. Formally, indifferentiability measures to what degree the function resembles a random oracle when instantiated with an ideal underlying primitive. However, proving the indifferentiability security of hash functions has been challenging due to complex simulator designs and proof arguments. The Sponge construction is one of the prevalent hashing method used in various systems. The Sponge has been shown to be indifferentiable from a random oracle when initialized with a random permutation.

In this work, we first introduce GSponge, a generalized form of the Sponge construction offering enhanced flexibility in input chaining, field sizes, and padding types. GSponge not only captures all existing sponge variants but also unveils new, efficient ones. The generic structure of GSponge facilitates the discovery of two micro-optimizations for already deployed sponges. Firstly, it allows a new padding rule based on zero-padding and domain-separated inputs, saving one full permutation call in certain cases without increasing the generation time of zero-knowledge proofs. Secondly, it allows to absorb up to $c/2$ more elements (that can save another permutation call for certain message lengths) without compromising the indifferentiability security. These optimizations enhance hashing time for practical use cases such as Merkle-tree hashing and short message processing.

We then propose a new efficient instantiation of GSponge called Sponge2 capturing these micro-optimizations and provide a formal indifferentiability proof to establish both Sponge2 and GSponge’s security. This proof, simpler than the original for Sponges, offers clarity and ease of understanding for real-world practitioners. Additionally, it is demonstrated that GSponge can be safely instantiated with permutations defined over large prime fields, a result not previously formally proven.

KEYWORDS
Sponge, GSponge, Sponge2, Hashing Mode, Algebraic, Miden VM, Indifferentiability, Random Oracle, Generic Attacks

1 INTRODUCTION
Cryptographic hash functions play a fundamental role in modern cryptography. Hash functions are used for converting input data of varying lengths into fixed-size outputs, ensuring data integrity, authenticating transactions, and facilitating digital signatures. These cryptographic primitives are integral to a wide range of applications, including but not limited to data storage, authentication protocols, and secure communication channels. In recent decades, hash functions have also found specialized applications in algebraic hashing, a field essential for Blockchain scaling via L2 rollups (e.g., Polygon Miden).

Ideal primitives commonly used in hash function construction are random functions, random permutations, or ideal ciphers. In practice, such primitives are expected to generate pseudorandom outputs for given inputs, ensuring unpredictability and resistance against cryptographic attacks. By combining these ideal primitives within specific modes, hash functions can achieve desired security properties such as collision resistance, preimage resistance, and second preimage resistance. The selection of appropriate primitives is crucial in designing hash functions that meet the stringent security requirements of modern cryptographic applications.

Hashing Mode Strategies. The choice of hashing mode is as critical as the selection of primitives, impacting both the security and efficiency of the resulting hash function. In cryptographic literature, the two mainly used hashing strategies are tree hashing (a.k.a. parallel hashing) and cascade hashing (a.k.a. sequential hashing).

Tree hashing techniques, exemplified by Merkle trees [5] and ABR trees [1], involve the parallel computation of hash values across multiple branches or levels of a hash tree. These methods offer efficient verification and authentication of large datasets, making them suitable for applications requiring scalable and parallel processing; however, at a cost of larger (length dependent) state to be maintained.

In contrast, cascade hashing strategies, such as Merkle-Damgård [12, 24] (MD) style hashes and sponge [7] based hashes, process input data sequentially, iteratively updating a relatively small internal state to produce the final hash value. MD-style hashes such as Wide Pipe [22], Fast Wide Pipe [28], Chop-MD [11] and p-MD[11] by design rely of random functions as primitives, whereas sponge-based hashes such as Sponge [7], Overwrite Sponge [14] and JH [30] can incorporate both random functions and permutations. Due to their versatility and wide applicability, the sponge-based approach has garnered huge attention in recent years. This approach, notably employed in SHA-3 and algebraic hashing systems such as Polygon Miden [25], offers inherent flexibility, particularly in accommodating algebraic primitives as most of them are permutations.

Sponge Hashing. At a fundamental level, Sponge-based constructions are characterized by a state of $b$ bits, comprising a $c$-bit inner state, known as the capacity, and an $r$-bit outer state, referred to as the rate, where the total state size $b$ equals the sum of $c$ and $r$. Traditionally, in Sponge-like modes, data absorbing operations occur via the rate part, processing $r$ bits at a time.
A notable sponge variant documented in the literature is the overwrite sponge mode [14]. This variant improves the efficiency of regular sponge by dropping the top $r$ bits of each permutation output instead of cascading them, resulting in $r$-bit smaller state among the permutation calls and removal of all XOR operations during absorption. Originally, (overwrite) sponge was proposed for primitives operating on bits, i.e., binary fields. However, recent works such as Rescue [29], Poseidon [15] and XHash [4] highlights that the applicability of sponge construction can be extended to prime field setting when algebraic primitives are preferred or required. The rate and capacity of sponge under algebraic primitive is measured in field elements.

**Handling Arbitrary Lengths.** By design, sponge variants can only handle messages that has lengths multiple of the rate size. Hence, to handle messages of arbitrary lengths, a sponge-compliant padding rule is used to preprocess the message into a rate-aligned message, i.e., a message whose length is a multiple of the rate size. One of the simple and popular padding rules is the pad10+ padding [7]. This padding simply appends the input with a 1 followed by the minimum number of 0s such that the length of the result is a multiple of the rate. We note that despite the simple definition, this padding adds another $r$-bit pad to even already rate-aligned messages, and hence imposes an extra primitive call to process them. Such drawback of a padding can affect the performance significantly in applications where reducing even one permutation call is highly beneficial such as applications that process small size messages.

**Security Analysis.** Security analysis of hash functions involves a two-step approach: Firstly, demonstrating the generic security of the hashing mode, initialized with an ideal primitive, by proving desired security properties such as collision resistance or indistinguishability. Subsequently, instantiating the ideal primitive with a concrete function that goes through multiple cryptanalysis for validation.

**Generic Attacks.** In a generic attack, adversaries exploit vulnerabilities in a crypto-algorithm by assuming perfect behavior of its underlying primitives. For instance, consider a hash function $H : \mathbb{F}_p^n \rightarrow \mathbb{F}_p^n$ utilizing a random permutation $P : \mathbb{F}_p^n \rightarrow \mathbb{F}_p^n$ for some positive integers $m$ and $n$. This creates an opportunity for generic attacks, which exploit weaknesses in $H$ with fewer resources compared to a larger random oracle $RO : \mathbb{F}_p^n \rightarrow \mathbb{F}_p^n$. Generic attacks on hash functions are widely documented in cryptographic literature. Examples include Joux’s multi-collision attack [18], the Kelsey–Schneier expandable-message second pre-image attack [20], and the Kelsey–Kohno herding attack [19], all targeting the prevalent Merkle–Damgård construction. Additionally, various generic attacks like pre-image, second pre-image, collision, multi-collision, and herding attacks have been identified for numerous other hash functions [2, 9, 13, 16–18], expanding beyond the basic Merkle–Damgård structure. These attacks typically assume the hash function’s fundamental primitive behaves optimally, fulfilling the criteria of generic attacks. Consequently, the prevalence of such attacks significantly influenced the security assessment of cryptographic hash functions, emphasizing the necessity to develop hash modes capable of withstanding such threats.

**Security by Indifferentiability.** The concept of indifferentiability, introduced by Maurer et al. [23] in 2004, and later employed by Coron et al. [11] in 2005, offers a means to assess the resilience of hash modes against generic attacks. This framework measures how closely a hash function resembles a random oracle when instantiated with an underlying primitive behaving in an ideal manner. Many cryptographic protocols employ random oracles and therefore depend on the indifferentiability security of the underlying hash function for engineering purposes. In fact, by contemporary best practices, indifferentiability has become a common prerequisite for hash mode adoption, given its effectiveness in safeguarding against generic attacks.

Despite being a crucial requirement, proving and validating the indifferentiability security of hash functions has not been an easy task. All indifferentiability proofs rely on the concept of a simulator, which is used to simulate the idealized primitive when the hash function is replaced with a true random oracle. Simulators are defined carefully to be both indistinguishable from the ideal primitive on the one hand, and consistent with the outputs of the random oracle on the other. Due to these correlated requirements, the description of simulators in indifferentiability proofs sometime becomes very contrived which makes both the simulator design and the proof arguments less intuitive and hard to follow. See [3, 8, 10, 26] for examples of simulator descriptions, indifferentiability proof approaches and proof sizes.

Further, it is also sometimes hard to directly adapt a given indifferentiability proof of a hashing mode to even minor variations due to the specifically defined simulator definition or dedicated bad case analysis.

### 1.1 Our Contribution
Our contribution in this work is twofold: (1) We generalize sponge mode into GSponge over input chaining type (capturing regular sponge and overwrite), field type (capturing binary and algebraic setting) and padding type (capturing all injective paddings). In comparison with the regular sponge, GSponge provides extra rate of $t_0 < c$ elements for the first permutation call where $c$ denotes its capacity. These elements can also be used for domain separation to directly achieve multi-rate and multi-protocol security.

We show that GSponge provides same security as any efficient instantiation of it. We then propose Sponge2, as an efficient instantiation of GSponge.

(2) We provide an intuitive yet formal proof of indifferentiability for Sponge2. To reduce the design complexity of the simulator and improve the proof intuition, we introduce a new ideal object for sponges called capacity-collision-free random functions. Idealizing primitives as capacity-collision-free random functions naturally makes the indifferentiability proof simple and easily verifiable.

We note that by design, Sponge2 saves one full permutation call in cases where the unpadded message’s length is already an integral multiple of the rate (e.g., in 2-to-1 Merkle-tree hashing) without increasing the proof generation time in zkVMs like Polygon Miden. Further, it also allows to absorb up to $c/2$ more elements in the first permutation call, again resulting in saving a permutation call for
some message lengths, without any loss in the security size, i.e., still $\approx c \log_2 p/2$ bits.

2 RELATED WORK

The Parazoa family, proposed by Andreeva et al. [3], can also be seen as a generalization of the sponge construction. However, unlike GSponge, it is limited to injective padding functions and provides the same rate for all primitive calls. Consequently, it cannot accommodate interesting sponge instantiations like Sponge2, which works with non-injective paddings to save one full permutation call and provides an extra rate for the first primitive call.

Furthermore, the provable security results of Parazoa says nothing about its multi-rate and multi-protocol security. As mentioned earlier, the simulator descriptions and proof approaches of existing indifferentiability proofs are lengthy and difficult to follow, and this complexity is exacerbated by the generalization of Parazoa. For GSponge, we achieve multi-rate and multi-protocol security with a simple and intuitive proof.

Naito et al. improved the indifferentiability result of PHOTON’s hashing mode in [27]. As a side result, they also claimed that the rate of the first permutation call in PHOTON’s sponge-like hashing mode could be improved by half of the capacity size, although they omitted the proof, relying on their main indifferentiability result of PHOTON’s hashing mode. While one may indeed be able to derive this proof for PHOTON’s hashing mode in the binary setting, we have considered it for a generalized sponge that captures various sponge-like modes in both binary and algebraic settings.

3 PAPER ORGANIZATION

We provide the preliminaries in Section 4. We then propose the sponge generalization GSponge and argue its security in Section 5. In Section 6, we provide Sponge2 as an efficient instantiation of GSponge and formally prove its security in Section 7. Finally, we prove a discussion and conclude the paper in Section 8.

4 PRELIMINARIES

4.1 Notation

**Vectors.** We let $\mathbb{F}_p$ to denote a finite field of order $p = 2^k$ with a prime and $b$ a positive integer. A vector $S$ of size $n$ is denoted by $S = (S[0], \ldots, S[n-1])$. We use $\oplus$ to denote addition over the finite field $\mathbb{F}_p$. For simplicity, we refer to a vector of $n > 0$ many $\mathbb{F}_p$ elements as an $(n, p)$-vector. Note that in particular $(n, 2)$-vectors are analogous to $n$-bit binary strings. We denote the set of all $(n, p)$-vectors by $\mathbb{F}_p^n$. The set of vectors of any possible length i.e., arbitrary $n$ is denoted by $\mathbb{F}_p^n$. The set of all permutations of $\mathbb{F}_p^n$ is denoted by $\text{Perm}_p(n)$ and the set of all functions/maps from $\mathbb{F}_p^n$ (respectively, $\mathbb{F}_p^2$) to $\mathbb{F}_p^m$ is denoted by $\text{Func}_p(m, n)$ (respectively, $\text{Func}_p(2, n)$). For any $(n, p)$-vector $A$, $|A|$ is the length of $A$ in elements i.e., $|A| = n$. For any two sets $A$ and $B$, their Cartesian product is defined as $A \times B = \{(i, j) \mid i \in A, j \in B\}$ and the term $A|B$ denotes the largest subset of $A$ that shares no element with $B$.

**Partitions.** Given a vector $A$ and an integer $n > 0$ such that $|A| = an + d$, where $a$ is a positive integer and $0 < d \leq n$, the notation $A_1, A_2, \ldots, A_{a+1} \iff A$ is used to indicate the partitioning of $A$ into a maximum number of $(n, p)$-vectors a.k.a. block vectors. Each block vector $A_i$ has a length of $n$ for $1 \leq i \leq a$, and the last block vector $A_{a+1}$ has a length of $d$. If $d = n$, we say $A$ is $n$-aligned. Similarly, we also use $[A_1, A_2, \ldots, A_{a+1}]$ to denote the ordered union a.k.a. concatenation of these vectors i.e., $A$.

**Miscellaneous.** We use $\langle i \rangle$ to denote a bijective encoding of $0 \leq i < p$ in $\mathbb{F}_p$. The notation $x \leftarrow X$ indicates the random sampling of an element $x$ from a finite set $X$ with a uniform distribution. The symbol $\perp$ is used to represent an undefined value or an error.

4.2 Statistical Distance

The statistical distance between two random variables (or distributions) is defined as follows

**Definition 4.1 (Statistical Distance).** Let $X$ and $Y$ be two random variables taking values from a finite set $X$. The statistical distance between $X$ and $Y$ is defined as

$$SD(X, Y) = \frac{1}{2} \sum_{x \in X} \left| \Pr[X = x] - \Pr[Y = x] \right| .$$

4.3 Indifferentiability

We now recall the main indifferentiability theorem, due to Maurer et al. [23] for hash functions under the random permutation model. If a hash function $H$ based on a public permutation $\mathcal{P}$ (with inverse $\mathcal{P}^{-1}$) is indifferentiable from a random oracle $RO$, then a cryptosystem $C$ based on $RO$ (in the random oracle model) is at least as secure as $C$ based on $H$ (in the random permutation model). This highlights the importance of indifferentiability in ensuring the security of cryptographic protocols that use hash functions to instantiate random oracles.

![Figure 1: The indifferentiability notion](image-url)
from RO is defined as
\[ \text{Adv}_{\text{RO-indiff}}^R(\mathcal{A}) = \min_S |\Pr[\mathcal{A}^R,S \xrightarrow{0,1} | \Pr[\mathcal{A}^R,S \xrightarrow{0,1}]| \]
where \( S: \mathbb{F}_p^m \rightarrow \mathbb{F}_p^m \) is a simulator algorithm simulating \( \mathcal{P} \) for \( \mathcal{R} \).

The simulator \( S \) has oracle access to \( \mathcal{R} \), but it cannot directly observe past queries made to \( \mathcal{R} \) by \( \mathcal{A} \) (refer to Fig. 1 for a visual representation of the indistinguishability concept).

5 GSpunge: A GENERALIZED SPONGE MODE

In this section, we provide a generalization of the popular sponge [7] mode to capture all existing sponge variants such as overwrite sponge, prepadded sponge, domain-separated sponge, etc. We then argue its security by its indistinguishability from a random oracle (RO).

GSpunge \( u \), \( \pi \) : \( \mathbb{F}_p^m \rightarrow \mathbb{F}_p^m \) is based on a permutation \( \pi: \mathbb{F}_p^b \rightarrow \mathbb{F}_p^b \) with size \( b = r+c \) for some positive integers \( r \) and \( c \) called the rate and capacity of \( \pi \), respectively, and an injective padding function \( \text{pad}_{r,0}: \mathbb{F}_p^m \rightarrow \mathbb{F}_p^b \) that for a given \( r_0 < c \) maps any arbitrary length vector to a unique \( ar + r_0 \)-length vector for some \( a \geq 1 \). An input \( M \in \mathbb{F}_p^m \) to GSpunge is first padded to \( P = \text{pad}_{r,0}(M) \) and then processed using \( \pi \) as shown in Fig. 2. Here \( u \in \{0,1\} \) that decides if the rate output of a permutation is chained to the next rate input or not.

![Figure 2: GSpunge (block diagram). Here \( \circ \) represents field multiplication with \( u \in \{0,1\} \) which returns zero or the input itself, depending on the value of \( u \). The upper input (or output) part of \( \pi \) corresponds to its rate whereas the lower input (or output) part corresponds to its capacity.](image)

5.1 Security of GSpunge

We target the security of GSpunge as its indistinguishability from an RO. We highlight that indistinguishability under sufficiently large digest size implicitly provides basic cryptographic hash security properties such as (second) pre-image resistance and collision resistance.

We note that the input space of GSpunge is invariant of \( u \) as every unique input under \( u = 0 \) matches with some unique input under \( u = 1 \). This implies that the outputs of GSpunge are differently ordered a.k.a. permuted for different values of \( u \), however, with the same output multiset and thus same output distribution.

Thus, the indistinguishability of GSpunge is independent of \( u \) and therefore, it is sufficient to argue its indistinguishability with \( u = 0 \), i.e., as overwrite GSpunge mode.

Further, we also note that the indistinguishability of GSpunge \( u, r_0 \) remains same over all choices of the injective paddings. This holds because injective paddings are bijective to each other and when modeled as an RO, the statistical distance between the output distributions of GSpunge \( u \), \( r_0 \) under two different injective paddings is zero. Therefore, the indistinguishability of GSpunge \( u, r_0 \) can be argued by proving the same for any efficient instantiation of it. In the next section, we provide such an efficient instantiation of GSpunge and formally prove its security. We emphasize that this identical output distribution argument only works when GSpunge \( u, r_0 \) (for any injective padding) is shown indistinguishable from an RO.

6 Sponge2: AN EFFICIENT GSpunge INSTANTIATION

In this section, we provide a new sponge variant called Sponge2 as an efficient instantiation of GSpunge. Sponge2 : \( \mathbb{F}_p^m \rightarrow \mathbb{F}_p^m \) is based on a permutation \( \pi: \mathbb{F}_p^b \rightarrow \mathbb{F}_p^b \) with size \( b = r+c \) for some positive integers \( r \) and \( c \) called the rate and capacity of \( \pi \), respectively. An input \( M \in \mathbb{F}_p^m \) to Sponge2 is processed using \( \pi \) as shown in Fig. 3 (b).

**Domain Separation in** Sponge2. For a message \( M \) of length \( 0 < |M| \leq r_0 \), the domain separator is defined as \( i = r + r_0 - |M| \) and otherwise, it is defined as \( i = r - ((|M| - r_0) \mod r) \) \( \mod r \). It is easy to see that \( i \) is a unique value that varies from 0 to \( r + r_0 - 1 \) for messages with different last block lengths and is same as the number of zero elements padded after the message to make it rate-aligned i.e., the number of zeros used in \( (0)^* \) defines \( i \) for every message.

Sponge2 differs from the popular overwrite-style sponge hash in the padding rule and the underlying first primitive’s input as shown in Fig. 3. This difference allows Sponge2 to 1) process \( r_0 < c \) many more \( \mathbb{F}_p \) elements and to 2) additionally reduce the overall cost by one \( \pi \) call than the regular overwrite sponge mode. More specifically, unlike Sponge, no extra \( \pi \) call is required at the end to finalize rate-aligned messages. This gain could be reasonably large for applications that work with stringent resource constraints and/or typically hash small size messages. The targeted application of Miden VM in this work fits to both of these categories which makes Sponge2 a viable choice for it.
We defer the proof of Theorem 6.1 to Section 7. We now define a formal claim about the indifferentiability (from an RO) from the result of Theorem 6.1. More specifically, with $r$ as in Theorem 6.1, for any adversary $\mathcal{A}$ who makes at most $q_p$ RO queries (to both forward and inverse oracles in total), we have

$$\text{Adv}^{\text{ro-indiff}}_{\text{Sponge}2[\pi]}(\mathcal{A}) \leq \frac{q_p^2}{2p^2} + \frac{q_p}{p^{c/2}} \left( 1 + \frac{1}{p-1} \right) \frac{p^2}{p^c} \left( 1 + \frac{1}{p-2} \right) \leq \frac{q_p^2}{p^{c/2}} + \frac{q_p}{p^{c/2}} \left( \frac{3}{2} - \frac{2}{3} \right) \leq 3q_p/p^{c/2}.$$

Finally, we drop the assumption $q_p \leq p^{c/2}/3$ as for $q_p > p^{c/2}/3$, this bound becomes void anyways. This completes the proof of Corollary 6.2.

\section{6.2 Sponge2 under Multi-rate and Multi-protocol Setting}

In real-world applications, the same hash function is sometimes used under the same input space but with different rate sizes or different protocols. In such situations, we need domain separations to ensure the security of the hash function across these use cases.

Sponge2 provides security for multi-rate and multi-protocol applications by simply adding a domain separator in the input of the used rate size or protocol. The security of Sponge2 (in bits) under multi-rate and multi-protocol applications is upper bounded by the security of Sponge2 (in bits) under the maximum allowed rate $r$ as captured in Theorem 6.1.

\section{7 SECURITY ANALYSIS}

In this section, we provide the deferred proof of Theorem 6.1.

\textbf{Proof of Theorem 6.1. \textbf{Replacing $\pi$.}} We treat duplicate queries and cross-oracle known response queries (i.e., querying the $\pi^{-1}$ oracle with an output of previously queried $\pi$ or vice versa) as trivial queries and the rest as non-trivial queries. We note that trivial queries cannot help $\mathcal{A}$ in increasing its advantage as their output is already known and thus independent of the queried oracle. Hence, we can assume w.l.o.g., that $\mathcal{A}$ only makes non-trivial queries.

We now recall that as per the standard RP-RF switching lemma [6], a randomly sampled $(\pi, \pi^{-1})$ pair with $\pi \leftarrow \text{Perm}_p(b)$ is indistinguishable up to the birthday bound (in the output size) from a randomly sampled function pair $(f_1, f_2)$ for non-trivial oracle queries where $(f_1, f_2) \leftrightarrow \text{Func}_p(b, b) \times \text{Func}_p(b, b)$.

More formally, for any adversary $\mathcal{A}$ that makes at most $q_p$ many non-trivial $\pi$ queries (to both forward and inverse oracles in total), we have that

$$\Pr[\mathcal{A}^{\pi, \pi^{-1}} \Rightarrow 1] - \Pr[\mathcal{A}^{f_1, f_2} \Rightarrow 1] \leq \frac{q_p(q_p - 1)}{2p^b}.$$

Let us denote Sponge2[\pi] by Sponge2[\pi, \pi^{-1}]. Then, with the above inequality we get

$$b \geq c, p > 2, q_p \leq p^{c/2}/3$$ and Theorem 6.1, we get

$$\text{Adv}^{\text{ro-indiff}}_{\text{Sponge}2[\pi]}(\mathcal{A}) \leq \frac{q_p^2}{2p^2} + \frac{q_p}{p^{c/2}} \left( 1 + \frac{1}{p-1} \right) \frac{p^2}{p^c} \left( 1 + \frac{1}{p-2} \right) \leq \frac{q_p^2}{p^{c/2}} + \frac{q_p}{p^{c/2}} \left( \frac{3}{2} - \frac{2}{3} \right) \leq 3q_p/p^{c/2}.$$

We defer the proof of Theorem 6.1 to Section 7. We now define a formal claim about the indifferentiability (from an RO) from the result of Theorem 6.1. More specifically, with $r$ as in Theorem 6.1, for any adversary $\mathcal{A}$ who makes at most $q_p$ RO queries (to both forward and inverse oracles in total), we have

$$\text{Adv}^{\text{ro-indiff}}_{\text{Sponge}2[\pi]}(\mathcal{A}) \leq \frac{q_p^2}{2p^2} + \frac{q_p}{p^{c/2}} \left( 1 + \frac{1}{p-1} \right) \frac{p^2}{p^c} \left( 1 + \frac{1}{p-2} \right) \leq \frac{q_p^2}{p^{c/2}} + \frac{q_p}{p^{c/2}} \left( \frac{3}{2} - \frac{2}{3} \right) \leq 3q_p/p^{c/2}.$$

We defer the proof of Theorem 6.1 to Section 7. We now define a formal claim about the indifferentiability bound for Sponge2 with fields of odd characteristic i.e., when $p > 2$ and when $r_0$ is fixed to $c/2$. In simple words, Corollary 6.2’s bound states that in the AO context, Sponge2 provides at least $c \cdot \log_2 p - 4)/2$ bits of security. This results in a concrete security of at least $\approx 126$ bits when $p \approx 2^{64}$ and $c = 4$ i.e., $p$ has 4 $2^p$ elements as the capacity.

**Corollary 6.2. Let Sponge2[\pi] be the hash function as defined above in Fig. 3 with $\pi \leftarrow \text{Perm}_p(b)$ and let $r_0 = c/2$, and $p > 2$. Then for any adversary $\mathcal{A}$ who makes at most $q_p$ RO queries, we have

$$\text{Adv}^{\text{ro-indiff}}_{\text{Sponge}2[\pi]}(\mathcal{A}) \leq \frac{2q_p}{p^{c/2}}.$$

**Proof of Corollary 6.2. The proof of Corollary 6.2 follows from the result of Theorem 6.1. More specifically, with $r_0 = c/2,$
with form for every input query (to either of the two functions) the output is $p$ free. Making $f$ differentiable proof, we will later need to restrict the primitive to not $p$ many queries, we get

$$
\text{Adv}_{\text{Sponge}2}^{\text{ro-indiff}}[f, f_1](\mathcal{A}) \leq \text{Adv}_{\text{Sponge}2}^{\text{ro-indiff}}[g_1, g_2](\mathcal{A}) + \frac{qp(2p^2 - 2p + 1)}{2p^b}.
$$

**Blacklisting Outputs for $f_1$ and $f_2$.** For a smooth indifferentiability proof, we will later need to restrict the primitive to not return a particular form of outputs, hence we hereby update the primitive to easily achieve this restriction later. Let $\mathcal{L}$ be a set of restricted outputs in $\mathbb{F}_p^b$ with size $|\mathcal{L}| = \Delta$ and let $g_1$ and $g_2$ be uniform random functions from $\mathbb{F}_p^b$ to $\mathbb{F}_p^b \setminus \mathcal{L}$. Observing that the statistical distance between $f_1$ and $g_1$ (similarly, between $f_2$ and $g_2$) is $\Delta/p$ and since we evaluate the primitives $f_1$ and $f_2$ in total $qp$ many times, we get

$$
\text{Adv}_{\text{Sponge}2}^{\text{ro-indiff}}[f_1, f_2](\mathcal{A}) \leq \text{Adv}_{\text{Sponge}2}^{\text{ro-indiff}}[g_1, g_2](\mathcal{A}) + \frac{qp\Delta}{p^b}.
$$

**Making $g_1$ and $g_2$. Capacity-collision-free.** Now, since $g_1$ and $g_2$ are uniform random functions over $\mathbb{F}_p^b \setminus \mathcal{L}$, they can still have random collisions in the last $c$ elements a.k.a. the capacity or the inner part. Such collisions are undesirable as they can be used to construct hash outputs for not-yet-queried inputs and thus make the hash differentiable from an RO. We therefore now replace the primitive with a capacity-collision-free variant.

We also note that for every input of the form $[x_1, x_2, \ldots, x_r, x_{r+1}, \ldots, x_{r+c}]$ with $x_i \in \mathbb{F}_p$, the outputs of $g_1$ and $g_2$ functions are sampled uniformly at random from $(\mathbb{F}_p^r \times \mathbb{F}_p^c) \setminus \mathcal{L}$ with form $[y_1, y_2, \ldots, y_r, y_{r+1}, \ldots, y_{r+c}]$ where $y_i \in \mathbb{F}_p$.

We now define a new class of functions called capacity-collision-free or CCF. CCF functions are defined in pairs.

**Definition 7.1.** A CCF pair $(f_1^{\text{ccf}}, f_2^{\text{ccf}})$ is a pair of $\mathbb{F}_p^r \times \mathbb{F}_p^c \rightarrow (\mathbb{F}_p^r \times \mathbb{F}_p^c) \setminus \mathcal{L}$ functions. Each of these functions takes $r + c$ elements in $\mathbb{F}_p$ as input and maps them to $r + c$ elements in $\mathbb{F}_p$ with the added restriction that the capacity i.e., the last $c$ elements of any output never collides/matches with the capacity of any previously queried input or output of both of the CCF functions.

One can notice that a CCF pair (i.e., both functions in total) cannot be exhausted in fewer than $p^r/2$ queries and is necessarily exhausted after at most $p^r - 1$ queries as by then all $\mathbb{F}_p^r$ many capacity values are exhausted. Further, a CCF pair can be constructed plainly by keeping a mapping table that requires a memory of at least $p^r/2$ (when all queried inputs and outputs have different capacities) to at most $p^r - 1$ (when only the queried outputs have different capacities) input-output pairs.

A randomly sampled CCF pair is defined as a CCF pair where for every input query (to either of the two functions) the output is sampled uniformly at random from the remaining space of possible outputs. We replace $(g_1, g_2)$ with a random CCF pair $(f_1^{\text{ccf}}, f_2^{\text{ccf}})$ and get

$$
\text{Adv}_{\text{Sponge}2}^{\text{ro-indiff}}[g_1, g_2](\mathcal{A}) \leq \text{Adv}_{\text{Sponge}2}^{\text{ro-indiff}}[f_1^{\text{ccf}}, f_2^{\text{ccf}}](\mathcal{A}) + \left| \Pr[\mathcal{A}|f_1^{\text{ccf}} \Rightarrow 1] - \Pr[\mathcal{A}|g_1 \Rightarrow 1] \right|.
$$

Note that the above probability difference is specific to the adversary $\mathcal{A}$ and its strategy. We know that this difference can never be higher than the actual statistical distance between the input-output distributions of the $qp$ queries to $(f_1^{\text{ccf}}, f_2^{\text{ccf}})$ and $(g_1, g_2)$.

Let $\mathcal{A}$ be a query bounded adversary making $q_1$ many queries to the first oracle and $q_2$ many queries to the second such that $q_1 + q_2 = q_p$. Let $\Theta_f = (\Theta_X, \Theta_Y, f) = ((X_1)_{i=1}^{q_1}, (Y_1)_{i=1}^{q_1})$ and $\Theta_g = (\Theta_X, \Theta_Y, g) = ((X_1)_{i=1}^{q_2}, (Y_1)_{i=1}^{q_2})$ be the random variables for the input-output distributions of the $qp$ queries with any adversarial choice of oracle order to $(f_1^{\text{ccf}}, f_2^{\text{ccf}})$ and $(g_1, g_2)$, respectively. Here $X_i$ and $Y_i$ represent the input and output of the $i$th query to the corresponding oracle, respectively. $\Theta_X$ takes values from $S_X = (\mathbb{F}_p^b)^{q_p}$ whereas $\Theta_Y, f$ and $\Theta_Y, g$ take values from $S_Y = (\mathbb{F}_p^b)^{q_p}$. Let $S = S_X \times S_Y$ and $S_{\text{ccf}} \subseteq S$ be the set of all possible $qp$ input-output tuples for oracles $(f_1^{\text{ccf}}, f_2^{\text{ccf}})$ i.e., the set of all possible capacity-collision-free query-response tuples. We get

$$
\Pr[\mathcal{A}|f_1^{\text{ccf}} \Rightarrow 1] \leq \Pr[\mathcal{A}|g_1 \Rightarrow 1]
$$

$$
\leq \Pr(\Theta_Y = \Theta_Y^1) \leq \Pr(\Theta_X = \Theta_X^1) = \Pr(\Theta_X = \Theta_X^1) / |S_Y|.
$$

Here the last equality holds as by definition, $\Theta_f$ only returns outputs from $S_{\text{ccf}}$. Now, since by design $\mathcal{A}$ does not make trivial queries we know that the $qp$ outputs of $g_1$ and $g_2$ are each sampled uniformly and independently at random from $\mathbb{F}_p^b \setminus \mathcal{L}$ and hence for $\theta = (\theta_X, \theta_Y)$, we get $\Pr[\Theta_Y = \theta] = \Pr[\Theta_X = \theta_X] \cdot \Pr[\Theta_Y = \theta_Y] = \Pr[\Theta_X = \theta_X] / |S_Y|$. Similarly, for a given query-tuple $\theta_X$, we also know that the corresponding $qp$ response-tuple $\theta_Y$ w.r.t. $f_1^{\text{ccf}}$ and $f_2^{\text{ccf}}$ is sampled uniformly at random from the set of all possible capacity-collision-free response tuples that correspond to the query tuple $\theta_X$, i.e., from the set $S_{\text{ccf}}^Y \subseteq S_Y$ defined as $\{ j \mid (i, j) \in S_{\text{ccf}}^Y \}$. This gives us

$$
\frac{1}{2} \sum_{\theta_X \in S_X} \left[ \Pr[\Theta_f = \theta_X] \cdot \frac{1}{|S_{\text{ccf}}^Y|} - \frac{1}{|S_Y|} \right] + \frac{1}{2} \sum_{\theta_Y \in S_Y} \left[ \Pr[\Theta_f = \theta_Y] \cdot \frac{1}{|S_{\text{ccf}}^X|} - \frac{1}{|S_X|} \right]
$$

$$
= \frac{1}{2} \sum_{\theta_X \in S_X} \Pr[\Theta_X = \theta_X] \cdot \left[ \sum_{\theta_Y \in S_{\text{ccf}}^Y} \frac{1}{|S_{\text{ccf}}^Y|} - \frac{1}{|S_Y|} \right]
$$

$$
+ \frac{1}{2} \sum_{\theta_Y \in S_Y} \Pr[\Theta_Y = \theta_Y] \cdot \left[ \sum_{\theta_X \in S_{\text{ccf}}^X} \frac{1}{|S_{\text{ccf}}^X|} - \frac{1}{|S_X|} \right].
$$

Tomer Ashur and Amit Singh Bhati
We are now left with bounding the term $\text{Adv}_\text{Sponge2}[\pi](\mathcal{A}) \leq \text{Adv}_\text{Sponge2}[f_1^{\text{ccf}}, f_2^{\text{ccf}}](\mathcal{A}) + \frac{q(p-1)}{2p^b} + \frac{q p^2 \beta}{p^b} + \frac{p^b - \Delta}{p}$.

We are now left with bounding the term $\text{Adv}_\text{Sponge2}[f_1^{\text{ccf}}, f_2^{\text{ccf}}](\mathcal{A})$.

We note that the above inequality holds for any value of $\Delta$; however, we are only interested in the smallest value of $\Delta$ for which the above upper bound minimizes.

**Reordering Input Space.** We recall that in Sponge2', the inputs are post-padded with minimum number of zeros to make the input rate aligned as shown in Fig. 3 (b). We define a new variant of Sponge2' named Sponge2\(^{†}\) where the padded zeros are pulled to the start of the message (see Fig. 4). For example, a message $M = [x_1, x_2, \ldots, x_{r+q+2}]$ of size $r + r_0 + 2$ elements is padded as $[x_1, x_2, \ldots, x_{r_0}, x_{r_0+1}, \ldots, x_{r+q+2}, (0), (0), \ldots, (0)]$ with $r = 2$ postpadded zeros when processed under Sponge2\(^{†}\) whereas the same message would become $(0), (0), \ldots, (0), x_1, x_2, \ldots, x_{r_0+1}, \ldots, x_{r+q+2}$ with $r = 2$ prepadded zeros when processed under Sponge2\(^{†}\).

**Figure 4:** Sponge2\(^{†}\) (block diagram) with prepadding.

We stress that the hash functions Sponge2' and Sponge2\(^{†}\) only differ in the padding rules and there is a bijective mapping between both of these padding rules. This implies that the output distributions of these two hash functions is identical and hence provide the same security (by indistinguishability from an RO). More concretely, we have

$$\text{Adv}_\text{ro-indiff}^{\text{Sponge2}[f_1^{\text{ccf}}, f_2^{\text{ccf}}]}(\mathcal{A}) = \text{Adv}_\text{ro-indiff}^{\text{Sponge2}[f_1^{\text{ccf}}, f_2^{\text{ccf}}]}(\mathcal{A})$$

$$\leq |\text{Pr}[\mathcal{A}^{\text{Sponge2}[f_1^{\text{ccf}}, f_2^{\text{ccf}}]}(\mathcal{A}) \Rightarrow 1] - \text{Pr}[\mathcal{A}^{\text{RO}, S_1, S_2} \Rightarrow 1]|$$

where $S_1$ and $S_2$ are two oracles defined to simulate adversarial-query response for $f_1^{\text{ccf}}$ and $f_2^{\text{ccf}}$ in the ideal world, respectively. Note that the above expression is an inequality instead of an equality as $S_1$ and $S_2$ may not necessarily be the best simulators to define the advantage tightly.

**DEFINING SIMULATORS.** In order to show that Sponge2\(^{†}\) is indistinguishable from an RO, we need to construct two simulators $S_1$ and $S_2$ to model $f_1^{\text{ccf}}$ and $f_2^{\text{ccf}}$ in the ideal world, respectively.

Let for any $b$-element vector $X$ with $b = r+c$, $X^r$ and $X^c$ represent the first $r$ and the last $c$ elements (as a vector) of $X$. In other words, $X = [X^r, X^c]$. We define $S_2 = f_2^{\text{ccf}}$ and $S_1$ as shown in Fig. 5. In the algorithmic description of $S_1$, $\mathcal{U}$ is a set of pairs of the form $(\alpha, \beta)$ where $\alpha$ is a $c$-element vector and $\beta$ is an arbitrary size vector. We set $\mathcal{U} = \{\}$ before $\mathcal{A}$ makes its first query.

For a fresh primitive query $S_1(X) = Y$ of $\mathcal{A}$, an entry $(\alpha, \beta)$ is added to $\mathcal{U}$ where $\alpha$ represents the last $c$ elements of the output $Y$ whereas $\beta$ represents the longest message whose hash output can be constructed using the already made $S_1$ queries (including the current query) with a condition that the final $S_1$ call of this hash is $S_1(X)$.

Note that $S_1$ and $f_1^{\text{ccf}}$ differs in the generation of top $r$ elements of the output, however, in both cases these $r$ elements are sampled
uniformly at random from $\mathbb{F}_p$ for each value of $X$. This implies $(S_1, S_2)$ is indistinguishable from $(f_1^{ccf}, f_2^{ccf})$.

**Indifferentiability from an RO.** We first define the set $L$ consisting restricted outputs for both primitives (introduced above for Exp. 2) to simplify the remaining analysis as the set of all possible first primitive call inputs of Sponge$^2$. More concretely, $L = \{X \in \mathbb{F}_p^r \mid$ for some $i \in \{0, r + r_0 - 1\}, [X[r + r_0 i] = (0)^i, (i), (0)^{c - r_0 - i})\}$. With $L$ defined, we get $\Delta = |L| = \sum_{r=0}^{r_0 - r_0} p^{r - r_0 - 1} = p(1 - p^{r_0 - 1})/(p - 1).

Let us refer the first oracle in both worlds (i.e., Sponge$^2$ in the real and RO in the ideal world) as the hash oracle and the second and third oracles in both worlds as the forward and backward primitive oracles, respectively. It is easy to notice that when the forward and backward primitives are restricted to return outputs only from $\mathbb{F}_p \setminus L$ and are capacity-collision-free, $\mathcal{A}$ can compute a hash oracle output only by either constructing the same using the old forward primitive query-response pairs or by making a fresh hash oracle query. We call the former type of queries — constructed queries and the latter type of queries — direct queries.

We emphasize here that the backward primitive calls become useless for $\mathcal{A}$ as they can’t be used to replace the forward primitive calls in any hash query. This holds as backward primitive queries never return outputs that belong to $L$ (i.e., the set of restricted outputs for primitives) or have capacity collisions with forward primitive query outputs (as they are capacity-collision-free).

Note that in the real world, all constructed and direct hash queries are consistent as per the definition of Sponge$^2$ i.e., on same message they both return same outputs. Similarly, in the ideal world, with the defined simulator $S_1$, all constructed and direct hash queries are consistent as per the definition of $S_1$. See Fig. 6 for an example showing how a constructed query in the ideal world is consistent with its corresponding direct query. The example shows that for some input $M$, the constructed query returns $\mathcal{R}O(\text{unpad}_1([P_0, \ldots, P_1]))$ whereas the direct query returns $\mathcal{R}O(M)$ and since by definition, $M = \text{unpad}_1([P_0, \ldots, P_1])$, we have $\mathcal{R}O(\text{unpad}_1([P_0, \ldots, P_1])) = \mathcal{R}O(M)$. Here, for any vector $X$, unpad$_1([0]^i, X) = X$ and $\bot$, otherwise.

This means the only way left for $\mathcal{A}$ to differentiate (Sponge$^2$ $f_1^{ccf}, f_2^{ccf}$) from (RO, $S_1$, $S_2$) is by distinguishing the direct query outputs of Sponge$^2$ ($f_1^{ccf}$) (i.e., hash outputs in the real world) from the direct query outputs of RO (i.e., hash outputs in the ideal world). Now, since $f_1^{ccf}$ is a capacity-collision-free function, we have that for distinct direct queries to Sponge$^2$ ($f_1^{ccf}$), the last $f_1^{ccf}$ calls will always have a unique input and thus the rate part of its output (which is the final hash output) will be sampled uniformly at random from $\mathbb{F}_p$. Similarly, since the outputs of RO are also sampled uniformly at random from $\mathbb{F}_p$, for distinct direct queries, we have that

$$\text{Adv}_{\text{indiff}}^{\text{ro-indiff}}(\mathcal{S}_2 | f_1^{ccf}, f_2^{ccf}) (\mathcal{A}) = 0 \quad (8)$$

We finalize the indifferentiability bound by combining Exp. 6, 7 and 8 with $\Delta = (1 + (p - 1)1)(p^{r_0 - 1})$ and get

$$\text{Adv}_{\text{indiff}}^{\text{ro-indiff}}(\mathcal{S}_2 | f_1^{ccf}) (\mathcal{A}) \leq \frac{q_p(p - 1)}{2p^b} \frac{q_p + p c^{r_0 - 1}}{p - 1} \left(1 + \frac{1}{p - 1}\right)$$

and thus the result of Theorem 6.1. □

**8 DISCUSSION AND CONCLUSION**

Arithmetization oriented hash functions are a crucial building block in ZKP systems deployed in the real world (e.g., blockchain L2 rollups). In many cases, the efficiency of the hash function is one of the major cost drivers for the entire system. Consequently, optimizing certain aspects of the hash function results in a net improvement to the entire system. Work in this field employed the permutation based cryptography paradigm to construct Sponge-based hash functions. When employing this paradigm, a designer comes up with a suitable permutation; i.e., a carefully crafted permutation admitting no distinguishable properties that is efficient to evaluate on the target architecture. Then, the permutation is used to instantiate a Sponge-function, resulting in a versatile algorithm that can be used as a hash function.
Indeed, most work in this domain focused on designing more efficient permutations (for some definition of efficiency). In this work we take a different approach and focus instead on the Sponge construction itself. As a first contribution, which is of independent interest, we generalize the sponge construction to GSponge. We then argue the security of the generalized construction with permutations defined over large prime fields. This latter result work we take a different approach and focus instead on the Sponge hashing, short messages, etc.). As a vision for future work, we hope to improve the hashing time of practical use-cases (e.g., Merkle-tree optimizations for deployed sponges. First, we introduce a new type of padding rule based on zero-padding and domain-separated inputs. This padding rule never extends the message length by a full rate block, and consequently, saves one full permutation call in cases where the unpadded message’s length is already an integral multiple of the rate (e.g., in 2-to-1 Merkle-tree hashing) without increasing the proof generation time in zkVMs like Polygon Miden. Secondly, we show that in the first permutation call it is possible to absorb up to $c \log_2 p_1/2$ more elements, again resulting in saving a permutation call for some message lengths, without any loss in the security size, i.e., still $\approx c \log_2 p_1/2$ bits.

While not asymptotic, these micro-optimizations can be used to improve the hashing time of practical use-cases (e.g., Merkle-tree hashing, short messages, etc.). As a vision for future work, we hope that this paper will inspire further work on modes of operation tailored for permutations defined over large prime fields.

\footnote{As an exception, we note the SAFE API \cite{Z%202023} providing a proof for one, somewhat contrived, instance of the Sponge construction.}

ACKNOWLEDGEMENTS
Amit Singh Bhati was supported by CyberSecurity Research Flanders with reference number VR20192203, in part by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things with contract number C16/15/058 and by the Flemish Government through FWO Project G.0835.16 A security Architecture for IoT.

REFERENCES
\begin{thebibliography}{9}
\end{thebibliography}


