The Billion Dollar Merkle Tree

Thomas Coratger [1] Dmitry Khovratovich [1] Bart Mennink [2]
Benedikt Wagner [1]

[1] Ethereum Foundation
{firstname.lastname}@ethereum.org
[2] Maastricht University
{firstname.lastname}@maastrichtuniversity.nl

Abstract

The Plonky3 Merkle tree implementation has become one of the most widely deployed Merkle tree constructions due to its high efficiency, and—through its integration into numerous succinct-argument systems—it currently helps secure an *estimated $4 billion in assets*. Somewhat paradoxically, however, the underlying 2-to-1 compression function is *not collision-resistant, nor even one-way*, which at first glance appears to undermine the security of the entire Merkle tree. The prevailing ad-hoc countermeasure is to pre-hash data before using them as leaves in this otherwise insecure Merkle tree.

In this work, we provide the first rigorous security analysis of this Merkle tree design and show that the Plonky3 approach is, in fact, sound. Concretely, we show (strong) position-binding and extractability.

This is the full version of the paper accepted for publication at the 33rd ACM Conference on Computer and Communications Security (ACM CCS 2026). The ACM Version of Record will appear in the ACM Digital Library.

Contents

1 Introduction **3**
- 1.1 Our Contribution . . . 4
- 1.2 Related Work . . . 5

2 Preliminaries **5**
- 2.1 Vector Commitments . . . 6
- 2.2 Hash Functions . . . 7
- 2.3 Sponge Mode . . . 8

3 The Analyzed Merkle Tree Scheme **9**

4 Proof of Strong Position-Binding **10**
- 4.1 Proof of Lemma 3 . . . 12
- 4.2 Proof of Lemma 4 . . . 13
- 4.3 Proof of Lemma 5 . . . 14

5 Proof of Strong Extractability **17**

6 Counterexample for Dependent Leaf Hashing **21**

7 Proof of Strong Extractability for Sponges **22**

8 Conclusion **24**

A More Details on Adoption of Plonky3 **31**
- A.1 Zero-Knowledge Virtual Machines (zkVMs) . . . 31
- A.2 Infrastructure and Future Consensus . . . 31

B More on Tree Hashing and Permutation-Based Hashing **32**

C Plonky3's MMCS **34**

1 Introduction

The landscape of succinct arguments (SNARKs, hereafter) has undergone a paradigm shift toward hash-based small-field constructions, driven by the efficiency of fields such as Goldilocks, BabyBear, KoalaBear, and the Mersenne31 prime [Grund, HLP24]. At the epicenter of this ecosystem lies *Plonky3* [Plo25], a modular toolkit for Polynomial Interactive Oracle Proofs (PIOPs) [BSCS16] that has effectively established itself as the standard cryptographic backend for this modern industry.

The Plonky3 library serves as the foundational infrastructure for a new generation of zkVMs (Zero-Knowledge Virtual Machines)[1] [Suc24, RIS25, Ope25, Pol25a, Pol24a], high-performance rollups [Sta25, Mat25, Scr25, Con25], and verifiable computation networks [Suc25c, Fer25]. One crucial component in Plonky3 is its Merkle tree implementation, which is highly efficient. We study this implementation in this work, thereby focusing on a primitive that currently secures billions of dollars in digital assets.

Industrial Adoption and Criticality. The ubiquity of the Plonky3 Merkle tree is best illustrated by its integration into high-value protocols. Most notably, the SP1 VM [Suc24] by Succinct Labs relies on this implementation. With a Total Value Secured (TVS) estimated in the multi-billions across major platforms like Polygon, Celestia, and Avail, the economic weight resting on this primitive is substantial (see Appendix A for a detailed breakdown of adoption statistics).

Beyond SP1, Plonky3 has effectively defined the technological baseline for the industry's race to real-time proving for Ethereum L1[2]. It serves as the underlying framework for a diverse array of emerging zkVMs—such as Valida, OpenVM, Ziren, and Pico—and is actively being adopted by privacy-oriented projects like Polygon Miden [Pol24b]. Furthermore, the framework is a leading candidate for the future Ethereum L1 consensus layer to aggregate post-quantum signatures [Lea25].

Given that the security of these high-value systems rests on the integrity of the Merkle tree, one would expect this tree to adhere to conservative cryptographic principles. However, a closer inspection reveals a design choice that appears, at first glance, to be dangerously fragile.

The Merkle Tree Anomaly. A Merkle tree [Mer79] allows a prover to commit to a vector of data by recursively applying a 2-to-1 compression function h, e.g., mapping 2λ bits of input to λ bits of output. The function is applied to pairs of nodes until a single root is obtained. In standard cryptographic literature, the security of such a tree is inextricably linked to the collision and preimage resistance of this compression function h [Mer90]. The basic security property that we require from such a tree is *position-binding*. Simply put, position-binding ensures that once a commitment (the root) is published, a computationally bounded adversary cannot equivocate—that is, they cannot produce valid opening proofs for two different values at the same index. Traditionally, this property is proven via a straightforward reduction: if an adversary can successfully open the same position to two different values, they must have found a collision in h somewhere along the authentication path. Thus, if h is collision-resistant, the Merkle tree is position-binding.

In contrast to this established paradigm, Plonky3 adopts a modular architecture that supports diverse compression backends, ranging from standard constructions like SHA-256 to arithmetic-friendly primitives. However, to maximize efficiency of recursion within small-field SNARKs, practitioners overwhelmingly favor algebraic permutations such as Poseidon2 [GKS23]. In these widely used configurations, the compression function h is instantiated not as a one-way function, but as a *truncated permutation* (see Figure 1).

This design choice creates a significant theoretical gap. A truncated permutation is efficiently *invertible*[3], and thus the compression function lacks collision and preimage resistance. This allows an adversary to manufacture valid authentication paths by simply unwinding the tree from the root. Consequently, the construction appears to be insecure, defying standard cryptographic intuitions and rendering classical security arguments inapplicable (see Section 1.2).

Intuitive Security vs. Formal Rigor. Given the trivial invertibility of the compression function, security of the Plonky3 Merkle tree appears, at first, to be broken. Fortunately, security may be restored

[1] While the industry has adopted the term "zkVM" (Zero-Knowledge Virtual Machine), we note that these systems are primarily utilized for succinctness and usually do not achieve the cryptographic zero-knowledge property.

[2] See `https://ethproofs.org/`.

[3] To invert, one can just pad and then apply the inverse permutation.

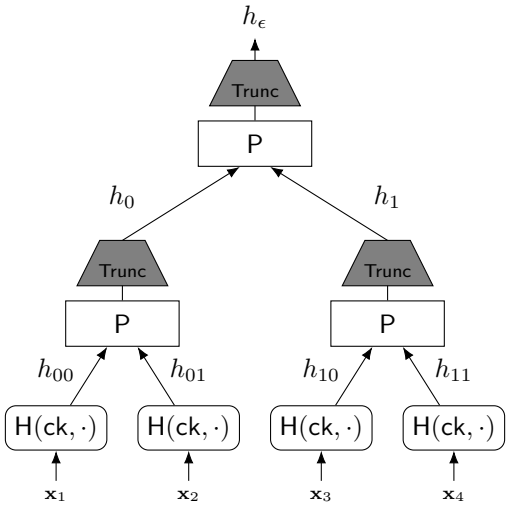


Figure 1: Visualization of the Merkle tree construction based on permutations, formally defined in Figure 3.

via the processing of the input data. In the Plonky3 architecture, the data (pre-leaves) is not fed directly into the invertible compression; instead, it is first processed by a distinct, *secure* cryptographic hash function, which we refer to as the *leaf hash*. The leaf hash transforms the pre-leaves into actual leaves, which are then compressed using the truncated permutation.

This initial step acts as a cryptographic boundary. While the internal compression function allows an adversary to easily generate colliding paths consisting of arbitrary field elements, these elements are not necessarily valid leaves. To successfully equivocate on the commitment, the adversary must find a collision path that descends to the bottom of the tree *and* results in a value that matches a valid output of the leaf hash. Thus, intuitively, the security of the tree relies on the difficulty of finding a preimage for the leaf hash that coincides with the values produced by inverting the compression function.

However, relying on merely intuitive arguments for a primitive that secures over $4 billion in assets represents a significant gap. We cannot simply postulate that the constraints imposed by the leaf hash are sufficient to preclude all attacks against the Merkle tree; we must demonstrate it. We therefore ask:

Can we prove the security of a Merkle tree implementation following the Plonky3 design—specifically one without a collision-resistant compression function?

1.1 Our Contribution

We answer this question positively, and give the *first security proofs for Plonky3's Merkle tree*. The tree is defined by a permutation P and a leaf hash function H. In practice, H may be instantiated by users of the Plonky3 library in various ways, which are covered by our proofs as follows:

- We show *strong position-binding* of the Merkle tree assuming H is a collision-resistant and preimage-resistant hash function;
- We show *strong extractability* when H is modeled as an independent random oracle;
- We show *strong extractability* when H implements the (overwrite) sponge mode [BDPV07] using P.

Notably, strong extractability implies strong position-binding (cf. Lemma 1), and extractability is the notion that the Merkle tree needs to satisfy to yield security of the BCS transform [BCS16] used to build modern SNARKs. All of our proofs model P as an ideal permutation. Our results are summarized in Table 1.

Table 1: Overview of our results for the analysis of Plonky3's Merkle tree (cf. Figures 1 and 3). All of our results additionally model the permutation P as an ideal permutation.

Result	Assumptions on H	Reference	Comment
Strong Position-Binding	Coll.-Res. + Pre.-Res.	Section 4	H in standard model
Strong Extractability	Random Oracle	Section 5	Extr. $\Rightarrow$ Pos.-Bind.
Strong Extractability	Sponge using P	Section 7	Extr. $\Rightarrow$ Pos.-Bind.

Our first two results assume that H is implemented independently[4] from P. One may wonder what happens if H is indeed implemented using P. To address this question, we give a (contrived!) counterexample (cf. Section 6): an instantiation of H using P that is collision-resistant, but such that the resulting Merkle tree is completely insecure. In addition to our third positive result above, this highlights the subtlety of choosing the right leaf hash H.

1.2 Related Work

We briefly revisit results related to our work. We refer to Appendix B for a more in-depth discussion.

It was already mentioned that security of the Merkle tree follow from the security of the underlying compression function h [Mer90]. However, in many applications, one requires a hash function to appear like a random oracle [BR93], a security property covered by the concept of *indifferentiability* [MRH04, CDMP05]. Yet, the basic Merkle tree (treated as a hash function) is not indifferentiable from a random oracle, as it is vulnerable to the length extension attack. A body of research [DRRS09, BDPV14b, DDV11], culminating in a systematization of knowledge of Daemen et al. [DMV18], presented sufficient conditions for sound tree hashing. These conditions also apply in case the compression function is a truncated permutation. Of particular interest in the eventual systematization of knowledge is their minimal tree hashing mode based on a truncated permutation. Assuming we take a permutation P on b bits and truncate it to n bits, we devote two bits of the input to P for domain separation: 00 for leaves, 11 for final, and 10 for other truncated permutation evaluations. For the leaves, we reserve an additional m bits for a dedicated initial value IV. For the rest, the tree operates as default. In this way, one obtains a tree hasher that is indifferentiable from a random oracle. These observations straightforwardly generalize to elements from larger fields.

One may observe that the Plonky3 Merkle tree structurally differs as it does not include an initial value IV, but rather takes the hash output of any message. This idea was analyzed before in the context of indifferentiability of the truncated permutation (i.e., not for the Merkle tree in general) by Grassi and Mennink [GM22]. As such, the Merkle tree variant in Plonky3 can be seen as a hybrid between the minimal permutation-based tree hasher of Daemen et al., but without domain separators, and the hash-then-permute-then-truncate approach of Grassi and Mennink. However, the lack of domain separation makes Plonky3's Merkle tree with simple message encoding vulnerable to a length extension attack and thus differentiable from a random oracle. This, concretely, already demonstrates that all earlier results on the indifferentiability of tree hashers are futile for Plonky3's tree hasher. Fortunately, this is *not a problem*: indifferentiability would even be an undesirably strong requirement. Indeed, indifferentiability implies that the tree hasher behaves like a monolithic random oracle, but we are interested in *vector commitments* and to prove strong position-binding and extractability, we need to define and analyze the concept of openings. Additionally, a direct proof of these two security properties, as we deliver in this work, gives a tighter bound with lighter assumptions on the hash function (see Table 1).

2 Preliminaries

We define our notation and the relevant cryptographic primitives.

Sets, Sampling, Distributions. For a finite set $\mathcal{X}$, we denote $\mathcal{X}^* = \bigcup_{i=0}^{\infty} \mathcal{X}^i$. For an integer r, we define $[r] := \{1, \ldots, r\} \subseteq \mathbb{N}$. We use $\perp$ to denote a dedicated abort symbol. The notation $\mathbb{Z}_q$ for some

[4]This is the most common case in practice, where P is Poseidon2-16 and H is Poseidon2-24 in overwrite sponge mode [BDPV07].

integer q refers to the set $\{0, \ldots, q-1\}$, which can be treated as a ring or field. Sampling an element uniformly from a finite set X is denoted by $x \xleftarrow{\$} X$, and for sampling from a distribution $\mathcal{D}$, we write $x \leftarrow \mathcal{D}$.

Strings and Vectors. The notation $\mathsf{rep}_2(i)$ denotes the representation of an integer i as a bitstring. For a bitstring $s \in \{0,1\}^k$, we denote by $s_{\leq l}$ the l-bit prefix of s. We denote by $\epsilon \in \{0,1\}^0$ the empty string. It has length 0 and is the prefix $s_{\leq 0}$ of every bitstring s. Each bitstring is also a prefix of itself. By $\|$, we denote string concatenation. By $\oplus$ we denote the bitwise exclusive or of two bitstrings of the same length. For a string, sequence, or vector $\mathbf{x}$, we denote by $\mathbf{x}_i$ its ith component. In this case, our indexing starts with 1. For a bitstring s of length at least a bits, we denote by $\mathsf{LBits}_a(s)$ its leftmost a bits and $\mathsf{RBits}_a(s)$ its rightmost a bits.

Cryptographic Notation and Algorithms. We denote the security parameter by λ. All algorithms get λ implicitly as input. PPT stands for probabilistic polynomial-time. A negligible function is a function vanishing faster than any inverse polynomial. For a deterministic algorithm Alg, we denote its output on input x by $y = \mathsf{Alg}(x)$. If Alg is randomized, we write $y \leftarrow \mathsf{Alg}(x)$ to indicate that the output is generated using fresh uniform random coins. When the randomness is fixed explicitly, we write $y = \mathsf{Alg}(x; \rho)$. The notation $y \in \mathsf{Alg}(x)$ refers to any output produced with positive probability on input x. We use $\mathbf{T}(\mathsf{Alg})$ to denote the running time of Alg. A function is said to be efficiently computable if it can be computed by a deterministic polynomial-time algorithm.

Idealized Models. In the *random oracle model*, we model a hash function $\mathsf{H}\colon \mathcal{X} \to \mathcal{Y}$ as a function picked uniformly at random, which is given to all parties as an oracle. In the *ideal permutation model*, we model a permutation $\mathsf{P}\colon \mathcal{X} \to \mathcal{X}$ that is used in the construction as a *perfectly random permutation*. Precisely, this means that P is sampled at the onset of the security experiment uniformly at random from the set of all bijections of the form $\mathcal{X} \to \mathcal{X}$. Then, throughout the experiment, the adversary has oracle access to both P and the inverse P^{-1}. In our specific case, we will have $\mathcal{X} = \mathcal{H} \times \mathcal{H}$ for some space $\mathcal{H}$, i.e., $\mathsf{P}\colon \mathcal{H} \times \mathcal{H} \to \mathcal{H} \times \mathcal{H}$.

2.1 Vector Commitments

We introduce the syntax of vector commitments. Intuitively, vector commitments allow to commit to a vector and later open individual positions of that vector.

Definition 1 (Vector Commitment Scheme)**.** A vector commitment scheme for length ℓ vectors over a domain $\mathcal{D}$ is a tuple of PPT algorithms $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{ck}$.
- $\mathsf{Com}(\mathsf{ck}, \mathbf{x}) \to \mathsf{com}$ for $\mathbf{x} \in \mathcal{D}^\ell$.
- $\mathsf{Open}(\mathsf{ck}, \mathbf{x}, i) \to \tau$ for $\mathbf{x} \in \mathcal{D}^\ell$, $i \in [\ell]$.
- $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, x, \tau) \to b \in \{0,1\}$, deterministic, for $x \in \mathcal{D}, i \in [\ell]$.

We require the scheme to be correct in the following sense: for any $\mathsf{ck} \in \mathsf{Setup}(1^\lambda)$, any $\mathbf{x} \in \mathcal{D}^\ell$, and any $i \in [\ell]$, we have

$$\Pr\left[\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, \mathbf{x}_i, \tau) = 1 \;\middle|\; \begin{array}{c} \mathsf{com} \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{x}), \\ \tau \leftarrow \mathsf{Open}(\mathsf{ck}, \mathbf{x}, i) \end{array}\right] = 1.$$

The first security notion that we will define (and prove) is *strong position-binding*. Intuitively, (weak) position-binding ensures that no efficient adversary can open a single vector commitment com to two conflicting values $x, x' \in \mathcal{D}$ in one position i. That is, the commitment is binding for every position i individually. In this work, we consider a stronger notion, in the sense that the adversary also wins if $x = x'$ but the provided openings τ and τ' differ.

Definition 2 (Strong Position-Binding)**.** Let $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ be a vector commitment scheme. We say that VC is (strongly) position-binding, if for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{posbind}}_{\mathsf{VC}}(\mathcal{A}) := \Pr\left[\begin{array}{ll} & \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, x, \tau) = 1 \\ \wedge & \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, x', \tau') = 1 \\ \wedge & (x, \tau) \neq (x', \tau') \end{array} \;\middle|\; \begin{array}{c} \mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda), \\ (\mathsf{com}, i, x, \tau, x', \tau') \leftarrow \mathcal{A}(\mathsf{ck}) \end{array}\right].$$

We also define a stronger notion, namely, *extractability*, that is needed in the application of vector commitments in succinct arguments [BCS16]. We define it for the case where the vector commitment is implemented from a set of idealized primitives, e.g., a random oracle or an ideal permutation. Intuitively, it states that one can efficiently extract a "preimage" of the commitment from the commitment (by looking at oracle queries), such that any opening provided by an adversary is consistent with this extraction. Our extractability definition is again *strong* in the following sense: the extractor not only extracts the vector of data $\mathbf{x}$, but also openings, and the adversary wins if the opening that it provides is inconsistent with the extracted one. Further, we note that our extractability definition is *adaptive*, in the sense that the adversary gets the extracted information before deciding which position to open and which opening to output. We also show that strong extractability implies strong position-binding.

Definition 3 (Strong Extractability)**.** Let $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ be a vector commitment scheme, which is defined with respect to an oracle[5] O. We say that VC is (strongly) extractable, if there is a PPT algorithm Ext, such that for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{extract}}_{\mathsf{VC},\mathsf{Ext}}(\mathcal{A}) := \Pr\left[\mathsf{Ver}^{\mathrm{O}}(\mathsf{ck}, \mathsf{com}, i, x, \tau) = 1 \wedge (x, \tau) \neq (\mathbf{x}_i, \tau_i) \;\middle|\; \begin{array}{c} \mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda), \\ (\mathsf{com}, St) \leftarrow \mathcal{A}^{\mathrm{O}}(\mathsf{ck}), \\ (\mathbf{x}, (\tau_i)_{i\in[\ell]}) \leftarrow \mathsf{Ext}(\mathsf{com}, \mathcal{Q}), \\ (i, x, \tau) \leftarrow \mathcal{A}^{\mathrm{O}}(St, \mathbf{x}, (\tau_i)_{i\in[\ell]}) \end{array}\right],$$

where $\mathcal{Q}$ denotes the ordered list of all oracle queries (and their responses) that $\mathcal{A}$ made up to that point.

Lemma 1 (Strong Extractability $\implies$ Strong Position-Binding)**.** *Assume that* $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ *is a a vector commitment scheme that is strongly extractable with extractor* Ext*. Then* VC *is strongly position-binding. Namely, for any PPT algorithm* $\mathcal{A}$ *there exists a PPT algorithm* $\mathcal{B}$ *with* $\mathbf{T}(\mathcal{A}) \approx \mathbf{T}(\mathcal{B})$ *and*

$$\mathsf{Adv}^{\mathsf{posbind}}_{\mathsf{VC}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{extract}}_{\mathsf{VC},\mathsf{Ext}}(\mathcal{B}).$$

Proof. This follows from a straightforward reduction: assume $\mathcal{A}$ breaks position-binding by producing a commitment com and valid openings $(x, \tau) \neq (x', \tau')$ at the same index i. The reduction outputs com and gets back what the extractor extracted. It is impossible for the extracted $(\mathbf{x}_i, \tau_i)$ to equal both (x, τ) and (x', τ'). Therefore, at least one of the adversary's openings must contradict the extracted values, thereby allowing the reduction to break extractability. □

2.2 Hash Functions

We recall the definition of a hash function and its security.

Definition 4 (Hash Function)**.** A hash function with key space $\mathcal{K}$, domain $\mathcal{D}$, and hash space $\mathcal{H}$ is an efficiently computable function of the form

$$\mathsf{H}\colon \mathcal{K} \times \mathcal{D} \to \mathcal{H}.$$

Formally, $\mathcal{K}, \mathcal{D}, \mathcal{H}$ may implicitly depend on the security parameter.

In order to define collision-resistance and preimage-resistance in a meaningful way in the standard model, we use keyed hash functions. To be precise, above definition is exactly the one used in the hash function security model formalization of Rogaway and Shrimpton [RS04], and also below definition of collision-resistance is taken from their work. Below definition of preimage-resistance is slightly weaker than the preimage-resistance notions of Rogaway and Shrimpton [RS04]: it is the range-oriented version

[5] This oracle can of course implement multiple oracles by taking as its first input a pointer to the desired oracle.

of preimage-resistance as formalized by Andreeva and Stam [AS11]. Note that our goal is not to prove collision-resistance and preimage-resistance of our hash functions, in fact, we will use them as assumptions on the hash function in our vector commitment scheme to be able to prove position-binding. We finally note that, in practice, the reader may think of H as a keyless hash function. In this case, collision-resistance and preimage-resistance still make sense in the random oracle model.

Definition 5 (Collision-Resistance)**.** Let $\mathsf{H}\colon \mathcal{K} \times \mathcal{D} \to \mathcal{H}$ be a hash function. We say that H is collision-resistant, if for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{collres}}_{\mathsf{H}}(\mathcal{A}) := \Pr\left[\begin{array}{c|c} x \neq x' \wedge & K \xleftarrow{\$} \mathcal{K}, \\ \mathsf{H}(K,x) = \mathsf{H}(K,x') & (x,x') \leftarrow \mathcal{A}(K) \end{array}\right].$$

Definition 6 (Preimage-Resistance)**.** Let $\mathsf{H}\colon \mathcal{K} \times \mathcal{D} \to \mathcal{H}$ be a hash function. We say that H is preimage-resistant, if for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{preres}}_{\mathsf{H}}(\mathcal{A}) := \Pr\left[\mathsf{H}(K,x) = h \;\middle|\; \begin{array}{c} K \xleftarrow{\$} \mathcal{K},\ h \xleftarrow{\$} \mathcal{H} \\ x \leftarrow \mathcal{A}(K,h) \end{array}\right].$$

2.3 Sponge Mode

The sponge mode [BDPV07] provides a generic framework for building variable-length hash functions from a fixed-width permutation P. It is originally defined to operates on bits, i.e., using a b-bit permutation P and operating on a state of size b bits that is split into an inner part of c bits and an outer part of r bits, processing messages by absorbing data into the outer part r bits at a time, and then squeezing digests r bits at a time. In our application, however, we will only need a simplified version of it, namely one that operates on a permutation $\mathsf{P}\colon \mathcal{H} \times \mathcal{H} \to \mathcal{H} \times \mathcal{H}$, where both the inner part and the outer part are represented by $\mathcal{H}$, which contains some default element $\mathbf{0}$. Additionally, we will not consider the sponge that adds the message into the outer part, but rather the one that overwrites the outer part by the message. To make the distinction clear, we will consistently refer to this as the overwrite sponge (oSponge). The function $\mathsf{oSponge}$ that we will use in our application is given in Figure 2.

Algorithm $\mathsf{oSponge}^{\mathsf{P}}(\mathbf{m}, n)$

// initialize state
$S = (S_1, S_2) := (\mathbf{0}, \mathbf{0})$

// parse message
Write $\mathbf{m} = (\mathbf{m}_1, \ldots, \mathbf{m}_k) \in \mathcal{H}^k$

// absorb
for $i \in [k]$: $S := (\mathbf{m}_i, S_2),\ S := \mathsf{P}(S)$

// squeeze
for $j \in [n]$: $h_j := S_1,\ S := \mathsf{P}(S)$
return $(h_1, \ldots, h_n)$.

Figure 2: The overwrite sponge hash function $\mathsf{oSponge}^{\mathsf{P}}$ built using the permutation P. The function takes as input a message $\mathbf{m} \in \mathcal{H}^*$ and an integer n and outputs the vector of n values from $\mathcal{H}$.

As per Definition 4, the hash function $\mathsf{H}\colon \mathcal{K} \times \mathcal{D} \to \mathcal{H}$ that we employ in Section 7 is defined as

$$\mathsf{H}(K, M) := \mathsf{oSponge}^{\mathsf{P}}(K \| M, 1),$$

where $K \in \mathcal{K} := \mathcal{H}$ and $M \in \mathcal{D} := \mathcal{H}^*$. Refer to Appendix B for an extensive discussion on related work and the security bounds on the sponge.

Remark 1. Note that this description (as well as all security results below) requires the message to be an integral number of blocks from $\mathcal{H}$. In case of arbitrarily length messages an injective padding must be applied to assure that any string in $\mathcal{H}^*$ corresponds to at most one message.

3 The Analyzed Merkle Tree Scheme

In this section, we define the specific Merkle tree variant utilized by Plonky3. We restrict our focus to the core cryptographic structure: a binary tree of height k that commits to a vector $\mathbf{x}$ of size $\ell = 2^k$. In Appendix C, we give more details on how this abstract scheme is used in Plonky3 to commit to a collection of matrices.

Building Blocks and Structure. The construction relies on the following three primitives:

- **Leaf Hash.** $\mathsf{H}\colon \mathcal{K} \times \mathcal{D} \to \mathcal{H}$. A (keyed) cryptographic hash function (Definition 4) used to map input data from domain $\mathcal{D}$ to the digest space $\mathcal{H}$.
- **Internal Permutation.** $\mathsf{P}\colon \mathcal{H} \times \mathcal{H} \to \mathcal{H} \times \mathcal{H}$. An efficiently computable cryptographic permutation acting on pairs of digests.
- **Truncation Map.** $\mathsf{Trunc}\colon \mathcal{H} \times \mathcal{H} \to \mathcal{H}$, defined by $(h_1, h_2) \mapsto h_1$.

We index the nodes of the Merkle tree via binary strings $s \in \{0,1\}^{\leq k}$, representing the path from the root. The empty string ϵ denotes the root node h_ϵ, while strings $s \in \{0,1\}^k$ of length k denote the leaves. Intuitively, the commitment scheme operates in two phases, described next.

Leaf Generation. The tree is initialized by hashing the data vector $\mathbf{x} \in \mathcal{D}^\ell$. For each index $i \in [\ell]$, the corresponding leaf node h_s is computed as:

$$h_s := \mathsf{H}(\mathsf{ck}, \mathbf{x}_i),$$

where $s = \mathsf{rep}_2(i-1)$ is the k-bit binary representation of the index. We may occasionally call h_s the *leaf* and $\mathbf{x}_i$ the corresponding *pre-leaf*.

Internal Compression. Internal nodes are derived recursively by compressing their children. Unlike standard Merkle trees, which typically employ a collision-resistant compression function, this scheme utilizes a *truncated permutation*. For any internal node s with children $s\|0$ (left) and $s\|1$ (right), the parent value is computed as:

$$h_s := \mathsf{Trunc}\big(\mathsf{P}(h_{s\|0}, h_{s\|1})\big).$$

The complete formal definition of the vector commitment scheme $\mathsf{VC}^{\mathsf{mt}} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ is detailed in Figure 3, and we show a visualization in Figure 1.

Remark 2 (Poseidon2 Instantiation). For concreteness, the reader may think of P as being the Poseidon2 permutation [GKS23] over $\mathbb{F}^{16} = \mathbb{F}^8 \times \mathbb{F}^8$, but our results are generic in P and H.

The Plonky3 library is designed to be modular, allowing the leaf hash H to be instantiated in several ways. The most common choice in practice is the $\mathsf{oSponge}$ of Section 2.3 (referred to as `PaddingFreeSponge` in the codebase) using the Poseidon2 permutation. More precisely, width-24 Poseidon2 is used for H (in overwrite sponge with rate 16 and capacity 8, outputting 8 field elements as the digest), whereas width-16 Poseidon2 is used for P. Alternative instantiations of H include classical hash functions such as Keccak-f [Nat15], SHA-256 [Nat02], or Blake3 [OANWO21], though these are less common in practice due to their higher arithmetization cost within SNARKs.

Remark 3 (Sparse Trees). In our definition of $\mathsf{VC}^{\mathsf{mt}}$, we assumed that each leaf/pre-leaf pair has the same depth k, i.e., is represented by a bitstring $s \in \{0,1\}^k$. That is, we work with balanced full binary trees. However, we note that all of our results easily *generalize* to the *sparse* case, i.e., to the setting where some leafs of the tree have depth less than k, i.e., are indexed by a bitstring $s \in \{0,1\}^{<k}$, as long as these leafs are also generated as hashes of pre-leafs. As a cryptographic primitive, we could think of such trees as a variant of vector commitments, namely, commitments to certain partial functions $\{0,1\}^{\leq k} \to \mathcal{D}$. Note that in this case we assume that the topology of the tree (i.e., which leafs have which depth) is fixed.

Algorithm $\mathsf{Setup}(1^\lambda)$
01 **return** $\mathsf{ck} \xleftarrow{\$} \mathcal{K}$

Algorithm $\mathsf{Compress}(h^{\mathsf{left}}, h^{\mathsf{right}})$
02 **return** $\mathsf{Trunc}(\mathsf{P}(h^{\mathsf{left}}, h^{\mathsf{right}}))$

Algorithm $\mathsf{GenTree}(\mathsf{ck}, (h_s)_{s\in\{0,1\}^k})$
// build the tree bottom-up
03 **for** $l = k-1, \ldots, 0$:
04 **for** $s \in \{0,1\}^l$:
05 $h_s := \mathsf{Compress}(h_{s\|0}, h_{s\|1})$
06 **return** $((h_s)_{s\in\{0,1\}^l})_{l=0}^k$

Algorithm $\mathsf{GenLeafs}(\mathsf{ck}, \mathbf{x})$
// map pre-leafs to leafs using H
07 **for** $i \in [\ell]$:
08 $s := \mathsf{rep}_2(i-1)$
09 $h_s := \mathsf{H}(\mathsf{ck}, \mathbf{x}_i)$
10 **return** $(h_s)_{s\in\{0,1\}^k}$

Algorithm $\mathsf{Com}(\mathsf{ck}, \mathbf{x})$
// generate leafs and tree
11 $(h_s)_{s\in\{0,1\}^k} := \mathsf{GenLeafs}(\mathsf{ck}, \mathbf{x})$
12 $((h_s)_{s\in\{0,1\}^l})_{l=0}^k := \mathsf{GenTree}(\mathsf{ck}, (h_s)_{s\in\{0,1\}^k})$
// return the root
13 **return** $\mathsf{com} := h_\epsilon$

Algorithm $\mathsf{Open}(\mathsf{ck}, \mathbf{x}, i)$
// recompute tree
14 $(h_s)_{s\in\{0,1\}^k} := \mathsf{GenLeafs}(\mathsf{ck}, \mathbf{x})$
15 $((h_s)_{s\in\{0,1\}^l})_{l=0}^k := \mathsf{GenTree}(\mathsf{ck}, (h_s)_{s\in\{0,1\}^k})$
// construct co-path
16 $s := \mathsf{rep}_2(i-1)$
17 **for** $l \in [k]$: $\bar{s}_l = s_{\le l} \oplus (0^{k-1}\|1)$
18 $\tau := (h_{\bar{s}_l})_{l=1}^k$
19 **return** τ

Algorithm $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, x, \tau)$
// compute leaf from pre-leaf
20 $s := \mathsf{rep}_2(i-1)$
21 $h_s := \mathsf{H}(\mathsf{ck}, x)$
// parse co-path
22 **for** $l \in [k]$: $\bar{s}_l = s_{\le l} \oplus (0^{k-1}\|1)$
23 **parse** $\tau = (h_{\bar{s}_l})_{l=1}^k$
// re-compute root and compare
24 **for** $l = k-1, \ldots, 0$:
25 $h_{s_{\le l}} := \mathsf{Compress}(h_{s_{\le l}\|0}, h_{s_{\le l}\|1})$
26 **if** $h_\epsilon = \mathsf{com}$: **return** 1
27 **return** 0.

Figure 3: The vector commitment $\mathsf{VC}^{\mathsf{mt}} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ representing the Merkle tree implementation in Plonky3. Here, $\ell = 2^k$ is a power of two, $\mathsf{H}\colon \mathcal{K}\times\mathcal{D} \to \mathcal{H}$ is a hash function, $\mathsf{P}\colon \mathcal{H}\times\mathcal{H} \to \mathcal{H}\times\mathcal{H}$ is a permutation, and $\mathsf{Trunc}\colon \mathcal{H}\times\mathcal{H} \to \mathcal{H}$ is the truncation map. The notation rep_2 denotes the representation of an integer as a bitstring.

4 Proof of Strong Position-Binding

We now give our first security proof for Plonky3's Merkle tree. Concretely, we show (strong) position-binding, where the leaf hash can be a standard model hash function. In particular, it does not need to be modeled as an idealized object such as a random oracle. We prove the following theorem.

Theorem 1 (Strong Position-Binding). *Consider the vector commitment* $\mathsf{VC}^{\mathsf{mt}}$ *in Figure 3. Assume that* $\mathsf{H}\colon \mathcal{K}\times\mathcal{D} \to \mathcal{H}$ *is collision-resistant and preimage-resistant (cf. Definitions 5 and 6) with* $|\mathcal{H}| \ge 2$ *and that* $\mathsf{P}\colon \mathcal{H}\times\mathcal{H} \to \mathcal{H}\times\mathcal{H}$ *is modeled as an ideal permutation (cf. Section 2). Assume that* $|\mathcal{H}| \ge \omega(\log\lambda)$.

Then, $\mathsf{VC}^{\mathsf{mt}}$ *is (strongly) position-binding (cf. Definition 2). Concretely, for any PPT algorithm* $\mathcal{A}$ *making at most* $q \le |\mathcal{H}|$ *oracle queries in total, there are PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2$ *with* $\mathbf{T}(\mathcal{B}_i) \approx \mathbf{T}(\mathcal{A})$, $i \in \{1,2\}$ *and*

$$\mathsf{Adv}^{\mathsf{posbind}}_{\mathsf{VC}^{\mathsf{mt}}}(\mathcal{A}) \le \frac{1.5q^2}{|\mathcal{H}|} + \mathsf{Adv}^{\mathsf{collres}}_{\mathsf{H}}(\mathcal{B}_1) + 2q \cdot \mathsf{Adv}^{\mathsf{preres}}_{\mathsf{H}}(\mathcal{B}_2).$$

Remark 4 (Concrete Security). Assuming that we also model H as an idealized object (concretely, a random oracle), the advantages against collision-resistance and preimage-resistance are bounded by $\binom{q_\mathsf{H}}{2}/|\mathcal{H}| \le 0.5q_\mathsf{H}^2/|\mathcal{H}|$ and $q_\mathsf{H}/|\mathcal{H}|$, respectively, via standard bounds, where q_H is the number of queries to H. Substituting these into Theorem 1, and letting $q_* := \max\{q, q_\mathsf{H}\}$, we obtain a bound:

$$\mathsf{Adv}^{\mathsf{posbind}}_{\mathsf{VC}^{\mathsf{mt}}}(\mathcal{A}) \le \frac{1.5q_*^2}{|\mathcal{H}|} + \underbrace{\frac{0.5q_*^2}{|\mathcal{H}|}}_{\mathsf{collres}} + \underbrace{\frac{2q_*^2}{|\mathcal{H}|}}_{\mathsf{preres}} = \frac{4q_*^2}{|\mathcal{H}|}.$$

That is, for any adversary that runs in time $\mathbf{T}(\mathcal{A}) \ge q_*$, we have $\mathsf{Adv}^{\mathsf{posbind}}_{\mathsf{VC}^{\mathsf{mt}}}(\mathcal{A}) \le 4q_*^2/|\mathcal{H}|$. In use cases of

Plonky3 (cf. Remark 2), we have $|\mathcal{H}| \approx 2^{8 \cdot 31}$. Then, the bound guarantees around $(8 \cdot 31 - \log 4)/2 = 123$ bits of security.

Proof. The proof of Theorem 1 is given throughout this section.

Security Game. We consider the strong position-binding game for adversary $\mathcal{A}$. Recall this game. First, a commitment key $\mathsf{ck} \xleftarrow{\$} \mathcal{K}$ is sampled, which will be used as the hashing key for H. The key is given to $\mathcal{A}$, and $\mathcal{A}$ also gets access to the ideal permutation oracles $\mathsf{P}(\cdot)$ and $\mathsf{P}^{-1}(\cdot)$. After some computation, $\mathcal{A}$ outputs $(\mathsf{com}, i, M, \tau, M', \tau')$, and it breaks position-binding if $(M, \tau) \neq (M', \tau')$ and $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M, \tau) = \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M', \tau') = 1$.

Assumptions and Notation. We denote by $x_i \in \mathcal{H} \times \mathcal{H}$ the ith forward query, i.e., the ith query to P. Similarly, we denote by $y_j \in \mathcal{H} \times \mathcal{H}$ the jth backward query, i.e., the jth query to P^{-1}. Without loss of generality, we can make the following assumptions:

- The adversary never makes the same query twice. That is, the x_i are pairwise distinct, and the y_j are pairwise distinct[6].
- The adversary never makes both a forward query and the corresponding backwards query. That is, there are no i, j such that $\mathsf{P}(x_i) = y_j$. In other words, for each input-output pair, the adversary decided which query to make.
- For each pair x, y such that $\mathsf{P}(x) = y$ is queried in the final verification that the game performs (i.e., in $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M, \tau)$ and $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M', \tau')$), the adversary already made a forward or backward query, i.e., $x = x_i$ for some i or $y = y_j$ for some j. In particular, the game itself does not need to make these queries explicitly.

These assumptions are without loss of generality: namely, we can always build a wrapper adversary around $\mathcal{A}$ that (1) remembers each query and answers repeated queries from its memory; (2) before making a backward (resp. forward) query y (resp. x) on behalf of $\mathcal{A}$, checks if y (resp. x) was the output of a previous forward (resp. backward) query, and then answers the query from memory[7]; and (3) runs $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M, \tau)$ and $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M', \tau')$ on its own before producing its output.

Events. We define the following events:

- Event Win: This event occurs if $\mathcal{A}$ breaks position-binding, i.e., its output $(\mathsf{com}, i, M, \tau, M', \tau')$ satisfies $(M, \tau) \neq (M', \tau')$ and $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M, \tau) = \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M', \tau') = 1$.
- Event $\mathsf{LeafColl}$: This event occurs if $M \neq M'$ but $\mathsf{H}(\mathsf{ck}, M) = \mathsf{H}(\mathsf{ck}, M')$. That is, the pre-leafs are distinct, but the leafs are the same.
- Forward Collisions:
  - Event $\mathsf{FwTrColl}_{i,j}$: This event occurs for $i \neq j$, if $\mathsf{Trunc}(\mathsf{P}(x_i)) = \mathsf{Trunc}(\mathsf{P}(x_j))$.
  - Event $\mathsf{FwTrColl}$: This event occurs if for any i, j the event $\mathsf{FwTrColl}_{i,j}$ occurs, i.e., $\mathsf{FwTrColl} = \bigvee_{i \neq j} \mathsf{FwTrColl}_{i,j}$.
- Backward-Forward Collisions:
  - Event $\mathsf{BaFwTrColl}^{\mathsf{left}}_{i,j}$: This event occurs if we have $\mathsf{P}^{-1}(y_i) = (\mathsf{Trunc}(\mathsf{P}(x_j)), h')$ for some $h' \in \mathcal{H}$.
  - Event $\mathsf{BaFwTrColl}^{\mathsf{right}}_{i,j}$: This event occurs if $\mathsf{P}^{-1}(y_i) = (h', \mathsf{Trunc}(\mathsf{P}(x_j)))$ for some $h' \in \mathcal{H}$.
  - Event $\mathsf{BaFwTrColl}$: This event occurs if for any i, j the event $\mathsf{BaFwTrColl}^{\mathsf{left}}_{i,j}$ or the event $\mathsf{BaFwTrColl}^{\mathsf{right}}_{i,j}$ occurs. That is, $\mathsf{BaFwTrColl} = \bigvee_{i \neq j} \mathsf{BaFwTrColl}^{\mathsf{left}}_{i,j} \vee \mathsf{BaFwTrColl}^{\mathsf{right}}_{i,j}$.

[6] It could however be that $x_i = y_j$ for some i, j.

[7] For instance, if $\mathcal{A}$ first makes the query $\mathsf{P}(x)$ and learns $y = \mathsf{P}(x)$, and then submits y to P^{-1}, the wrapper can simply return x from memory instead of really making the query $\mathsf{P}^{-1}(y)$

Top-Level Proof. Our goal is to upper bound the probability of Win. To this end, we first write

$$\begin{aligned}\Pr[\mathsf{Win}] \leq \quad & \Pr[\mathsf{LeafColl}] \\ + \ & \Pr[\mathsf{FwTrColl}] \\ + \ & \Pr[\mathsf{BaFwTrColl}] \\ + \ & \Pr[\mathsf{Win} \mid \neg(\mathsf{LeafColl} \vee \mathsf{FwTrColl} \vee \mathsf{BaFwTrColl})].\end{aligned}$$

Intuitively, we will upper bound the first term by collision-resistance of H, the second and third term using statistical arguments that rely on P being a random permutation, and the final term using preimage-resistance of H. We do so using the following lemmata, which are proven directly or in the following sections. Together, they yield an upper bound for the probability of Win, as desired.

Lemma 2. *There is a PPT algorithm $\mathcal{B}_1$ with $\mathbf{T}(\mathcal{B}_1) \approx \mathbf{T}(\mathcal{A})$ such that*

$$\Pr[\mathsf{LeafColl}] \leq \mathsf{Adv}^{\mathsf{collres}}_{\mathsf{H}}(\mathcal{B}_1).$$

Proof. The reduction $\mathcal{B}_1$ gets as input a hashing key and uses it as the commitment key $\mathsf{ck} \in \mathcal{K}$. With this key, it simulates the experiment for $\mathcal{A}$, and outputs M, M' once $\mathcal{A}$ produces its output. Clearly, if $\mathsf{LeafColl}$ occurs, then $\mathcal{B}_1$ breaks collision-resistance. The running time of $\mathcal{B}_1$ is roughly that of $\mathcal{A}$. □

Lemma 3. *We have* $\Pr[\mathsf{FwTrColl}] \leq q^2/(2|\mathcal{H}| + 2)$.

Proof. The proof is postponed to Section 4.1. □

Lemma 4. *We have* $\Pr[\mathsf{BaFwTrColl}] \leq q^2/(|\mathcal{H}| + 1)$.

Proof. The proof is postponed to Section 4.2. □

Lemma 5. *There is a PPT algorithm $\mathcal{B}_2$ with $\mathbf{T}(\mathcal{B}_2) \approx \mathbf{T}(\mathcal{A})$ such that*

$$\Pr[\mathsf{Win} \mid \neg(\mathsf{LeafColl} \vee \mathsf{FwTrColl} \vee \mathsf{BaFwTrColl})] \leq 2q \cdot \mathsf{Adv}^{\mathsf{preres}}_{\mathsf{H}}(\mathcal{B}_2).$$

Proof. The proof is given in Section 4.3. □

□

4.1 Proof of Lemma 3

Proof of Lemma 3. By a standard union bound over all $\binom{q}{2} \leq q^2/2$ pairs of queries $i \neq j$, we can simply fix a pair $i \neq j$, upper bound the probability of $\mathsf{FwTrColl}_{i,j}$, and then multiply the result by $q^2/2$. The probability will always be taken over the random choice of the permutation P.

Let $H := |\mathcal{H}|$, $H \geq 1$. We claim that the following holds:

$$\forall h \in \mathcal{H}\colon \quad \Pr[\mathsf{Trunc}(\mathsf{P}(x_i)) = h \mid \mathsf{Trunc}(\mathsf{P}(x_j)) = h] = \frac{1}{H+1}. \tag{1}$$

Using (1), we conclude the proof with

$$\begin{aligned}\Pr[\mathsf{FwTrColl}_{i,j}] &= \sum_{h \in \mathcal{H}} \Pr[\mathsf{Trunc}(\mathsf{P}(x_i)) = \mathsf{Trunc}(\mathsf{P}(x_j)) = h] \\ &= \sum_{h \in \mathcal{H}} \Pr[\mathsf{Trunc}(\mathsf{P}(x_i)) = h \mid \mathsf{Trunc}(\mathsf{P}(x_j)) = h] \cdot \Pr[\mathsf{Trunc}(\mathsf{P}(x_j)) = h] \\ &= \sum_{h \in \mathcal{H}} \Pr[\mathsf{Trunc}(\mathsf{P}(x_i)) = h \mid \mathsf{Trunc}(\mathsf{P}(x_j)) = h] \cdot \frac{1}{H} = \sum_{h \in \mathcal{H}} \frac{1}{H+1} \cdot \frac{1}{H} = \frac{1}{H+1}.\end{aligned}$$

Now, we argue that (1) holds. Fix $h \in \mathcal{H}$. Denote

$$A := |\{\text{Permutations } \mathsf{P} \text{ with } \mathsf{Trunc}(\mathsf{P}(x_j)) = h \wedge \mathsf{Trunc}(\mathsf{P}(x_i)) = h\}|$$

and

$$B := |\{\text{Permutations } \mathsf{P} \text{ with } \mathsf{Trunc}(\mathsf{P}(x_j)) = h\}|.$$

We now count A and B. We start with B. If the first component of $\mathsf{P}(x_j)$ is fixed to be h, we have H choices for the second component. For any such choice, we have now fixed one preimage-image pair, and the rest of P can freely be determined as a bijection on $H^2 - 1$ elements. Thus $B = H(H^2-1)!$. For counting A, say that the first component of both $\mathsf{P}(x_i)$ and $\mathsf{P}(x_j)$ is fixed to be h. Now, we can freely choose the second component of $\mathsf{P}(x_i)$ among H elements. For the second component of $\mathsf{P}(x_j)$, we now only have $H-1$ choices left, as we cannot make the same choice as for $\mathsf{P}(x_i)$. Now, for any such choice among $H(H-1)$ options, we have fixed exactly two preimage-image pairs, and can pick the rest of P as a bijection on H^2-2 elements. Thus $A = H(H-1)(H^2-2)!$ This means that

$$\begin{aligned}\Pr\left[\mathsf{Trunc}(\mathsf{P}(x_i)) = h \mid \mathsf{Trunc}(\mathsf{P}(x_j)) = h\right] &= \frac{A}{B} = \frac{H(H-1)(H^2-2)!}{H(H^2-1)!} \\ &= \frac{(H-1)(H^2-2)!}{(H^2-1)!} \\ &= \frac{H-1}{H^2-1} = \frac{1}{H+1}.\end{aligned}$$

This proves (1), finishing the proof of this lemma. □

4.2 Proof of Lemma 4

Proof of Lemma 4. For the proof, we will fix i, j and simply upper bound the probability of the event $\mathsf{BaFwTrColl}^{\mathsf{left}}_{i,j}$. By symmetry, the same upper bound holds for $\mathsf{BaFwTrColl}^{\mathsf{right}}_{i,j}$. Therefore, using a standard union bound, we get the upper bound on $\mathsf{BaFwTrColl}$ by multiplying our bound by $2 \cdot q^2/4 = q^2/2$, where the 2 is for left and right, the $q^2/4$ for the number of pairs of backward and forward queries[8] i, j. Note that $\mathsf{BaFwTrColl}^{\mathsf{left}}_{i,j}$ can be equivalently written as $\mathsf{Trunc}(\mathsf{P}^{-1}(y_i)) = \mathsf{Trunc}(\mathsf{P}(x_j))$.

Let $H = |\mathcal{H}|$. We follow the strategy from the proof of Lemma 3, and recommend that the reader reads this proof before (Section 4.1). Namely, exactly as we did there, it is sufficient to show

$$\forall h \in \mathcal{H}\colon \quad \Pr\left[\mathsf{Trunc}(\mathsf{P}^{-1}(y_i)) = h \mid \mathsf{Trunc}(\mathsf{P}(x_j)) = h\right] \leq \frac{2}{H+1}. \tag{2}$$

Denote

$$A := |\{\text{Permutations } \mathsf{P} \text{ with } \mathsf{Trunc}(\mathsf{P}^{-1}(y_i)) = h \wedge \mathsf{Trunc}(\mathsf{P}(x_j)) = h\}|$$

and

$$B := |\{\text{Permutations } \mathsf{P} \text{ with } \mathsf{Trunc}(\mathsf{P}(x_j)) = h\}|.$$

The value of B has been determined in the proof of Lemma 3 as $B = H(H^2-1)!$. We now compute an upper bound on A, by splitting into three cases:

Case 1. y_i and x_j start with h. The first component of $\mathsf{P}(x_j)$ and $\mathsf{P}^{-1}(y_i)$ are fixed to be h. Now, we can either have $y_i = \mathsf{P}(x_j)$, in which case we have $(H^2-1)!$ choices for the rest of the bijection. Or, we have $y_i \neq \mathsf{P}(x_j)$. In this case, there are $H-1$ choices for the second component of $\mathsf{P}(x_j)$ (second component of y_i not allowed) and $H-1$ choices for the second component of $\mathsf{P}^{-1}(y_i)$ (second component of x_j not allowed). Once these choices are fixed, there are $(H^2-2)!$ choices for the rest of the bijection. Thus, $A = (H^2-1)! + (H-1)(H-1)(H^2-2)! = 2H(H-1)(H^2-2)!$ in this case.

Case 2. Neither y_i nor x_j start with h. The first component of $\mathsf{P}(x_j)$ and $\mathsf{P}^{-1}(y_i)$ are fixed to be h. There are now H choices for the second component of $\mathsf{P}(x_j)$, and for each of those, H choices for the second component of $\mathsf{P}^{-1}(y_i)$. Because neither starts with h, this fixes exactly two preimage-image pairs for P, and we can freely fill the rest by a bijection on H^2-2 elements. Therefore, we get $A = H \cdot H \cdot (H^2-2)!$ in this case.

Case 3. Exactly one of y_i, x_j starts with h. By symmetry, we can simply assume that y_i starts with h, but x_j does not. The first component of $\mathsf{P}(x_j)$ and $\mathsf{P}^{-1}(y_i)$ are fixed to be h. We now have H choices for

[8] Observe that the number of backward and forward queries b and f satisfy $b + f = q$, and so their maximum product $b \cdot f$ is always at most $q^2/4$.

the second component of $\mathsf{P}^{-1}(y_i)$. For the second component of $\mathsf{P}(x_j)$, we have $H-1$ choices (we are not allowed to obtain y_i here). Then, exactly two preimage-image pairs for P are fixed, and we fill the rest with an arbitrary bijection on H^2-2 elements. Therefore, we get $A = H \cdot (H-1) \cdot (H^2-2)!$ in this case.

This bound in Case 3 is smaller than both the bound in Case 1 and Case 2. The bound in Case 1 is larger than the bound in Case 2 if and only if $H \geq 2$, which we assumed.

Considering all cases, this means that

$$\begin{aligned}\Pr\left[\mathsf{Trunc}(\mathsf{P}^{-1}(y_i)) = h \mid \mathsf{Trunc}(\mathsf{P}(x_j)) = h\right] = \frac{A}{B} &\leq \frac{2H(H-1)(H^2-2)!}{H(H^2-1)!} \\ &= \frac{2(H-1)}{H^2-1} = \frac{2}{H+1}.\end{aligned}$$

This proves (2), finishing the proof of this lemma. □

4.3 Proof of Lemma 5

We will prove Lemma 5 via a guessing argument, namely, a reduction that embeds its preimage-resistance challenge in a random backward query. To make this work, we first argue that $\mathsf{H}(\mathsf{ck}, M)$ or $\mathsf{H}(\mathsf{ck}, M')$ is the result of a backward query. Our reduction will then guess this particular query, and the side on which to embed the preimage-resistance challenge.

Lemma 6. *If* $\neg(\mathsf{LeafColl} \vee \mathsf{FwTrColl} \vee \mathsf{BaFwTrColl}) \wedge \mathsf{Win}$ *occurs, then there is a backward query* $(h_1, h_2) = \mathsf{P}^{-1}(y_j)$ *such that* $\{\mathsf{H}(\mathsf{ck}, M), \mathsf{H}(\mathsf{ck}, M')\} \cap \{h_1, h_2\} \neq \emptyset$.

Proof of Lemma 6. Assume that neither of the events $\mathsf{LeafColl}, \mathsf{FwTrColl}, \mathsf{BaFwTrColl}$ occurs, and that Win occurs. Let $\mathcal{H}^{\text{in}}$ and $\mathcal{H}^{\text{out}}$ be two distinguishable copies of $\mathcal{H}$, i.e., containing elements h^{in} and h^{out}, respectively, for each $h \in \mathcal{H}$. We define the following directed graph $G = (V, E)$, where $V = \mathcal{H} \cup (\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}}) \cup (\mathcal{H}^{\text{out}} \times \mathcal{H}^{\text{out}})$ consist of three partitions, with the following edges E:

- *Extension Edges.* For each $h \in \mathcal{H}$, we have an edge $(h, (h^{\text{in}}, r^{\text{in}}))$ (called a *right extension edge*) and an edge[9] $(h, (l^{\text{in}}, h^{\text{in}}))$ (*left extension edge*) for each $r, l \in \mathcal{H}$.
- *Permutation Edges.* For each $h_1, h_2 \in \mathcal{H}$ and for $(u_1, u_2) = \mathsf{P}(h_1, h_2)$, we have an edge $((h_1^{\text{in}}, h_2^{\text{in}}), (u_1^{\text{in}}, u_2^{\text{in}}))$.
- *Truncation Edges.* For each $h_1, h_2 \in \mathcal{H}$, we have an edge $((h_1^{\text{out}}, h_2^{\text{out}}), h_1)$.

That is, each vertex in $\mathcal{H}$ has out-degree $2|\mathcal{H}|$, each vertex in $\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}}$ has out-degree 1, and each vertex in $\mathcal{H}^{\text{out}} \times \mathcal{H}^{\text{out}}$ has out-degree 1. Edges from $\mathcal{H}$ go into $\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}}$, edges from $\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}}$ go into $\mathcal{H}^{\text{out}} \times \mathcal{H}^{\text{out}}$, and edges from $\mathcal{H}^{\text{out}} \times \mathcal{H}^{\text{out}}$ go back into $\mathcal{H}$.

Now, note that from the successful calls $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M, \tau) = \mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, M', \tau') = 1$, we can identify two paths in this directed graph: one path p going from $\mathsf{H}(\mathsf{ck}, M) \in \mathcal{H}$ to the Merkle root $\mathsf{com} \in \mathcal{H}$, and one path p' going from $\mathsf{H}(\mathsf{ck}, M') \in \mathcal{H}$ to $\mathsf{com} \in \mathcal{H}$. We make the following observations about these paths:

- *Same length.* Both paths have the same length L. Denote the paths by $(e_1, \ldots, e_L)$ and $(e'_1, \ldots, e'_L)$, where e_i, e'_i are edges in our graph.
- *Parallel Traversal.* The two paths go through the partitions of the graph simultaneously. That is, if $e_s \in (\mathcal{U}, \mathcal{W})$ for $\mathcal{U}, \mathcal{W} \in \{\mathcal{H}, (\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}}), (\mathcal{H}^{\text{out}} \times \mathcal{H}^{\text{out}})\}$, then $e'_s \in (\mathcal{U}, \mathcal{W})$ as well.
- *Permutation Edges Queried.* By our assumption that each permutation call in Ver has been made, we know that each permutation edge that is on these paths has the form $(x_j, \mathsf{P}(x_j))$ or $(\mathsf{P}^{-1}(y_j), y_j)$. We say that this is a *forward-queried permutation edge* or a *backward-queried permutation edge*, respectively.

[9] h^{in} is the distinguishable copy of h in the set $\mathcal{H}^{\text{in}}$.

- *Partially Determined Extension Edges.* If these paths traverse an extension edge (denoted e_s, e'_s) at some step s, then it is determined from i and s whether this is a left or a right extension edge. In particular, if we traverse the two paths in parallel, then at the same time, when they traverse an extension edge, they either both take a left extension edge or both take a right extension edge.

- *Determined $\mathcal{H}^{in}$-order.* Similar to the previous point, it is determined from i and s which component (left or right) of an $(\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}})$-vertex that the paths are visiting comes from the provided opening.

- *Divergence Point.* Traversing the two paths in parallel, we now claim that there must be at least one step s_0 that visits the $(\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}})$ partition in which the two paths are not visiting the same vertex. Denote the corresponding edges ending in two different vertices of $(\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}})$ by e_{s_0}, e'_{s_0}. To see that this holds, first note that by definition of Win and $\neg\mathsf{LeafColl}$, we have $(\mathsf{H}(\mathsf{ck}, M), \tau) \neq (\mathsf{H}(\mathsf{ck}, M'), \tau')$. Second, note that if all vertices visited in the $(\mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}})$ were the same for both paths, we would have $(\mathsf{H}(\mathsf{ck}, M), \tau) = (\mathsf{H}(\mathsf{ck}, M'), \tau')$ due to the determined $\mathcal{H}^{\text{in}}$-order property.

- *Meeting Point.* Again, traversing the two paths in parallel *starting from the divergence point*, we can find the first vertex in $\mathcal{H}$ in which both paths meet. This vertex must exist, as they meet in the root, and this vertex is not the very first vertex on the path, as it must be after the divergence point. More precisely, find the minimum $s \in [L], s > s_0$ such that $e_s, e'_s \in (\mathcal{H}^{\text{out}} \times \mathcal{H}^{\text{out}}) \times \mathcal{H}$ (i.e., e_s, e'_s are truncation edges) and the end vertex of e_s, e'_s is the same. Denote this vertex by $\mathsf{par}^* \in \mathcal{H}$, and the corresponding $s \in [L]$ by s^*. Note that $s^* \geq 3$, as the paths first traverse an extension and a permutation edge.

Now, we consider three cases. For the first two, we will derive a contradiction, i.e., they are impossible. They are visualized in Figure 4. For the third one, we will then derive the claimed statement.

Case 1. $e_{s^*} \neq e'_{s^*}$ and e_{s^*-1} and e'_{s^*-1} are both forward-queried permutation edges. Note that in this case, the starting points of e_{s^*}, e'_{s^*} must differ. This is because the end points are the same (namely, par^*). These starting points are the end points of e_{s^*-1}, e'_{s^*-1}, respectively. Those are permutation edges, and as P is a function, it means that also the starting points of e_{s^*-1}, e'_{s^*-1} must be distinct. Thus, we have two edges with distinct starting points which are both forward-queried, but after truncation, we end up in the same value. See Figure 4, left. In particular, this means that $\mathsf{FwTrColl}$ occurs, a contradiction.

Case 2. $e_{s^*} = e'_{s^*}$. In this case, we first observe that we also must have $e_{s^*-1} = e'_{s^*-1}$. This is because those are permutation edges and P is a permutation. Precisely, the start vertex of $e_{s^*} = e'_{s^*}$ is the end vertex of both e_{s^*-1}, e'_{s^*-1}. Because P is a permutation (and in particular injective), this implies that the start vertex of e_{s^*-1}, e'_{s^*-1} is also the same. In particular, we now know that $s_0 < s^* - 2$. Denote this start vertex of both e_{s^*-1}, e'_{s^*-1} by $(h_1^{\text{in}}, h_2^{\text{in}}) \in \mathcal{H}^{\text{in}} \times \mathcal{H}^{\text{in}}$. We are in the situation in Figure 4, right. Now, we claim that $e_{s^*-2} = e'_{s^*-2}$ as well, which is a contradiction as s^* was chosen minimally. To see why this holds, recall the *partially determined extension edges* property, which tells us that the extension edges e_{s^*-2}, e'_{s^*-2} are either both left or both right extension edges. Assume that they are both left extension edges (the other case follows in the same way). Then, the starting point of both must be $h_2 \in \mathcal{H}$. In particular, this means that $e_{s^*-2} = e'_{s^*-2}$ as claimed. Therefore, the contradiction follows and this case is not possible.

Case 3. $e_{s^*} \neq e'_{s^*}$ and at least one of e_{s^*-1} and e'_{s^*-1} is a backward-queried permutation edge. We visualize this case in Figure 5. Say without loss of generality that e_{s^*-1} on path p is a backward-queried permutation edge. We claim that this implies that *all* permutation-edges on p must be backward queried. To see this, assume this does not hold. Then there must be a situation as in Figure 5, right, namely, there must be an index k such that e_{s^*-k} is a forward-queried permutation edge but e_{s^*-k+3} is a backward-queried permutation edge. In this case, note that we have a contradiction to $\neg\mathsf{BaFwTrColl}$. Thus, we know that permutation edges on p must be backward-queried. This in particular holds for the first permutation edge on p, which is e_2. This means that there is a backward query $(h_1, h_2) = \mathsf{P}^{-1}(y_j)$ such that $\mathsf{H}(\mathsf{ck}, M) \in \{h_1, h_2\}$, which implies our lemma. Note that if we had assumed that e'_{s^*-1} on path p' is a backward-queried permutation edge, we would have concluded that $\mathsf{H}(\mathsf{ck}, M') \in \{h_1, h_2\}$, which also implies our lemma. □

Proof of Lemma 5. The reduction $\mathcal{B}_2$ gets as input a hashing key, which it takes as the commitment key $\mathsf{ck} \in \mathcal{K}$, and it gets an image $h^* \in \mathcal{H}$. The goal of $\mathcal{B}_2$ is to find a preimage of h^* under $\mathsf{H}(\mathsf{ck}, \cdot)$. The

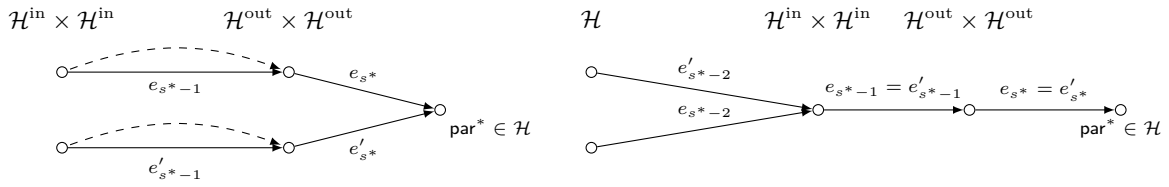


Figure 4: Case 1 (left) and Case 2 (right) in the proof of Lemma 6. In Case 1, the two permutation edges are both forward-queried permutation edges, indicated by the two dashed edges.

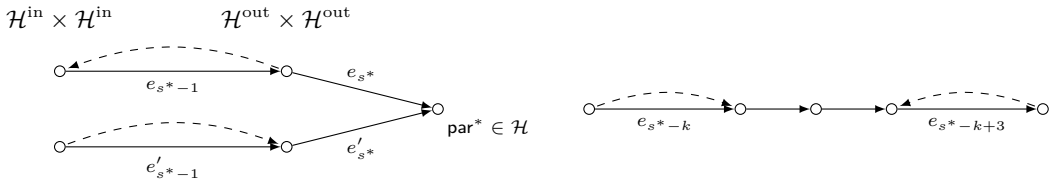


Figure 5: Case 3 in the proof of Lemma 6. Left: the situation in which we start the argument. Right: we arrive at a case where a forward-queried permutation edge is followed by a backward-queried one on path p. This is shown to be impossible.

reduction samples a random $j^* \xleftarrow{\$} [q]$ and $b^* \xleftarrow{\$} \{0,1\}$, and starts running the position-binding $\mathcal{A}$ on input ck, and with access to $\mathsf{P}(\cdot)$, $\mathsf{P}^{-1}(\cdot)$ which are implemented in the standard lazy manner while respecting the fact that P is a bijection. The only exception is the j^*-th query, if it is a $\mathsf{P}^{-1}(\cdot)$ query, denoted by $\mathsf{P}^{-1}(y)$, which the reduction answers as follows:

1. If $b^* = 0$:
   (a) Let Free denote the set of $h \in \mathcal{H}$ such that (h^*, h) has no image under P yet. If $\mathsf{Free} = \emptyset$, $\mathcal{B}_2$ aborts its execution.
   (b) Otherwise, sample $h \xleftarrow{\$} \mathsf{Free}$, and define and return $\mathsf{P}^{-1}(y) := (h^*, h)$.
2. If $b^* = 1$:
   (a) Let Free denote the set of $h \in \mathcal{H}$ such that (h, h^*) has no image under P yet. If $\mathsf{Free} = \emptyset$, $\mathcal{B}_2$ aborts its execution.
   (b) Otherwise, sample $h \xleftarrow{\$} \mathsf{Free}$, and define and return $\mathsf{P}^{-1}(y) := (h, h^*)$.

Finally, when the adversary outputs $(\mathsf{com}, i, M, \tau, M', \tau')$ and Win occurs, the reduction checks if $\mathsf{H}(\mathsf{ck}, M) = h^*$ or $\mathsf{H}(\mathsf{ck}, M') = h^*$, and outputs M or M' as a solution if so.

Let us analyze this reduction. Clearly, its running time is about the same as that of $\mathcal{A}$. Further, assuming it does not abort, one can see that the view provided to $\mathcal{A}$ is exactly as in the real game, and in particular independent of j^* and b^*. Note that because $|\mathcal{H}|$ is at least the number of queries (by our assumptions on $\mathcal{A}$), the set Free cannot be empty and therefore $\mathcal{B}_2$ never aborts. Finally, conditioning on $\neg(\mathsf{LeafColl} \vee \mathsf{FwTrColl} \vee \mathsf{BaFwTrColl})$, we know by Lemma 6 that $\mathsf{H}(\mathsf{ck}, M)$ or $\mathsf{H}(\mathsf{ck}, M')$ coincide at least one component (left or right) of at least one backward query. If the reduction guesses this query and component correctly by sampling the right q^*, b^*, and Win occurs, we know that it breaks preimage-resistance. This happens with probability $1/(2q)$, which finishes the proof of this lemma. □

Remark 5 (The Sparse Case). It is easy to see that the proof directly generalizes to the sparse case (cf. Remark 3). Namely, the events $\mathsf{LeafColl}, \mathsf{FwTrColl}$, and $\mathsf{BaFwTrColl}$ can be bounded in exactly the same way (they do not depend on the tree structure). For Lemma 5, the proof also directly carries over, where bitstrings $s \in \{0,1\}^{\leq k}$ indexing leafs take the role of the index i.

5 Proof of Strong Extractability

We now extend our proof of (strong) position-binding to also show (strong) extractability. This assumes that H is modeled as a random oracle. Namely, we prove the following theorem.

Theorem 2 (Strong Extractability). *Consider the vector commitment $\mathsf{VC}^{\mathsf{mt}}$ in Figure 3. Assume that $\mathsf{H}\colon \mathcal{K} \times \mathcal{D} \to \mathcal{H}$ is modeled as a random oracle with $|\mathcal{H}| \geq 2$, and that $\mathsf{P}\colon \mathcal{H} \times \mathcal{H} \to \mathcal{H} \times \mathcal{H}$ is modeled as an ideal permutation (cf. Section 2). Assume that $|\mathcal{H}| \geq \omega(\log \lambda)$.*

Then, $\mathsf{VC}^{\mathsf{mt}}$ is (strongly) extractable (cf. Definition 3). Concretely, there is a PPT algorithm Ext such that for any PPT algorithm $\mathcal{A}$ making at most $q \leq |\mathcal{H}|$ oracle queries to $\mathsf{P}, \mathsf{P}^{-1}$ in total, and at most q_{H} many queries to H, we have

$$\mathsf{Adv}^{\mathsf{extract}}_{\mathsf{VC}^{\mathsf{mt}},\mathsf{Ext}}(\mathcal{A}) \leq \frac{0.5q_{\mathsf{H}}^2 + q_{\mathsf{H}}\ell + 2q_{\mathsf{H}}q}{|\mathcal{H}|} + \frac{q^2+q}{|\mathcal{H}|-1} + \frac{1.5q^2}{|\mathcal{H}|+1}.$$

Remark 6 (Concrete Security). Simplifying the bound a bit using appropriate upper bounds[10], for any adversary that runs in time $\mathbf{T}(\mathcal{A}) \geq q_* := \max\{q, q_{\mathsf{H}}\}$, we have $\mathsf{Adv}^{\mathsf{extract}}_{\mathsf{VC}^{\mathsf{mt}}}(\mathcal{A}) \lesssim 6q_*^2/|\mathcal{H}|$. In use cases of Plonky3 (cf. Remark 2), we have $|\mathcal{H}| \approx 2^{8\cdot 31}$. The bound gives us around $(8 \cdot 31 - \log 6)/2 \approx 122.71$ bits of security.

Proof. The proof is given throughout this section, and we recommend that the reader reads our position-binding proof (proof of Theorem 1) first. As we are working with a random oracle, we can assume that the key ck is fixed and omit it as an explicit input to H. As the proof works for any fixed key, it also works for random keys.

Security Game. We recall the strong extractability game: the adversary $\mathcal{A}$ gets the commitment key ck. It also gets access to oracles H, P, P^{-1}. It outputs a commitment com. Then, the extractor (specified below) is given the ordered list of all oracle queries and responses $\mathcal{Q}$ made so far and the commitment com. It needs to produce a vector $\mathbf{x} \in \mathcal{D}^\ell$ and openings $\tau_i, i \in [\ell]$ and the adversary $\mathcal{A}$ continues its execution with the same oracles and input $\mathbf{x}$ and $(\tau_i)_{i\in[\ell]}$. It finally outputs a position $i \in [\ell]$, a symbol $x \in \mathcal{D}$ and an opening proof τ. It wins if the opening verifies (note that verification may involve further queries that are not in $\mathcal{Q}$) and $(x, \tau) \neq (\mathbf{x}_i, \tau_i)$. That is, the provided opening is valid and inconsistent with the extracted one.

Assumptions and Notation. We make the same assumptions without loss of generality and use the same notation as in the proof of Theorem 1. Namely, $x_i \in \mathcal{H} \times \mathcal{H}$ denotes the ith forward query and $y_j \in \mathcal{H} \times \mathcal{H}$ denotes the jth backward query, and we assume:

- The adversary never makes the same query twice. This also implies that $\mathcal{Q}$ contains no duplicates.
- The adversary never makes both a forward query and the corresponding backwards query. That is, there are no i, j such that $\mathsf{P}(x_i) = y_j$.
- For each pair x, y such that $\mathsf{P}(x) = y$ is queried in the final verification that the game performs, the adversary already made a forward or backward query, i.e., $x = x_i$ for some i or $y = y_j$ for some j. Hence, the game itself does not need to make these queries explicitly. Note, however, that these queries are not necessarily in $\mathcal{Q}$, as they could be made after $\mathcal{A}$ received the extracted $\mathbf{x}$ (we call this late queries below).

Additionally, we make the following assumptions, related to the random oracle H:

- The adversary never makes an H query twice. Together with the above, this implies that $\mathcal{Q}$ contains no duplicates.
- All H queries in the final verification have already been made by $\mathcal{A}$. Again, this does not mean that these queries are necessarily in $\mathcal{Q}$.

As in the proof of Theorem 1, we can argue that these assumptions are without loss of generality, by simply building a wrapper adversary around $\mathcal{A}$.

[10] Concretely: upper bound ℓ by q_*, ignore the isolated q term, assume all denominators are $|\mathcal{H}|$.

Late Queries. We will use the following terminology: A *late query* is a query that has been made after $\mathcal{A}$ received the extracted $\mathbf{x}$. By our assumptions above, this also means that this query has not been made before. In particular, these queries are not recorded in the list $\mathcal{Q}$ passed to the extractor.

Extractor. We now specify our extractor Ext. As $\mathcal{A}$ has access to three oracles H, P, P^{-1}, each entry in $\mathcal{Q}$ has one of the following forms:

- (x, H, h) for $x \in \mathcal{D}, h \in \mathcal{H}$. This indicates that $\mathcal{A}$ queried $\mathsf{H}(x)$ and obtained h.
- (x, P, y) for $x, y \in \mathcal{H} \times \mathcal{H}$. This indicates that $\mathcal{A}$ queried $\mathsf{P}(x)$ and obtained y.
- (y, P^{-1}, x) for $x, y \in \mathcal{H} \times \mathcal{H}$. This indicates that $\mathcal{A}$ queried $\mathsf{P}^{-1}(y)$ and obtained x.

We assume without loss of generality that each triple is contained only once. Otherwise the extractor may first eliminate duplicates. On input com and $\mathcal{Q}$, our extractor Ext works as follows:

1. Set $h_\epsilon := \mathsf{com} \in \mathcal{H}$, where ϵ denotes the empty string.
2. Initialize a queue $L := \{h_\epsilon\}$.
3. While $L \neq \emptyset$:
   (a) Pop h_s from L, for $s \in \{0,1\}^{\leq k}$, $h_s \in \mathcal{H}$.
   (b) If $|s| = k$, skip this iteration. Note that h_s has been removed from L now.
   (c) Find $(h_{s\|0}, h_{s\|1}, y) \in \mathcal{H}^3$ s.t. $((h_{s\|0}, h_{s\|1}), \mathsf{P}, (h_s, y)) \in \mathcal{Q}$.
   (d) Consider the following cases:
   - If there are multiple such triples, set $\mathsf{BadLeaves} := 1$ and abort.
   - If there is no such triple, set $h_{s\|w} := \perp$ for all $w \in \{0,1\}^{k-|s|}$.
   - If there is exactly one such triple, then push both $h_{s\|0}, h_{s\|1}$ into L.

   Note: now, all the leaves h_s, $s \in \{0,1\}^k$, are specified (some could be $\perp$).
4. Set all $\mathbf{x}_i := \perp$, $i \in [\ell]$.
5. For each $i \in [\ell]$ with $h_s \neq \perp$ for $s := \mathsf{rep}_2(i-1)$:
   (a) Find x such that $(x, \mathsf{H}, h_s) \in \mathcal{Q}$.
   (b) Consider the following cases:
   - If there are two such x, set $\mathsf{BadPreLeaves} := 1$ and abort.
   - If there is no such x, keep $\mathbf{x}_i = \perp$.
   - If there is exactly one, set $\mathbf{x}_i := x$.
6. Construct the co-paths as Open does. That is, for each $i \in [\ell]$:
   - If $\mathbf{x}_i = \perp$, set $\tau_i := \perp$.
   - Else, set $s := \mathsf{rep}_2(i-1)$, $\bar{s}_l = s_{\leq l} \oplus (0^{k-1}\|1)$, $l \in [k]$, and $\tau_i := (h_{\bar{s}_l})_{l=1}^k$.

   Note that all these $h_{\bar{s}_l}$ are defined and not $\perp$.
7. Return $\mathbf{x}, (\tau_i)_{i \in [\ell]}$.

It is easy to see that the extractor runs in polynomial time.

Events. We now define some events, extending the events that we defined in the proof of Theorem 1:

- Event Win: This event occurs if $\mathcal{A}$ breaks extractability, i.e., if $\mathsf{Ver}(\mathsf{ck}, \mathsf{com}, i, x, \tau) = 1 \wedge x \neq \mathbf{x}_i$.
- Event $\mathsf{ExtAbort}$: This event occurs if the extractor aborts, i.e., if $\mathsf{BadLeaves} = 1$ or $\mathsf{BadPreLeaves} = 1$.
- Bad events:
  - Event $\mathsf{FwTrColl}$, $\mathsf{BaFwTrColl}$: exactly as in the proof of Theorem 1.
  - Event HColl: This event occurs if there are two queries $\hat{x} \neq \hat{x}'$ to H resulting in the same hash, i.e., with $\mathsf{H}(\hat{x}) = \mathsf{H}(\hat{x}')$.
  - Event $\mathsf{HashHitsBw}$: This event occurs if a query $\mathsf{H}(x)$ outputs h and a backwards query $\mathsf{P}^{-1}(y_j)$ outputs (h, h') or (h', h) for some $h' \in \mathcal{H}$. That is, a hash query hits the output of a backward query.
  - Event $\mathsf{LateTargetFw}$: This event occurs if a *late* forward query to P of the form $\mathsf{P}(x_i)$ satisfies $\mathsf{Trunc}(\mathsf{P}(x_i)) = h_s$ for some $s \in \{0,1\}^{<k}$.
  - Event $\mathsf{LateTargetH}$: This event occurs if a *late* query $\mathsf{H}(\hat{x})$ satisfies $\mathsf{H}(\hat{x}) = h_s$ for some $s \in \{0,1\}^{k}$, i.e., hits a leaf extracted by Ext.
  - Event Bad: This event occurs if one of the previous bad events (not including $\mathsf{Win}, \mathsf{ExtAbort}$) occurs, i.e., it is just a short name for $\mathsf{FwTrColl} \vee \mathsf{BaFwTrColl} \vee \mathsf{HColl} \vee \mathsf{LateTargetFw} \vee \mathsf{LateTargetH} \vee \mathsf{HashHitsBw}$.

Top-Level Proof. Now, our goal is to bound the probability of Win. To do so, we first bound it as follows:

$$\begin{aligned}\Pr[\mathsf{Win}] \leq \quad & \Pr[\mathsf{Bad}] \\ + \ & \Pr[\mathsf{ExtAbort} \mid \neg\mathsf{Bad}] \\ + \ & \Pr[\mathsf{Win} \wedge \mathbf{x}_i \neq \bot \mid \neg(\mathsf{Bad} \vee \mathsf{ExtAbort})] \\ + \ & \Pr[\mathsf{Win} \wedge \mathbf{x}_i = \bot \mid \neg(\mathsf{Bad} \vee \mathsf{ExtAbort})].\end{aligned}$$

Of course, we can rewrite the probability of Bad as

$$\begin{aligned}\Pr[\mathsf{Bad}] \leq \quad & \Pr[\mathsf{HColl}] + \Pr[\mathsf{FwTrColl}] + \Pr[\mathsf{BaFwTrColl}] \\ + \ & \Pr[\mathsf{LateTargetFw}] + \Pr[\mathsf{LateTargetH}] + \Pr[\mathsf{HashHitsBw}].\end{aligned}$$

We bound each of these terms individually below. Recall that events $\mathsf{FwTrColl}$, $\mathsf{BaFwTrColl}$ are bounded in the proof of Theorem 1, precisely, in Lemmata 3 and 4. For the remaining terms, we state the following lemmas, deferring the proofs for more complex cases to the following subsections.

Lemma 7. *We have* $\Pr[\mathsf{ExtAbort} \mid \neg\mathsf{Bad}] = 0$.

Proof. It can easily be seen that we have the following implications: $\neg\mathsf{FwTrColl} \implies \mathsf{BadLeaves} = 0$ and $\neg\mathsf{HColl} \implies \mathsf{BadPreLeaves} = 0$. Consequently, if $\neg\mathsf{Bad}$, then $\neg(\mathsf{FwTrColl} \vee \mathsf{HColl})$, and then Ext does not abort, and so the probability is zero. □

Lemma 8. *We have* $\Pr[\mathsf{Win} \wedge \mathbf{x}_i \neq \bot \mid \neg(\mathsf{Bad} \vee \mathsf{ExtAbort})] = 0$.

Proof. Condition on $\neg\mathsf{Bad}$ and $\neg\mathsf{ExtAbort}$. Recall that for strong extractability, the event Win implies that $\mathcal{A}$ outputs a valid opening (x, τ) for position i such that $(x, \tau) \neq (\mathbf{x}_i, \tau_i)$, where $(\mathbf{x}_i, \tau_i)$ are the value and opening extracted by Ext. If $\mathsf{Win} \wedge \mathbf{x}_i \neq \bot$ and $\neg\mathsf{ExtAbort}$, we can view the combination of Ext and $\mathcal{A}$ as an adversary winning the strong position-binding game. More formally, one could build a reduction that runs in the strong position-binding game, and simulates the extractability game for $\mathcal{A}$, while forwarding access to all oracles and maintaining the list $\mathcal{Q}$. When $\mathcal{A}$ outputs the commitment, the reduction provides it together with $\mathcal{Q}$ to the extractor Ext. Since Ext does not abort and $\mathbf{x}_i \neq \bot$, the extractor outputs a valid pair $(\mathbf{x}_i, \tau_i)$. Because Win occurred, the adversary's output (x, τ) satisfies $(x, \tau) \neq (\mathbf{x}_i, \tau_i)$. Thus, the reduction holds two valid openings for the same commitment and index that

differ in value or path. This constitutes a break of strong position-binding. Now, we apply Lemma 6 to this reduction. We can do this because we condition on ¬Bad, which implies that FwTrColl, BaFwTrColl, and LeafColl (as defined in the proof of Theorem 1) do not occur. The lemma then guarantees that there is a backward query with an output component that is the result of a hash query, i.e., that HashHitsBw occurs. However, this is a contradiction, as we conditioned on ¬Bad (which includes ¬HashHitsBw). Consequently, the probability is zero. □

Lemma 9. *We have* $\Pr[\mathsf{Win} \wedge \mathbf{x}_i = \bot \mid \neg(\mathsf{Bad} \vee \mathsf{ExtAbort})] = 0$.

Proof. Condition on ¬Bad and ¬ExtAbort. Let $s = \mathsf{rep}_2(i-1)$ be the bitstring that represents the ith leaf. For each prefix w of s, let $\hat{h}_w \in \mathcal{H}$ be the corresponding hash value in the tree computed during verification of the opening (x, τ) that the adversary provided. This includes values provided as part of τ, and the values recomputed during verification. For instance, $\hat{h}_s = \mathsf{H}(x)$ and $\hat{h}_\epsilon = \mathsf{com} = h_\epsilon$. Define $s^* \in \{0,1\}^{\leq k}$ to be the longest prefix of s such that $h_{s^*} \neq \bot$ *and* $h_{s^*} = \hat{h}_{s^*}$. Note that such an s^* must exist, as $s^* = \epsilon$ satisfies this condition. Because all queries for the verification call have been made (either in forward or backward direction), we know that the relation

$$h_{s^*} = \hat{h}_{s^*} = \mathsf{Trunc}(\mathsf{P}(\hat{h}_{s^*\|0}, \hat{h}_{s^*\|1})) \tag{3}$$

has been established either via the forward query $\mathsf{P}(\hat{h}_{s^*\|0}, \hat{h}_{s^*\|1})$ or by making a backward query with h_{s^*} as left or right input. Then, it must be that one of the following cases holds:

- *Case 1.* $s^* = s$. Then we have $\mathsf{H}(x) = h_s$ as part of verification, but $\mathbf{x}_i = \bot$ by assumption. Hence, this hash query must be a late query. Thus, LateTargetH occurs. Contradiction.
- *Case 2.* $s^* \neq s$ *and* (3) *comes from a forward query.* We consider two sub-cases:
  - *Case 2a. This forward query* $\mathsf{P}(\hat{h}_{s^*\|0}, \hat{h}_{s^*\|1})$ *is a late query.* Then LateTargetFw occurs. Contradiction.
  - *Case 2b. This forward query* $\mathsf{P}(\hat{h}_{s^*\|0}, \hat{h}_{s^*\|1})$ *is not a late query.* Then it must be part of the list $\mathcal{Q}$ by definition of forward queries. Because the extractor did not abort (in particular ¬FwTrColl), it must be that the extractor has defined $(h_{s^*\|0}, h_{s^*\|1}) := (\hat{h}_{s^*\|0}, \hat{h}_{s^*\|1})$. But that contradicts the choice of s^*. Namely, the one-bit longer prefix of s (which is $s^*\|0$ or $s^*\|1$) would also satisfy the constraint above. Contradiction.
- *Case 3.* $s^* \neq s$ *and* (3) *comes from a backward query.* Exactly as in the position-binding proof (precisely, Lemma 6), we can argue that the leaf $\mathsf{H}(x)$ computed during verification must then be the part of an output of a backwards query. This argument uses that we conditioned on ¬Bad, and in particular on ¬BaFwTrColl. But then HashHitsBw occurs. Contradiction.

All cases lead to a contradiction, which means that the probability is zero. □

Lemma 10. *We have* $\Pr[\mathsf{HColl}] \leq 0.5 q_\mathsf{H}^2 / |\mathcal{H}|$.

Proof. This follows by a standard birthday bound, and using $\binom{q_\mathsf{H}}{2} \leq q_\mathsf{H}^2/2$. □

Lemma 11. *We have* $\Pr[\mathsf{HashHitsBw}] \leq 2 q_\mathsf{H} q / |\mathcal{H}|$.

Proof. Fix any hash query and any backwards query. The probability that the output of the hash query is the same as the left component of the output of the backwards query is $1/|\mathcal{H}|$. Similarly, the probability that the output of the hash query is the same as the right component of the output of the backwards query is $1/|\mathcal{H}|$. Therefore, the probability that HashHitsBw occurs for this particular pair is at most $2/|\mathcal{H}|$. There are at most $q_\mathsf{H} q$ such pairs, so a union bound yields the stated lemma. □

Lemma 12. *We have* $\Pr[\mathsf{LateTargetH}] \leq q_\mathsf{H} \ell / |\mathcal{H}|$.

Proof. We use a union bound. Fix any $s \in \{0,1\}^k$ with $h_s \neq \bot$. There are at most $2^k = \ell$ of these. Fix any late query to H. There are at most q_H of these. The probability that the output of this late query hits h_s is $1/|\mathcal{H}|$. Via the union bound over all $\ell \cdot q_\mathsf{H}$ pairs, we get the bound. □

Lemma 13. *We have* $\Pr[\mathsf{LateTargetFw}] \leq (q^2+q)/(|\mathcal{H}|-1)$.

Proof. To analyze $\mathsf{LateTargetFw}$, we first define the following events, where we denote the jth forward query x_j as $x_j = (x_{j,0}, x_{j,1}) \in \mathcal{H} \times \mathcal{H}$:

- Event $\mathsf{FwChain}^b_{i,j}$ for $i > j$ and $b \in \{0,1\}$: This event occurs if $\mathsf{Trunc}(\mathsf{P}(x_i)) = x_{j,b}$. That is, the truncated output of the ith query hits component b of the jth query, which occurred before.
- Event $\mathsf{LateHitRoot}_i$: This event occurs if $\mathsf{Trunc}(\mathsf{P}(x_i)) = h_\epsilon$ for a late forward query x_i. That is, the truncated output of the ith query hits the provided commitment $\mathsf{com} = h_\epsilon$.

One can see easily that if $\mathsf{LateTargetFw}$ occurs, then $\mathsf{FwChain}^b_{i,j}$ must occur for some $i > j$ and $b \in \{0,1\}$, or $\mathsf{LateHitRoot}_i$ occurs. Consequently, we get

$$\Pr[\mathsf{LateTargetFw}] \leq \left(\sum_{b\in\{0,1\}} \sum_{i>j} \mathsf{FwChain}^b_{i,j}\right) + \left(\sum_i \mathsf{LateHitRoot}_i\right).$$

In the left sum, there are at most $2\binom{q}{2} \leq q^2$ terms. In the right sum, there are at most q terms. We bound each term individually. We start with the terms in the left sum. To this end, fix $i > j$ and $b \in \{0,1\}$. Consider the moment in time where the ith forward query x_i is made, and condition on everything that happened before. In particular, there is a set of already *occupied* images $\mathsf{Occ} \subseteq \mathcal{H} \times \mathcal{H}$ that already have preimages under P. Clearly, $|\mathsf{Occ}| \leq q \leq |\mathcal{H}|$. We need to bound the probability that $\mathsf{Trunc}(\mathsf{P}(x_i)) = x_{j,b}$. Because the ith query is made after the jth query, we know that $x_{j,b}$ is fixed. In total, we have $|\mathcal{H}|^2 - |\mathsf{Occ}|$ choices for $\mathsf{P}(x_i)$. There are at most $\mathcal{H}$ cases in which the truncation hits $x_{j,b}$, each given by one choice of $x_{j,1-b}$. Therefore, the probability of $\mathsf{Trunc}(\mathsf{P}(x_i)) = x_{j,b}$ is at most $|\mathcal{H}|/(|\mathcal{H}|^2 - |\mathsf{Occ}|)$. By our bounds on $|\mathsf{Occ}|$ above, this is at most $|\mathcal{H}|/(|\mathcal{H}|^2 - |\mathcal{H}|) = 1/(|\mathcal{H}|-1)$. This finishes the proof of this lemma.

Now, for the terms in the right sum, exactly the same argument works, where h_ϵ takes the role of $x_{j,b}$. □

□

Remark 7 (The Sparse Case). Similar to our proof for position-binding, the proof trivially extends to the sparse case (cf. Remark 3). This assumes that the tree topology is fixed and thus known to the extractor.

6 Counterexample for Dependent Leaf Hashing

The proofs of strong position-binding (Section 4) and strong extractability (Section 5) require independence between H and P. As a matter of fact, it is possible to construct a (contrived) hash function H based on P for which the proofs fall apart. For ease of presentation, we consider an instantiation on bits. In other words, we take $\mathcal{H} = \{0,1\}^n$, and consider a permutation $\mathsf{P}\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$ and a hash function $\mathsf{H}\colon \{0,1\}^n \to \{0,1\}^n$ (ignoring the key input ck for simplicity; it may be implicit as first part of the message):

$$\mathsf{H}(m) = \begin{cases} \mathsf{LBits}_n\left(\mathsf{P}^{-1}(0^n \| \mathsf{LBits}_{n-1}(m) \| 0)\right), & \text{if } \mathsf{RBits}_1(m) = 0, \\ \mathsf{RBits}_n\left(\mathsf{P}^{-1}(0^n \| \mathsf{LBits}_{n-1}(m) \| 0)\right), & \text{if } \mathsf{RBits}_1(m) = 1. \end{cases}$$

One can see that H is collision-resistant and preimage-resistant. However, if we restrict our focus to the vector commitment $\mathsf{VC}^{\mathsf{mt}}$ of Figure 3 for $k = 1$ and thus $\ell = 2^k = 2$, we can observe that $\mathsf{Com}((z\|0, z\|1)) = 0^n$ for any $z \in \{0,1\}^{n-1}$. This leads to a trivial position-binding attack of the form

$$(\mathsf{com}, i, x, \tau, x', \tau') = \left(0^n, 0, z\|0, \mathsf{RBits}_n\left(\mathsf{P}^{-1}(0^n \| z \| 0)\right), z'\|0, \mathsf{RBits}_n\left(\mathsf{P}^{-1}(0^n \| z' \| 0)\right)\right),$$

for any distinct $z, z' \in \{0,1\}^{n-1}$.

7 Proof of Strong Extractability for Sponges

In Sections 4 and 5, we have shown that the Merkle tree commitment that uses a truncated permutation P for the tree hashing is strongly position-binding, and strongly extractable if the leaf hash H is modeled as a random oracle. These results assume independence of the leaf hash H and the permutation P, and our contrived counterexample of Section 6 shows that we cannot hope for security in general if this independence assumption is released.

Nonetheless, we next show that strong extractability still holds in the special case where the leaf hash H is the Overwrite Sponge $\mathsf{oSponge}^{\mathsf{P}}$ (cf. Section 2.3) instantiated with the same permutation P as used in the tree.

Theorem 3 (Strong Extractability for Overwrite Sponge). *Consider the vector commitment* $\mathsf{VC}^{\mathsf{mt}}$ *in Figure 3. Assume that* $\mathsf{H}\colon \mathcal{K} \times \mathcal{D} \to \mathcal{H}$ *is implemented as the overwrite sponge hash as per Figure 2, denoted as* $\mathsf{H} = \mathsf{oSponge}^{\mathsf{P}}$*, using the same permutation* P*, i.e.,* $\mathcal{D} = \mathcal{H}^*$*, and that* $\mathsf{P}\colon \mathcal{H} \times \mathcal{H} \to \mathcal{H} \times \mathcal{H}$ *is modeled as an ideal permutation (cf. Section 2). Assume that* $|\mathcal{H}| \geq \omega(\log \lambda)$*.*

Then, $\mathsf{VC}^{\mathsf{mt}}$ *is (strongly) extractable (cf. Definition 3). Concretely, there is a PPT algorithm* Ext *such that for any PPT algorithm* $\mathcal{A}$ *making at most* $q \leq |\mathcal{H}|$ *oracle queries to* $\mathsf{P}, \mathsf{P}^{-1}$ *in total, we have*

$$\mathsf{Adv}^{\mathsf{extract}}_{\mathsf{VC}^{\mathsf{mt}},\mathsf{Ext}}(\mathcal{A}) \leq \frac{4q^2 + 2q}{|\mathcal{H}| - 1}.$$

Proof. To prove the theorem, we first recall the security game and assumptions, then define our extractor, and then analyze it.

Security Game and Assumptions. We deal with the same security game and the same assumptions as in Section 5, except that we do not have the random oracle H anymore. As in Section 5, the assumptions about queries to P and P^{-1} are:

- a query does not repeat.
- the inverse query is made only if the corresponding forward query has not been made before and vice versa.
- The queries made by the Ver algorithm have been previously made by the adversary.

Extractor. The list $\mathcal{Q}$ now only contains entries of the form $((h_1, h_2), \mathsf{P}, (h_3, h_4))$ and $((h_3, h_4), \mathsf{P}^{-1}, (h_1, h_2))$, Our extractor Ext operates as the extractor in Section 5, but with some changes:

1.-4. Same as in Section 5.

5. For each $i \in [\ell]$ with $h_s \neq \perp$ for $s := \mathsf{rep}_2(i-1)$:
   (a) Initialize set $\mathcal{W}_i$ to $\emptyset$.
   (b) Find $(h_1, h_2, y) \in \mathcal{H}^3$ s.t. $((h_1, h_2), \mathsf{P}, (h_s, y)) \in \mathcal{Q}$.
   (c) If there are multiple such triples, set $\mathsf{BadPreLeaves} := 1$ and abort.
   (d) If there is no such triple, set $\mathbf{x}_i := \perp$.
   (e) If there is exactly one such triple and $h_2 \in \mathcal{W}_i$, set $\mathsf{Cycle} := 1$ and abort.
   (f) If there is exactly one such triple and $h_2 \notin \mathcal{W}_i$ set $\mathbf{x}_i := h_1 || \mathbf{x}_i$ and add h_2 to $\mathcal{W}_i$.
   (g) If $h_2 = \mathbf{0}$ then proceed to the next i; otherwise set $y' := h_2$ and find $(h_1, h_2, y) \in \mathcal{H}^3$ s.t. $((h_1, h_2), \mathsf{P}, (y, y')) \in \mathcal{Q}$ then go to Step (c).
6. Construct the co-paths τ_i as in Section 5.
7. Return $\mathbf{x}, (\tau_i)_{i \in [\ell]}$. Note that $\mathbf{x}_i \in \mathcal{D} \cup \{\perp\}$.

Note that the extractor terminates. This can be seen by noting that the loop that is induced by jumping back to Step (c) can run at most $|\mathcal{Q}|$ times for each fixed i. Namely, either it terminates with $\mathbf{0}$ being found, or it terminates because of event Cycle after at most $|\mathcal{Q}|$ steps, because each iteration of Step 5 adds at least one new element from $\mathcal{Q}$ to $\mathcal{W}_i$.

Auxiliary Graph. In the strong extractability game, we consider a directed graph $\mathcal{G} = (V, E)$ where nodes in V are labeled with elements of $\mathcal{H}$, and directed edges are also labeled with those. We denote an edge from a to b labeled with c as $a \xrightarrow{c} b$. Our graph is in fact a multigraph, i.e., it can contain multiple edges between the same pair of nodes with different labels. Initially, the graph contains only one node labeled with $\mathbf{0}$. The graph is built as follows throughout the game:

- For each forward query $((h_1, h_2), \mathsf{P}, (h_3, h_4))$ or backward query $((h_3, h_4), \mathsf{P}^{-1}, (h_1, h_2))$ we add nodes h_1, h_2, h_3, h_4 to V if they are not already present, and edges $h_1 \xrightarrow{h_2} h_3$, $h_2 \xrightarrow{h_1} h_3$, $h_1 \xrightarrow{h_2} h_4$, $h_2 \xrightarrow{h_1} h_4$.
- When the adversary outputs the commitment com, we add it to the graph as a node if it is not already present.

Events. We consider the following bad events:

- Event $\mathsf{GrFwColl}$: the adversary queries $((h_1, h_2), \mathsf{P}, (h_3, h_4))$ but h_3 or h_4 is already in $\mathcal{G}$.
- Event $\mathsf{GrInvColl}$: the adversary queries $((h_3, h_4), \mathsf{P}^{-1}, (h_1, h_2))$ but h_1 or h_2 is already in $\mathcal{G}$.
- Event $\mathsf{GraphColl}$: one of the above events occurs.

Note that we do not distinguish whether the collision happens before or after the adversary outputs com.

Lemma 14. *We have* $\Pr[\mathsf{GraphColl}] \leq (4q^2 + 2q)/(|\mathcal{H}| - 1)$.

Proof. Each query can create at most four new nodes in $\mathcal{G}$, plus the initial node labeled with $\mathbf{0}$ and the com node. Thus before the ith query, $\mathcal{G}$ has at most $4(i-1) + 2 = 4i - 2$ nodes. This ith query results in a collision with an existing node h if the query's output is either (h, y) or (y, h) for some $y \in \mathcal{H}$. This gives $2(4i-2)|\mathcal{H}|$ bad outputs per query, whereas the total number of possible outputs is at most $|\mathcal{H}|^2 - (i-1)$ (as at most $i-1$ outputs in $\mathcal{H}^2$ are already fixed by previous queries). Therefore, the ith query results in a collision with an existing node with probability at most

$$\frac{(8i-2)|\mathcal{H}|}{|\mathcal{H}|^2 - (i-1)} \leq \frac{(8i-2)|\mathcal{H}|}{|\mathcal{H}|^2 - q} \leq \frac{8i-2}{|\mathcal{H}| - 1},$$

where we used $q \leq |\mathcal{H}|$. Summing over $i = 1, \ldots, q$ yields our bound

$$\sum_{i=1}^{q} \frac{8i-2}{|\mathcal{H}|-1} = \frac{8q(q+1)/2 - 2q}{|\mathcal{H}|-1} = \frac{4q^2 + 2q}{|\mathcal{H}|-1}.$$

□

Lemma 15. $\mathsf{BadLeaves} \vee \mathsf{BadPreLeaves} \vee \mathsf{Cycle}$ *implies* $\mathsf{GraphColl}$.

Proof. By definition, if $\mathsf{BadPreLeaves}$ or $\mathsf{BadLeaves}$ occurs, there are two different queries in $\mathcal{Q}$ with the same truncated output. This implies that $\mathsf{GraphColl}$ occurs.

On the other hand, if Cycle occurs, then at some point in Step 5(e) of the extractor, we have found a chain of queries

$$(h_1, h_2) \overset{\mathsf{P}}{\mapsto} (h_3, h_4) \overset{\mathsf{P}}{\mapsto} (h_5, h_6) \overset{\mathsf{P}}{\mapsto} \cdots (h_{2n-1}, h_{2n}) \overset{\mathsf{P}}{\mapsto} (h_{2n+1}, h_2).$$

If the last query in the chain is also chronologically last, then it hits the already existing node h_2, which implies that $\mathsf{GraphColl}$ occurs. Otherwise the chronologically last query in the chain hits some already existing node h_{2k} for some $2 \leq k \leq n$. Therefore in both cases the event $\mathsf{GraphColl}$ occurs. □

Lemma 16. *If the adversary wins, then by the end of the game, there is a directed path from* $\mathbf{0}$ *to* com *in* $\mathcal{G}$.

Proof. Winning implies that the adversary's opening gets verified by Ver. By our assumption, all queries made by Ver have been made by the adversary as well (either in forward or backward direction). Then, from the definition of the Ver algorithm we conclude that its queries induce a path from $\mathbf{0}$ to com in $\mathcal{G}$. This concludes the proof of this lemma. □

An example of such a path is illustrated in Figure 6.

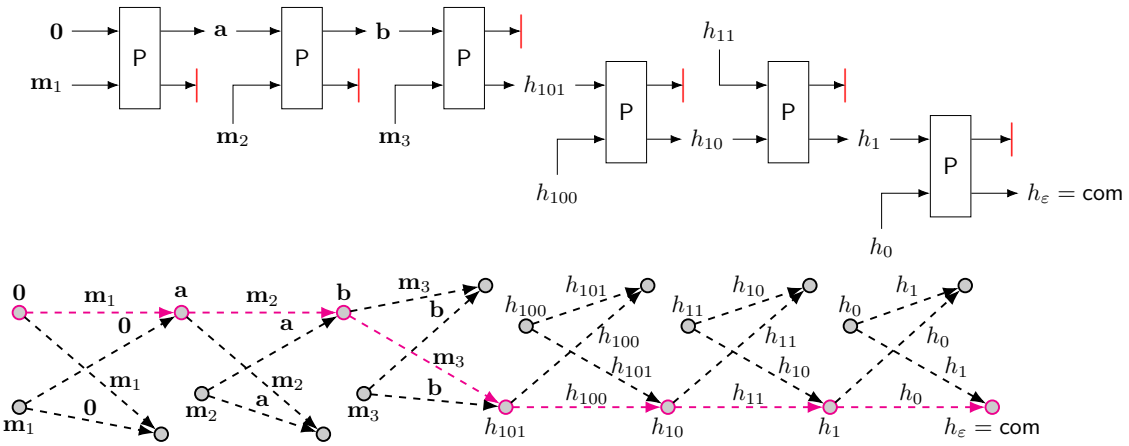


Figure 6: Example of the verification process in the case of the overwrite sponge. Here we verify the commitment to the message $\mathbf{x}_6 = (\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3) \in \mathcal{H}^3$ with opening $\tau = (h_{100}, h_{11}, h_0)$. Below the verification process we depict parts of the graph $\mathcal{G}$ (as defined in the proof of Theorem 3) that are relevant for the verification. The verification queries induce a path from $\mathbf{0}$ to com which is marked in magenta.

Main Proof. Assume that the adversary wins the strong extractability game with commitment com and final output (i, x, τ). Consider the function $\mathsf{Ver}(i, x, \tau)$ that computes the verification as in Figure 3. Consider the following cases:

- *Extractor aborts.* This can happen only if BadPreLeaves or BadLeaves or Cycle occurs, which implies GraphColl by Lemma 15.
- *Extractor does not abort.* The adversary wins the game so there is a path p from $\mathbf{0}$ to com in the graph $\mathcal{G}$ by Lemma 16. We distinguish between the following cases
  - The first edge on the path p (i.e., from $\mathbf{0}$) is produced by a backward query. Then the query hits the existing $\mathbf{0}$ node, leading to event GraphColl.
  - The first edge on the path p is produced by a forward query, but there is one further edge that is produced by a backward query. Then there is a node a on the path that is produced by both a backward and a forward query, leading to event GraphColl.
  - If the path is built by forward queries only, then look at it in the backward direction from com to $\mathbf{0}$. For each node h_3 on the path there then exists a forward query $((h_1, h_2), \mathsf{P}, (h_3, h_4))$. Since the extractor did not abort, for each h_3 on the path there was at most one such query in $\mathcal{Q}$. However, not all of them can be in $\mathcal{Q}$, as otherwise the extractor would have extracted exactly the same path (which is not the case as the adversary wins the game). Therefore, there exists at least one node h_3 on the path such that the query $((h_1, h_2), \mathsf{P}, (h_3, h_4))$ was not present in $\mathcal{Q}$. The last of such queries hits the existing h_3 node, leading to event GraphColl.

Therefore, in all cases, if the adversary wins the strong extractability game, then event GraphColl occurs. By Lemma 14 we have the theorem statement. □

8 Conclusion

This work provides the first formal security analysis of the Plonky3 Merkle tree, a primitive that currently secures billions of dollars in digital assets. We have shown that despite relying on an efficiently invertible compression function the construction can achieve strong position-binding and extractability. These results offer much-needed assurance that the widely deployed Plonky3 Merkle tree is theoretically sound, if instantiated with appropriate permutation and leaf hash.

However, there is still some discrepancy between current industrial targets and provable security margins. While the prevailing target for SNARKs using Plonky3's Merkle tree is 128 bits of security, the

most common parameter configurations fall slightly short of this goal. Specifically, implementations using 31-bit fields (e.g., BabyBear, Mersenne31) typically produce a root of size $8 \times 31 = 248$ bits. Due to a birthday attack, security is therefore capped at approximately $248/2 = 124$ bits.

To achieve a full 128-bit security level, we see two paths forward. One solution is to use a wider permutation and larger digests, thereby pushing the birthday bound beyond the threshold. Alternatively, if one is interested in 128-bit soundness only for the SNARKs that use the Plonky3 Merkle tree, one could pursue an improved theoretical analysis that avoids a reduction to position-binding altogether. For instance, a work by Chiesa and Yogev [CY21] shows that for Micali's SNARK construction [Mic00], 128-bit soundness can be achieved with smaller than 256-bit hashes. This technique is likely adaptable to modern IOP-based SNARKs, but requires a Merkle tree built in the classical way from a random oracle. Tweaking this methodology to the Plonky3 Merkle tree construction is an interesting direction for future research.

Finally, we highlight the nuances regarding the instantiation of the leaf hash. While our (contrived) counterexample in Section 7 illustrates that certain correlations between the leaf hash and the internal permutation can compromise the tree's integrity, our positive results for the Sponge mode demonstrate that security is achievable even when these primitives share the same underlying permutation. Consequently, practitioners should exercise care when selecting a leaf hash, using our formal analysis as an initial set of guidelines.

Acknowledgments. Bart Mennink was supported by the Netherlands Organisation for Scientific Research (NWO) under grant VI.Vidi.203.099.

References

[ABR21] Elena Andreeva, Rishiraj Bhattacharyya, and Arnab Roy. Compactness of hashing modes and efficiency beyond Merkle tree. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 92–123. Springer, Cham, October 2021. (Cited on Page 32.)

[AGP+19] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger. Feistel structures for MPC, and more. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *ESORICS 2019, Part II*, volume 11736 of *LNCS*, pages 151–171. Springer, Cham, September 2019. (Cited on Page 33.)

[AHMN10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 1–15. Springer, Berlin, Heidelberg, August 2010. (Cited on Page 33.)

[AKMQ23] JP Aumasson, Dmitry Khovratovich, Bart Mennink, and Porçu Quine. SAFE: Sponge API for field elements. Cryptology ePrint Archive, Report 2023/522, 2023. (Cited on Page 34.)

[AMP11] Elena Andreeva, Bart Mennink, and Bart Preneel. Security reductions of the second round SHA-3 candidates. In Mike Burmester, Gene Tsudik, Spyros S. Magliveras, and Ivana Ilic, editors, *ISC 2010*, volume 6531 of *LNCS*, pages 39–53. Springer, Berlin, Heidelberg, October 2011. (Cited on Page 32.)

[AMP12] Elena Andreeva, Bart Mennink, and Bart Preneel. The parazoa family: generalizing the sponge hash functions. *Int. J. Inf. Sec.*, 11(3):149–165, 2012. (Cited on Page 33.)

[AS11] Elena Andreeva and Martijn Stam. The symbiosis between collision and preimage resistance. In Liqun Chen, editor, *13th IMA International Conference on Cryptography and Coding*, volume 7089 of *LNCS*, pages 152–171. Springer, Berlin, Heidelberg, December 2011. (Cited on Page 8.)

[Ava25] Avail Project. Avail. Official Website, 2025. https://availproject.org/. Accessed: 2025-12-13. (Cited on Page 31.)

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Berlin, Heidelberg, October / November 2016. (Cited on Page 4, 7.)

[BDPV07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007, 2007. (Cited on Page 4, 5, 8, 32.)

[BDPV08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the Sponge construction. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 181–197. Springer, Berlin, Heidelberg, April 2008. (Cited on Page 33.)

[BDPV11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The KECCAK reference, 2011. https://keccak.team/files/Keccak-reference-3.0.pdf. (Cited on Page 33.)

[BDPV14a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sakura: A flexible coding for tree hashing. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 2014*, volume 8479 of *LNCS*, pages 217–234. Springer, Cham, June 2014. (Cited on Page 34.)

[BDPV14b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sufficient conditions for sound tree and sequential hashing modes. *Int. J. Inf. Sec.*, 13(4):335–353, 2014. (Cited on Page 5, 32, 34.)

[BKL+11] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. Spongent: A lightweight hash function. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 312–325. Springer, Berlin, Heidelberg, September / October 2011. (Cited on Page 33.)

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. (Cited on Page 5, 32.)

[Bre25] Brevis Network Contributors. Pico. GitHub repository, 2025. https://github.com/brevis-network/pico. Accessed: 2025-12-13. (Cited on Page 31.)

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography Conference*, pages 31–60. Springer, 2016. (Cited on Page 3.)

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, Berlin, Heidelberg, August 2005. (Cited on Page 5, 32.)

[Cel25] Celestia Foundation. Celestia. Official Website, 2025. https://celestia.org/. Accessed: 2025-12-13. (Cited on Page 31.)

[CLL19] Wonseok Choi, ByeongHak Lee, and Jooyoung Lee. Indifferentiability of truncated random permutations. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 175–195. Springer, Cham, December 2019. (Cited on Page 33.)

[CN24] Debasmita Chakraborty and Mridul Nandi. Lower bound on number of compression calls of a collision-resistance preserving hash. *CiC*, 1(3):22, 2024. (Cited on Page 32.)

[CO25] Alessandro Chiesa and Michele Orrù. A fiat–shamir transformation from duplex sponges. Cryptology ePrint Archive, Report 2025/536, 2025. (Cited on Page 33.)

[Con25] ConsenSys. Linea. Official Website, 2025. https://linea.build/. Accessed: 2025-12-13. (Cited on Page 3.)

[CY21] Alessandro Chiesa and Eylon Yogev. Tight security bounds for Micali's SNARGs. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part I*, volume 13042 of *LNCS*, pages 401–434. Springer, Cham, November 2021. (Cited on Page 25.)

[DDV11] Joan Daemen, Tony Dusenge, and Gilles Van Assche. Sufficient conditions for sound hashing using a truncated permutation. Cryptology ePrint Archive, Report 2011/459, 2011. (Cited on Page 5, 34.)

[Def25] DefiLlama. Scroll total value locked. DefiLlama Analytics, 2025. https://defillama.com/chain/scroll. Accessed: 2025-12-13. (Cited on Page 31.)

[DKKW25a] Justin Drake, Dmitry Khovratovich, Mikhail Kudinov, and Benedikt Wagner. Hash-based multi-signatures for post-quantum ethereum. Cryptology ePrint Archive, Paper 2025/055, 2025. (Cited on Page 31.)

[DKKW25b] Justin Drake, Dmitry Khovratovich, Mikhail Kudinov, and Benedikt Wagner. Technical note: LeanSig for post-quantum ethereum. Cryptology ePrint Archive, Paper 2025/1332, 2025. (Cited on Page 31.)

[DKMN21] Yevgeniy Dodis, Dmitry Khovratovich, Nicky Mouha, and Mridul Nandi. T_5: Hashing five inputs with three compression calls. In Stefano Tessaro, editor, *ITC 2021*, volume 199 of *LIPIcs*, pages 24:1–24:23. Schloss Dagstuhl, July 2021. (Cited on Page 32.)

[DMV18] Joan Daemen, Bart Mennink, and Gilles Van Assche. Sound hashing modes of arbitrary functions, permutations, and block ciphers. *IACR Trans. Symm. Cryptol.*, 2018(4):197–228, 2018. (Cited on Page 5, 34.)

[DRRS09] Yevgeniy Dodis, Leonid Reyzin, Ronald L. Rivest, and Emily Shen. Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to MD6. In Orr Dunkelman, editor, *FSE 2009*, volume 5665 of *LNCS*, pages 104–121. Springer, Berlin, Heidelberg, February 2009. (Cited on Page 5, 32, 33.)

[Fer25] Fermah. Fermah: The universal proof generation layer. Official Documentation, 2025. https://docs.fermah.xyz/. Accessed: 2025-12-13. (Cited on Page 3.)

[GDM20] Aldo Gunsing, Joan Daemen, and Bart Mennink. Errata to sound hashing modes of arbitrary functions, permutations, and BCs. *IACR Trans. Symm. Cryptol.*, 2020(3):362–366, 2020. (Cited on Page 34.)

[GKL+22] Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Reinforced concrete: A fast hash function for verifiable computation. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1323–1335. ACM Press, November 2022. (Cited on Page 33.)

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021. (Cited on Page 33.)

[GKS23] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the poseidon hash function. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *AFRICACRYPT 23*, volume 14064 of *LNCS*, pages 177–203. Springer, Cham, July 2023. (Cited on Page 3, 9.)

[GM22] Lorenzo Grassi and Bart Mennink. Security of truncated permutation without initial value. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 620–650. Springer, Cham, December 2022. (Cited on Page 5, 33.)

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 222–239. Springer, Berlin, Heidelberg, August 2011. (Cited on Page 33.)

[Grund] Angus Gruen. Small prime fields in Plonky3. HackMD Post, n.d. https://hackmd.io/51_gB_3ZSBSZ4KafGGvkUw. Accessed: 2025-12-13. (Cited on Page 3.)

[Haz25] Hazeflow Research. Proof generation on Succinct and SP1. Hazeflow Research Blog, September 2025. https://research.hazeflow.xyz/p/proof-generation-on-succinct-and. Accessed: 2025-12-13. (Cited on Page 31.)

[Hir18] Shoichi Hirose. Sequential Hashing with Minimum Padding. *Cryptogr.*, 2(2):11, 2018. (Cited on Page 33.)

[HLP24] Ulrich Haböck, David Levit, and Shahar Papini. Circle starks. *Cryptology ePrint Archive*, 2024. (Cited on Page 3.)

[KBM23] Dmitry Khovratovich, Mario Marhuenda Beltrán, and Bart Mennink. Generic security of the SAFE API and its applications. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VIII*, volume 14445 of *LNCS*, pages 301–327. Springer, Singapore, December 2023. (Cited on Page 34.)

[LBM25] Charlotte Lefevre, Mario Marhuenda Beltrán, and Bart Mennink. To pad or not to pad? Padding-free arithmetization-oriented sponges. *IACR Trans. Symm. Cryptol.*, 2025(1):97–137, 2025. (Cited on Page 33.)

[Lea25] LeanMultisig Contributors. Leanmultisig. GitHub repository, 2025. https://github.com/leanEthereum/leanMultisig. Accessed: 2025-12-13. (Cited on Page 3, 31.)

[Lef23] Charlotte Lefevre. Indifferentiability of the Sponge construction with a restricted number of message blocks. *IACR Trans. Symm. Cryptol.*, 2023(1):224–243, 2023. (Cited on Page 33.)

[LM22] Charlotte Lefevre and Bart Mennink. Tight preimage resistance of the Sponge construction. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 185–204. Springer, Cham, August 2022. (Cited on Page 33.)

[LM24] Charlotte Lefevre and Bart Mennink. Permutation-based hash chains with application to password hashing. *IACR Trans. Symm. Cryptol.*, 2024(4):249–286, 2024. (Cited on Page 33.)

[LM25] Charlotte Lefevre and Bart Mennink. SoK: Security of the ascon modes. *IACR Trans. Symm. Cryptol.*, 2025(1):138–210, 2025. (Cited on Page 33.)

[Mat25] Matter Labs. Zksync. Official Website, 2025. https://www.zksync.io/. Accessed: 2025-12-13. (Cited on Page 3.)

[Mer79] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979. (Cited on Page 3, 32, 33.)

[Mer90] Ralph C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 428–446. Springer, New York, August 1990. (Cited on Page 3, 5, 32, 33.)

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. (Cited on Page 25.)

[MP16] Bart Mennink and Bart Preneel. Efficient parallelizable hashing using small non-compressing primitives. *Int. J. Inf. Sec.*, 15(3):285–300, 2016. (Cited on Page 32.)

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, Berlin, Heidelberg, February 2004. (Cited on Page 5, 32.)

[Nat02] National Institute of Standards and Technology. Secure hash standard. *Federal Information Processing Standards Publication (FIPS)*, 2002. available at https://csrc.nist.gov/files/pubs/fips/180-2/final/docs/fips180-2.pdf. (Cited on Page 9.)

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-based hash and extendable-output functions. *Federal Information Processing Standards Publication (FIPS)*, 2015. (Cited on Page 9, 33, 34.)

[NO14] Yusuke Naito and Kazuo Ohta. Improved indifferentiable security analysis of PHOTON. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14*, volume 8642 of *LNCS*, pages 340–357. Springer, Cham, September 2014. (Cited on Page 33.)

[OANWO21] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. one function, fast everywhere. *url: https://github. com/BLAKE3-team/BLAKE3-specs/blob/master/blake3. pdf*, 2021. (Cited on Page 9.)

[Ope25] OpenVM Contributors. Openvm. GitHub repository, 2025. https://github.com/openvm-org/openvm. Accessed: 2025-12-13. (Cited on Page 3, 31.)

[Plo25] Plonky3 Contributors. Plonky3. GitHub repository, 2025. https://github.com/Plonky3/Plonky3/tree/3899100a68374fa1e9e4cf74f5fc508de06ac533. Accessed: 2025-12-13. (Cited on Page 3.)

[Pol24a] Polygon Miden Team. Miden vm. GitHub repository, 2024. https://github.com/0xMiden/miden-vm. (Cited on Page 3.)

[Pol24b] Polygon Miden Team. Migration to Plonky3 backend. GitHub Pull Request #2258, 2024. https://github.com/0xMiden/miden-vm/pull/2258. Accessed: 2025-12-13. (Cited on Page 3, 31.)

[Pol25a] Polygon Hermez Team. Zisk. GitHub repository, 2025. https://github.com/0xPolygonHermez/zisk. Accessed: 2025-12-13. (Cited on Page 3.)

[Pol25b] Polygon Labs. Polygon. Official Website, 2025. https://polygon.technology/. Accessed: 2025-12-13. (Cited on Page 31.)

[RAB+09] Ronald Rivest, Benjamin Agre, Daniel V. Bailey, Christopher Crutchfield, Yevgeniy Dodis, Kermin Elliott Fleming, Asif Khan, Jayant Krishnamurthy, Yuncheng Lin, Leo Reyzin, Emily Shen, Jim Sukha, Drew Sutherland, Eran Tromer, and Yiqun Lisa Yin. The MD6 hash function – A proposal to NIST for SHA-3, 2009. (Cited on Page 33.)

[RIS25] RISC Zero Team. Risc zero. GitHub repository, 2025. https://github.com/risc0/risc0. Accessed: 2025-12-13. (Cited on Page 3.)

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal K. Roy and Willi Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 371–388. Springer, Berlin, Heidelberg, February 2004. (Cited on Page 7.)

[RSS11] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 487–506. Springer, Berlin, Heidelberg, May 2011. (Cited on Page 32.)

[Scr25] Scroll Foundation. Scroll. Official Website, 2025. https://scroll.io/. Accessed: 2025-12-13. (Cited on Page 3.)

[SMK+25] Meltem Sönmez Turan, Kerry A. McKay, Jinkeon Kang, John Kelsey, and Donghoon Chang. Ascon-Based Lightweight Cryptography Standards for Constrained Devices: Authenticated Encryption, Hash, and Extendable Output Functions. NIST SP 800-232, August 2025. https://csrc.nist.gov/pubs/sp/800/232/final. (Cited on Page 33.)

[Sta25] Starknet Foundation. Starknet. Official Website, 2025. https://www.starknet.io/. Accessed: 2025-12-13. (Cited on Page 3.)

[Suc24] Succinct Labs. Sp1. GitHub repository, 2024. https://github.com/succinctlabs/sp1. (Cited on Page 3, 31.)

[Suc25a] Succinct Labs. SP1 adoption and token security metrics. Succinct Blog, July 2025. https://blog.ju.com/succinct-network-zkvm-prove-token/. Accessed: 2025-12-13. (Cited on Page 31.)

[Suc25b] Succinct Labs. Succinct. Official Website, 2025. https://www.succinct.xyz/. Accessed: 2025-12-13. (Cited on Page 31.)

[Suc25c] Succinct Labs. Succinct prover network. GitHub repository, 2025. https://github.com/succinctlabs/network. Accessed: 2025-12-13. (Cited on Page 3.)

[Tai25] Taiko Labs. Taiko. Official Website, 2025. https://taiko.xyz/. Accessed: 2025-12-13. (Cited on Page 31.)

[Val23] Valida XYZ Contributors. Valida. GitHub repository, 2023. https://github.com/valida-xyz/valida. Archived repository. Accessed: 2025-12-13. (Cited on Page 31.)

[ZKM25] ZKM Contributors. Ziren. GitHub repository, 2025. https://github.com/ProjectZKM/Ziren. Accessed: 2025-12-13. (Cited on Page 31.)

Appendix

A More Details on Adoption of Plonky3

Plonky3 has established itself as the foundational cryptographic backend for the modern small-field SNARK industry. Its modular Merkle tree implementation is a critical dependency for a wide array of high-value applications, ranging from general-purpose Zero-Knowledge Virtual Machines (zkVMs) to next-generation consensus layers. In this section, we survey the ecosystem to highlight the criticality of the primitive analyzed in this work.

Remark 8. Despite their name, most zkVMs are actually not zero-knowledge in the cryptographic sense. The industry has misused the term "zk" when actually *succinctness* is meant.

A.1 Zero-Knowledge Virtual Machines (zkVMs)

The most significant adoption of the Plonky3 Merkle tree is found in the burgeoning zkVM landscape, where it serves as the commitment scheme.

- **SP1 (Succinct Labs).** As the flagship implementation of a Plonky3-based zkVM, SP1 relies heavily on this Merkle tree construction [Suc24]. The economic weight resting on this implementation is substantial:
  - Estimates of the aggregate value secured by SP1 vary by methodology but consistently indicate multi-billion dollar scale. Recent analyses of ZK infrastructure and L2 ecosystems place the TVS at approximately $3.14 billion [Haz25].
  - Official metrics indicate over 35 ecosystem partners, with internal projections of TVS exceeding $4 billion [Suc25b]. This includes integrations with major modular blockchain platforms such as Polygon, Celestia, Avail, and Taiko [Suc25a, Pol25b, Cel25, Ava25, Tai25].
- **Emerging zkVM Architectures.** Beyond SP1, the Plonky3 Merkle tree is the standard backend for several competing architectures currently racing to achieve real-time proving for Ethereum:
  - **Valida:** A STARK-based zkVM that utilizes the Plonky3 tree for its trace commitments [Val23].
  - **OpenVM:** A modular zkVM framework targeting adoption by high-TVS Layer 2 networks such as Scroll [Ope25]. While OpenVM is an open-source framework, its target implementation partner, Scroll, currently secures approximately $200 million in assets [Def25].
  - **Ziren (formerly zkMIPS):** A MIPS-based zkVM developed by ZKM, also utilizing the Plonky3 backend for its proving system [ZKM25].
  - **Pico:** A coprocessor-centric zkVM developed by Brevis Network, optimized and modular by design [Bre25].

A.2 Infrastructure and Future Consensus

The utility of the Plonky3 Merkle tree extends beyond virtual machines into core blockchain infrastructure and future consensus mechanisms.

- **Polygon Miden.** While originally built on a custom STARK stack, the Miden VM is actively transitioning its cryptographic backend to a Plonky3-based architecture [Pol24b]. This migration will inherit the Merkle tree construction analyzed in this work, placing the security of the Miden rollup—and its future assets—directly on this primitive.
- **Post-Quantum Ethereum.** Perhaps the most critical future application is the proposal to utilize Plonky3-based hash functions for the Ethereum consensus layer. The *LeanSig* proposal advocates for aggregating post-quantum signatures using this specific tree structure to ensure the long-term security of the Ethereum Layer 1 network [DKKW25a, DKKW25b, Lea25].

Summary of Criticality. The Plonky3 Merkle tree is not merely an experimental library but the root of trust for a rapidly expanding ecosystem. Between the currently secured assets in SP1 (estimated at $3–4 billion) and the strategic roadmap of Ethereum's post-quantum security, the integrity of this implementation is a matter of systemic importance.

B More on Tree Hashing and Permutation-Based Hashing

Given the vast amount of research already performed on tree hashing and on permutation-based cryptography, it may come as a surprise that existing results do not capture security of Plonky3's Merkle tree. However, there are multiple reasons for that, as we will outline below.

Tree Hashing Based on Random Compression Function. The idea of Merkle trees originates from Merkle's PhD thesis [Mer79]. In its simplest form, assuming a compression function h from $2n$ to n bits, an input message M is first injectively padded to an integral number of $2n$-bit blocks $M_1, \ldots, M_\ell$ where ℓ is a power of two. At the leaves, the message blocks are hashed in parallel to $h_{0,i} = h(M_i)$. At the next layer, these values are subsequently hashed to $h_{1,i} = h(h_{0,2i-1} \| h_{0,2i})$, and so on, until the root $h_{\log_2(\ell),1}$ is obtained. One can prove that, with proper strengthening, collision resistance of a Merkle tree follows from the collision resistance of the underlying function h [Mer90]. The definition and collision result generalize to sparser trees or trees with a different arity. There have also been works that extended to non-compressing one-way functions [MP16] and that aimed to optimize the amount of data hashed per compression function call [DKMN21, ABR21, CN24].

However, in many applications, one requires more of a hash function than just collision resistance: one requires it to appear like a random oracle [BR93]. The security model that captures this for hash functions is *indifferentiability* of Maurer et al. [MRH04] and Coron et al. [CDMP05]. This is a powerful notion, because it implies that the hash function generically behaves like a random oracle up to the proven bound, and can be used as such in *single-stage settings*.[11] Andreeva et al. [AMP11, Appendix A] demonstrated that the security of a hash function against specific attacks (e.g., a collision attack) is determined by the indifferentiability of that hash function from a random oracle plus the security of that random oracle against the particular attack.

The Merkle tree with strengthening is, however, despite its collision resistance preservation is not indifferentiable from a random oracle: it is vulnerable to the length extension attack. On the positive side, with proper encoding, one can prove that it behaves like a random oracle, roughly up to the birthday bound in the output size of h. Concretely, assuming that h is a random function, Dodis et al. [DRRS09] derived five conditions required for a tree hasher to be secure, and Bertoni et al. [BDPV14b] likewise derived three general sufficient conditions and proved indifferentiability of the mode.

Permutation-Based Hashing and the Sponge. However, these observations all require the compression function h to be random, whereas Plonky3's Merkle tree uses a truncated permutation as a compression function:[12]

$$\mathsf{LBits}_n\left(\mathsf{P}(X)\right). \tag{4}$$

Before diving into the existing results of tree hashing with a truncated permutation (next paragraph), it makes sense to first revisit the state of the art on permutation-based hashing.

Bertoni et al. [BDPV07] pioneered the idea of permutation-based hashing with the invention of sponge functions. This construction is initially defined on bits. If instantiated with a b-bit permutation P, it maintains a b-bit state $S \in \{0,1\}^b$, initialized to the $\mathbf{0}^b$-string. This state is split into an r-bit outer part and a c-bit inner part, where $b = r + c$. To hash a message M, it is injectively padded to an integral number of r-bit blocks $M_1, \ldots, M_\ell$, and it is processed r bits at a time by adding them into the outer r bits of the state, interleaved with an evaluation of P: $S = \mathsf{P}(S \oplus (M_i \| \mathbf{0}^c))$. Once absorption is finished, digest blocks are output one-by-one by taking the leftmost r bits from the state, denoted $\mathsf{LBits}_r(S)$, and the state is permuted ($S = \mathsf{P}(S)$) and another digest block is output, as often as needed.

[11] Ristenpart et al. [RSS11] demonstrated that indifferentiability is not sufficient for a hash function to be securely used in multi-stage settings.

[12] Here, the compression function is explained over bits for simplicity; refer to Section 3 for the details.

The sponge function has quickly found adoption since its inception. Most notably, the US NIST standard SHA-3 [Nat15] *is* a sponge instantiated with the Keccak-f permutation [BDPV11]. However, it turned out to also be a powerful design for lightweight hashing (e.g., QUARK [AHMN10], PHOTON [GPP11], SPONGENT [BKL+11], and the NIST Lightweight Cryptography scheme Ascon-Hash [SMK+25]) and for arithmetization-oriented hashing (e.g., GMiMCHash [AGP+19], Poseidon [GKR+21], and Reinforced Concrete [GKL+22]). In particular, if used in larger fields, the permutation is specified to work on a domain of the form $\mathcal{B} = \mathbb{F}^b$, which is split into $\mathcal{R} \times \mathcal{C}$ for $\mathcal{R} = \mathbb{F}^r$ and $\mathcal{C} = \mathbb{F}^c$, the message is injectively padded into elements of $\mathcal{R}$, and absorption is then simple addition in the corresponding field. All generic security bounds on the sponge (as discussed below) generalize to the case of permutations over larger fields, and we mainly discuss them for the bitwise setting.

The generic security of the sponge, and thus of the modes underpinning each and every one of the functions in the previous paragraph, is founded upon a decent base of research. Already in 2008, Bertoni et al. [BDPV08] proved that the sponge construction (with a random permutation) is indifferentiable from a random oracle in the aforementioned model, as long as the number of permutation calls does not exceed $2^{c/2}$. This particularly implies that the sponge construction behaves like a random oracle up to this bound and can be used as such in single-stage settings.

For a hash function with outputs of n bits, this indifferentiability result implies collision resistance up to $\min\{2^{c/2}, 2^{n/2}\}$ permutation calls, and first and second preimage resistance up to $\min\{2^{c/2}, 2^n\}$ permutation calls. An improved security bound for preimage resistance, up to $\min\{2^{c/2} + 2^{n-r}, 2^n\}$ permutation calls, was derived by Lefevre and Mennink [LM22]. We refer to the systematization of knowledge on the Ascon modes of Lefevre and Mennink [LM25], and specifically Section 8 of their work, for a more in-depth overview and treatment of the generic security of the sponge. It was also demonstrated by Lefevre [Lef23] that improved security is obtained in case the messages are of a restricted length. In addition, various works considered security of variants of the sponge, the most notable examples being Andreeva et al. [AMP12], who introduced the parazoa hash function family as a generalization of the sponge (covering more general absorption and squeezing), Naito and Ohta [NO14], who proved indifferentiability up to a comparable bound with larger initial absorption and larger squeezing, and Lefevre et al. [LBM25], who considered the sponge where the inner part of the state is transformed with a non-cryptographic permutation after message absorption (inspired by Hirose [Hir18]). It should also be remarked that, while the sponge construction is indifferentiable from a random oracle and thus behaves as such in applications, sometimes direct proofs lead to better bounds: for example, Lefevre and Mennink [LM24] considered password-based hash chains based on the sponge and Chiesa and Orrù considered the security of the Fiat-Shamir transform based on the sponge [CO25], both with a direct proof relying on the strength of the permutation rather than relying on the indifferentiability of the sponge. (In our work, specifically in Section 7, we also consider security in a direct proof without relying on the indifferentiability of the sponge.)

Tree Hashing With Truncated Permutation. However, despite its popularity as a general-purpose hash function, it is inherently *sequential*, meaning that it is not the optimal choice for settings requiring massive parallelism, such as committing to large datasets. Several works have investigated tree-style hashing using a permutation, all inspired by the classical Merkle trees [Mer79, Mer90], and these works are arguably closest to Plonky3's Merkle tree.

The idea of tree-based hashing with a cryptographic permutation first appeared in MD6 [RAB+09], and the five conditions of Dodis et al. [DRRS09] for a tree hasher to be secure were, in fact, derived in the context of the MD6 hash function. However, fundamental to their work was that the (in this case, permutation-based) compression function is random. In MD6, this was achieved by including an initial value IV in every permutation call:

$$\mathsf{LBits}_n\left(\mathsf{P}(X \parallel IV)\right). \tag{5}$$

By proving that this compression function is indifferentiable from a random function up to $\min\{2^{b-m}, 2^{(b-n)/2}\}$ queries [DRRS09], indifferentiability of the overall tree follows by composition. This bound was improved by Choi et al. [CLL19] to security up to $\min\{2^{(2b-n)/3}, 2^{b-n}, 2^m\}$ queries (ignoring logarithmic terms).

Grassi and Mennink [GM22] later observed that for this particular construction, one can replace this IV by a random oracle output of any message, as in

$$\mathsf{LBits}_n\left(\mathsf{P}(X \parallel \mathsf{H}(M))\right), \tag{6}$$

and security only degrades by an additional term $2^{m/2}$, which accounts for hash function collisions. The idea appeared in disguise in SAFE, a sponge construction where the inner part of the initial state is not $\mathbf{0}^c$ but $\mathsf{H}(M)$ for certain auxiliary input M [AKMQ23, KBM23]. In essence, one may see the approach in SAFE as a hybrid between the sponge and the Grassi-Mennink approach of (6), though SAFE has several side-contributions on its own. Here, we remark that hashing data into the initial state of a new hash function sounds counter-intuitive, but this approach is particularly useful if, for example, part of the data is in a different field, or if they are stable for many hash function evaluations.

Despite these positive results, it is obvious that placing a IV at each compression function call (as was done in MD6) sacrifices efficiency in exchange for composability: it is also possible to obtain an indifferentiable tree hasher without the compression function itself being indifferentiable from a random function. The three general sufficient conditions of Bertoni et al. [BDPV14b] only applied to the case of random compression functions, but Daemen et al. [DDV11] later appended this work with an extra condition in case the compression function is a truncated permutation (something that we will also do). Eventually, Daemen et al. [DMV18] presented a systematization of knowledge on conditions for sound tree hashing, and derived bounds in case the underlying building block is an arbitrary compression function, a truncated block cipher, or a truncated permutation. Gunsing et al. [GDM20] presented a corrigendum on this work to resolve an issue in the proof on the instantiation with a truncated permutation and block cipher. Note that, despite the fact that SHA-3 [Nat15] is intrinsically a sequential mode, the extendible output functions (XOFs) in the standard are designed to be compatible with the Sakura coding for tree hashing [BDPV14a].

Of particular interest in the eventual systematization of knowledge of Daemen et al. [DMV18] is their minimal tree hashing mode based on a truncated permutation. Assuming we take a permutation P on b bits and truncate it to n bits, we devote two bits of the input to P for domain separation: 00 for leaves, 11 for final, and 10 for other truncated permutation evaluations. As for the leaves, we reserve an additional m bits for a dedicated initial value IV. For the rest, the tree operates as default. This way, one obtains a tree hasher that is indifferentiable from a random oracle up to $\min\{2^{n/2}, 2^{b-m}\}$ queries [DMV18, GDM20].

The Plonky3 Anomaly: A Hybrid Approach. The Merkle tree variant in Plonky3 resembles several of the above ideas. To be precise, it operates as a simple Merkle tree, but with a truncated permutation evaluation on all nodes, except for the leaf nodes where a cryptographic hash function H is used (see Section 3 for the details). Stated differently, one may see this tree variant as a hybrid between the minimal tree hasher of Daemen et al. [DMV18], but without domain separation, and the Grassi-Mennink approach of (6): replace the dedicated initial values IV by hashed data. However, the lack of domain separation makes Plonky3's Merkle tree with simple message encoding vulnerable to a length extension attack and thus differentiable from a random oracle. This, concretely, already demonstrates that all earlier results on the indifferentiability of tree hashers are mostly futile for Plonky3's tree hasher.

Fortunately, this is *not a problem* for Plonky3's tree hasher: indifferentiability would even be an undesirably strong requirement. Indeed, indifferentiability implies that the tree hasher behaves like a monolithic random oracle, but we are interested in vector commitment and to prove strong position-binding and extractability, we need to define and analyze the concept of openings. Additionally, a direct proof of these two security properties, as we deliver in this work, gives a tighter bound with lighter assumptions on the hash function (see Table 1). (We remark that, even if we were to rely on the indifferentiability result, we would only be able to do so for position-binding, as extractability is a two-stage game and all aforementioned indifferentiability results apply to single-stage settings only.)

C Plonky3's MMCS

In practice, Plonky3 instantiates a *Mixed Matrix Commitment Scheme* (MMCS). This construction generalizes standard Merkle trees to efficiently commit to a heterogeneous collection of matrices—a requirement imposed by the structure of modern SNARK execution traces. In this section, we explain why our security results remain applicable despite these complexities.

The Need for Mixed Matrix Commitments. In the context of zkVMs and modular proof systems, the prover does not commit to a single flat vector. Instead, they commit to a set of trace matrices $\mathbf{M} = \{M_0, \ldots, M_{n-1}\}$ representing polynomial evaluations. These matrices exhibit significant irregularity:

- Varying widths: different execution traces require varying amounts of data per step (e.g., wide main traces vs. narrow auxiliary traces).
- Varying heights: polynomials often differ in degree, resulting in matrices of disparate row counts (e.g., a main memory trace of height 2^{20} versus a precompile trace of height 2^{18}).

The naive approach of padding all matrices to the maximum height would introduce unacceptable prover overhead. The MMCS solves this via two structural strategies that differ from standard Merkle trees: *row-wise aggregation* and *height-based injection*.

Row-Wise Aggregation. Although the MMCS is presented as committing to a collection of distinct matrices, it is structurally isomorphic to the vector commitment defined in Section 3. In our formal analysis, the input vector $\mathbf{x}$ represents the sequence of aggregated rows; explicitly, for any index i, the pre-leaf data x_i corresponds to the concatenation of the i-th rows of all matrices:

$$x_i := M_0[i] \parallel M_1[i] \parallel \cdots \parallel M_m[i].$$

The leaf is then computed as $h_i = \mathsf{H}(\mathsf{ck}, x_i)$.

Jagged Construction via Injection. The most significant structural deviation is the handling of varying heights. The tree is constructed bottom-up starting from the tallest matrices (height $H_{\max}$). Shorter matrices are not padded; instead, they are *injected* into the tree structure at the intermediate layer corresponding to their native height. Formally, the transition from layer k to $k+1$ involves more than just compressing sibling digests. If specific matrices $M \in \mathbf{M}$ have a height corresponding to layer $k+1$, their row data is hashed and mixed into the accumulator. More precisely, one first compresses the left and right child, and then compresses the result and the H-hash of the row data, both using function $\mathsf{Compress}$ from Figure 3. Crucially, this injection strategy preserves the security properties that we have shown. Namely, this operation corresponds to a standard binary tree of increased depth with sparsity, where the injected rows function as leaves at intermediate levels. As the injection points are fixed by the public matrix dimensions, this construction falls strictly within the scope of our sparse tree analysis (see Remark 3).