

Nothing Up My Matrix: Kleptographic Backdoors in ZK-friendly Hash Functions

Hyunsik Jeong¹, Malte Sander Leip¹, Mincheol Son^{1,2*}

¹*Zellic* ²*KAIST*

¹{hyunsik, malte, mincheol}@zellic.io

²encrypted.def@kaist.ac.kr

Abstract

ZK-friendly hash functions are often built from algebraic SPN permutations. The matrix provides diffusion by mixing the state elements after the S-box layer. However, there is no uniform convention for selecting matrices for the linear layer across primitives. Matrix selection may follow a transparent nothing-up-my-sleeve procedure or prioritize implementation efficiency. Some specifications instead treat any MDS matrix as admissible. This parameter-selection freedom also persists in practice, as some deployed implementations replace the concrete matrix proposed in the original specification with an alternative instantiation. We show that adversarial use of this freedom can create a kleptographic attack surface.

In this paper, we study kleptographic backdoors embedded in the matrices of ZK-friendly hash functions. Assuming that a malicious designer controls matrix selection, the designer can choose a matrix that maps a chosen input to a chosen output under appropriate round and parameter conditions. The resulting matrix is MDS and passes the additional matrix security checks required by the target primitive.

Three case studies show that such a backdoor could have critical security consequences in real-world deployments. In Plonky3, it would allow a prover to control a Fiat–Shamir challenge and make the verifier accept an invalid claim. In Neptune Cash, it would permit creating a digest collision between the program that checks whether a transaction is valid and one that omits this check, enabling counterfeit currency. Finally, in Plonky2, it would enable a forged Merkle-tree membership proof for an attacker-chosen element. Our results show that satisfying the MDS condition and other security requirements is insufficient to establish that a matrix is trustworthy. Its generation process must also be transparent and verifiable.

1 Introduction

Zero-knowledge proof systems [8, 9, 16, 22, 29, 46, 47] rely extensively on cryptographic hash functions. They use hashes

to commit to polynomial evaluations, derive Fiat–Shamir challenges, and authenticate programs and state. Some applications evaluate hash functions inside the zero-knowledge systems, for example to support proof recursion [16, 18, 32] or to compute a hash of a witness hidden from the verifier. In many widely used proof systems, such computations are expressed as arithmetic constraints or polynomial identities over a large prime field $\mathbb{F}_p$, and their cost is largely determined by the complexity of the resulting algebraic representation.

Although standardized hash functions such as SHA-2 [34] and Keccak [14] offer strong security and efficient in modern CPUs, their bit-oriented operations are costly to express as arithmetic constraints over large prime fields. This mismatch has motivated a class of ZK-friendly hash functions [2–4, 15, 23, 25, 30] whose arithmetic can be represented efficiently inside a proof system. Their security depends on the nonlinear S-box and the public matrix that specifies the linear layer. This matrix is typically chosen to have the MDS property to provide for maximum diffusion.

As summarized in Table 1, matrix selection often leaves designers some freedom. For instance, Poseidon [25] uses a Cauchy MDS matrix, Tip5 [42] uses circulant MDS matrices, and Polocolo [30] uses MDS matrices with low addition cost found through heuristic search. This flexibility persists in practice. As shown in Table 2, some implementations replace the matrix proposed in the original specification with an alternative instantiation.

1.1 Matrix Selection as a Supply-Chain Attack Surface

The literature does not clearly establish how much freedom is safe in matrix selection. Poseidon [25, Sections 2.3 and 4.1] cautions that not every MDS matrix provides the same level of security, requires candidates to be screened against long subspace trails [28], and explicitly notes that allowing an attacker to choose either the round constants or the matrix freely can lead to attacks. In contrast, Rescue-Prime [41, Section 4.2] states that any MDS matrix can be used and assigns

*This work was conducted at Zellic.

high confidence to such alternative instantiations. Griffin [23, Section 5.1] takes an intermediate position. For state sizes $t \in \{3, 4\}$, it requires the matrix to be MDS while presenting particular efficient matrices only as suggested choices. See Table 1 for further details on these differing positions and the corresponding matrix-generation methods.

These differing positions leave a gray area rather than a clear consensus. A maliciously chosen matrix may satisfy all stated algebraic and structural requirements, which do not reveal how or why the matrix was generated. The deployments surveyed in Table 2 further show that using a matrix different from the one proposed in the original specification has not prevented these instantiations from being deployed and used in real-world systems.

We ask whether this freedom remains benign under adversarial parameter selection. Although allowing an attacker to choose the matrix of a hash function may initially appear artificial, such control can arise in a supply-chain attack. A service provider, library maintainer, or system integrator may be permitted to choose a matrix that appears valid and passes the expected tests. As previous examples of potential cryptographic backdoors [7, 10, 17] illustrate, such an attack may be feasible when the selecting party exercises sufficient control over the system or provides a plausible justification for its parameter choices. In this setting, can a matrix be considered secure solely because it satisfies the MDS condition and exhibits no known structural weaknesses?

1.2 Our Contributions

We answer this question in the negative by presenting a generic construction that enables an attacker to construct an MDS linear layer under which an SPN permutation maps an attacker-chosen input to an attacker-chosen output. At a high level, the attacker reverses the usual instantiation process. The attacker first chooses the permutation input and output, then constructs intermediate states consistent with the nonlinear operations and round constants, and finally selects a matrix that satisfies the resulting linear input–output constraints.

A naive construction with N matrix multiplications gives N distinct conditions $M\vec{u}_i = \vec{v}_i$, whereas the basic interpolation handles at most t linearly independent input–output pairs. Our generic construction pairs matrix multiplications so that both multiplications in each pair share the same input–output condition, reducing the number of distinct conditions by approximately one half. For a permutation of width t in which the state is multiplied by the matrix N times, the generic construction succeeds when

$$N \leq 2t - 1. \quad (1)$$

The generic construction does not apply directly to Poseidon [25] or Poseidon2 [26] because each permutation performs too many matrix multiplications relative to its state width. Instead, we present separate attacks that exploit the

partial rounds of Poseidon and the distinct internal and external matrices of Poseidon2.

We further demonstrate the protocol-level impact of malicious matrix selection in three case studies. In the first case study, we use Plonky3 [38] to demonstrate how malicious matrix selection can manipulate the value of a Fiat–Shamir challenge. The attacker first chooses an invalid witness together with a challenge so that the verifier’s batched constraint check is satisfied and then selects the external matrix of the Poseidon2-based challenger so that hashing the corresponding transcript yields that challenge. The resulting STARK proof is accepted by the unmodified verifier configured to use the backdoored matrix in the challenger.

In the second case study, we use Neptune Cash [35] to demonstrate how malicious selection of Tip5’s matrix [42] can produce a targeted digest collision between two different virtual machine programs. The legitimate `NativeCurrency` program checks that a transaction conserves the native currency, whereas the malicious `AcceptAll` program terminates successfully without performing this check. Neptune Cash identifies the program being proved by its digest rather than by comparing its code byte for byte. The collision therefore allows a valid STARK proof of an `AcceptAll` execution to be accepted as a proof of `NativeCurrency`, enabling the attacker to create and later spend counterfeit currency.

Finally, we present an attack on Plonky2 [37] that forges a Merkle-tree membership proof, allowing an attacker to prove that an arbitrary element belongs to a committed tree.

Together, these results demonstrate the possibility of a supply-chain attack in which an attacker replaces the intended matrix of a ZK-friendly hash instantiation with a maliciously crafted one, thereby implanting a kleptographic backdoor. Our case studies further show that such manipulation can have concrete impact on real-world systems, causing an otherwise unmodified verifier to accept a false statement, validate an inflationary transaction, or accept a forged Merkle-tree membership proof.

Remark (Concurrent work). Concurrently and independently of our work, Jo [31] and Vitto [43] also investigated malicious matrix selection in Poseidon. Our work additionally studies the resulting protocol-level consequences through three case studies involving Plonky3, Neptune Cash, and Plonky2. As these works appeared only shortly before our submission, we will provide a detailed technical comparison in a future revision.

2 Preliminaries

2.1 Notation

Throughout the paper, p denotes a prime and $\mathbb{F}_p$ denotes the finite field of order p . We use t for the permutation width, and $P: \mathbb{F}_p^t \rightarrow \mathbb{F}_p^t$ denotes a permutation on the correspond-

ing state space. We write P_M for the permutation obtained when its matrix is set to M . When P is used in a sponge construction, r and c denote the numbers of field elements in the rate and capacity portions of the state, respectively, with $r + c = t$. Vectors are column vectors and are written as $\vec{x} = (x_1, \dots, x_t) \in \mathbb{F}_p^t$; we write $\vec{x}[i]$ for the i -th coordinate of $\vec{x}$ and $\vec{x}[i]$ for the $(x_1, \dots, x_i) \in \mathbb{F}_p^i$. The concatenation of vectors $\vec{x}$ and $\vec{y}$ is denoted by $\vec{x} \parallel \vec{y}$. We write $(\vec{x}_1 \mid \dots \mid \vec{x}_N)$ for the matrix whose columns are $\vec{x}_1, \dots, \vec{x}_N$.

Matrices are denoted by uppercase letters. We write $\text{span}\{\vec{x}_1, \dots, \vec{x}_k\}$ for the linear span of a collection of vectors. For a scalar function $S : \mathbb{F}_p \rightarrow \mathbb{F}_p$, the notation $S(\vec{x})$ means that S is applied coordinatewise to $\vec{x}$.

2.2 Sponge Construction

The sponge construction [11] uses a fixed-width permutation to process and produce sequences of arbitrary length. It underlies Keccak [14] and many ZK-friendly hash functions [3, 23–25, 30]. Let $P : \mathbb{F}_p^t \rightarrow \mathbb{F}_p^t$ be a permutation. For notational convenience, we place the r rate elements before the c capacity elements, where $r + c = t$. After padding, the input is parsed into blocks in $\mathbb{F}_p^r$. The sponge starts from the all-zero state. In the standard construction considered here, absorption adds each input block to the rate portion of the state and then applies P . Once all blocks have been absorbed, the first output block is read from the rate portion. Further output blocks are obtained by applying P between successive reads of the rate.

CICO problem. When P is modeled as an ideal random permutation and r and c are chosen for the desired security level, the sponge construction is expected to resist generic preimage and collision attacks up to the standard bounds [12]. Cryptanalysis of a concrete ZK-friendly hash function instead seeks to exploit algebraic or structural properties of P that are absent from the ideal model. Preimage attacks of this kind can be studied by constraining selected coordinates of the input and output of P , leading to a constrained-input constrained-output (CICO) problem. A CICO problem parameterized by (n_I, n_O) fixes the values of n_I coordinates of the permutation input and n_O coordinates of the permutation output, and asks for an input–output pair satisfying those constraints.

Problem 1 formulates preimage attack for an r -element target as a CICO problem with $(n_I, n_O) = (c, r)$.

Problem 1 (Preimage). *Given a permutation $P : \mathbb{F}_p^t \rightarrow \mathbb{F}_p^t$, positive integers r and c satisfying $r + c = t$, and a target $\vec{z} \in \mathbb{F}_p^r$, find an input $\vec{x} \in \mathbb{F}_p^r$ and a vector $\vec{y} \in \mathbb{F}_p^c$ such that $P(\vec{x} \parallel \vec{y}) = \vec{z} \parallel \vec{y}$.*

2.3 Substitution-Permutation Network

A substitution–permutation network (SPN) [20] constructs a permutation by iterating nonlinear substitution layers and

invertible linear layers over a fixed-width state. The substitution layer applies a nonlinear S-box S independently to every state element:

$$\text{SBoxLayer}(\vec{x}) = (S(x_1), \dots, S(x_t)).$$

The S-box S must be invertible so that the resulting SPN is a well-defined permutation.

The linear layer is typically defined as

$$\text{LinLayer}(\vec{x}) = M\vec{x} + \vec{k},$$

where M is a $t \times t$ matrix and $\vec{k} \in \mathbb{F}_p^t$ is a round constant that is typically chosen at random, with a different value used in each round. The substitution layer provides nonlinearity, while the linear layer diffuses dependencies across the state. An SPN permutation is obtained by repeatedly composing these two layers.

A common design approach for ZK-friendly hash functions is to first construct an SPN permutation by selecting an S-box and a linear layer, and then build a hash function from this permutation using the sponge construction [11]. Representative examples following this design approach include Rescue-Prime [41], Tip5 [42], Griffin [23], and Polocolo [30].

2.4 HADES Design Strategy

HADES [27] is a variant of the SPN structure used in Poseidon [25] and Poseidon2 [26]. Instead of applying the S-box to every state element throughout the permutation, HADES uses R_F full rounds, in which the S-box is applied to every state element, and R_P partial rounds, in which the S-box is applied only to one state element. The permutation consists of $R_F/2$ full rounds, followed by R_P partial rounds, and another $R_F/2$ full rounds. The partial rounds increase the algebraic degree using fewer S-box evaluations, while the surrounding full rounds provide diffusion across the entire state. This combination reduces the arithmetic cost of the permutation while retaining resistance to both statistical and algebraic attacks. Throughout this paper, $R_f := R_F/2$ for the number of full rounds on either side of the partial-round block.

Poseidon. Poseidon [25] instantiates HADES over $\mathbb{F}_p^t$ with the power S-box $S(x) = x^\alpha$, where $\gcd(\alpha, p-1) = 1$. Each round consists, in order, of *AddRoundConstant*, *SubWords*, and *MixLayer*. In round i , *AddRoundConstant* maps the state as $\vec{x} \mapsto \vec{x} + \vec{k}_i$. *SubWords* then applies S to every state element in a full round or only to the first element in a partial round. Finally, *MixLayer* maps the state as $\vec{x} \mapsto M\vec{x}$ using the same $t \times t$ MDS matrix M in every round. Its round sequence therefore consists of R_f full rounds, R_P partial rounds, and another R_f full rounds.

Poseidon2. Poseidon2 [26] was proposed to address both the practical inefficiency of Poseidon’s linear layer and a security problem in the original design [6]. It preserves the HADES

round sequence and the power S-box but makes two main changes related to the linear layer. First, it uses separate matrices for the full and partial rounds. The external matrix M_E is applied to full rounds, whereas the internal matrix M_I is applied to partial rounds. The matrix M_I has all off-diagonal entries equal to 1, which makes its multiplication particularly efficient. Rather than requiring a full-state MDS matrix for M_E , Poseidon2 adopts the lower-cost matrix construction introduced for Griffin [23]. Second, Poseidon2 adds an additional linear layer at the beginning of the permutation, before the first round.

2.5 Linear Layer and MDS Matrices

In an SPN, the linear layer provides diffusion by spreading a change in one state element across many elements of the state.

For $\vec{x} \in \mathbb{F}_p^t$, let $\text{wt}(\vec{x})$ denote the number of its nonzero elements. The branch number of a matrix $M \in \mathbb{F}_p^{t \times t}$ is

$$B(M) = \min_{\vec{x} \neq \vec{0}} (\text{wt}(\vec{x}) + \text{wt}(M\vec{x})).$$

For an invertible $t \times t$ matrix, $B(M) \leq t + 1$. A matrix attaining the maximum value $t + 1$ is called maximum distance separable (MDS). Equivalently, M is MDS if every square submatrix of M is invertible. Matrices in SPNs are commonly chosen to be MDS because they provide the maximum possible diffusion and help resist statistical attacks. Some designs, including Griffin [23], Arion [40], and Poseidon2 [26], instead use non-MDS matrices in some linear layers to improve efficiency, accepting less than maximum diffusion.

2.6 Matrix Selection in ZK-Friendly Hash Functions

ZK-friendly hash functions use different methods to construct their matrices. Table 1 summarizes whether each specification fixes its matrix, whether the matrix is MDS, and how the matrix is constructed.

Poseidon [25] and Rescue-Prime [41] both require the matrix to be MDS and provide suggested matrices. Poseidon explicitly notes that allowing an attacker to choose either the round constants or the matrix freely can lead to attacks. However, Rescue-Prime explicitly states that any MDS matrix may be used.

Griffin [23] allows some flexibility in matrix selection, but the permitted choices depend on the state size. For $t \in \{3, 4\}$, it requires the matrix to be MDS while presenting M_3 and M_4 only as suggested choices. For $t \geq 8$, the full matrix cannot be chosen freely. Instead, it is constructed as $\text{diag}(M_4, M_4, \dots, M_4) \cdot \text{circ}(2I, I, \dots, I)$, where the 4×4 matrix M_4 must be MDS.

The remaining designs, namely Poseidon2 [26], Arion [40], Polocolo [30], and Tip5 [42], and Rescue-Prime Opti-

Table 1: Matrix selection in ZK-friendly hash-function specifications. In the Fixed matrices column, $\checkmark$, $\blacktriangle$, and $\times$ denote a matrix fixed by the standard, limited selection freedom under structural restrictions, and an MDS requirement with suggestions, respectively. The MDS column indicates whether the matrix is MDS. The Matrix construction column indicates how the matrix is constructed.

Hash Function	Fixed matrices	MDS	Matrix construction
Poseidon	$\times$	$\checkmark$	Cauchy
Rescue-Prime	$\times$	$\checkmark$	Vandermonde
Griffin	$\blacktriangle^\dagger$	$\times$	Structured
Poseidon2	$\checkmark$	$\times$	Structured
Arion	$\checkmark$	$\times$	Circulant
Polocolo	$\checkmark$	$\checkmark$	Heuristic
Tip5	$\checkmark$	$\checkmark$	Circulant
Rescue-Prime Optimized	$\checkmark$	$\checkmark$	Circulant

[†] For $t \in \{3, 4\}$, Griffin requires only that the matrix be MDS and presents M_3 and M_4 as suggested choices rather than fixed requirements. For $t \geq 8$, it instead specifies $\text{diag}(M_4, M_4, \dots, M_4) \cdot \text{circ}(2I, I, \dots, I)$.

mized [5], use fixed matrices. Rescue-Prime Optimized, Polocolo, and Tip5 use MDS matrices, whereas Poseidon2 and Arion use non-MDS matrices for efficiency.

3 Kleptographic Backdoors

Kleptography studies cryptographic systems that appear benign but contain a backdoor exploitable by a party who knows its hidden structure [44]. Such a backdoor is distinct from an accidental weakness or an implementation error. The system exhibits its expected behavior for ordinary users, while the informed party can trigger or exploit exceptional behavior.

3.1 Kleptography in Symmetric Primitives

Symmetric primitives may appear to offer limited room for a kleptographic backdoor because their algorithms and parameters are typically public and their behavior is deterministic. Nevertheless, malicious structure can be embedded in the design or its parameters while preserving the public interface and ordinary behavior of the primitive.

Prior work gives several examples of such design-level backdoors. Rijmen and Preneel [39] constructed S-boxes containing hidden linear correlations. Knowledge of these correlations enables linear cryptanalysis of the resulting block cipher, while the construction was intended to make the correlations difficult to recover from the public S-boxes. Young and Yung [45] showed that even a deterministic block cipher can carry a subliminal channel. Their construction encodes a key-dependent message across ciphertext blocks, allowing the

malicious designer to recover the user’s key after observing enough plaintext–ciphertext pairs.

Backdoors can also be embedded through apparently ordinary public parameters. Albertini et al. [1] modified only a small number of bits in the SHA-1 round constants and obtained collisions between valid binary files for the resulting hash function. Peyrin and Wang [36] built a high-probability related-tweak differential path into a tweakable block cipher. A designer who knows the corresponding tweak relation can use this path for key recovery, although the cipher otherwise retains its usual public interface and efficiency. These examples use S-boxes, output encoding, round constants, or tweak relations to carry hidden behavior within a symmetric design. Our setting uses a matrix for the same purpose.

3.2 Matrix Selection as a Kleptographic Attack Surface

We discover that a kleptographic backdoor can be embedded in a matrix. As discussed in Section 2.6, ZK-friendly hash specifications permit different ranges of choices when selecting a matrix. Rescue-Prime [41], for example, explicitly permits any MDS matrix, allowing a malicious parameter selector to introduce an attacker-crafted matrix without violating the stated MDS requirement. In practice, some real-world deployments even use matrices different from those presented in the corresponding specifications, as summarized in Table 2.

A deployment may also replace a specified matrix because of concerns about its security. Recent work shows that the non-MDS external matrices of Poseidon2 instances enable improved algebraic attacks [33]. Although these attacks do not necessarily invalidate the claimed security of the recommended parameters, they may provide a security rationale for replacing the specified external matrix with an MDS matrix. This result is specific to Poseidon2 but more generally, concerns about non-MDS matrices may motivate their replacement with MDS matrices in designs such as Arion [40] and Griffin [23].

More broadly, prior work on potential cryptographic backdoors [7, 10, 17] has repeatedly considered malicious modification of public cryptographic components or parameters. This threat is particularly relevant to matrix selection because there is no uniform convention for what constitutes an acceptable matrix in ZK-friendly hash functions.

Taken together, these observations support that malicious matrix selection represents a plausible supply-chain attack surface when a library maintainer, service provider, or system integrator is authorized to select or optimize the matrix.

3.3 Threat Model and Scope

Based on our analysis of matrices in hash-function specifications and real-world deployments, we adopt the following threat model for the remainder of this paper.

Table 2: Matrix selection in real-world deployments. The Spec Match column indicates whether the deployed matrix is identical to the matrix proposed in the original paper. In the Transparent Generation column, ✓ denotes a reproducible nothing-up-my-sleeve generation procedure, while ▲ indicates that the construction principle is clear but the procedure used to generate the matrix entries is not provided.

Hash Function	Project	Spec Match	Transparent Generation
Poseidon	Mina Protocol	✓	✓
	Starknet	✗	▲
	Polygon zkEVM	✗	▲
	Aleo	✓	✓
	Loopring v3	✗	✓
Poseidon2	Plonky3	✗	▲
	Aztec Protocol	✓	✓
Tip5	Neptune Cash	✓	✓
Rescue-Prime Optimized	Miden	✓	▲
Rescue	zkSync Lite	✗	✓

Threat Model. We consider a threat model in which the attacker can choose the public matrix before deployment. The chosen matrix must be MDS and must pass any additional matrix-security checks specified by the design. We impose the MDS requirement even when the target specification uses a non-MDS matrix, treating MDS as a conservative baseline for matrix security. The attacker is not required to follow the default matrix construction specified by the target hash function.

Scope. Under this threat model, the malicious matrix produced by our constructions meets the security requirements of the target design. Our constructions ensure functional validity and the required matrix security properties, but do not establish that the resulting matrices are indistinguishable from those generated by a benign procedure. In particular, a dense constructed matrix may appear nonstandard when the target implementation normally uses matrices with specific structures or small entries.

The construction allows the designer to plant either an input that hashes to a chosen digest or two distinct inputs that hash to the same digest. The relation $P_M(\vec{a}) = \vec{b}$ alone is unremarkable, since every input to a permutation has a corresponding output. Therefore, if the planted input or inputs remain secret, exploiting the backdoor requires finding those inputs using only the public hash instance. We expect this

search problem to be comparable to generic search against a benignly instantiated hash function.

We emphasize that every matrix surveyed in Table 2 is either reproducible from a public generation procedure or has a clearly identifiable construction principle. Our results are not attacks on Poseidon, Poseidon2, or any other hash function instantiated with the matrix proposed in its original paper. Similarly, the case studies describe the consequences that could follow if a maliciously generated matrix were substituted into the corresponding system. These case studies do not show that any surveyed project has been compromised or that its deployed matrix contains a backdoor.

4 Adversarial Matrix Construction

In this section, under the threat model described in Section 3.3, we show how an attacker who controls the choice of the matrix M can make an SPN permutation map an attacker-chosen input to an attacker-chosen output.

We first present a generic adversarial matrix construction that does not depend on the particular choice of invertible S-box. It does not apply directly to Poseidon or Poseidon2 because the linear constraints accumulated over their many rounds make the resulting system overdetermined. We therefore give separate constructions that exploit Poseidon’s partial-round structure and Poseidon2’s use of distinct internal and external matrices.

In the sponge setting, this control directly yields preimages and collisions. A solution to Problem 1 gives a preimage, while enforcing equal rate outputs for two distinct one-block messages gives a collision.

4.1 Generic Construction

We consider an SPN permutation $P: \mathbb{F}_p^t \rightarrow \mathbb{F}_p^t$ of width t . Let $M \in \mathbb{F}_p^{t \times t}$ be the matrix chosen by the attacker, and let N be the total number of times that matrix multiplies the state. For $1 \leq i \leq N$, let $\vec{u}_i$ and $\vec{v}_i$ denote the input and output when M multiplies the state for the i -th time, respectively. These values impose the conditions

$$M\vec{u}_i = \vec{v}_i \quad \text{for } 1 \leq i \leq N. \quad (2)$$

Let $\vec{a}, \vec{b} \in \mathbb{F}_p^t$ be the desired permutation input and output, respectively. The attacker’s goal is to choose a matrix M such that

$$P_M(\vec{a}) = \vec{b}.$$

Let $\vec{k}_1, \dots, \vec{k}_N$ be the round constants. By the SPN round structure, the first matrix input is $\vec{u}_1 = S(\vec{a} + \vec{k}_1)$, the final matrix output is $\vec{v}_N = \vec{b}$, and $\vec{u}_{i+1} = S(\vec{v}_i + \vec{k}_{i+1})$ for $1 \leq i < N$.

Basic Construction. Starting from the fixed value $\vec{u}_1$, the attacker chooses $\vec{v}_i$ uniformly at random and then sets $\vec{u}_{i+1} = S(\vec{v}_i + \vec{k}_{i+1})$ for $i = 1, \dots, N-1$. The final value $\vec{v}_N = \vec{b}$ is

already fixed by the desired output. This procedure produces all N input–output pairs consistently with the SPN structure. The attacker then chooses M to satisfy Equation (2). Write

$$U = (\vec{u}_1 \mid \dots \mid \vec{u}_N), \quad V = (\vec{v}_1 \mid \dots \mid \vec{v}_N),$$

so that the matrix conditions become $MU = V$.

Assume that $\vec{u}_1, \dots, \vec{u}_N$ are linearly independent, as are $\vec{v}_1, \dots, \vec{v}_N$. Since $\mathbb{F}_p^t$ has dimension t , each list can contain at most t linearly independent vectors. For $N \leq t$, append columns to U and V until both are invertible $t \times t$ matrices. The matrix $M = VU^{-1}$ is then invertible and satisfies all the original conditions. The resulting bound is

$$N \leq t. \quad (3)$$

Reducing Constraints by Pairing. In the basic construction, each matrix multiplication contributes a distinct input–output constraint on M . We increase the bound on N from t to $2t-1$ by making two multiplications share the same constraint. For $N > t$, set $L = N - t$ and create L such duplicate conditions. For $i = 1, \dots, L$, write $j := N + 1 - i$ for the index paired with i . Their conditions on M are identical when

$$\vec{u}_j = \vec{u}_i, \quad \vec{v}_j = \vec{v}_i. \quad (4)$$

We enforce these equalities recursively. At the first step, the desired permutation input determines $\vec{u}_1$, and the desired permutation output determines $\vec{v}_N$. For $i = 1$, we have $j = N$; set $\vec{v}_1 := \vec{v}_N$ and $\vec{u}_N := \vec{u}_1$. The two matrix conditions then both reduce to $M\vec{u}_1 = \vec{v}_N$.

Continuing this recursion, suppose that $\vec{u}_i$ and $\vec{v}_j$ are known at the i -th step. Set $\vec{v}_i := \vec{v}_j$ and $\vec{u}_j := \vec{u}_i$, making the two matrix conditions identical. We then compute $\vec{u}_{i+1} = S(\vec{v}_i + \vec{k}_{i+1})$ and $\vec{v}_{j-1} = S^{-1}(\vec{u}_j) - \vec{k}_j$. Since $j-1 = N+1-(i+1)$, these are precisely the two known values needed for the next pairing step. Repeating this process for $i = 1, \dots, L$ establishes Equation (4) for all L pairs. Any intermediate outputs left undetermined after pairing are chosen uniformly at random. The L pairings reduce the number of distinct matrix conditions from N to

$$N - L = N - (N - t) = t.$$

These t distinct pairs can therefore be used to construct M as in the basic construction.

To see that this recursion is well defined, consider $N > t$. At pairing step i , the construction fixes $\vec{u}_j$ and $\vec{v}_i$, then computes $\vec{u}_{i+1}$ and $\vec{v}_{j-1}$. No state variable is assigned twice as long as $i+1 < j$. It suffices to check the final step $i = L$, where $j = N + 1 - L$. The condition becomes $2L < N$; substituting $L = N - t$ gives $N < 2t$, or equivalently

$$N \leq 2t - 1. \quad (5)$$

The bound in Equation (5) is met by parameter sets of several ZK-friendly hash functions. Appendix B reports the total

number of times that matrix multiplies the state for Poseidon, Rescue-Prime, Griffin, Poseidon2, Arion, Polocolo, and Tip5 and evaluates which instances satisfy the bound.

Appendix A presents pseudocode for the complete construction as Algorithm 1.

Matrix Validation. Algorithm 1 returns $\perp$ if either U or V is not full rank. When the attacker samples the unconstrained coordinates of $\vec{a}$ and $\vec{b}$, together with the free vectors in the algorithm, uniformly at random, we expect this failure probability to be negligible over the large fields considered here¹. Additionally, consistency with our threat model requires a returned matrix M to be MDS and to satisfy the other security requirements of the target design, such as the absence of long subspace trails. Experimentally, we find that a random matrix satisfies the long-subspace-trail condition with high probability. Next, the MDS condition requires every square minor of M to have nonzero determinant. A $t \times t$ matrix has

$$\sum_{k=1}^t \binom{t}{k}^2 = \binom{2t}{t} - 1$$

such minors. Heuristically treating the candidate matrices produced by the construction as uniformly random and the probability that any given determinant is zero as $1/p$, we expect to find an MDS matrix with reasonable probability when $\binom{2t}{t} < p$. For $t = 16$, $\binom{32}{16} \approx 2^{29.16}$, while for $t = 24$, $\binom{48}{24} \approx 2^{44.87}$. These values are below the sizes of 32-bit and 64-bit prime fields, respectively. If any of these checks fail, the attacker repeats the construction with fresh random choices consistent with the CICO constraints.

4.2 Instantiation for Poseidon

The generic construction of Section 4.1 requires the total number of times that M multiplies the state to be at most $2t - 1$. In Poseidon, M multiplies the state once per round, and $R = R_F + R_P > 2t - 1$ for the parameter sets we consider. The generic construction therefore does not apply directly. We increase this bound by exploiting the structure of Poseidon’s partial rounds.

At a high level, our key observation is that the S-box layer of a partial round acts as the identity when its single S-box input is zero. Instead of constraining the entire state, we force only this S-box input to zero, reducing the number of linear constraints contributed by each partial round from t to 1. Let $R_f := R_F/2$ denote the number of full rounds in each of the two full-round blocks. For the full rounds, we apply the pairing technique of Section 4.1. This leaves $R_f + 1$ distinct matrix

¹When two or more input–output relations are imposed on the same permutation, independently applying Algorithm 1 with the same pairing schedule and combining the resulting constraints can introduce deterministic linear dependencies that force the constructed matrix to be singular. Possible ways to avoid these dependencies include using relation-dependent proportional pairings, with $(\vec{u}_j, \vec{v}_j) = C(\vec{u}_i, \vec{v}_i)$ for a nonzero scalar C , or leaving selected rounds unpaired and choosing their state vectors independently.

conditions, which contribute $t(R_f + 1)$ scalar constraints. The resulting bound is

$$R_P + t(R_f + 1) \leq t^2. \quad (6)$$

However, reducing the number of linear constraints per partial round from t to 1 introduces a complication. Unlike in Section 4.1, the remaining $t - 1$ state elements in each partial round are not fixed before solving for M . Instead, they are expressions in M determined by the preceding rounds. Consequently, expressing these zero-input constraints directly in terms of M introduces powers $M, M^2, M^3, \dots$, so the resulting equations are no longer linear in the entries of M . We therefore restrict M to a special form that makes these conditions linear.

Reducing Partial-Round Constraints. As in Section 4.1, let $\vec{u}_i$ and $\vec{v}_i = M\vec{u}_i$ denote the input and output of the i -th matrix multiplication. The matrix inputs in the partial rounds are $\vec{u}_{R_f+1}, \dots, \vec{u}_{R_f+R_P}$, and we write $\vec{k}_i = (k_{i,1}, \dots, k_{i,t})^\top$ for the i -th round constant. To describe this reduction, we first treat $\vec{u}_{R_f}$, which is the input to the matrix multiplication immediately before the partial-round block, as a constant. We force the S-box input of each partial round to zero and write the corresponding matrix input as

$$\vec{u}_i = (0, u_{i,2}, \dots, u_{i,t})^\top \quad \text{for } R_f + 1 \leq i \leq R_f + R_P. \quad (7)$$

We write M as follows.

$$M = \begin{pmatrix} M_{1,1} & M_{1,2} & M_{1,3} & \cdots & M_{1,t} \\ M_{2,1} & M_{2,2} & M_{2,3} & \cdots & M_{2,t} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{t,1} & M_{t,2} & M_{t,3} & \cdots & M_{t,t} \end{pmatrix}.$$

Instead of treating all t^2 entries of M as variables, we treat only the first-column entries $M_{1,1}, \dots, M_{t,1}$ as variables. The attacker randomly chooses the remaining entries $M_{j,k}$ for $1 \leq j \leq t$ and $2 \leq k \leq t$ and treats them as constants when solving the linear system.

Since $u_{i,1} = 0$, the terms $M_{j,1}u_{i,1}$ vanish in every partial-round matrix multiplication, so the first-column entries need not be fixed before solving. Treating the entries $M_{j,k}$ with $k \geq 2$ as constants avoids bilinear products between matrix entries and state elements.

The purpose of this structure is to ensure that, throughout the partial-round block, the first element of every $\vec{u}_i$ is zero while all its other elements are linear in the first-column entries of M .

We begin with the first partial round. The input to its S-box layer is

$$M\vec{u}_{R_f} + \vec{k}_{R_f+1}.$$

Since $\vec{u}_{R_f}$ is fixed, requiring its first element to be zero gives the linear condition

$$M_{1,1}u_{R_f,1} + \sum_{k=2}^t M_{1,k}u_{R_f,k} = -k_{R_f+1,1}.$$

The S-box maps this zero to zero, while the remaining state elements pass through unchanged. Hence, the first partial-round matrix input $\vec{u}_{R_f+1}$ satisfies $u_{R_f+1,1} = 0$ and, for $2 \leq j \leq t$,

$$u_{R_f+1,j} = M_{j,1}u_{R_f,1} + \sum_{k=2}^t M_{j,k}u_{R_f,k} + k_{R_f+1,j}.$$

Thus, in $\vec{u}_{R_f+1}$, the first element is zero and every other element is linear in the first-column entries of M . We now show that these properties carry from $\vec{u}_i$ to $\vec{u}_{i+1}$.

Suppose that $R_f + 1 \leq i < R_f + R_p$, $u_{i,1} = 0$, and each $u_{i,j}$ for $2 \leq j \leq t$ is linear in the first-column entries of M . The matrix multiplication in round i gives

$$v_{i,j} = \sum_{k=2}^t M_{j,k}u_{i,k} \quad \text{for } 1 \leq j \leq t,$$

which shows that every $v_{i,j}$ is also linear in the first-column entries of M .

The input to the $(i+1)$ -th S-box layer is $\vec{v}_i + \vec{k}_{i+1}$. Requiring its first element to be zero gives

$$\sum_{k=2}^t M_{1,k}u_{i,k} = -k_{i+1,1},$$

which gives one linear constraint. The output of the S-box layer then satisfies

$$u_{i+1,1} = 0, \quad u_{i+1,j} = v_{i,j} + k_{i+1,j} \quad \text{for } 2 \leq j \leq t.$$

This update only adds round constants to the elements of $\vec{v}_i$. Therefore, $u_{i+1,1} = 0$, and every other element of $\vec{u}_{i+1}$ is again linear in the first-column entries of M . Repeating this argument covers the entire partial-round block and shows that requiring the single S-box input to be zero imposes 1 linear constraint in each partial round.

Increasing the Number of Variables. The current structure of M , in which only the first column is variable, reduces the number of constraints contributed by each partial round from t to 1. However, it also reduces the number of variables from t^2 to t , so it does not achieve the bound in Equation (6).

Let M' have the structure introduced to reduce the partial-round constraints. Its first-column entries $M'_{j,1}$ are variables, and its remaining entries $M'_{j,k}$ for $2 \leq k \leq t$ are randomly chosen and treated as constants. We increase the number of variables by taking the actual matrix M to be a matrix similar to M' , i.e., $M = T^{-1}M'T$ for an invertible matrix T .

To keep the S-box input of each partial round equal to zero, we set $T_{1,2} = \dots = T_{1,t} = 0$. Since multiplying T by a common nonzero scalar does not change $T^{-1}M'T$, we normalize $T_{1,1} = 1$ and define T as

$$T = \begin{pmatrix} 1 & 0 & \dots & 0 \\ T_{2,1} & T_{2,2} & \dots & T_{2,t} \\ \vdots & \vdots & \ddots & \vdots \\ T_{t,1} & T_{t,2} & \dots & T_{t,t} \end{pmatrix}, \quad (8)$$

where every entry denoted by $T_{j,k}$ is a variable.

Intuitively, this similarity transformation allows each condition $M\vec{u} = \vec{v}$ to be written equivalently as $M'T\vec{u} = T\vec{v}$. The transformation increases the number of variables while retaining the reduction in the number of constraints contributed by each partial round.

Under these structures, both $M'T$ and T are linear in the matrix variables. Consequently, each equivalent matrix condition

$$M'T\vec{u} = T\vec{v}$$

remains linear.

The matrices T and M' contribute $t^2 - t$ and t variables, respectively, giving t^2 variables in total.

Full Instantiation. We apply pairing to the two full-round blocks and use the reduced-constraint recurrence inside the partial-round block. Let $R = R_f + R_p$. We first pair the first $R_f - 1$ full rounds with the last $R_f - 1$ full rounds, matching round i with round $R + 1 - i$. Each pair is constructed to impose the same matrix condition, reducing these $2(R_f - 1)$ conditions to $R_f - 1$. The two remaining full rounds give the independent linear conditions $M\vec{u}_{R_f} = \vec{v}_{R_f}$ and $M\vec{u}_{R_f+R_p+1} = \vec{v}_{R_f+R_p+1}$. Hence, the full rounds contribute $(R_f - 1) + 2 = R_f + 1$ distinct matrix conditions. Starting from the desired permutation input and output, these pairings determine $\vec{u}_{R_f}$ and $\vec{v}_{R_f+R_p+1}$, respectively. We then choose $\vec{u}_{R_f+R_p+1}$ and set

$$\vec{v}_{R_f+R_p} = S^{-1}(\vec{u}_{R_f+R_p+1}) - \vec{k}_{R_f+R_p+1},$$

We then apply the reduced partial-round constraints. Let $\vec{u}'_i$ denote the transformed partial-round states, which are recursively computed as linear expressions rather than introduced as independent variables. The combined equations are

$$M'T\vec{u}_i = T\vec{v}_i, \quad 1 \leq i < R_f, \quad (9)$$

$$\vec{u}'_{R_f+1} := M'T\vec{u}_{R_f} + T\vec{k}_{R_f+1}, \quad (10)$$

$$\vec{u}'_{i+1} := M'\vec{u}'_i + T\vec{k}_{i+1}, \quad (11)$$

$$u'_{i,1} = 0, \quad (12)$$

$$M'\vec{u}'_{R_f+R_p} = T\vec{v}_{R_f+R_p}, \quad (13)$$

$$M'T\vec{u}_{R_f+R_p+1} = T\vec{v}_{R_f+R_p+1}. \quad (14)$$

Equation (9) contributes $t(R_f - 1)$ scalar constraints. Equations (10) and (11) express the intermediate partial-round states as linear expressions in the matrix variables and do not add constraints to the system, where Equation (11) applies for $R_f + 1 \leq i < R_f + R_p$. Equation (12), imposed for $R_f + 1 \leq i \leq R_f + R_p$, contributes R_p scalar constraints, and Equations (13) and (14) each contribute t . Thus, the system has

$$t(R_f - 1) + R_p + 2t = t(R_f + 1) + R_p$$

scalar constraints. Since there are t^2 matrix variables, the system can be solved as long as Equation (6) holds. Appendix A

presents pseudocode for construction as Algorithm 2. If M is non-MDS or fails a required matrix-security check, we repeat the algorithm with fresh random choices.

4.3 Instantiation for Poseidon2

Unlike Poseidon, Poseidon2 applies the external matrix M_E once at the beginning of the permutation, in addition to applying it after each of the R_F full-round S-box layers. Thus, M_E multiplies the state $R_F + 1$ times. The internal matrix M_I is used only in the partial rounds. Our key observation is that this separation allows the attacker to modify only M_E while leaving M_I , the round constants, and all partial-round operations unchanged. The complete partial-round block can then be treated as a fixed invertible transformation and imposes no additional conditions on M_E .

Index the applications of M_E by $0, \dots, R_F$, and let $\vec{u}_i$ and $\vec{v}_i$ be the input and output of application i , respectively. In particular, $\vec{u}_0 = \vec{a}$ and $\vec{v}_{R_F} = \vec{b}$. Let Φ_{M_I} denote the fixed invertible transformation from $\vec{v}_{R_F}$, the output of the last external-matrix application in the first full-round block, to $\vec{u}_{R_F+1}$, the input to the first external-matrix application in the second full-round block. This transformation consists of the R_P partial rounds followed by the round-constant addition and S-box layer of the first full round in the second block.

Once the partial-round block is absorbed into this fixed transformation, the remaining conditions involve only the $R_F + 1$ applications of M_E . We can therefore apply the generic construction of Section 4.1, and in particular its pairing technique, directly to M_E . We pair application i with application $R_F - i$ for $0 \leq i < R_F$ by setting

$$\vec{u}_{R_F-i} = \vec{u}_i, \quad \vec{v}_{R_F-i} = \vec{v}_i.$$

The states are computed inward from the desired input and output. Within the two full-round blocks, the next state is obtained using the S-box and round constant as in the generic pairing construction. At the partial-round block, the backward computation instead uses $\vec{v}_{R_F} = \Phi_{M_I}^{-1}(\vec{u}_{R_F+1})$. The R_F pairs impose R_F distinct conditions on M_E , and application R_F imposes one additional condition. Hence, the construction requires only $R_F + 1$ conditions of the form $M_E \vec{u}_i = \vec{v}_i$. These conditions can be extended to determine an external matrix whenever

$$R_F + 1 \leq t. \quad (15)$$

Appendix A presents pseudocode for the complete construction as Algorithm 3. As in Section 4.1, the attacker repeats the random extension step if the resulting matrices are not full rank or if M_E fails a required matrix-security check.

5 Case Study 1: Forging a Plonky3 Proof

In this section we discuss how choosing the matrix of ZK-friendly hash functions such as Poseidon2 could be used by

protocol implementers to backdoor cryptographic protocols in which a challenger built from a hash function plays the role of the random oracle after Fiat–Shamir transformation [21] of a public-coin interactive protocol. For such protocols to be sound, it is crucial that the prover’s earlier messages are fixed before the verifier’s challenge is known. Our backdoor allows breaking this assumption: An attacker that is able to choose the hash function’s matrix proceeds by first choosing the challenge jointly with the earlier prover messages in such a way that the verifier’s check is satisfied, and then constructs the backdoored hash instantiation with precisely this input-output pair.

5.1 Plonky3

Proofs or arguments of knowledge are one source of examples for such protocols. To enable proof recursion, the verifiers are themselves often encoded in a constraint system, and thus make use of ZK-friendly hash functions for their challenger. The sketched attack against such systems that an implementer may carry out by backdooring the ZK-friendly hash function applies in general. To show this in a concrete example, we build upon Plonky3 [38].

Plonky3 is a widely used set of Rust crates that offers, in particular, an implementation of a STARK proving system, as well as primitives required for it, such as hash functions and polynomial commitment schemes. The STARK system is highly customizable: Projects using it can configure it to use a Fiat-Shamir challenger and polynomial commitment scheme of their choice. The provided `DuplexChallenger`² is in turn highly configurable, with in particular the cryptographic permutation configurable, with Plonky3’s provided Poseidon2 permutation being a standard choice.

In the rest of this section, we describe how a backdoored external matrix for the Poseidon2 permutation used in the `DuplexChallenger` could be used to make the Plonky3 STARK verifier accept an incorrect proof.

5.2 The attacked constraint system

We make use of a simple toy circuit in our proof of concept. The public values are two field elements (n, m) , and the statement the circuit is intended to enforce is $m = 2 \cdot n$. This is accomplished with a circuit whose witness consists of $h = 16$ rows of width 8, in which we impose the following constraints:

1. The first row’s first two values are $(0, 0)$.
2. Each row $\vec{x}$ satisfies, for $0 \leq i < 3$ that $\vec{x}[1 + 2 \cdot (i + 1)] = \vec{x}[1 + 2 \cdot i] + 1$ and $\vec{x}[1 + 2 \cdot (i + 1) + 1] = \vec{x}[1 + 2 \cdot i + 1] + 2$.

²A Fiat–Shamir challenger following the duplex sponge construction from [13] as analyzed in [19].

3. Each row $\vec{y}$ that is not the first, with $\vec{x}$ the previous row, satisfies $\vec{y}[1] = \vec{x}[7] + 1$ and $\vec{y}[2] = \vec{x}[8] + 2$.
4. The last two entries of the last row are equal to (n, m) .

Our goal is to backdoor the external matrix used in the challenger such that we can produce a proof for this constraint system with public values $(1, 3)$, seemingly suggesting that $3 = 2 \cdot 1$.

5.3 Constraint batching in the Plonky3 STARK verifier

The first reduction carried out by the protocol implemented by the Plonky3 STARK verifier is to reduce checking a list of constraints to checking a single constraint, a standard application of the Schwartz-Zippel lemma.

Simplifying the exposition by only considering transition constraints³, the initial task of the verifier is to check for a list of polynomials in 16 variables $f_0, \dots, f_k$ that the committed trace satisfies, for each successive rows $\vec{x}$ and $\vec{y}$ and each $0 \leq i \leq k$, that $f_i(\vec{x} \parallel \vec{y}) = 0$. For this the verifier samples a random element γ of an extension field $\mathbb{F}_{p^e}$ large enough to obtain negligible soundness error, and is left to check that for each successive rows $\vec{x}$ and $\vec{y}$ it holds that $(\sum_{i=0}^k f_i \cdot \gamma^{k-i})(\vec{x} \parallel \vec{y}) = 0$.

In the (non-interactive) implementation, γ is obtained using the `DuplexChallenger`, which implements a variant of the sponge construction described in Section 2.2, the main difference being that rather than adding the new elements in, absorption overwrites (parts of) the rate portion of the state. The challenger initially absorbs some configuration parameters, a commitment to the trace, and the public values. The first e elements squeezed are then assembled into an element γ of $\mathbb{F}_{p^e}$.

5.4 Summary of the proof of concept

The proof of concept now works as follows: The attacker⁴ chooses public values of their choice⁵, and computes a witness together with a value γ so that the batched constraints are satisfied. In our implementation, we allow free choice of γ . We then fill the witness with legitimate values in all but the last row, as well as the first two entries of the last row. This ensures that the constraints will be satisfied for all but possibly the last row. We then solve the batched constraints for the last row for the remaining six entries.

³Those involving two successive rows, like constraints of type 3 in our example. We emphasize that we only simplify the notation here, in practice all constraints are batched together, not just transition constraints.

⁴In this scenario, the attacker would be someone involved in the development of a library such as Plonky3, or in the development of a project that makes use of Plonky3.

⁵In our implementation, we use $(1, 3)$.

Having obtained γ and our witness, we compute the commitment for the witness, with which we can now compute the input to be absorbed by the `DuplexChallenger`. Now the attack described in Section 4.3 can be applied⁶ to compute an external matrix for the given input-output pair. This allows the attacker to construct a false proof that Plonky3's standard, unmodified verifier accepts, by only changing the external matrix used in the Poseidon2 permutation function used for the challenger. Note that after solving for the matrix, a full check for the MDS property is then carried out. On failure of the matrix to be MDS, the attack is retried with fresh randomness. As the resulting matrix is MDS, unlike the standard external matrix for Poseidon2 for widths greater than four, the attacker may justify their deviation from the standard Poseidon2 matrix by referencing recent work [33] and arguing that their provably MDS matrix is more secure than the standard one. See sections 3.2 and 3.3 for more details.

Our proof of concept uses a configuration of the Plonky3 STARK verifier with a `DuplexChallenger` using a custom Poseidon2 permutation of width $t = 16$ and rate $r = 8$. The field used is KoalaBear where $p = 2^{31} - 2^{24} + 1$, with γ sampled from a field extension of degree $e = 4$. The custom Poseidon2 permutation differs from the standard Plonky3 implementation only in that the external matrix is configurable. We choose some other STARK configuration parameters differently than usual defaults⁷ to obtain a small instance, in which the number of field elements absorbed before squeezing γ is 13, thus requiring two permutation applications. Other configurations may involve three or more permutation applications, which will increase the number of constraints, making the attack more difficult.

5.5 Impact and Scope

The presented proof of concept shows how choice of the matrices used in the Poseidon2 permutation in the challenger could be used to insert a backdoor that allows creating specific false proofs that an otherwise sound and unmodified proof system accepts. The Plonky3 STARK verifier acts as an example; the same technique applies to many similar proving systems.

We note that the standard Plonky3 configuration would also use a Poseidon2 permutation with the same width as the challenger in the polynomial commitment system. This would complicate the creation of a backdoor using the described method if the same matrices are used for both, as the commitment to the witness would then also depend on the matrix to be backdoored. This would make the attack significantly more difficult and possibly infeasible with the described technique. In our proof of concept, we only replace the standard

⁶While Section 4.3 focuses only on the case of backdooring the evaluation of a single permutation, the configuration that we target applies the permutation twice before sampling γ . Beyond increasing the number of constraints imposed this does not cause any issues however.

⁷Without intended impact on the soundness of the protocol.

external matrix in the instantiation for the challenger. In a real-world attempt to backdoor a similar protocol, the attacker may use various arguments to avoid suspicions based on making use of two Poseidon2 instantiations with the same width but different matrices. For example, they may use a polynomial commitment scheme that is not hash based at all, come up with reasonable explanations why a different hash function is more efficient in the polynomial commitment scheme, or make use of a polynomial commitment scheme from a library as a black box while the challenger is implemented from scratch.

6 Case Study 2: Creating Counterfeit Currency in Neptune Cash

Neptune Cash [35] is a cryptocurrency that verifies transaction validity using STARK proofs of virtual machine executions. Tip5 [42] is a hash function constructed by instantiating the sponge construction with an SPN permutation, as discussed in Section 2.3. It is designed for recursive STARKs and serves as a core hashing primitive in Neptune Cash. Its S-box decomposes each element of $\mathbb{F}_p$ into smaller integers using its binary representation, applies a permutation to each integer, and reconstructs a field element from the results. This S-box has high algebraic complexity over $\mathbb{F}_p$, allowing Tip5 to use small rounds. Its specification fixes a state width of 16 and 5 rounds, making it particularly well suited to the generic construction described in Section 4.1.

In this section, we show that an adversary who can choose Tip5’s matrix can embed a collision that bypasses Neptune’s native-currency conservation check. This allows the adversary to create and spend unauthorized native currency.

6.1 High-Level Overview

Neptune checks transactions with a virtual machine program called `NativeCurrency`. Neptune identifies a virtual machine program by its Tip5 digest. A verifier checks a STARK proof for the required digest, but it does not compare the supplied program with `NativeCurrency` byte for byte. In virtual machine, a program execution succeeds if it reaches `halt` without an error.

Now consider an `AcceptAll` program that accepts a transaction without checking whether it creates new currency. If the attacker can use this program in place of `NativeCurrency`, the attacker can easily create counterfeit coins. Let $\vec{p}_{\text{NC}}$ and $\vec{p}_{\text{A}}$ denote serialized form of program `NativeCurrency` and `AcceptAll`, respectively. The attacker chooses a matrix M such that $H_M(\vec{p}_{\text{NC}}) = H_M(\vec{p}_{\text{A}})$. Since the two program digests are equal, Neptune accepts this proof as the required `NativeCurrency` proof. The STARK proof is valid, but it proves the execution of `AcceptAll` instead of `NativeCurrency`.

6.2 Constructing the Program Collision

Tip5 has state width $t = 16$, rate $r = 10$, capacity $c = 6$, and 5 rounds. Tip5 absorbs a block by overwriting the 10 rate elements rather than adding the block to them. Thus, if two states have the same 6 capacity elements, absorbing the same next block makes the entire states identical. This overwrite rule allows the attacker to change only the first block and keep all subsequent blocks identical.

The encoded `NativeCurrency` program contains 2,074 field elements. After padding, Tip5 divides the program into 208 blocks of 10 elements and absorbs each block once. Controlling all 208 permutation calls with one matrix is infeasible. Instead, the attacker constructs $\vec{p}_{\text{A}}$ so that only its first block differs from $\vec{p}_{\text{NC}}$; the remaining 207 blocks are identical.

The first blocks are

$$\begin{aligned}\vec{x}_{\text{NC}} &= (33, 15, 33, 15, 33, 15, 33, 15, 33, 15), \\ \vec{x}_{\text{A}} &= (73, 5, 73, 5, 73, 5, 0, 8, 8, 8).\end{aligned}$$

The vector $\vec{x}_{\text{A}}$ encodes 3 `read_io 5` instructions, followed by `halt` and 3 `nop` instructions. It reads all 15 public input elements and halts without checking them. The remaining instructions are copied from `NativeCurrency`, but they are not executed. Thus, `AcceptAll` accepts every transaction.

The attacker chooses M so that the 6 capacity elements are equal after the first blocks are permuted. Since all subsequent blocks are identical, the next absorption makes the two states identical. All remaining hash operations then produce the same result, so the complete programs have the same digest.

Applying the pairing construction from Section 4.1 requires 3 mapping constraints for each program. The attacker uses these 6 constraints to construct an MDS matrix M .

6.3 Impact and Scope

The attacker creates a transaction whose output exceeds its input by 1 coin. Although `NativeCurrency` would reject this transaction, the program collision described in Section 6.2 allows the attacker to replace it with `AcceptAll`, which performs no balance check. Neptune therefore accepts the proof and adds the counterfeit coin to the ledger. The attacker can later spend this coin using the genuine `NativeCurrency`. Even if another party later discovers the same colliding `AcceptAll`, the zero-knowledge proof does not reveal whether a past transaction executed `NativeCurrency` or `AcceptAll` since programs have the same digest.

This gives the matrix designer plausible deniability. The designer can claim that the collision occurred unintentionally, while the matrix and transaction history alone provide no evidence of how the matrix was generated or whether the collision was exploited. Unless the generation process or other external evidence is available, it is difficult to refute this claim. The construction therefore functions effectively as a backdoor while concealing both intent and use.

7 Case Study 3: Forging Merkle Membership in Plonky2

Plonky2 [37] provides a generic native Merkle-tree abstraction whose internal nodes use Poseidon-based compression. We show that a malicious parameter designer can construct a dense matrix with a targeted collision in its compression output. In the demonstrated tree built using this matrix, the collision allows an absent leaf to be accepted at the index and depth of an honest leaf after changing only the first sibling in its authentication path.

The matrix is generated before the tree is built and is used consistently by the resulting hash instance. Our result is an existence demonstration for a selected tree instance.

7.1 High-Level Overview

Merkle membership verification begins with the digest of the claimed leaf and repeatedly compresses it with the ordered sibling at each level. The claimed index determines the child ordering, and the final value must be equal to the agreed root. When a leaf contains exactly four field elements, Plonky2 uses those elements directly as its digest rather than first hashing them. Thus, verification begins from the leaf itself.

Let $\vec{x}, \vec{y} \in \mathbb{F}_p^4$ be an adjacent pair in the honest tree, and $\vec{x}', \vec{y}' \in \mathbb{F}_p^4$ a replacement pair with the same ordering. We construct the matrix so that both pairs compress to the same parent. An authentication path for $\vec{y}'$ can then replace the honest first sibling $\vec{x}$ with $\vec{x}'$. At the first level, the forged computation reaches the honest parent and immediately rejoins the original path. The remaining siblings, path length, index, and root are therefore unchanged. This one planted collision suffices for the demonstrated membership forgery, but does not give arbitrary openings for unrelated trees.

7.2 Constructing the Compression Collision

Plonky2's Poseidon instance operates over the Goldilocks field $\mathbb{F}_p$, where $p = 2^{64} - 2^{32} + 1$, with $t = 12$, $\alpha = 7$, $R_F = 8$, and $R_P = 22$. We retain its round constants and replace only M . Compression of two child digests is

$$H_M(\vec{x}, \vec{y}) := P_M(\vec{x} \parallel \vec{y} \parallel 0 \parallel 0 \parallel 0 \parallel 0)[4]. \quad (16)$$

The target relation is

$$H_M(\vec{x}, \vec{y}) = H_M(\vec{x}', \vec{y}'). \quad (17)$$

The full permutation outputs differ in our construction, so Equation (17) is a collision in H_M , not in P_M .

Realizing Equation (17) requires one matrix to satisfy two complete Poseidon input-output relations.

Applied independently, the construction of Section 4.2 imposes $R_P + t(R_F + 1) = 82$ scalar constraints for each relation. The two relations would consequently impose 164 constraints

on the same 144 matrix variables. To remove this obstruction, we generalize the full-round equality pairings to scaled pairings. For a nonzero scalar ρ , the full-round S-box satisfies $S(\rho\vec{z}) = \rho^\alpha S(\vec{z})$. Because the collision constrains only the retained coordinates, neither full permutation output must be fixed in advance. Writing $C = \rho^\alpha$, we use this freedom to arrange one additional proportional pairing in each trace:

$$\vec{u}_{R_f+R_P+1} = C\vec{u}_1, \quad \vec{v}_{R_f+R_P+1} = C\vec{v}_1.$$

By linearity of M , the final full-round condition $M\vec{u}_{R_f+R_P+1} = \vec{v}_{R_f+R_P+1}$ is then just C times the first condition $M\vec{u}_1 = \vec{v}_1$. Thus, Equation (14) contributes no independent constraints. Each relation consequently contributes $t(R_F - 1) + R_P + t = tR_f + R_P = 70$ constraints, yielding a joint 140-by-144 affine system rather than an overconstrained one.

The resulting endpoint of each trace is an affine function of its compression input before M is solved. After choosing $\vec{y}'$, we therefore derive $\vec{x}'$ so that the retained outputs agree. Any solution to the joint system then realizes both complete traces and hence the collision in Equation (17). Appendix C gives the round-by-round scaled-state assignments, the derivation of $\vec{x}'$, and the resulting joint system.

For the demonstrated instance, this system has rank 140 and hence a four-dimensional affine solution space. We select a solution M from this space. It is MDS and passes the applicable Poseidon matrix-security checks for invariant subspaces and long subspace trails. These are verified separately; the constraint count does not imply that every pair selection or solution produces an admissible matrix.

7.3 The Forged Membership Proof

We evaluate the construction on a depth-three tree of eight four-element leaves, placing $(\vec{x}, \vec{y})$ at indices 4 and 5. We first construct and verify an authentication path for $\vec{y}$ at index 5. We then claim $\vec{y}'$ at the same index and replace only its first sibling, $\vec{x}$, with $\vec{x}'$. Plonky2's native verifier accepts this path with the same two upper siblings, path depth, root, and index. Neither replacement occurs among the tree's leaves.

Plonky2's optimized Poseidon implementation hard-wires its structured matrix rather than accepting an arbitrary dense matrix at runtime. We therefore supply a dense Poseidon evaluator over $\mathbb{F}_p$ through the native tree's generic hash interface. The tree construction, the authentication-path construction, and the verification logic remain unchanged. We also instantiate this evaluator with Plonky2's shipped matrix and cross-check it against the default Poseidon implementation.

7.4 Impact and Scope

The planted collision violates the binding property of the Merkle commitment at the targeted index. By presenting the replacement leaf $\vec{y}'$ with the replacement first sibling $\vec{x}'$, an attacker can produce a valid opening under the root computed

from the honest tree containing $(\vec{x}, \vec{y})$, although neither replacement leaf occurred in that tree. For example, consider a zero-knowledge application that commits account records in an indexed Poseidon Merkle tree. A malicious parameter designer could preselect a replacement leaf $\vec{y}'$ containing a much larger balance than the honest record at the targeted index. The planted collision would make that absent record verify under the committed root, allowing downstream state-transition logic to treat nonexistent funds as genuine and propagate their effects into the successor commitment. This is a serious state-integrity failure even though the attack does not provide arbitrary openings elsewhere in the tree.

The matrix M is constructed for a specific honest adjacent pair $(\vec{x}, \vec{y})$ and a selected replacement leaf $\vec{y}'$, with $\vec{x}'$ derived as part of the construction. These values are therefore fixed when the matrix is generated, and the attack succeeds only if the honest pair later appears at the targeted positions in the tree. This condition can hold when the relevant leaf values and their placement are predictable or attacker-controlled; the construction does not permit an arbitrary replacement to be chosen after seeing an independently generated tree.

8 Conclusion

In this paper, we showed that an attacker who controls the matrix of ZK-friendly hash functions can embed attacker-chosen input–output relations while producing an MDS matrix that passes the additional matrix-security checks required by the target design. The generic construction pairs matrix multiplications to reduce the number of distinct constraints on the matrix. We extend this construction to Poseidon by exploiting its partial rounds and to Poseidon2 by modifying only its external matrix. These results show that the MDS property and the matrix-security checks do not by themselves rule out malicious matrix selection.

We demonstrate the consequences in three case studies. A malicious external matrix in the Poseidon2 challenger allows a Plonky3 verifier to accept a false proof. A collision embedded in the Tip5 matrix gives an unchecked virtual machine program the same digest as Neptune Cash’s native-currency program, allowing counterfeit currency to be created. A collision embedded in the Poseidon matrix used by Plonky2 allows a forged Merkle-membership claim. These are attacks that assume malicious matrix selection under our threat model. They are not attacks on the matrices proposed in the original papers or evidence that a deployed matrix contains a backdoor.

Our constructions search over general MDS matrices and do not require the result to follow one of the matrix-construction methods in Table 1. Future work should determine whether attacker-chosen relations can be embedded while requiring the resulting matrix to be Cauchy, Vandermonde, circulant, or to satisfy another construction used by an existing specification. This would extend our attacks to a stricter threat model in which the original matrix-construction

method is also enforced.

References

- [1] Ange Albertini, Jean-Philippe Aumasson, Maria Eichlseder, Florian Mendel, and Martin Schl  ffer. Malicious hashing: Eve’s variant of SHA-1. In *Selected Areas in Cryptography*, volume 8781 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2014.
- [2] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016*, volume 10031, pages 191–219. Springer, 2016.
- [3] Abdelrahman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols. *ToSC 2020*, 2020(3), Sep. 2020.
- [4] Tomer Ashur and Siemen Dhooghe. MARVELlous: a STARK-Friendly Family of Cryptographic Primitives. IACR Cryptology ePrint Archive, Report 2018/1098, 2018. <https://eprint.iacr.org/2018/1098>.
- [5] Tomer Ashur, Al Kindi, Willi Meier, Alan Szepieniec, and Bobbin Threadbare. Rescue-prime optimized. Cryptology ePrint Archive, Report 2022/1577, 2022. <https://eprint.iacr.org/2022/1577>.
- [6] Augustin Bariant, Cl  mence Bouvier, Ga  tan Leurent, and L  o Perrin. Algebraic attacks against some arithmetization-oriented primitives. *ToSC 2022*, pages 73–101, 2022.
- [7] Christof Beierle, Tim Beyne, Patrick Felke, and Gregor Leander. Constructing and deconstructing intentional weaknesses in symmetric ciphers. In *Annual International Cryptology Conference*, pages 748–778. Springer, 2022.
- [8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.
- [9] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT 2019*, pages 103–128. Springer, 2019.
- [10] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. Dual ec: A standardized back door. In *The new codebreakers: essays dedicated to David Kahn on the occasion of his 85th birthday*, pages 256–281. Springer, 2016.

- [11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007, 2007.
- [12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *EUROCRYPT 2008*, pages 181–197. Springer, 2008.
- [13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2012.
- [14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *EUROCRYPT 2013*, pages 313–314. Springer, 2013.
- [15] Clémence Bouvier, Pierre Briaud, Pyrros Chaidos, Léo Perrin, Robin Salen, Vesselin Velichkov, and Danny Willems. New Design Techniques for Efficient Arithmetization-Oriented Hash Functions: Anemoi Permutations and Jive Compression Mode. In *CRYPTO 2023*, pages 507–539. Springer, 2023.
- [16] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive*, 2019.
- [17] Stephen Checkoway, Jacob Maskiewicz, Christina Gorman, Joshua Fried, Shaanan Cohny, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual ec incident. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 468–479, 2016.
- [18] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 769–793. Springer, 2020.
- [19] Alessandro Chiesa and Michele Orrù. A Fiat–Shamir transformation from duplex sponges. In *Theory of Cryptography – TCC 2025*, volume 16268 of *Lecture Notes in Computer Science*, pages 452–474. Springer, 2026. <https://eprint.iacr.org/2025/536>.
- [20] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.
- [21] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO’ 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.
- [22] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.
- [23] Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schafnegg, Roman Walch, and Qingju Wang. Horst Meets Fluid-SPN: Griffin for Zero-Knowledge Applications. In *Advances in Cryptology – CRYPTO 2023*, volume 14083 of *Lecture Notes in Computer Science*, pages 573–606. Springer, 2023. <https://eprint.iacr.org/2022/403>.
- [24] Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schafnegg, and Roman Walch. Reinforced concrete: a fast hash function for verifiable computation. In *CCS 2022*, pages 1323–1335, 2022.
- [25] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schafnegg. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX 2021*, pages 519–535. USENIX Association, August 2021.
- [26] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schafnegg. Poseidon2: A faster version of the poseidon hash function. In *International Conference on Cryptology in Africa*, pages 177–203. Springer, 2023.
- [27] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schafnegg. On a generalization of substitution-permutation networks: The HADES design strategy. In *EUROCRYPT 2020*, pages 674–704. Springer, 2020.
- [28] Lorenzo Grassi, Christian Rechberger, and Markus Schafnegg. Proving resistance against infinitely long subspace trails: How to choose the linear layer. *Cryptology ePrint Archive*, 2020.
- [29] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016*, pages 305–326. Springer, 2016.
- [30] Jincheol Ha, Seongha Hwang, Jooyoung Lee, Seungmin Park, and Mincheol Son. Polocolo: A zk-friendly hash function based on s-boxes using power residues. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 303–332. Springer, 2025.
- [31] Sunghyeon Jo. Midpoint reset: A full-round poseidon collision from an adaptively chosen MDS matrix. *Cryptology ePrint Archive*, Paper 2026/1760, 2026.

- [32] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*, pages 359–388. Springer, 2022.
- [33] Simon-Philipp Merz and Àlex Rodríguez García. Skipping class: Algebraic attacks exploiting weak matrices and operation modes of poseidon2(b). *Cryptology ePrint Archive*, 2026.
- [34] National Institute of Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-4, August 2015. <https://doi.org/10.6028/NIST.FIPS.180-4>.
- [35] Neptune Cash Developers. Neptune Core. <https://github.com/Neptune-Crypto/neptune-core>, 2026. Version 0.15.0, commit 030a1bc. Accessed 20-August-2026.
- [36] Thomas Peyrin and Haoyang Wang. The malicious framework: Embedding backdoors into tweakable block ciphers. In *Annual international cryptology conference*, pages 249–278. Springer, 2020.
- [37] Polygon Labs. Plonky2. <https://github.com/0xPolygonZero/plonky2>, 2025.
- [38] Polygon Labs. Plonky3. <https://github.com/Plonky3/Plonky3>, 2026.
- [39] Vincent Rijmen and Bart Preneel. A family of trapdoor ciphers. In *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 1997.
- [40] Arnab Roy, Matthias Johann Steiner, and Stefano Trevisani. Arion: Arithmetization-oriented permutation and hashing from generalized triangular dynamical systems. arXiv preprint arXiv:2303.04639, 2023. <https://arxiv.org/abs/2303.04639>.
- [41] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. Rescue-prime: A standard specification (sok). *Cryptology ePrint Archive*, Report 2020/1143, 2020. <https://eprint.iacr.org/2020/1143>.
- [42] Alan Szepieniec, Alexander Lemmens, Jan Ferdinand Sauer, Bobbin Threadbare, and Al-Kindi. The tip5 hash function for recursive starks. *Cryptology ePrint Archive*, Report 2023/107, 2023. <https://eprint.iacr.org/2023/107>.
- [43] Giuseppe Vitto. Slipway: Accessing finite subspace trails in poseidon. *Cryptology ePrint Archive*, Paper 2026/1579, 2026.
- [44] Adam Young and Moti Yung. The dark side of “black-box” cryptography or: Should we trust capstone? In *Advances in Cryptology – CRYPTO*, pages 89–103, 1996.
- [45] Adam Young and Moti Yung. A subliminal channel in secret block ciphers. In *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 198–211. Springer, 2005.
- [46] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. Baloo: Nearly optimal lookup arguments. 2022. <https://eprint.iacr.org/2022/1565>.
- [47] Yuncong Zhang, Shi-Feng Sun, and Dawu Gu. Efficient KZG-Based Univariate Sum-Check and Lookup Argument. In *PKC 2024*, pages 400–425. Springer, 2024.

A Matrix Construction Algorithms

Algorithm 1 A Generic matrix construction with pairing

Input: Permutation input $\vec{a}$ and output $\vec{b}$

Output: A matrix M under which the SPN permutation maps $\vec{a}$ to $\vec{b}$, or $\perp$

```

1:  $\vec{u}_1 \leftarrow S(\vec{a} + \vec{k}_1)$ ,  $\vec{v}_N \leftarrow \vec{b}$ 
2:  $L \leftarrow \max\{N - t, 0\}$ 
3: for  $i = 1, \dots, L$  do
4:    $j \leftarrow N + 1 - i$ 
5:    $\vec{v}_i \leftarrow \vec{v}_j$ 
6:    $\vec{u}_j \leftarrow \vec{u}_i$ 
7:    $\vec{u}_{i+1} \leftarrow S(\vec{v}_i + \vec{k}_{i+1})$ 
8:    $\vec{v}_{j-1} \leftarrow S^{-1}(\vec{u}_j) - \vec{k}_j$ 
9: end for
10: for  $i = L + 1, \dots, N - L - 1$  do
11:   Choose  $\vec{v}_i$  uniformly from  $\mathbb{F}_p^t$ 
12:    $\vec{u}_{i+1} \leftarrow S(\vec{v}_i + \vec{k}_{i+1})$ 
13: end for
14: if  $N < t$  then
15:   Choose  $\vec{u}_{N+1}, \dots, \vec{u}_t$  uniformly from  $\mathbb{F}_p^t$ 
16:   Choose  $\vec{v}_{N+1}, \dots, \vec{v}_t$  uniformly from  $\mathbb{F}_p^t$ 
17: end if
18:  $U \leftarrow (\vec{u}_1 \mid \dots \mid \vec{u}_t)$ 
19:  $V \leftarrow (\vec{v}_1 \mid \dots \mid \vec{v}_t)$ 
20: if  $\text{rank}(U) < t$  or  $\text{rank}(V) < t$  then
21:   return  $\perp$ 
22: end if
23:  $M \leftarrow VU^{-1}$ 
24: return  $M$ 

```

Algorithm 2 Matrix construction for Poseidon

Input: Permutation input $\vec{a}$ and output $\vec{b}$ **Output:** A matrix M under which Poseidon maps $\vec{a}$ to $\vec{b}$, or $\perp$

```

1:  $R_f \leftarrow R_F/2$  and  $R \leftarrow R_F + R_P$ 
2:  $\vec{u}_1 \leftarrow S(\vec{a} + \vec{k}_1)$  and  $\vec{v}_R \leftarrow \vec{b}$ 
3: for  $i = 1, \dots, R_f - 1$  do
4:    $j \leftarrow R + 1 - i$ 
5:    $\vec{v}_i \leftarrow \vec{v}_j$  and  $\vec{u}_j \leftarrow \vec{u}_i$ 
6:    $\vec{u}_{i+1} \leftarrow S(\vec{v}_i + \vec{k}_{i+1})$ 
7:    $\vec{v}_{j-1} \leftarrow S^{-1}(\vec{u}_j) - \vec{k}_j$ 
8: end for
9: Choose  $\vec{u}_{R_f+R_P+1}$  uniformly at random from  $\mathbb{F}_p^t$  and set
    $\vec{v}_{R_f+R_P} \leftarrow S^{-1}(\vec{u}_{R_f+R_P+1}) - \vec{k}_{R_f+R_P+1}$ 
10: Choose the entries  $M'_{j,k}$  with  $1 \leq j \leq t$  and  $2 \leq k \leq t$ 
    uniformly at random from  $\mathbb{F}_p$ 
11: Treat the free entries of  $T$  in Equation (8) and
     $M'_{1,1}, \dots, M'_{t,1}$  as the  $t^2$  variables of the linear system
12: Initialize a linear system  $\mathcal{E}$  with  $M'T\vec{u}_i = T\vec{v}_i$  for  $1 \leq i < R_f$ 
13:  $\vec{u}'_{R_f+1} \leftarrow M'T\vec{u}_{R_f} + T\vec{k}_{R_f+1}$ 
14: for  $i = R_f + 1, \dots, R_f + R_P - 1$  do
15:   Add  $u'_{i,1} = 0$  to  $\mathcal{E}$ 
16:    $\vec{u}'_{i+1} \leftarrow M'\vec{u}'_i + T\vec{k}_{i+1}$ 
17: end for
18: Add  $u'_{R_f+R_P,1} = 0$ ,  $M'\vec{u}'_{R_f+R_P} = T\vec{v}_{R_f+R_P}$ , and
     $M'T\vec{u}_{R_f+R_P+1} = T\vec{v}_{R_f+R_P+1}$  to  $\mathcal{E}$ 
19: Solve  $\mathcal{E}$  for the free entries of  $T$  and  $M'_{1,1}, \dots, M'_{t,1}$ 
20: if no solution has invertible  $T$  then
21:   return  $\perp$ 
22: end if
23: Set  $M \leftarrow T^{-1}M'T$ 
24: return  $M$ 

```

B Coverage of the Generic Bound

This appendix evaluates the condition $N \leq 2t - 1$ from Equation (5) for representative parameter sets of several ZK-friendly hash functions. For a design with a single matrix, N denotes the total number of times that matrix multiplies the state during one permutation evaluation. For Poseidon2, which has two matrices, N counts the times that either matrix multiplies the state. We use the 128-bit parameter sets given in the original specifications or their parameter comparisons [5, 23, 25, 26, 30, 40–42]. Rescue-Prime Optimized also defines a 160-bit instance at $t = 16$, which we include to cover both widths defined by its specification. For Poseidon, Rescue-Prime, Griffin, Poseidon2, and Arion, we use parameter families with the S-box $x \mapsto x^5$ over approximately 256-bit prime fields.

Table 3 gives N for each width. Generally, N is equal to the

Algorithm 3 External-matrix construction for Poseidon2

Input: Permutation input $\vec{a}$ and output $\vec{b}$ **Output:** An external matrix M_E under which Poseidon2 maps $\vec{a}$ to $\vec{b}$, or $\perp$

```

1:  $R_f \leftarrow R_F/2$ 
2: Let  $\Phi_{M_I}$  be the fixed transformation from  $\vec{v}_{R_f}$  to  $\vec{u}_{R_f+1}$ 
3:  $\vec{u}_0 \leftarrow \vec{a}$  and  $\vec{v}_{R_f} \leftarrow \vec{b}$ 
4: for  $i = 0, \dots, R_f - 1$  do
5:    $j \leftarrow R_f - i$ 
6:    $\vec{v}_i \leftarrow \vec{v}_j$  and  $\vec{u}_j \leftarrow \vec{u}_i$ 
7:    $\vec{u}_{i+1} \leftarrow S(\vec{v}_i + \vec{k}_{i+1})$ 
8:   if  $j = R_f + 1$  then
9:      $\vec{v}_{R_f} \leftarrow \Phi_{M_I}^{-1}(\vec{u}_{R_f+1})$ 
10:  else
11:     $\vec{v}_{j-1} \leftarrow S^{-1}(\vec{u}_j) - \vec{k}_{j+R_P}$ 
12:  end if
13: end for
14:  $U_0 \leftarrow (\vec{u}_0 \mid \dots \mid \vec{u}_{R_f})$  and  $V_0 \leftarrow (\vec{v}_0 \mid \dots \mid \vec{v}_{R_f})$ 
15: Extend  $U_0$  and  $V_0$  independently with uniformly sampled
    columns to obtain  $t \times t$  matrices  $U$  and  $V$ 
16: if  $\text{rank}(U) < t$  or  $\text{rank}(V) < t$  then
17:   return  $\perp$ 
18: end if
19:  $M_E \leftarrow VU^{-1}$ 
20: return  $M_E$ 

```

number of rounds because the matrix multiplies the state once per round. The exceptions are as follows. In Rescue-Prime and Rescue-Prime Optimized, the matrix multiplies the state twice per round. Rescue-Prime Optimized has seven rounds for both $t = 12$ and $t = 16$, giving $N = 14$. Griffin, Arion, and Polocolo include an initial multiplication in addition to one per round. In Poseidon2, M_E multiplies the state once at the beginning and after each of its R_F full-round S-box layers, while M_I multiplies the state in each of its R_P partial rounds. Counting both matrices therefore gives $N = R_F + R_P + 1$.

Table 4 reports the applicability of the generic attack for each width t . The generic attack applies to the listed Rescue-Prime Optimized instances, the Griffin widths $t \geq 8$, the listed Arion and Polocolo widths $t \geq 4$, and Tip5. It does not apply to the listed Poseidon, Rescue-Prime, or Poseidon2 instances.

Table 3: The total number of times that matrix multiplies the state. “-” is used to indicate that the hash function is not defined for the given value of t .

Hash function	Width t											
	2	3	4	5	6	7	8	12	16	20	24	
Poseidon	64	64	64	64	65	65	65	65	65	65	65	
Rescue-Prime	-	28	22	-	16	-	16	-	-	-	-	
Griffin	-	15	12	-	-	-	10	10	10	10	10	
Poseidon2	64	64	64	-	-	-	65	65	65	65	66	
Arion	-	7	6	6	6	-	5	-	-	-	-	
Polocolo	-	7	6	6	6	6	6	-	-	-	-	
Tip5	-	-	-	-	-	-	-	-	5	-	-	
Rescue-Prime Optimized	-	-	-	-	-	-	-	14	14	-	-	

Table 4: Applicability of the generic attack for each width t . “-” is used to indicate that the hash function is not defined for the given value of t .

Hash function	Width t											
	2	3	4	5	6	7	8	12	16	20	24	
Poseidon	X	X	X	X	X	X	X	X	X	X	X	
Rescue-Prime	-	X	X	-	X	-	X	-	-	-	-	
Griffin	-	X	X	-	-	-	✓	✓	✓	✓	✓	
Poseidon2	X	X	X	-	-	-	X	X	X	X	X	
Arion	-	X	✓	✓	✓	-	✓	-	-	-	-	
Polocolo	-	X	✓	✓	✓	✓	✓	-	-	-	-	
Tip5	-	-	-	-	-	-	-	-	✓	-	-	
Rescue-Prime Optimized	-	-	-	-	-	-	-	✓	✓	-	-	

C Detailed Plonky2 Two-Relation Construction

This appendix gives the round-by-round construction deferred from Section 7.2. Let $\vec{a}$ denote the starting state of one of the two permutation evaluations. We have

$$\begin{aligned}\vec{v}_i &= M\vec{u}_i & (1 \leq i \leq 30), \\ \vec{u}_1 &= S(\vec{a} + \vec{k}_1), \\ \vec{u}_i &= S(\vec{v}_{i-1} + \vec{k}_i) & (2 \leq i \leq 30).\end{aligned}$$

For each desired input–output relation, the attacker chooses four nonzero field elements $\rho_1, \rho_2, \rho_3, \rho_4$. The value ρ_i is the factor by which the entire input of a full-round S-box in the final full-round block is scaled. Since the full-round S-box raises every coordinate to the power α , its output is then scaled

by $C_i := \rho_i^\alpha$. We use these four factors to impose

$$\begin{aligned}\vec{u}_{30} &= C_1\vec{u}_1, & \vec{v}_{30} &= C_1\vec{v}_1, \\ \vec{u}_{29} &= C_2\vec{u}_2, & \vec{v}_{29} &= C_2\vec{v}_2, \\ \vec{u}_{28} &= C_3\vec{u}_3, & \vec{v}_{28} &= C_3\vec{v}_3, \\ \vec{u}_{27} &= C_4\vec{u}_1, & \vec{v}_{27} &= C_4\vec{v}_1.\end{aligned}\tag{18}$$

In particular, the last line is the proportionality relation used in Section 7.2. The four relations between the matrix inputs are obtained by imposing the following relations immediately before the corresponding S-boxes:

$$\begin{aligned}\vec{v}_{29} + \vec{k}_{30} &= \rho_1(\vec{a} + \vec{k}_1), \\ \vec{v}_{28} + \vec{k}_{29} &= \rho_2(\vec{v}_1 + \vec{k}_2), \\ \vec{v}_{27} + \vec{k}_{28} &= \rho_3(\vec{v}_2 + \vec{k}_3), \\ \vec{v}_{26} + \vec{k}_{27} &= \rho_4(\vec{a} + \vec{k}_1).\end{aligned}$$

Applying the S-box to these equalities gives the four relations between the $\vec{u}_i$ in Equation (18). The relations between the $\vec{v}_i$ in the same equation are imposed directly. Because matrix multiplication is linear, each final-round condition is then a scalar multiple of its early-round condition and adds no independent rows. When $\rho_i = 1$, this reduces to the equality used in Section 4.2.

These assignments have no circular dependency. Their main dependency chain is

$$\vec{a} \longrightarrow \vec{v}_{29} \longrightarrow \vec{v}_2 \longrightarrow \vec{v}_{27} \longrightarrow \vec{v}_1.$$

The chain alternates between solving an S-box-input relation and a scaled $\vec{v}$ -pair; no line uses a value defined later. For fixed $\vec{a}$, scales, and round constants, evaluate this chain in the displayed order:

$$\begin{aligned}\vec{v}_{29} &:= \rho_1(\vec{a} + \vec{k}_1) - \vec{k}_{30}, \\ \vec{v}_2 &:= C_2^{-1}\vec{v}_{29}, \\ \vec{v}_{27} &:= \rho_3(\vec{v}_2 + \vec{k}_3) - \vec{k}_{28}, \\ \vec{v}_1 &:= C_4^{-1}\vec{v}_{27}.\end{aligned}$$

Once $\vec{v}_1$ is known, determine the remaining values as

$$\begin{aligned}\vec{v}_{28} &:= \rho_2(\vec{v}_1 + \vec{k}_2) - \vec{k}_{29}, \\ \vec{v}_3 &:= C_3^{-1}\vec{v}_{28}, \\ \vec{v}_{30} &:= C_1\vec{v}_1, \\ \vec{v}_{26} &:= \rho_4(\vec{a} + \vec{k}_1) - \vec{k}_{27}.\end{aligned}$$

Here the first and last lines solve the remaining S-box-input relations, while the middle two solve $\vec{v}_{28} = C_3\vec{v}_3$ and $\vec{v}_{30} = C_1\vec{v}_1$. The first three lines of Equation (18) replace the three full-round equalities already used in Section 4.2. The last line is the additional relation that makes Equation (14) linearly dependent.

The same assignments also determine the desired permutation output before M is known. Substituting the main dependency chain into $\vec{v}_{30} = C_1 \vec{v}_1$ gives

$$\begin{aligned}\vec{v}_{30} &= \frac{C_1}{C_4} \left(\rho_3 (\vec{v}_2 + \vec{k}_3) - \vec{k}_{28} \right) \\ &= \frac{C_1}{C_4} \left[\frac{\rho_3}{C_2} \left(\rho_1 (\vec{a} + \vec{k}_1) - \vec{k}_{30} \right) + \rho_3 \vec{k}_3 - \vec{k}_{28} \right] \\ &= \gamma_p \vec{a} + \beta_p,\end{aligned}$$

where

$$\begin{aligned}\gamma_p &:= \frac{C_1 \rho_3 \rho_1}{C_2 C_4}, \\ \beta_p &:= \frac{C_1}{C_4} \left[\frac{\rho_3}{C_2} (\rho_1 \vec{k}_1 - \vec{k}_{30}) + \rho_3 \vec{k}_3 - \vec{k}_{28} \right].\end{aligned}$$

We denote this prescribed endpoint by $G_p(\vec{a}) := \vec{v}_{30}$. Thus, G_p is not an additional permutation: it is the final state forced by the intermediate-state assignments above.

The attacker now chooses any desired replacement $\vec{y}' \in \mathbb{F}_p^4$ with $\vec{y}' \neq \vec{y}$ and two scale tuples ρ_H, ρ_F with $\gamma_F \neq 0$, where $\gamma_H := \gamma_{\rho_H}$ and $\gamma_F := \gamma_{\rho_F}$, and likewise for β_H, β_F . It derives

$$\vec{x}' := \gamma_F^{-1} (\gamma_H \vec{x} + \beta_H[:4] - \beta_F[:4]). \quad (19)$$

Now fix the two complete relations as

$$\begin{aligned}\vec{a}_H &= \vec{x} \parallel \vec{y} \parallel 0 \parallel 0 \parallel 0 \parallel 0, & O_H &= G_{\rho_H}(\vec{a}_H), \\ \vec{a}_F &= \vec{x}' \parallel \vec{y}' \parallel 0 \parallel 0 \parallel 0 \parallel 0, & O_F &= G_{\rho_F}(\vec{a}_F).\end{aligned}$$

By Equation (19),

$$O_H[:4] = \gamma_H \vec{x} + \beta_H[:4] = \gamma_F \vec{x}' + \beta_F[:4] = O_F[:4].$$

Hence any matrix realizing these two complete relations gives the required compression collision.

The partial-round recurrence remains affine even when its active S-box input is a fixed nonzero value. For $5 \leq i \leq 26$, let a_i be that fixed input and let $\vec{u}'_i = T \vec{u}_i$ be the transformed matrix input. Then

$$\vec{u}'_i = T(\vec{v}_{i-1} + \vec{k}_i) + (a_i^\alpha - a_i) T e_1.$$

Its first element is the known constant a_i^α . Since only the first column of M' is variable and all other columns are fixed, $M' \vec{u}'_i$ is still linear in the 144 matrix variables. We use the zero schedule from Section 4.2 for the first relation and a fixed nonzero schedule for the second relation.

After the additional scaled pairing, each input–output relation contributes

$$3 \cdot 12 + 22 + 12 = 70$$

independent scalar constraints: three early full-round conditions, 22 active partial-round conditions, and the condition

connecting the partial block to round 27. The two relations share T , M' , and M , giving a joint system of

$$2 \cdot 70 = 140 \text{ equations in 144 variables.} \quad (20)$$

The system in our proof of concept has rank 140 and hence an affine solution space of dimension four. We select a solution with invertible T satisfying the required matrix checks and set

$$M = T^{-1} M' T.$$

This matrix realizes both fixed relations, so $H_M(\vec{x}, \vec{y}) = H_M(\vec{x}', \vec{y}')$ as required.

D Ethical Considerations

This work considers a hypothetical malicious parameter-selection scenario. It does not identify a backdoor in any deployed hash-function instantiation or project, nor does it exploit a real-world system. All experiments were performed locally using proof-of-concept configurations. We did not interact with production networks, user accounts, assets, or private data, and the study involved no human participants.

The groups most directly affected by this work are designers and implementers of ZK-friendly hash functions, blockchain projects and their users, and the broader cryptographic research community. Publishing the attack constructions may provide information useful to a malicious parameter designer. However, our attacks assume that such a designer can already select the matrix, and withholding the results would leave the risks of opaque parameter selection less understood. We limit potential harm by stating our threat model and scope explicitly and by making no claim that any surveyed deployment contains a backdoor. Our results show that satisfying the MDS condition and other security requirements is insufficient to establish that a matrix is trustworthy. Its generation process must also be transparent and verifiable. We believe that openly documenting this risk is necessary to improve the security of the blockchain community by encouraging reproducible parameter generation and independent verification.

E AI Use Disclosure

Implementations in Sections 5, 6, and 7 were developed with the assistance of ChatGPT and Claude, including implementation-level suggestions. The authors manually reviewed, modified, and tested AI-assisted code. The final implementation was validated by independently running the experimental pipeline and confirming that the reported experimental results were reproducible.

Parts of the manuscript were edited with the assistance of ChatGPT and Claude for language polishing, including phrasing, grammar, typo correction, and clarity. All AI-assisted

suggestions were reviewed and revised by the authors, who take full responsibility for the correctness and integrity of the manuscript.