# Data Matching in Unequal Worlds

## and Applications to Smart Contracts

Dmitry Khovratovich [1] Mikhail Vladimirov [2] Benedikt Wagner [1]

[1] Ethereum Foundation
{dmitry.khovratovich,benedikt.wagner}@ethereum.org
[2] ABDK Consulting
mikhail.vladimirov@gmail.com

**Abstract**

SNARKs enable compact proofs that an NP statement is true and that the prover knows a valid witness. They have become a key building block in modern smart contract applications, including rollups and privacy-focused cryptocurrencies. In the widely used Groth16 framework, however, long statements incur high costs. A common workaround is to pass the statement's hash to the SNARK and move the statement into the witness. The smart contract then hashes the statement first, and the circuit that is proven additionally checks consistency of the hash and the statement. Unfortunately, virtually any hash function is expensive to call either in a smart contract (in terms of gas) or in the proven circuit (in terms of prover time).

We demonstrate a novel solution to this dilemma, which we call *hybrid compression*. Our method allows us to use two different hash functions—one optimized for the proof circuit, and another optimized for on-chain verification—thereby combining the efficiency advantages of both. We prove the security of this approach in the standard model under reasonable assumptions about the two hash functions, and our benchmarks show that it achieves near-optimal performance in both gas usage and prover time. As an example, compressing an 8 KB statement with our approach results in a 10-second prover time and a smart contract spending 270K gas, whereas the existing approaches either need a much longer proof generation (290 seconds for SHA-256 hashing) or a much more expensive contract (5M gas for Poseidon hashing).

Along the way, we develop a two-party protocol of independent interest in communication complexity: an efficient deterministic method for checking input equality when the two parties do not share the same hash function.

# Contents

# 1 Introduction

**Smart Contracts and their Applications.** Smart contracts, first introduced in Ethereum, are programs that execute in a decentralized manner; that is, no single entity runs them. Instead, they are executed collectively by all nodes participating in the blockchain protocol. To ensure fair access to this decentralized computation, each operation triggered by a smart contract call incurs a certain cost, measured in *gas* in the case of Ethereum[1]. The caller of the contract pays this cost in Ether, the native cryptocurrency in Ethereum, where the total gas consumption is converted to Ether according to the gas price established by the network.

Smart contracts enable a wide range of applications, from financial instruments[2] to privacy-focused cryptocurrencies [PSS19]. The latter are often implemented as collections of notes containing encrypted metadata (such as value and owner). To spend a note, its owner marks it as deprecated and creates one or more new notes, providing a *succinct proof* of ownership and correctness for the entire operation. An even more advanced application of succinct proofs in smart contracts is a rollup: a protocol that operates a separate blockchain on top of Ethereum without executing its own transactions on-chain. Instead, it submits a batch of transactions along with a state update and a *succinct proof* of correctness, which is verified by the rollup's smart contract. In this way, the transactions of the rollup do not need to be executed individually, thereby saving substantial gas costs.

**Succinct Arguments.** In the aforementioned smart contract applications, *succinct non-interactive arguments*[3] *of knowledge* (SNARKs) [Gro10] play a crucial role. A SNARK is parameterized by a relation $\Gamma$ over public and private inputs, referred to as the *statement* and the *witness*, respectively. For rollups, the statement typically consists of the old and new state along with the transactions, while the witness comprises the complete execution trace of these transactions. Any prover possessing a statement–witness pair that satisfies the relation can generate a succinct proof, which can then be verified efficiently using only the statement. This eliminates the need for the verifier to access the witness or recompute the relation directly. In the smart contract use cases discussed above, the contract call includes a statement and a proof; the contract verifies the proof and proceeds according to the verification result.

Two SNARK constructions have become particularly popular in smart contract applications due to their small proof sizes: Groth16 [Gro16] and Plonk [GWC19]. Both schemes require only a few pairing computations on the verifier side, making them practical for on-chain verification. In Groth16, the verifier first computes a multi-scalar multiplication (MSM) on elliptic curve points derived from the statement and then applies a small number of pairing operations to the MSM result and proof elements.

However, the MSM over the entire statement makes the gas cost grow linearly with the statement length, and hence becomes a performance bottleneck in Groth16. For applications where the statement consists of dozens of field elements (e.g., private cryptocurrencies) or even hundreds (e.g., transaction metadata in rollups), the verification cost becomes significant.

**Compressing the Statement to Save Gas.** A common approach to mitigate this high gas cost, which we have seen employed in several applications[4] [Tea24, Tea25, Lab21], is to *compress* a long statement $S$ into a short hash $\alpha = H(S)$ using a hash function $H$, and modify the proof so that it first "unpacks" $H(S)$. Concretely, the proof is now made for a modified relation $\Gamma'$, where the statement is replaced by $\alpha = H(S)$ and the original statement $S$ is included in the witness. The relation $\Gamma'$ ensures that $\alpha = H(S)$ and that the original relation $\Gamma$ holds. Importantly, the user still provides $S$ to the smart contract, which computes $\alpha = H(S)$ before invoking the SNARK verifier. Note that it is crucial here (even for correctness) that the smart contract and the relation use the *same* hash function $H$.

**The Hash Function Dilemma.** While this approach minimizes the number of elliptic curve operations in the verifier, it introduces a dilemma. If $H$ is a traditional hash function such as SHA-256 [Nat02], proving the correct execution of $H$ results in a large number of contraints for Groth16, which requires a gigantic trusted setup and is computationally expensive for the prover. Conversely, if $H$ is a SNARK-friendly

[1] We focus on Ethereum as the most popular smart contract platform at the time of writing, but our discussion applies also to other platforms.

[2] See, e.g., `https://app.uniswap.org/`.

[3] We use the terms *argument* and *proof* interchangeably, although typically proofs refer to systems that are secure even against computationally unbounded adversaries.

[4] Interestingly, a similar technique has also been adopted in Plonk-based protocols [Lab25].

hash function like Poseidon [GKR+21], evaluating $H$ in the smart contract becomes costly in terms of gas consumption[5]. Thus, the choice of $H$ represents a dilemma between prover efficiency and gas cost.

## 1.1 Our Contribution: Hybrid Compression

In this work, we overcome the above dilemma by introducing a new technique, which we call *hybrid compression*, that enables the use of *different* hash functions in the contract and in the relation.

More concretely, our starting point is a contract of the aforementioned type, which (among other inputs) takes a statement and a witness, checks a relation, and then proceeds based on the outcome of this check. For any such contract, we construct a new contract that *securely emulates* the original one using a SNARK. Importantly:

- The statement for the relation verified by the SNARK is short;
- The contract and the relation can employ *independently chosen* hash functions.

Concretely, this means the contract could use the built-in hash function SHA-256, while the relation can use a SNARK-friendly hash function such as Poseidon. We show that our construction is secure in the standard model under mild assumptions on the two hash functions, and that these assumptions hold unconditionally in the random oracle model.

Technically, our approach starts by taking the perspective of communication complexity [Yao79, AB09]. Namely, we reformulate our problem as one in which two parties aim to verify whether they hold the same input (Section 3.1). We then provide a solution for the (unstudied) setting where both parties are deterministic and have access to *different* hash functions (Section 3.2). Then, in Section 4, we translate these ideas to the context of smart contracts. There, we also formally define what it means for a smart contract to *securely emulate* a relation-checking contract and prove (in the standard model) that our hybrid compression construction satisfies this definition, assuming the hash functions satisfy what we call *joint UHF-hardness* (Definition 4). This is a slight generalization of collision resistance that holds unconditionally in the random oracle model.

We benchmark our construction against existing approaches (Section 5) and present a compact summary of the results in Table 1. The benchmarks indicate that our approach resolves the hash-function dilemma, achieving performance close to the best of both worlds: proving time and setup size are comparable to using Poseidon for statement compression, while gas costs are on par with using SHA-256.

| Benchmark | Setup Size | Proving Time | Contract Size | Gas |
|---|---|---|---|---|
| **Uncompressed** | 580 KB | 1.01 s | 25 574 B | 1 883 907 |
| **SHA-256** | 9.5 GB | 295.37 s | 1 615 B | 193 984 |
| **Poseidon** | 75 MB | 14.36 s | 22 712 B | 5 041 709 |
| **Our Approach** | 37 MB | 10.35 s | 1 983 B | 268 960 |

Table 1: Comparison of not compressing the statement, compressing with SHA-256, compressing with (Solidity-optimized) Poseidon, and our approach. For a more detailed version and explanations, we refer to Section 5 and Table 4. The numbers are for a statement of size $k = 256$ field elements. For an explanation why we can beat Poseidon in terms of proving time, we refer to Section 5.

## 1.2 Related work

We discuss more related work here.

**Combiners.** Our approach to sum the outputs of two different hash functions is related to the concept of the *XOR combiner*: $H(X) := H_1(X) + H_2(X)$ for two hash functions $H_1, H_2$. Hoch and Shamir [HS08] proved that the resulting function $H$ is indifferentiable from a random oracle up to $2^{n/2}$ queries if $H_1, H_2$ are $n$-bit random oracles. This bound is tight for collisions [HS08] and almost tight for

[5]This is because SHA-256 is built-in as a precompile into the virtual machine running smart contracts, whereas other hash functions need to be implemented from scratch.

preimages [LW15, BDG+20]. These results, however, do not directly relate to our setting as our $H_1$ and $H_2$ may be called with *distinct* inputs in our case.

**Communication Complexity.** As already outlined, our results can be viewed in the framework of communication complexity [Yao79, AB09]. The vast literature on this topic is outside of the scope of this paper, and a curious reader may refer to [KN97, RY20] for an extensive background on the topic. Shortly, the (two-party) communication complexity of a function $f$ on two inputs $X$ and $Y$ is the minimal number of bits that are required to be communicated two honest parties (one receiving $X$ and one receiving $Y$) who are trying to compute the function. In extensions of this setting, the parties may also have access to shared randomness or to private randomness, with respective protocols designated as public-coin and private-coin ones, respectively. The complexity of a non-randomized, *deterministic* protocol is calculated in the worst case over all inputs, whereas one may also consider the *average case* complexity. If a protocol fails on some inputs, it has a certain *error rate*. Our construction essentially makes two parties to compute the *equality function*

$$f(X,Y) = \begin{cases} 1, & \text{if } X = Y \\ 0, & \text{if } X \neq Y \end{cases}$$

for which various bounds have been studied and proven over decades. Particularly, it is known that the communication complexity of a deterministic protocols for $n$-bit inputs is exactly[6] $n$ [AB09]. Enabling randomness gives a great power to the parties: in the public-coin setting, there exists a randomized protocol with a constant error rate and constant communication complexity. The simplest version of such a protocol prescribes the parties to sample a random function $h$ with a fixed-size output, apply it to their own inputs, and check the match of the outputs [RY20]. In the private-coin setting, the communication complexity of the equality function is logarithmic in $n$ [KN97]. Notably, in these settings one assumes that the randomness is chosen after the inputs are fixed.

A natural extension of this framework introduces an adversarial control over the inputs, given the public randomness. For instance, Cohen and Naor [CN22] consider protocols where bad inputs exist but are hard to find by a computationally bounded adversary, and prove a connection between the existence of collision-resistant hash functions and breaking existing communication complexity bounds.

Our results can be interpreted as follows. We construct a deterministic protocol (as there is no shared nor private randomness) for the equality function, when the parties have access to distinct hash functions (or in the standard model: distinct hash function keys sampled at random). As in [CN22], bad inputs exist but are hard to find by a bounded adversary. In order to achieve $\lambda$ bits of security for $n$-bit inputs, it suffices for our protocol to work over a $(\log n + 2\lambda)$-bit field[7]. The communication consists of a small constant number of field elements, and so the communication complexity of the protocol is $O(\log n + \lambda)$. It remains an open problem whether the logarithmic term can be improved.

**Independent Work.** After the initial publication of this report, we became aware of independent work by Zhao et al. [ZSCZ24], who also address the hash function dilemma in the context of their folding-based recursive argument system, MicroNova. They faced the problem of expensive verification of the proof by a smart contract. Their solution employs techniques that are similar in spirit to ours, though not identical. While their focus is on resolving this issue within for a particular argument system, our contribution with regards to the hash function dilemma is more general: we formally establish that our approach applies to arbitrary relation-checking contracts. Moreover, we study the problem through the lens of communication complexity, offering a perspective that may be of independent interest. We believe that the hash function dilemma is a fundamental issue, and our results highlight its importance as a *standalone* research question worthy of dedicated investigation, independent of a specific argument system.

Additionally, Zhao et al.'s work focuses on recursive arguments, which inherently treat *random oracles as circuits* to be proven. Their technique also relies on the Fiat-Shamir paradigm. In contrast, our approach isolates a concrete *standard-model* property that the hash functions must satisfy for our technique to apply, using the random oracle model only as heuristic evidence supporting the plausibility of this property.

[6] This means that $n$ is both an upper and a lower bound, while of course there can be silly protocols communicating more.
[7] Actually, it suffices if we have an $f$-bit field with $f \geq \log(n/f) + 2\lambda$, see Remark 5.

# 2 Preliminaries

We adopt conventional notation from cryptography, such as a security parameter, negligible functions and PPT ( probabilistic polynomial-time) algorithms. The security parameter is denoted by $\lambda$, and we assume algorithms are (PPT) unless stated otherwise. For a deterministic algorithm $\mathsf{A}$, we use $y := \mathsf{A}(x)$ to indicate the output $y$ obtained by executing $\mathsf{A}$ on input $x$. If $\mathsf{A}$ is randomized, we write $y \leftarrow \mathsf{A}(x)$ to indicate sampling $y$ by running $\mathsf{A}$ with fresh uniform randomness. To emphasize the randomness $\rho$ explicitly, we write $y := \mathsf{A}(x; \rho)$. The expression $y \in \mathsf{A}(x)$ refers to any valid output $y$ that $\mathsf{A}$ might produce on input $x$. We write $x \xleftarrow{\$} X$ to denote the uniform sampling of $x$ from a finite set $X$. For integers, we use $[r]$ to represent the set $\{1, \ldots, r\} \subseteq \mathbb{N}$. We use $\mathbb{F}$ to denote a finite field.

**Definition 1** (Non-Interactive Argument System)**.** Let $\Gamma \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation. A non-interactive argument system for $\Gamma$ is a tuple $\mathsf{AS} = (\mathsf{ArgProve}, \mathsf{ArgVer})$ of PPT algorithms with the following syntax:

- $\mathsf{ArgProve}(\mathsf{stmt}, \mathsf{witn}) \to \pi$
- $\mathsf{ArgVer}(\mathsf{stmt}, \pi) \to b \in \{0,1\}$ (deterministic)

We require the scheme to be correct in the following sense: For every pair $(\mathsf{stmt}, \mathsf{witn}) \in \Gamma$, we have

$$\Pr\left[\mathsf{ArgVer}(\mathsf{stmt}, \pi) = 1 \;\middle|\; \pi \leftarrow \mathsf{ArgProve}(\mathsf{stmt}, \mathsf{witn}) \;\right] = 1.$$

*Remark 1* (Setup). Usually, one defines non-interactive argument systems with respect to a *common reference string* or a *random oracle*. We do not make this explicit to not overload our notation and to not commit to one variant. Our proofs work for both cases, and the reader may think of the argument system as having implicit access to either.

**Definition 2** (Knowledge-Soundness)**.** Let $\Gamma \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation. Let $\mathsf{AS} = (\mathsf{ArgProve}, \mathsf{ArgVer})$ be a non-interactive argument system for $\Gamma$. We say that $\mathsf{AS}$ is knowledge-sound, if there is a PPT algorithm $\mathsf{Ext}$ (called the *extractor*) such that for any PPT algorithm $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{ll} & \mathsf{ArgVer}(\mathsf{stmt}, \pi) = 1 \\ \wedge & (\mathsf{stmt}, \mathsf{witn}) \notin \Gamma \end{array} \;\middle|\; \begin{array}{l} (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(1^\lambda), \\ \mathsf{witn} \leftarrow \mathsf{Ext}(\mathsf{stmt}, \pi) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

*Remark 2* (Straight-line Extraction). For our security proofs to work, we need to work with straight-line extraction, as we have defined it. That is, the extractor cannot rewind the adversary. To still enable the extractor to get the witness from the typically much shorter $\pi$, one can equip the extractor with a trapdoor for the common reference string, or work in the random oracle or algebraic group model. As we have explained above, we do not want to commit to one variant and therefore leave this implicit.

# 3 Data Matching in Unequal Worlds

In this section, we present our main idea through the lens of communication complexity [Yao79, AB09]. Specifically, we consider a setting in which two parties aim to determine whether their inputs are identical while exchanging only a small amount of information. Looking ahead to our smart contract application, one party will correspond to the smart contract, and the other to the circuit that is verified via a succinct argument.

## 3.1 Two Party Communication Model

We first introduce our model.

**Data Matching ...** Consider two *deterministic* parties (i.e., interactive algorithms) Alice and Bob, receiving an input $x_{\mathsf{Alice}}$ and $x_{\mathsf{Bob}}$, respectively. The goal of Alice and Bob is to determine – with minimal communication effort – whether $x_{\mathsf{Alice}} = x_{\mathsf{Bob}}$. To accomplish this, they interact in some prescribed

protocol and exchange messages. We want that this protocol achieves (1) *completeness*, i.e., if $x_{\mathsf{Alice}} = x_{\mathsf{Bob}}$, then both parties accept, and (2) *soundness*, i.e., if $x_{\mathsf{Alice}} \neq x_{\mathsf{Bob}}$, then both parties reject. Notably, we assume that Alice and Bob always follow the protocol honestly and trust each other.

**... in Equal Worlds.** There is a simple protocol to achieve this: Alice can send $x_{\mathsf{Alice}}$ to Bob and Bob checks if $x_{\mathsf{Alice}} = x_{\mathsf{Bob}}$. It then sends the decision bit $b \in \{0, 1\}$ to Alice, and both output $b$. This protocol is complete and sound, and it has communication complexity $n + 1$ bits, if the inputs are $n$ bits long. Without any change of the model, this is provably optimal [AB09].

**The Cryptographic Setting.** If Alice and Bob both have access to a hash function $\mathsf{H}$ that outputs $\lambda$ bits, then a better protocol is possible: Alice sends $h_{\mathsf{Alice}} = \mathsf{H}(x_{\mathsf{Alice}})$, Bob sends $h_{\mathsf{Bob}} = \mathsf{H}(x_{\mathsf{Bob}})$, and both check if the hashes are the same, i.e., if $h_{\mathsf{Alice}} = h_{\mathsf{Bob}}$. For $n$-bit inputs, the communication complexity is only $2\lambda$ bits, independent of $n$. Completeness holds as before, but soundness is more subtle: of course, there exist collisions for $\mathsf{H}$, and so we cannot hope that the parties reject for *every* pair of distinct inputs. However, such collisions should be hard to find, which means that we have to relax the soundness condition to be computational: for any efficient adversary $\mathcal{A}$ that has access to the hash function, the probability that it finds $x_{\mathsf{Alice}} \neq x_{\mathsf{Bob}}$ such that Alice or Bob accepts is negligible.

Looking at the protocol, we can now easily argue this computational soundness in two ways: (a) we model $\mathsf{H}$ as a random oracle [BR93] and assume that $\mathcal{A}$ has access to it when picking the inputs. In this case, the probability is over the coins of $\mathcal{A}$ and the random oracle; (b) we model $\mathsf{H}$ as a keyed hash function with key $K \xleftarrow{\$} \mathcal{K}$ and the probability is over the coins of $\mathcal{A}$ and the choice of the key. In this case, collision-resistance is sufficient to argue soundness [CN22].

**Data Matching in Unequal Worlds.** In our work, we consider a similar setting, but where Alice and Bob have access to different hash functions $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$. Again, we can either model these as two independent random oracles or as two keyed hash functions, where Alice has access to the key $K^{(1)}$ for $\mathsf{H}^{(1)}$ and Bob has access to the key $K^{(2)}$ for $\mathsf{H}^{(2)}$. In both cases, we assume that the adversary has access to *both* hash functions (or keys) when picking the inputs.

## 3.2 Our Protocol

We now explain our protocol. The main challenge lies in the fact that Alice and Bob do not share a secure collision-resistant hash function.

**Adding a Third Hash Function.** Our first idea is to let them use a third hash function on the fly during the protocol. For that, we first let them agree on a key for this third hash function. To better distinguish this from the keys $K^{(1)}$ and $K^{(2)}$, we call it a *seed* from now on. Then, they can use this third hash function and exchange succinct hashes as before. But note that we required the parties to be *deterministic*, so there is no way for them to randomly sample a seed. In our solution, the parties will derive a seed using their hash functions, see below.

**A Good Third Hash Function.** Before addressing how this should be done, let us first consider which third hash function can be used. Our central observation is that when Alice and Bob begin their interaction, their inputs are already fixed, and the seed is sampled only afterward. Consequently, collision resistance is unnecessary; instead, any universal hash function suffices. For concreteness, say that the seed is a field element $\sigma \in \mathbb{F}$, the inputs are vectors in $\mathbb{F}^k$, and we use the universal hash function

$$x \longmapsto \sum_{i=1}^{k} \sigma^{i-1} x_i.$$

Assuming two inputs are fixed and $\sigma$ is sampled at random, the probability that the hashes collide is at most[8] $(k-1)/|\mathbb{F}|$, which is negligible for large enough fields.

**How to Sample the Seed.** What remains is to explain how Alice and Bob can agree on a seed $\sigma$. Indeed, we must make sure that the adversary cannot predict $\sigma$ before picking the inputs $x_{\mathsf{Alice}}, x_{\mathsf{Bob}}$, as otherwise, the universal hash function makes no guarantees. Naively, one could set $\sigma := \mathsf{H}(x_{\mathsf{Alice}}, x_{\mathsf{Bob}})$ for a random

[8] Some authors would call such a hash function $(k-1)$-universal instead of universal, but we do not make this distinction explicitly.

oracle $\mathsf{H}$, but Alice and Bob do not have access to such a shared random oracle, and none of them knows both inputs. We observe that one could instead define

$$\sigma = \underbrace{\mathsf{H}^{(1)}(x_{\mathsf{Alice}})}_{=:\alpha} + \underbrace{\mathsf{H}^{(2)}(x_{\mathsf{Bob}})}_{=:\beta},$$

where Alice can compute $\alpha$ and Bob can compute $\beta$. In some sense, we have compressed the input with both hash functions, and with a third shared hash function sampled on the fly, hence the name *hybrid compression*.

**Our Protocol.** To simplify notation, we now specify the universal hash function that we will use throughout the rest of the paper.

**Definition 3** (Universal Hash Function)**.** We define the following function $\mathsf{UHF}\colon \mathbb{F} \times \mathbb{F}^k \to \mathbb{F}$ taking a seed $\sigma \in \mathbb{F}$ and an input $x = (x_1, \ldots, x_k) \in \mathbb{F}^k$ as

$$\mathsf{UHF}(\sigma, x) := \sum_{j=1}^{k} \sigma^{j-1} x_j \in \mathbb{F}.$$

We now present our protocol, assuming Alice has access to $\mathsf{H}^{(1)}$ with key $K_1$ and Bob has access to $\mathsf{H}^{(2)}$ with key $K_2$. We assume that both functions map into $\mathbb{F}$. Alice gets as input $x_{\mathsf{Alice}} \in \mathbb{F}^k$ and Bob gets as input $x_{\mathsf{Bob}} \in \mathbb{F}^k$. Then:

1. Alice sends $\alpha := \mathsf{H}^{(1)}(K_1, x_{\mathsf{Alice}})$ to Bob; Bob sends $\beta := \mathsf{H}^{(2)}(K_2, x_{\mathsf{Bob}})$ to Alice.
2. Alice sends $\gamma_{\mathsf{Alice}} := \mathsf{UHF}(\alpha + \beta, x_{\mathsf{Alice}})$ to Bob; Bob sends $\gamma_{\mathsf{Bob}} := \mathsf{UHF}(\alpha + \beta, x_{\mathsf{Bob}})$ to Alice;
3. Both parties accept if and only if $\gamma_{\mathsf{Alice}} = \gamma_{\mathsf{Bob}}$.

Completeness of the protocol is clear. The communication complexity is independent of the input length $n = k \log |\mathbb{F}|$, namely, each party sends 2 field elements.

*Remark 3* (Alternative Structure). We can restructure our protocol to make it more suited for our smart contract application in Section 4. Namely, if only Bob has to output the decision, then he can first send $\beta$, and then Alice can send $(\alpha, \gamma_{\mathsf{Alice}})$. In this way, the number of messages is reduced from four to two.

**Soundness and Joint UHF Hardness.** One can analyze the soundness of the protocol in the random oracle model, using the intuition that we have provided above. However, as we will later use elements of this protocol in the context of succinct arguments, it is cleaner to define a standard model property of the two hash functions which implies soundness, and then heuristically show that this property is plausible in the random oracle model. Considering our specific protocol, an adversary that aims to break soundness wants to find two distinct inputs $x$ (for Alice) and $\tilde{x}$ for Bob such that the universal hash function outputs collide, with the seed derived as in the protocol. We phrase this as an abstract property, which we call *joint UHF hardness*.

**Definition 4** (Joint UHF Hardness)**.** Consider a finite field $\mathbb{F}$ and two keyed hash functions

$$\mathsf{H}^{(1)}\colon \mathcal{K}^{(1)} \times \mathbb{F}^k \to \mathbb{F}, \qquad \mathsf{H}^{(2)}\colon \mathcal{K}^{(2)} \times \mathbb{F}^k \to \mathbb{F}.$$

Then, we say that $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$ are jointly UHF-hard if for any PPT algorithm $\mathcal{A}$, we have

$$\Pr\left[x \neq \tilde{x} \wedge \mathsf{UHF}(\alpha + \beta, x) = \mathsf{UHF}(\alpha + \beta, \tilde{x})\right] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the following experiment:

1. Sample $K_1 \xleftarrow{\$} \mathcal{K}^{(1)}$ and $K_2 \xleftarrow{\$} \mathcal{K}^{(2)}$.
2. Run $(x, \tilde{x}) \leftarrow \mathcal{A}(K_1, K_2)$.
3. Set $\alpha := \mathsf{H}^{(1)}(K_1, x)$ and $\beta := \mathsf{H}^{(2)}(K_2, \tilde{x})$.

*Remark 4* (Keyed Hash Functions). As usual in the standard model, we have to consider keyed hash functions, as otherwise an adversary can have a solution to the problem hardcoded.

*Remark 5* (Random Oracles). If $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$ are both modeled as random oracles, then they are jointly UHF hard (with $\lambda$ bits of security), assuming $|\mathbb{F}| \geq k \cdot 2^{2\lambda}$. We show this in Appendix A.

*Remark 6* (The Same Hash Function). There is nothing that prevents $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$ from being the same, but with high probability, the keys will be different. Also, in our application to smart contracts, the interesting case is when $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$ are different.

By the above discussion, we get the following informal statement. We leave a formal application of joint UHF hardness to Section 4.

**Lemma 1** (Informal)**.** *If $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$ are jointly UHF-hard, then our data matching protocol is complete and computationally sound.*

# 4 Application to Smart Contracts

In this section, we present an application of our hybrid compression technique to smart contracts. We start with an informal overview, then give a formal model of what we want to achieve, and then present and analyze our construction.

## 4.1 Overview

We apply our technique to a specific class of smart contracts, improving their efficiency, and thereby reducing the amount of gas that they consume.

**Relation-Checking Contracts.** The class of contracts that we consider is what we call *relation-checking contracts*. Intuitively, when these contracts are called, the user provides as a statement $\mathsf{stmt}$ and a witness $\mathsf{witn}$ of some relation $\Gamma$, among other inputs. The contract then proceeds in two steps:

1. Check if $(\mathsf{stmt}, \mathsf{witn}) \in \Gamma$.
2. Perform some arbitrary computation based on the result of this check. The computation may depend on $\mathsf{stmt}$ and other inputs, but not on $\mathsf{witn}$.

Our goal is to transform such a contract into an *equivalent*[9] contract that is more efficient. We will first explain two transformation steps that are somewhat folklore, before explaining our final transformation.

**Compressing the Witness.** The first step is to *compress* the witness $\mathsf{witn}$. This can easily be achieved using a *succinct non-interactive argument of knowledge* (SNARK) for relation $\Gamma$. Namely, the user would now compute a (succinct!) argument of knowledge $\pi$ of the witness $\mathsf{witn}$, and supply $\pi$ instead of $\mathsf{witn}$ to the smart contract. The contract now checks that $\pi$ is valid instead of checking $(\mathsf{stmt}, \mathsf{witn}) \in \Gamma$. Intuitively, if the SNARK is secure, this check should be equivalent.

**Compressing the Statement.** Using a SNARK, we have now compressed the witness, but the statement $\mathsf{stmt}$ may still be large. In particular, the cost of verifying the SNARK in the contract may still be quite high, depending on the size of $\mathsf{stmt}$. To compress the statement, we will use a collision-resistant hash function and make a change to our relation $\Gamma$. Namely, the user still provides $\mathsf{stmt}$ and an argument $\pi$ (for a different relation, see below) to the contract[10] The contract, however, would now first hash the statement to get a short hash $\alpha = \mathsf{H}(\mathsf{stmt})$. Then, $\alpha$ takes the role of the statement and $\mathsf{stmt}$ is moved into the witness. More concretely, the SNARK now has to be for the relation $\Gamma'$ of pairs $(\mathsf{stmt}' = \alpha, \mathsf{witn}' = (\mathsf{witn}, \mathsf{stmt}))$ such that $\alpha = \mathsf{H}(\mathsf{stmt})$ and $(\mathsf{stmt}, \mathsf{witn}) \in \Gamma$. We will discuss below why this transformation is secure.

**The Problem: Choice of the Hash Function.** The transformation sketched so far is somewhat folklore and can be applied in practice. However, there is a fundamental issue with it: it is unclear what would be the best choice of the hash function $\mathsf{H}$:

[9] We will make formal what we mean by *equivalent* later.

[10] Indeed, as the computation of the contract can depend arbitrarily on $\mathsf{stmt}$, there is no way we can eliminate $\mathsf{stmt}$ entirely from the input to the contract.

- If we choose $\mathsf{H}$ to be a conventional hash like SHA-256, then executing $\mathsf{H}$ in the contract is cheap, but executing $\mathsf{H}$ inside the circuit computing relation $\Gamma'$ is expensive.
- On the other hand, if $\mathsf{H}$ is a modern algebraic hash function like Poseidon2, then $\mathsf{H}$ is cheap inside the circuit for $\Gamma'$, but expensive in the smart contract.

Our goal is to find a solution to this problem, avoiding any expensive use of a hash function.

**Our Solution.** To understand our solution and the relation to data matching in unequal worlds, it is instructive to think about why the transformation above with relation $\Gamma'$ is secure. To this end, recall that the SNARK is an argument of knowledge, i.e., in a security proof we can *extract* a witness $\mathsf{witn}' = (\mathsf{witn}, \tilde{\mathsf{stmt}})$ from $\pi$ with $\alpha = \mathsf{H}(\tilde{\mathsf{stmt}})$ and $(\tilde{\mathsf{stmt}}, \mathsf{witn}) \in \Gamma$. We want to argue that if the proof verifies, then $(\mathsf{stmt}, \mathsf{witn}) \in \Gamma$, where $\mathsf{stmt}$ is the statement that is input to the smart contract and then hashed to get $\alpha$. Our goal is to show that $\mathsf{stmt} = \tilde{\mathsf{stmt}}$ must hold, and for that we can simply use collision-resistance of $\mathsf{H}$. Looking at this from a different perspective, we can think of the smart contract – holding $\mathsf{stmt}$ – and the relation $\Gamma'$ – holding $\tilde{\mathsf{stmt}}$ as two honest players (Alice and Bob, respectively, with names as in Section 3) that want to check that their strings match. The user of the contract is the adversary that provides the two statements to these players. As they both have access to $\mathsf{H}$, the contract (i.e., Alice) can simply "send" the hash $\alpha$ to the relation (i.e., Bob) by using it as a statement for $\pi$ and the relation checks equality using the check $\alpha = \mathsf{H}(\tilde{\mathsf{stmt}})$.

Looking at it from this perspective, we can actually use data matching in unequal worlds to ensure that both players can use different hash functions. This is the main idea of our transformation.

**The Final Catch.** Unfortunately, it is not entirely clear how to apply our protocol from Section 3 for data matching in unequal worlds. This is because – in contrast to the protocol for data matching in equal worlds with one hash function – the protocol is a *two-message* protocol (cf. Remark 3): first, Bob sends $\beta$ and then Alice sends $(\alpha, \gamma_{\mathsf{Alice}})$. But in our application Bob is the relation and Alice is the smart contract, and there is no way that the relation can "send" anything to the contract (except the final decision bit). Thus, it appears that we are limited to using protocols in which only Alice sends exactly one message to Bob, which can be done via the statement for $\pi$. Luckily, we can solve this final issue by making the (untrusted!) user provide $\beta$ to the smart contract (i.e., Alice), and making the relation (i.e., Bob) check that $\beta$ has been computed correctly. This yields our final transformation, which we present and analyze formally below. We give an overview in Figure 1.

## 4.2 Definitions

To formally analyze our construction, we begin by providing a definition of smart contracts, followed by a corresponding security definition. We first define a smart contract as a function that, given an input, produces an output. To capture stateful behavior, we extend this model by allowing the function to take the current state as part of its input and return an updated state. By default, we denote the initial state as $St = \bot$, representing an empty or uninitialized state. For the purposes of our security definition, it is useful to partition the input into two components: a primary input and an auxiliary input.

**Definition 5** (Contract Function)**.** A contract function is a function of the form

$$\mathsf{Con}\colon\ (St, (\mathsf{In}, \mathsf{AuxIn})) \longmapsto (St', \mathsf{Out})\,,$$

which maps a contract state $St$, and an input $(\mathsf{In}, \mathsf{AuxIn})$ to an updated contract state $St'$ and an output $\mathsf{Out}$. The input $(\mathsf{In}, \mathsf{AuxIn})$ is composed of a main input $\mathsf{In}$ and an auxiliary input $\mathsf{AuxIn}$.

*Remark 7* (Parameters). A contract function may also take as input some public parameters, e.g., the common reference string of a non-interactive argument system or keys for a keyed hash function. We omit these as explicit input to not overload notation. One should however, specify how these parameters are generated when specifying a contract function, and experiments involving such contract functions generate the parameters during initialization, unless specified otherwise.

Our construction transforms a given smart contract into a more efficient one that preserves the same functionality, or *emulates* the original contract. To formalize this notion, we introduce definitions of *correctness* and *soundness*.

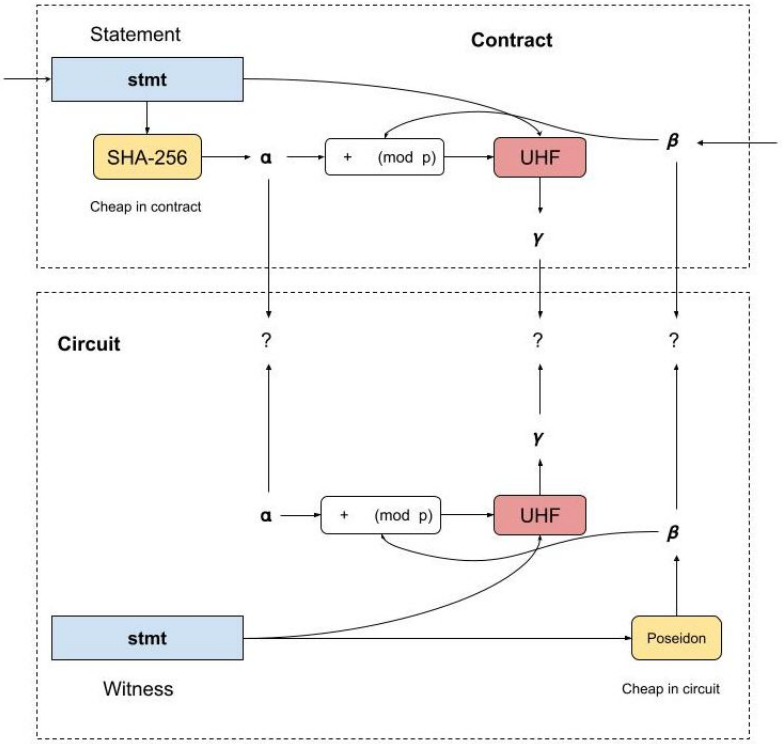


Figure 1: Overview of our construction instantiated with concrete hash functions SHA-256 and Poseidon.

Intuitively, correctness ensures that the original and the modified contracts exhibit the same input-output behavior for honest users: for any sequence of inputs, both contracts produce the same sequence of outputs[11]. In our construction, however, the modified contract may require additional auxiliary inputs that are not needed by the original contract. Specifically, we require the existence of an efficient algorithm that, given the original inputs, computes the corresponding auxiliary inputs for the modified contract. This is precisely the motivation for distinguishing between the main input and the auxiliary input.

Turning to soundness, we wish to ensure that no dishonest user can cause the two contracts to disagree on the same main input. A naive approach would allow an adversary to choose an input, two states, and two auxiliary inputs (one for each contract), and then claim a soundness violation if the outputs differ. However, this definition is too strong to be achievable in general. If the original contract relies on the auxiliary input in any meaningful way, an adversary could simply supply valid inputs to the original contract and garbage auxiliary inputs to the modified contract, thereby causing disagreement. Instead, we require the following notion: for any output that an adversary can cause the modified contract to produce, there must exist an execution (i.e., set auxiliary inputs) of the original contract on the same main input that yields the same output, and the adversary must have known it. In other words, any behavior observable in the modified contract must be reproducible by the original contract on honest inputs. We capture this formally via an *extraction-based* definition.

**Definition 6** (Secure Emulation)**.** Let $\mathsf{Con}$ and $\widehat{\mathsf{Con}}$ be contract functions. We say that $\widehat{\mathsf{Con}}$ securely emulates $\mathsf{Con}$, if the following two properties hold:

- **Emulation Completeness.** There is a PPT algorithm $\widehat{\mathsf{Usr}}$ such that for any sequence $(\mathsf{In}_i, \mathsf{AuxIn}_i)_{i=1}^{\ell}$, we have
$$\Pr\left[(\mathsf{Out}_1, \ldots, \mathsf{Out}_\ell) = \left(\widehat{\mathsf{Out}}_1, \ldots, \widehat{\mathsf{Out}}_\ell\right)\right] = 1,$$
where the probability is taken over the following experiment:
  1. Set $St = \bot$ and $\widehat{St} = \bot$.

[11]The internal state transitions of the two contracts may differ, of course.

2. For $i = 1, \ldots, \ell$: Run $(St, \mathsf{Out}_i) := \mathsf{Con}\,(St, (\mathsf{In}_i, \mathsf{AuxIn}_i))$.
3. For $i = 1, \ldots, \ell$: Set $\widehat{\mathsf{AuxIn}}_i := \widehat{\mathsf{Usr}}(\mathsf{In}_i, \mathsf{AuxIn}_i)$ and run $(\widehat{St}, \widehat{\mathsf{Out}}_i) := \widehat{\mathsf{Con}}\left(\widehat{St}, (\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)\right)$.

- **Emulation Soundness.** There is a PPT algorithm Ext (the extractor) such that for any PPT algorithm $\mathcal{A}$, we have

$$\Pr\left[(\mathsf{Out}_1, \ldots, \mathsf{Out}_\ell) \neq \left(\widehat{\mathsf{Out}}_1, \ldots, \widehat{\mathsf{Out}}_\ell\right)\right] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the following experiment:

1. Set $St = \bot$ and $\widehat{St} = \bot$.
2. Run $(\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)_{i=1}^{\ell} \leftarrow \mathcal{A}(1^\lambda)$ and $(\mathsf{AuxIn}_i)_{i=1}^{\ell} \leftarrow \mathsf{Ext}((\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)_{i=1}^{\ell})$.
3. For $i = 1, \ldots, \ell$: Run $(St, \mathsf{Out}_i) := \mathsf{Con}\,(St, (\mathsf{In}_i, \mathsf{AuxIn}_i))$.
4. For $i = 1, \ldots, \ell$: Run $(\widehat{St}, \widehat{\mathsf{Out}}_i) := \widehat{\mathsf{Con}}\left(\widehat{St}, (\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)\right)$.

*Remark 8* (Reductions of Knowledge). Note that our soundness notion shares some similarities with *reductions of knowledge* [KP23]. We leave it as an interesting open problem if reductions of knowledge can be applied to smart contracts in a meaningful way.

*Remark 9* (Privacy). Our construction can also be tweaked to add a form of *privacy* to the emulation. Namely, if the succinct arguments we use are zero-knowledge, then no information about the auxiliary inputs of the original contract is revealed to the modified contract. We leave out a formal treatment of this to not distract from the main points.

*Remark 10* (Extractor). Typically, the extractor in a notion like our soundness notion needs additional power to be able to extract from the adversary's output. This can be observing random oracles, programming an extraction trapdoor into a common reference string, or via generic or algebraic group models, just to mention a few. We decided not to specify which form of extraction is used in our definition and leave it implicit. This is because our construction builds on *any* succinct argument of knowledge with straight-line extraction and inherits the form of extraction from it. Note that rewinding-based extraction may cause issues as we will need to call the extractor of the argument system repeatedly.

*Remark 11* (Parameters). If the contract functions implicitly take as input parameters such as a common reference string, then we assume that this is sampled at the onset of the experiments for emulation completeness and emulation soundness. In this case, the adversary gets these parameters as input and the extractor may embed a trapdoor into these parameters, while the common reference string must be indistinguishable from an honestly generated common reference string.

## 4.3 Construction and Analysis

We now present our construction. As outlined in our overview, it applies to smart contracts that accept a statement and a witness as part of their input and internally verify a binary relation between them. Beyond this internal check, the witness should not be used explicitly by the contract. The contract may also take additional inputs. We begin by formally defining the class of contracts to which our construction applies.

**Definition 7** (Relation-Checking Contract Function)**.** A relation-checking contract function for a binary relation $\Gamma \subseteq \{0,1\}^* \times \{0,1\}^*$ is a contract function Con as in Definition 5 such that:

- **Syntax of Inputs.** The inputs can be written as $\mathsf{In} = (\mathsf{stmt}, \overline{\mathsf{In}})$ and $\mathsf{AuxIn} = (\mathsf{witn}, \overline{\mathsf{AuxIn}})$.
- **Functionality.** There is a function $f$ such that for every state $St$, and every inputs $\mathsf{In} = (\mathsf{stmt}, \overline{\mathsf{In}})$ and $\mathsf{AuxIn} = (\mathsf{witn}, \overline{\mathsf{AuxIn}})$, we have

$$\mathsf{Con}(St, (\mathsf{In}, \mathsf{AuxIn})) = f\left(St, b, \mathsf{stmt}, \overline{\mathsf{In}}, \overline{\mathsf{AuxIn}}\right) \text{ with } b = \begin{cases} 1 & \text{if } (\mathsf{stmt}, \mathsf{witn}) \in \Gamma, \\ 0 & \text{otherwise} \end{cases}.$$

Before we present our final construction, we want to make sure the reader is familiar with our notation and definitions. To this end, we present the simpler warm-up construction from our overview using our notation. For this construction, we omit detailed security proofs and only sketch them, as our main focus is our final construction.

**Construction 1** (Emulating Relation-Checking Contracts – Warmup)**.** *Let* $\mathsf{Con}$ *be a relation-checking contract function for a binary relation* $\Gamma \subseteq \{0,1\}^* \times \{0,1\}^*$*. Let* $\mathsf{AS} = (\mathsf{ArgProve}, \mathsf{ArgVer})$ *be a non-interactive argument system (Definition 1) for* $\Gamma$*. Consider the following contract function* $\widehat{\mathsf{Con}}$ *and associated algorithm* $\widehat{\mathsf{Usr}}$*:*

- *For* $\mathsf{In} = (\mathsf{stmt}, \overline{\mathsf{In}})$ *and* $\mathsf{AuxIn} = (\mathsf{witn}, \overline{\mathsf{AuxIn}})$*, the user runs*

$$\widehat{\mathsf{Usr}}(\mathsf{In}, \mathsf{AuxIn}) = \begin{cases} (\mathsf{ArgProve}(\mathsf{stmt}, \mathsf{witn}), \overline{\mathsf{AuxIn}}) & \text{if } (\mathsf{stmt}, \mathsf{witn}) \in \Gamma, \\ (\bot, \overline{\mathsf{AuxIn}}) & \text{otherwise} \end{cases}.$$

  *Denote the output of the user as* $\widehat{\mathsf{AuxIn}}$*.*

- *The new contract for* $\widehat{\mathsf{AuxIn}} = (\pi, \overline{\mathsf{AuxIn}})$ *is then defined as*

$$\widehat{\mathsf{Con}}\left(\widehat{St}, (\mathsf{In}, \widehat{\mathsf{AuxIn}})\right) = f\left(\widehat{St}, b, \mathsf{stmt}, \overline{\mathsf{In}}, \overline{\mathsf{AuxIn}}\right) \text{ with } b = \begin{cases} 1 & \text{if } \pi \neq \bot \wedge \mathsf{ArgVer}(\mathsf{stmt}, \pi), \\ 0 & \text{otherwise} \end{cases}.$$

Intuitively, if AS is knowledge-sound, then $\widehat{\mathsf{Con}}$ securely emulates Con. Let us sketch the argument for that. Emulation completeness follows from correctness of the argument system, and from the fact that if $(\mathsf{stmt}, \mathsf{witn}) \in \Gamma$, the user inputs $\bot$ to the contract, so the contract will set $b = 0$. For emulation soundness, recall that we have to define an extractor Ext, which gets as input the inputs to our new contract $\widehat{\mathsf{Con}}$ and must output auxiliary inputs to the original contract Con that reproduce the same behavior when input to Con. Concretely, this means that Ext needs to output auxiliary inputs $\mathsf{AuxIn}_i = (\mathsf{witn}_i, \overline{\mathsf{AuxIn}_i})$ given $\mathsf{In}_i = (\mathsf{stmt}_i, \overline{\mathsf{In}_i})$ and $\widehat{\mathsf{AuxIn}}_i = (\pi_i, \overline{\mathsf{AuxIn}_i})$. To accomplish this, the extractor can (for inputs resulting in $b = 1$ in $\widehat{\mathsf{Con}}$) obtain $\mathsf{witn}_i$ from $\pi_i$ via the knowledge extractor of AS. We now continue with our actual construction, which is based on our hybrid compression technique.

**Construction 2** (Emulating Relation-Checking Contracts)**.** *Let* $\mathsf{Con}$ *be a relation-checking contract function for a binary relation* $\Gamma \subseteq \{0,1\}^* \times \{0,1\}^*$*. Consider a finite field* $\mathbb{F}$ *and two keyed hash functions*

$$\mathsf{H}^{(1)} \colon \mathcal{K}^{(1)} \times \mathbb{F}^k \to \mathbb{F}, \quad \mathsf{H}^{(2)} \colon \mathcal{K}^{(2)} \times \mathbb{F}^k \to \mathbb{F},$$

*where we assume that statements of* $\Gamma$ *can be parsed as vectors of* $k$ *field elements. Let* $\mathsf{AS} = (\mathsf{ArgProve}, \mathsf{ArgVer})$ *be a non-interactive argument system (Definition 1) for the relation* $\Gamma'$*, which is defined as follows:*

$$\Gamma' := \left\{ \left( \underbrace{(K_1, K_2, \alpha, \beta, \gamma)}_{=\mathsf{stmt}'}, \underbrace{(\mathsf{witn}, \mathsf{stmt})}_{=\mathsf{witn}'} \right) \;\middle|\; \begin{array}{ll} & (\mathsf{stmt}, \mathsf{witn}) \in \Gamma \\ \wedge & \beta = \mathsf{H}^{(2)}(K_2, \mathsf{stmt}) \\ \wedge & \gamma = \mathsf{UHF}(\alpha + \beta, \mathsf{stmt}) \end{array} \right\}.$$

*Consider the following contract function* $\widehat{\mathsf{Con}}$ *and associated algorithm* $\widehat{\mathsf{Usr}}$*:*

- *In addition to any public parameters that* $\mathsf{Con}$ *involves,* $\widehat{\mathsf{Con}}$ *makes use of two random public hash function keys* $K_1 \xleftarrow{\$} \mathcal{K}^{(1)}$ *and* $K_2 \xleftarrow{\$} \mathcal{K}^{(2)}$*.*

- *For* $\mathsf{In} = (\mathsf{stmt}, \overline{\mathsf{In}})$ *where* $\mathsf{stmt} \in \mathbb{F}^k$ *and* $\mathsf{AuxIn} = (\mathsf{witn}, \overline{\mathsf{AuxIn}})$*, the user* $\widehat{\mathsf{Usr}}(\mathsf{In}, \mathsf{AuxIn})$ *runs:*

  1. *If* $(\mathsf{stmt}, \mathsf{witn}) \notin \Gamma$*, return* $\widehat{\mathsf{AuxIn}} := (\bot, \overline{\mathsf{AuxIn}})$*. Otherwise, proceed.*
  2. *Set* $\alpha := \mathsf{H}^{(1)}(K_1, \mathsf{stmt})$*,* $\beta := \mathsf{H}^{(2)}(K_2, \mathsf{stmt})$*, and* $\gamma := \mathsf{UHF}(\alpha + \beta, \mathsf{stmt})$*.*
  3. *Set* $\mathsf{stmt}' := (K_1, K_2, \alpha, \beta, \gamma)$*,* $\mathsf{witn}' := (\mathsf{witn}, \mathsf{stmt})$*, and run* $\pi \leftarrow \mathsf{ArgProve}(\mathsf{stmt}', \mathsf{witn}')$*.*
  4. *Return* $\widehat{\mathsf{AuxIn}} := ((\pi, \beta), \overline{\mathsf{AuxIn}})$*.*

- *The new contract* $\widehat{\mathsf{Con}}(\widehat{St}, (\mathsf{In}, \widehat{\mathsf{AuxIn}}))$ *for* $\widehat{\mathsf{AuxIn}} = ((\pi, \beta), \overline{\mathsf{AuxIn}})$ *is then defined as:*

  1. *If* $(\pi, \beta) = \bot$*, set* $b := 0$*. Else, determine* $b$ *as follows:*
     (a) *Set* $\alpha := \mathsf{H}^{(1)}(K_1, \mathsf{stmt})$ *and* $\gamma := \mathsf{UHF}(\alpha + \beta, \mathsf{stmt})$.
     (b) *Set* $\mathsf{stmt}' := (K_1, K_2, \alpha, \beta, \gamma)$ *and* $b := \mathsf{ArgVer}(\mathsf{stmt}', \pi)$.
  2. *Return* $f\left(\widehat{St}, b, \mathsf{stmt}, \overline{\mathsf{In}}, \overline{\mathsf{AuxIn}}\right)$.

  *Notably,* $\widehat{\mathsf{Con}}$ *never evaluates* $\mathsf{H}^{(2)}$ *and* $\Gamma'$ *never evaluates* $\mathsf{H}^{(1)}$.

We now show emulation completeness and emulation soundness, as defined in Definition 6.

**Lemma 2** (Emulation Completeness)**.** *Construction 2 satisfies emulation completeness as defined in Definition 6.*

*Proof.* To show emulation completeness, we need to consider any sequence $(\mathsf{In}_i, \mathsf{AuxIn}_i)_{i=1}^{\ell}$ and the following experiment:

1. Sample $K_1 \xleftarrow{\$} \mathcal{K}^{(1)}$ and $K_2 \xleftarrow{\$} \mathcal{K}^{(2)}$
2. Set $St = \bot$ and $\widehat{St} = \bot$.
3. For $i = 1, \ldots, \ell$: Run $(St, \mathsf{Out}_i) := \mathsf{Con}\,(St, (\mathsf{In}_i, \mathsf{AuxIn}_i))$.
4. For $i = 1, \ldots, \ell$: Set $\widehat{\mathsf{AuxIn}}_i := \widehat{\mathsf{Usr}}(\mathsf{In}_i, \mathsf{AuxIn}_i)$ and run $(\widehat{St}, \widehat{\mathsf{Out}}_i) := \widehat{\mathsf{Con}}\left(\widehat{St}, (\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)\right)$.

We need to show that

$$(\mathsf{Out}_1, \ldots, \mathsf{Out}_\ell) = \left(\widehat{\mathsf{Out}}_1, \ldots, \widehat{\mathsf{Out}}_\ell\right)$$

with probability 1. To this end, we first note that we can rewrite the sequence $(\mathsf{In}_i, \mathsf{AuxIn}_i)_{i=1}^{\ell}$ as $\mathsf{In}_i = (\mathsf{stmt}_i, \overline{\mathsf{In}}_i)$ and $\mathsf{AuxIn}_i = (\mathsf{witn}_i, \overline{\mathsf{AuxIn}}_i)$, as Con is a relation-checking contract. We also rewrite the experiment with the concrete definition of $\widehat{\mathsf{Con}}$ and $\widehat{\mathsf{Usr}}$, and using the fact that Con is a relation-checking contract:

1. Sample $K_1 \xleftarrow{\$} \mathcal{K}^{(1)}$ and $K_2 \xleftarrow{\$} \mathcal{K}^{(2)}$
2. Set $St = \bot$ and $\widehat{St} = \bot$.
3. For $i = 1, \ldots, \ell$: *// Run original contract*
   (a) Set $b_i := \begin{cases} 1 & \text{if } (\mathsf{stmt}_i, \mathsf{witn}_i) \in \Gamma, \\ 0 & \text{otherwise} \end{cases}$.
   (b) Set $(St, \mathsf{Out}_i) := f\left(St, b_i, \mathsf{stmt}_i, \overline{\mathsf{In}}_i, \overline{\mathsf{AuxIn}}_i\right)$.
4. For $i = 1, \ldots, \ell$: *// Run new user and modified contract*
   (a) If $(\mathsf{stmt}_i, \mathsf{witn}_i) \notin \Gamma$, set $\widehat{\mathsf{AuxIn}}_i := (\bot, \overline{\mathsf{AuxIn}}_i)$.
   (b) Else:
      i. Set $\alpha_i := \mathsf{H}^{(1)}(K_1, \mathsf{stmt}_i)$, $\beta_i := \mathsf{H}^{(2)}(K_2, \mathsf{stmt}_i)$, $\gamma_i := \mathsf{UHF}(\alpha + \beta, \mathsf{stmt}_i)$.
      ii. Set $\mathsf{stmt}'_i := (K_1, K_2, \alpha_i, \beta_i, \gamma_i)$, $\mathsf{witn}'_i := (\mathsf{witn}_i, \mathsf{stmt}_i)$, $\pi_i \leftarrow \mathsf{ArgProve}(\mathsf{stmt}'_i, \mathsf{witn}'_i)$.
      iii. Set $\widehat{\mathsf{AuxIn}}_i := ((\pi_i, \beta_i), \overline{\mathsf{AuxIn}}_i)$.
   (c) If $\widehat{\mathsf{AuxIn}}_i := (\bot, \overline{\mathsf{AuxIn}}_i)$, set $\hat{b}_i := 0$. Else, determine $\hat{b}_i$ as $\hat{b}_i := \mathsf{ArgVer}(\mathsf{stmt}'_i, \pi_i)$.
   (d) Set $(\widehat{St}, \widehat{\mathsf{Out}}_i) := f\left(\widehat{St}, \hat{b}_i, \mathsf{stmt}_i, \overline{\mathsf{In}}_i, \overline{\mathsf{AuxIn}}_i\right)$.

The reader can observe that it is sufficient to show that $(b_1, \ldots, b_\ell) = (\hat{b}_1, \ldots, \hat{b}_\ell)$ with probability 1. This can easily be seen: If $(\mathsf{stmt}_i, \mathsf{witn}_i) \notin \Gamma$, then $b_i = 0$, and $\hat{b}_i = 0$. On the other hand, if $(\mathsf{stmt}_i, \mathsf{witn}_i) \in \Gamma$, then $b_i = 1$ by definition, and $\hat{b}_i = 1$ by correctness of the argument system AS. □

**Lemma 3** (Emulation Soundness). *Assume that* $\mathsf{AS}$ *is knowledge-sound, as defined in Definition 2, and assume that* $\mathsf{H}^{(1)}$ *and* $\mathsf{H}^{(2)}$ *are jointly UHF-hard, as defined in Definition 4. Then, Construction 2 satisfies emulation soundness as defined in Definition 6.*

*Proof.* Denote the extractor for $\mathsf{AS}$ guaranteed by the knowledge-soundness by $\mathsf{AS.Ext}$. To show emulation soundness, we first need to define an extractor $\mathsf{Ext}$. This extractor gets a list of inputs $(\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)_{i=1}^{\ell}$ to the modified contract $\widehat{\mathsf{Con}}$ and needs to output auxiliary inputs $(\mathsf{AuxIn}_i)_{i=1}^{\ell}$ to the original contract, such that the same output behavior is obtained. Our extractor will work on each of these inputs independently. Recall that the inputs to the modified contract have the form $\mathsf{In}_i = (\mathsf{stmt}_i, \overline{\mathsf{In}}_i)$ and $\widehat{\mathsf{AuxIn}}_i = ((\pi_i, \beta_i), \overline{\mathsf{AuxIn}}_i)$, and the desired auxiliary inputs to the original contract have the form $\mathsf{AuxIn}_i = (\mathsf{witn}_i, \overline{\mathsf{AuxIn}}_i)$. The idea is now to extract $\mathsf{witn}_i$ from $\pi_i$ using the extractor $\mathsf{AS.Ext}$, and then use joint UHF-hardness to argue that these are indeed valid witnesses. In detail, we define our extractor $\mathsf{Ext}((\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)_{i=1}^{\ell})$ as follows[12]:

- For $i = 1, \ldots, \ell$:
  1. Set $\mathsf{BadExt}_i := 0$ and $\mathsf{BadColl}_i := 0$.
  2. Parse $\mathsf{In}_i = (\mathsf{stmt}_i, \overline{\mathsf{In}}_i)$, and $\widehat{\mathsf{AuxIn}}_i = ((\pi_i, \beta_i), \overline{\mathsf{AuxIn}}_i)$.
  3. Determine $\alpha_i, \gamma_i, \mathsf{stmt}'_i$, and $b_i$ in the way same way as $\widehat{\mathsf{Con}}$ does, namely:
     (a) If $(\pi_i, \beta_i) = \bot$, set $b_i := 0$ and do not define $\alpha_i, \gamma_i, \mathsf{stmt}'_i$.
     (b) Otherwise, set $\alpha_i := \mathsf{H}^{(1)}(K_1, \mathsf{stmt}_i)$ and $\gamma_i := \mathsf{UHF}(\alpha + \beta, \mathsf{stmt}_i)$.
     (c) Set $\mathsf{stmt}'_i := (K_1, K_2, \alpha_i, \beta_i, \gamma_i)$ and $b_i := \mathsf{ArgVer}(\mathsf{stmt}'_i, \pi_i)$.
  4. If $b_i = 0$, set $\mathsf{witn}_i := \bot$.
  5. If $b_i = 1$, determine $\mathsf{witn}_i$ as follows:
     (a) Run $\mathsf{witn}'_i \leftarrow \mathsf{AS.Ext}(\mathsf{stmt}'_i, \pi'_i)$. Note that $\mathsf{stmt}'_i$ is defined.
     (b) Parse $\mathsf{witn}'_i := (\mathsf{witn}_i, \tilde{\mathsf{stmt}}_i)$.
     (c) If $(\mathsf{stmt}'_i, \mathsf{witn}'_i) \notin \Gamma'$, set $\mathsf{BadExt}_i := 1$.
     (d) If $\tilde{\mathsf{stmt}}_i \neq \mathsf{stmt}_i$ and $\mathsf{BadExt}_i = 0$, set $\mathsf{BadColl}_i := 1$.
  6. Set $\mathsf{AuxIn}_i := (\mathsf{witn}_i, \overline{\mathsf{AuxIn}}_i)$.
- Output $(\mathsf{AuxIn}_i)_{i=1}^{\ell}$.

Now that we have defined our extractor, we need to show that for any PPT algorithm $\mathcal{A}$, we have

$$\Pr\left[(\mathsf{Out}_1, \ldots, \mathsf{Out}_\ell) \neq \left(\widehat{\mathsf{Out}}_1, \ldots, \widehat{\mathsf{Out}}_\ell\right)\right] \leq \mathsf{negl}(\lambda),$$

in the following experiment:

1. Sample $K_1 \xleftarrow{\$} \mathcal{K}^{(1)}$ and $K_2 \xleftarrow{\$} \mathcal{K}^{(2)}$
2. Set $St = \bot$ and $\widehat{St} = \bot$.
3. Run $(\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)_{i=1}^{\ell} \leftarrow \mathcal{A}(1^\lambda)$ and $(\mathsf{AuxIn}_i)_{i=1}^{\ell} \leftarrow \mathsf{Ext}((\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)_{i=1}^{\ell})$.
4. For $i = 1, \ldots, \ell$: Run $(St, \mathsf{Out}_i) := \mathsf{Con}\,(St, (\mathsf{In}_i, \mathsf{AuxIn}_i))$.
5. For $i = 1, \ldots, \ell$: Run $(\widehat{St}, \widehat{\mathsf{Out}}_i) := \widehat{\mathsf{Con}}\left(\widehat{St}, (\mathsf{In}_i, \widehat{\mathsf{AuxIn}}_i)\right)$.

As in the proof of emulation completeness, our goal will be to show that the sequence of $b$ values computed in the executions of the original and modified contract are the same. To this end, we first argue that this is the case assuming $\mathsf{BadExt}_i = 0$ and $\mathsf{BadExt}_i = 0$ for all $i$, and then bound the probability that $\mathsf{BadExt}_i = 1$ or $\mathsf{BadExt}_i = 1$ for some $i$. We do so in the following claims. Combining the claims, we obtain the result.

[12] We assume that for $\mathsf{witn} = \bot$, there exists no statements $\mathsf{stmt}$ such that $(\mathsf{stmt}, \mathsf{witn}) \in \Gamma$.

*Claim 1.* We have

$$\Pr\left[(\mathsf{Out}_1,\ldots,\mathsf{Out}_\ell) \neq \left(\widehat{\mathsf{Out}}_1,\ldots,\widehat{\mathsf{Out}}_\ell\right) \mid \nexists i \in [\ell]\colon\ \mathsf{BadExt}_i = 1 \vee \mathsf{BadColl}_i = 1\right] = 0.$$

*Proof of Claim.* This follows with the same arguments as in emulation completeness, by considering the two cases $b_i = 0$ and $b_i = 1$ separately. The assumption that $\mathsf{BadExt}_i = 0$ and $\mathsf{BadExt}_i = 0$ for all $i$ is only needed in the $b_i = 1$ case: as $\mathsf{BadExt}_i = 0$, we know that $(\mathsf{stmt}_i, \mathsf{witn}_i) \in \Gamma'$, which means that $(\tilde{\mathsf{stmt}}_i, \mathsf{witn}_i) \in \Gamma$, and as $\mathsf{BadColl}_i = 0$, we know that $\tilde{\mathsf{stmt}}_i = \mathsf{stmt}_i$, so $(\mathsf{stmt}_i, \mathsf{witn}_i) \in \Gamma$. This means that the original contract will also set $b = 1$ for this input.

*Claim 2.* Assume that $\mathsf{AS}$ is knowledge-sound. Then, we have

$$\forall i \in [\ell]\colon \quad \Pr\left[\mathsf{BadExt}_i = 1\right] \leq \mathsf{negl}(\lambda).$$

*Proof of Claim.* The proof is an easy reduction to knowledge soundness. The reduction simulates the emulation soundness game for $\mathcal{A}$ (without running $\mathsf{AS.Ext}$) and outputs $\mathsf{stmt}'_i$ and $\pi_i$ to the knowledge soundness game.

*Claim 3.* Assume that $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$ are jointly UHF-hard. Then, we have

$$\Pr\left[\exists i \in [\ell]\colon\ \mathsf{BadColl}_i = 1\right] \leq \mathsf{negl}(\lambda).$$

*Proof of Claim.* We prove this via a reduction to the joint UHF-hardness of $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$. The reduction gets as input keys $K_1 \xleftarrow{\$} \mathcal{K}^{(1)}$ and $K_2 \xleftarrow{\$} \mathcal{K}^{(2)}$ from the joint UHF-hardness game. It simulates the emulation soundness game for $\mathcal{A}$. Then, it identifies the smallest $i \in [\ell]$ such that $\mathsf{BadColl}_i = 1$ (i.e., $\tilde{\mathsf{stmt}}_i \neq \mathsf{stmt}_i$ and $\mathsf{BadExt}_i = 0$) and outputs $(\mathsf{stmt}_i, \tilde{\mathsf{stmt}}_i)$ to the joint UHF-hardness game. If no such $i$ exists, the reduction aborts. Clearly, the reduction simulates the game perfectly for $\mathcal{A}$ and is efficient if $\mathcal{A}$ is efficient. It remains to show that the reduction wins in the joint UHF-hardness game, assuming $\mathsf{BadColl}_i = 1$. Define $\alpha := \mathsf{H}^{(1)}(K_1, \mathsf{stmt}_i)$ and $\beta := \mathsf{H}^{(2)}(K_2, \tilde{\mathsf{stmt}}_i)$ as in the joint UHF-hardness game. By definition of $\mathsf{BadColl}_i$, we know that $\tilde{\mathsf{stmt}}_i \neq \mathsf{stmt}_i$, and so it just remains to argue that $\mathsf{UHF}(\alpha+\beta, \mathsf{stmt}_i) = \mathsf{UHF}(\alpha+\beta, \tilde{\mathsf{stmt}}_i)$.

We have $\alpha = \alpha_i$ by definition of $\alpha_i$. Further, as $\mathsf{BadExt}_i = 0$, we know that $(\mathsf{stmt}'_i, \mathsf{witn}'_i) \in \Gamma'$, which means that $\beta_i = \mathsf{H}^{(2)}(K_2, \tilde{\mathsf{stmt}}_i) = \beta$. By definition of $\Gamma'$, we therefore get

$$\gamma_i = \mathsf{UHF}(\alpha_i + \beta_i, \tilde{\mathsf{stmt}}_i) = \mathsf{UHF}(\alpha + \beta, \tilde{\mathsf{stmt}}_i).$$

And by definition of $\gamma_i$, we get

$$\gamma_i = \mathsf{UHF}(\alpha_i + \beta_i, \mathsf{stmt}_i) = \mathsf{UHF}(\alpha + \beta, \mathsf{stmt}_i).$$

Combined, this shows the desired equality. □

**Corollary 1** (Secure Emulation)**.** *Let* $\mathsf{Con}$ *be a relation-checking contract function and consider* $\widehat{\mathsf{Con}}$ *as defined in Construction 2. Assume that* $\mathsf{AS}$ *is knowledge-sound, as defined in Definition 2, and assume that* $\mathsf{H}^{(1)}$ *and* $\mathsf{H}^{(2)}$ *are jointly UHF-hard, as defined in Definition 4. Then,* $\widehat{\mathsf{Con}}$ *securely emulates* $\mathsf{Con}$ *as defined in Definition 6.*

## 4.4 Can We Do Better?

One may ask if it is necessary to have both $\alpha$ and $\beta$ in the statement in our construction. We show that if any of these two is removed from the statement of $\Gamma'$ to its witness, then (almost) any statement $\mathsf{stmt}^{(1)}$ that does not satisfy $\Gamma$ can be accepted by $\widehat{\mathsf{Con}}$. For the attack to work we need just one statement $\mathsf{stmt}^{(2)}$ that *does* satisfy $\Gamma$.

**Removing $\alpha$.** If $\alpha$ is not part of the statement of $\Gamma'$, but only as a witness, then an attacker succeeds as follows. First they compute $\beta = H^{(2)}(K_2, \mathsf{stmt}^{(2)})$, $\alpha^{(1)} = H^{(1)}(K_1, \mathsf{stmt}^{(1)})$, and $\gamma = \mathsf{UHF}(\alpha^{(1)} + \beta, \mathsf{stmt}^{(1)})$. For the relation $\Gamma'$ to hold for $\mathsf{stmt}' = (K_1, K_2, \beta, \gamma)$ it remains to find $\alpha^{(2)}$ such that $\gamma = \mathsf{UHF}(\alpha^{(2)} + \beta, \mathsf{stmt}^{(2)})$. Given $\beta, \gamma$, and $\mathsf{stmt}^{(2)}$ this reduces to solving a univariate polynomial equation, which is doable in polynomial time. In the (unlikely) case the equation does not have a root,

the adversary repeats the attack with different $\mathsf{stmt}^{(1)}, \mathsf{stmt}^{(2)}$. Given $\alpha^{(2)}$ the adversary then produces a proof $\pi$ for $\Gamma'$ with $\mathsf{stmt}' = (K_1, K_2, \beta, \gamma)$ and makes $\widehat{\mathsf{Con}}$ to accept $\mathsf{stmt}^{(1)}, \pi, \beta$.

**Removing $\beta$.** If $\beta$ is not available as part of the statement to $\Gamma'$, but only as a witness $\beta^{(2)}$, then an attacker succeeds as follows. First they compute $\beta^{(2)} = H^{(2)}(K_2, \mathsf{stmt}^{(2)})$, $\alpha = H^{(1)}(K_1, \mathsf{stmt}^{(1)})$, and $\gamma = \mathsf{UHF}(\alpha + \beta^{(2)}, \mathsf{stmt}^{(2)})$. Relation $\Gamma'$ now holds for $\mathsf{stmt}' = (K_1, K_2, \alpha, \gamma)$ so the adversary creates proof $\pi$. It remains to find $\beta^{(1)}$ such that $\gamma = \mathsf{UHF}(\alpha + \beta^{(1)}, \mathsf{stmt}^{(1)})$. Given $\alpha, \gamma$, and $\mathsf{stmt}^{(1)}$ this again reduces to solving a univariate polynomial equation, which is doable in polynomial time. With $\beta^{(1)}$ the adversary makes $\widehat{\mathsf{Con}}$ to accept $\mathsf{stmt}^{(1)}, \pi, \beta^{(1)}$.

# 5 Benchmarks

To assess the practicality of our approach, we benchmarked it in the context of the Ethereum blockchain. Smart contracts in the Ethereum blockchain currently support (through moderately expensive built-in operations) scalar multiplications and pairings over the BN254 curve. Throughout, we assume that $\mathbb{F}$ is the scalar field of this curve, with $|\mathbb{F}| \approx 2^{254}$.

## 5.1 Setting the Stage

Before we present our results, we explain which constructions we benchmarked and how.

**Business Logic.** Our method transforms a smart contract that checks a certain relation $\Gamma$ into an efficient smart contract that checks the same relation using a SNARK while compressing the statement and avoiding the hash function dilemma. For our benchmarks, we used the following example relation $\Gamma$, which verifies that the witness is a root of the polynomial given as the statement:

$$\Gamma = \left\{ \left( \underbrace{(x_1, \ldots, x_k)}_{=:\mathsf{stmt}}, \underbrace{\eta}_{=:\mathsf{witn}} \right) \in \mathbb{F}^k \times \mathbb{F} \;\middle|\; \sum_{i=1}^{k} x_i \eta^{i-1} = 0 \right\}.$$

**Schemes.** For comparison, we have implemented the schemes outlined in Section 4.1. Concretely, we compare the following schemes:

1. **Uncompressed.** This corresponds to Construction 1. The smart contract takes as input the statement $\mathsf{stmt} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ and a proof $\pi$. It simply runs the verifier for the proof and returns the result. The proof system is for relation $\Gamma$.
2. **SHA-256 Compressed.** The smart contract takes as input the statement $\mathsf{stmt} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ and a proof $\pi$. The contract computes the hash $\alpha := \mathsf{H}(\mathsf{stmt})$, where $\mathsf{H}$ is SHA-256 [Nat02]. It then verifies the proof $\pi$ with respect to statement $\alpha$. Here, $\pi$ is a proof for the relation

$$\Gamma' = \left\{ \left( \underbrace{\alpha}_{=:\mathsf{stmt}'}, \underbrace{(\overbrace{\mathsf{witn}}^{\mathsf{witn}=\eta\in\mathbb{F}}, \mathsf{stmt})}_{=:\mathsf{witn}'} \right) \;\middle|\; \begin{array}{ll} & (\mathsf{stmt}, \mathsf{witn}) \in \Gamma \\ \wedge & \alpha = \mathsf{H}(\mathsf{stmt}) \end{array} \right\}.$$

3. **Poseidon Compressed.** This is the same as the previous one, but $\mathsf{H}$ is Poseidon [GKR+21].
4. **Hybrid Compression.** This is our scheme, following Construction 2. The smart contract takes as input the statement $\mathsf{stmt} \in \mathbb{F}^k$, a hash $\beta$, and a proof $\pi$. The contract then
   (a) Computes $\alpha := \mathsf{H}^{(1)}(\mathsf{stmt})$ where $\mathsf{H}^{(1)}$ is SHA-256.
   (b) Computes $\gamma = \mathsf{UHF}(\alpha + \beta, \mathsf{stmt})$ (cf. Definition 3).
   (c) Verifies the proof $\pi$ for statement $(\alpha, \beta, \gamma)$.

Here, the relation for the proof system is

$$\Gamma' = \left\{ \left( \underbrace{(\alpha, \beta, \gamma)}_{=:\mathsf{stmt}'}, \underbrace{(\mathsf{witn}, \mathsf{stmt})}_{=:\mathsf{witn}'} \right) \;\middle|\; \begin{array}{ll} & (\mathsf{stmt}, \mathsf{witn}) \in \Gamma \\ \wedge & \beta = \mathsf{H}^{(2)}(K_2, \mathsf{stmt}) \\ \wedge & \gamma = \mathsf{UHF}(\alpha + \beta, \mathsf{stmt}) \end{array} \right\},$$

where $\mathsf{H}^{(2)}$ is Poseidon.

**Metrics.** To compare the schemes, we consider the metrics already outlined in the introduction. Namely, we consider the *amount of gas* consumed by a smart contract call, which corresponds to the financial cost of such a call. Related to that, we also consider the *code size of the verifier*, which in our case is the smart contract bytecode size. We then consider the *witness size* for the relation when implemented in Groth16 [Gro16] as it directly affects the *prover time* and *size of the trusted setup*[13]. Related to the size of the trusted setup and the proving efficiency, we count the *number of quadratic constraints* arising in the Groth16 representation of the circuit.

**Tooling and Implementation.** We have written the circuits implementing the relations above using the *Circom 2.2.2 framework using snarkjs 0.7.5*[14]. That is, we use the Circom framework to implement circuits which are then proven and verified using snarkjs, which implements Groth16. As usual in Ethereum, we have written the smart contracts in *Solidity*[15] To determine the amount of gas that is consumed, we used *ethereumjs*[16] to run the smart contracts in a local version of Ethereum Prague/Electra. We provide our code at

`https://github.com/khovratovich/two-worlds-ref`.

Our benchmarks are run on 10-core Macbook running MacOS 15.6 (kernel 24.6.0), with Apple M1 Max and 32GB of memory. Proving times are the sum of times for all cores.

**Details on Poseidon.** For the sake of compatibility and enabling apples-to-apples comparison, we sought for the implementation of Poseidon that is cross-compatible between the Circom and Solidity frameworks. We have selected the implementation which is the most efficient in gas[17] as it is the bottleneck for all Poseidon usages in Solidity. The available builtin is the $t$-to-1 compression function, which is implemented as the sponge-mode Poseidon over the state of $t+1$ field elements and denoted as Poseidon-$t$. We then implemented hashing of $k > t$ inputs as nested hashing[18] $\mathrm{Poseidon}(x_1, x_2, \ldots, x_k) = H(H(H(x_1, x_2, \ldots, x_t), x_{t+1}, \ldots, x_{2t-1}), \ldots, x_k)$ where $H$ is Poseidon-$t$. We now explain how we select the $t$ in Poseidon-$t$. It is known that bigger $t$ allows for smaller number of quadratic constraints in the Circom/circuit setting, but the bigger matrices it requires are heavy in the Solidity/native implementation. We thus maximize $t$ in our hybrid compression approach (concretely we select $t = 4$ for 4-input hashing and $t = 16$ otherwise), but take it low (setting $t = 4$) for the Poseidon compression in order to minimize the Solidity cost.

## 5.2 Results and Discussion

Our results are presented in Tables 2 to 4, and we briefly discuss them here.

**Performance of Existing Approaches.** The uncompressed variant works reasonably fine for small statements, but for big statement the gas cost and the contract size exceed the limit. The SHA-256 compression yields a reasonably small contract and is very efficient in gas. However, the proving time increases quickly, at a rate of about 2.5 seconds per statement element, which can be prohibitive for many applications. The Poseidon compression yields a large contract, which barely fits the maximum allowed bytecode size (22 vs 24 KB). It is also gas consuming as it spends, roughly, 20K gas per additional statement element, or 50 elements per million gas, thus quickly reaching the Ethereum block gas limit.

[13] Note that the Circom framework rounds up the required setup size to the nearest power of two.
[14] See `https://docs.circom.io/`.
[15] See `https://soliditylang.org/`.
[16] See `https://github.com/ethereumjs`.
[17] See `https://github.com/chancehudson/poseidon-solidity/`.
[18] We remark that this mode of operation is secure only when the input length is constant in a protocol.

**Performance of Our Approach.** We see that already with statements of size 4 our approach eliminates the bottlenecks of both SHA-256 and Poseidon compressions. The contract size is only marginally larger than the SHA-256 compression and smaller than the uncompressed case for statement size of 8 and larger. The proving time of our approach is comparable with the Poseidon compression. One may wonder why our proving time (and setup size) is even smaller than for Poseidon compression in some cases. This is because for Poseidon compression, we must use the Solidity-optimal small state size $t$, whereas we can use a larger state size in our hybrid compression approach, as Poseidon is never called in Solidity. The same state size would yield the same proving time. The gas cost of our approach is about the same as for the SHA-256 one. In conclusion:

*While our approach is not the best for any metric, it is close to optimal in all, and hence resolves the hash function dilemma.*

| Benchmark | Quadr. Constraints | Witness Size | Setup Size | Proving Time | Contract Size | Gas |
|---|---|---|---|---|---|---|
| **Uncompressed** | 3 | 332 B | 10 KB | 0.74 s | 1 664 B | 208 946 |
| **SHA-256 Compression** | 91 945 | 3 MB | 150 MB | 10.54 s | 1 610 B | 189 472 |
| **Poseidon Compression** | 303 | 24 KB | 1.1 MB | 1.24 s | 22 704 B | 250 008 |
| **Hybrid Compression (Ours)** | 306 | 24 KB | 1.1 MB | 1.25 s | 1 976 B | 203 946 |

Table 2: Results of our benchmarks for the statement size $k = 4$. Gas is the execution cost of a transaction, i.e., a contract call. Poseidon Compression Hybrid Compression both use Poseidon-4.

| Benchmark | Quadr. Constraints | Witness Size | Setup Size | Proving Time | Contract Size | Gas |
|---|---|---|---|---|---|---|
| **Uncompressed** | 15 | 1.1 KB | 38 KB | 0.75 s | 25 790 B | 288 627 |
| **SHA-256 Compression** | 280 069 | 9 MB | 600 MB | 36.22 s | 1 614 B | 189 582 |
| **Poseidon Compression** | 1 515 | 118 KB | 4.6 MB | 2.07 s | 22 705 B | 478 162 |
| **Hybrid Compression (Ours)** | 642 | 68 KB | 4.6 MB | 1.69 s | 1 980 B | 206 958 |

Table 3: Results of our benchmarks for the statement size $k = 16$. Gas is the execution cost of a transaction, i.e., a contract call. Poseidon Compression uses Poseidon-4, whereas Hybrid Compression uses Poseidon-16.

| Benchmark | Quadr. Constraints | Witness Size | Setup Size | Proving Time | Contract Size | Gas |
|---|---|---|---|---|---|---|
| **Uncompressed** | 255 | 16 KB | 580 KB | 1.01 s | 25 574 B | 1 883 907 |
| **SHA-256 Compression** | 4 042 549 | 130 MB | 9.5 GB | 295.37 s | 1 615 B | 193 984 |
| **Poseidon Compression** | 25 755 | 2 MB | 75 MB | 14.36 s | 22 712 B | 5 041 709 |
| **Hybrid Compression (Ours)** | 10 914 | 1.1 MB | 37 MB | 10.35 s | 1 983 B | 268 960 |

Table 4: Results of our benchmarks for the statement size $k = 256$. Gas is the execution cost of a transaction, i.e., a contract call. Poseidon Compression uses Poseidon-4, whereas Hybrid Compression uses Poseidon-16.

# References


[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009. (Cited on Page 4, 5, 6, 7.)

[BDG+20] Zhenzhen Bao, Itai Dinur, Jian Guo, Gaëtan Leurent, and Lei Wang. Generic attacks on hash combiners. *J. Cryptol.*, 33(3):742–823, 2020. (Cited on Page 5.)

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. (Cited on Page 7.)

[CN22] Shahar P. Cohen and Moni Naor. Low communication complexity protocols, collision resistant hash functions and secret key-agreement protocols. In *CRYPTO (3)*, volume 13509 of *Lecture Notes in Computer Science*, pages 252–281. Springer, 2022. (Cited on Page 5, 7.)

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021. (Cited on Page 4, 17.)

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 321–340. Springer, Berlin, Heidelberg, December 2010. (Cited on Page 3.)

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Berlin, Heidelberg, May 2016. (Cited on Page 3, 18.)

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. (Cited on Page 3.)

[HS08] Jonathan J. Hoch and Adi Shamir. On the strength of the concatenated hash combiner when all the hash functions are weak. In *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 616–630. Springer, 2008. (Cited on Page 4.)

[KN97] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997. (Cited on Page 5.)

[KP23] Abhiram Kothapalli and Bryan Parno. Algebraic reductions of knowledge. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 669–701. Springer, Cham, August 2023. (Cited on Page 12.)

[Lab21] Matter Labs. Zksync protocol. Github, 2021. available at = https://github.com/matter-labs/zksync/blob/master/docs/protocol.md. (Cited on Page 3.)

[Lab25] Matter Labs. ZkEVM Era. Github, 2025. available at = https://github.com/matter-labs/era-zkevm_circuits/blob/v1.5.0/src/scheduler/mod.rs. (Cited on Page 3.)

[LW15] Gaëtan Leurent and Lei Wang. The sum can be weaker than each part. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 345–367. Springer, 2015. (Cited on Page 5.)

[Nat02] National Institute of Standards and Technology. Secure hash standard. *Federal Information Processing Standards Publication (FIPS)*, 2002. available at `https://csrc.nist.gov/files/pubs/fips/180-2/final/docs/fips180-2.pdf`. (Cited on Page 3, 17.)

[PSS19] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4. *Tornado cash privacy solution version*, 1(6), 2019. (Cited on Page 3.)

[RY20] Anup Rao and Amir Yehudayoff. *Communication complexity: and applications*. Cambridge University Press, 2020. (Cited on Page 5.)

[Tea24] Term Structure Team. Zktrueup protocol. Github, 2024. available at = https://tinyurl.com/23628fyt. (Cited on Page 3.)

[Tea25] Railgun Team. Railgun protocol. Github, 2025. available at https://github.com/Railgun-Privacy/contract/blob/main/contracts/logic/Verifier.sol. (Cited on Page 3.)

[Yao79] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing (preliminary report). In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 209–213, 1979. (Cited on Page 4, 5, 6.)

[ZSCZ24] Jiaxing Zhao, Srinath Setty, Weidong Cui, and Greg Zaverucha. MicroNova: Folding-based arguments with efficient (on-chain) verification. Cryptology ePrint Archive, Paper 2024/2099, 2024. (Cited on Page 5.)

# Appendix

## A Random Oracle Bound for Joint UHF Hardness

Here, we show that two hash functions modeled as a random oracle are jointly UHF hard, assuming the field is large enough. Note that keys could even be omitted in the random oracle model.

**Lemma 4.** *Consider a finite field $\mathbb{F}$ and two keyed hash functions $\mathsf{H}^{(1)}\colon \mathcal{K}^{(1)} \times \mathbb{F}^k \to \mathbb{F}$ and $\mathsf{H}^{(2)}\colon \mathcal{K}^{(2)} \times \mathbb{F}^k \to \mathbb{F}$, both modeled a random oracle. Assume that $|\mathbb{F}| \geq k \cdot 2^{2\lambda}$. Then, $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$ are jointly UHF-hard.*

*Proof.* We have to consider the following experiment:

1. Sample $K_1 \xleftarrow{\$} \mathcal{K}^{(1)}$ and $K_2 \xleftarrow{\$} \mathcal{K}^{(2)}$.
2. Run $(x, \tilde{x}) \leftarrow \mathcal{A}^{\mathsf{H}^{(1)},\mathsf{H}^{(2)}}(K_1, K_2)$ ($\mathcal{A}$ gets oracle access to $\mathsf{H}^{(1)}$ and $\mathsf{H}^{(2)}$).
3. Set $\alpha := \mathsf{H}^{(1)}(K_1, x)$ and $\beta := \mathsf{H}^{(2)}(K_2, \tilde{x})$.

Without loss of generality, we assume that the adversary never issues the same random oracle query twice. In this experiment, we define the following events:

- Event $\mathsf{Win}$: This event occurs if $\sum_{i=1}^{k}(\alpha+\beta)^{i-1}x_i = \sum_{i=1}^{k}(\alpha+\beta)^{i-1}\tilde{x}_i$.
- Event $\mathsf{UHFColl}_{t,r}$: This event occurs for $t < r$, if for the $t$th random oracle query $\alpha = \mathsf{H}^{(b)}(K, x)$ and the $r$th random oracle query $\beta = \mathsf{H}^{(b')}(K', x')$ it holds that

$$x \neq x' \text{ and } \sum_{i=1}^{k}(\alpha+\beta)^{i-1}x_i = \sum_{i=1}^{k}(\alpha+\beta)^{i-1}x'_i.$$

It is clear that if $\mathsf{Win}$ occurs, then $\mathsf{UHFColl}_{k,r}$ must occur for some $t, r$. Now, fix $t < r$. When the $r$th query is made, the values $\alpha$ and $x \neq x'$ are fixed and $\beta$ is sampled at random from $\mathbb{F}$. Then, we get

$$\Pr\left[\mathsf{UHFColl}_{t,r}\right] = \Pr_{\beta \xleftarrow{\$} \mathbb{F}}\left[\sum_{i=1}^{k}(\alpha+\beta)^{i-1}x_i = \sum_{i=1}^{k}(\alpha+\beta)^{i-1}x'_i\right]$$
$$\Pr_{\alpha+\beta \xleftarrow{\$} \mathbb{F}}\left[\sum_{i=1}^{k}(\alpha+\beta)^{i-1}(x_i - x'_i) = 0\right] \leq \frac{k-1}{|\mathbb{F}|},$$

where the last inequality follows from the fact that $x$ and $x'$ define a non-zero polynomial of degree at most $k-1$.

Denoting the number of random oracle queries made by $\mathcal{A}$ by $q$, we can now combine what we have using a union bound:

$$\Pr\left[\mathsf{Win}\right] \leq \Pr\left[\exists t, r\colon \mathsf{UHFColl}_{t,r}\right] \leq \frac{q^2(k-1)}{|\mathbb{F}|} \leq \frac{q^2 k}{|\mathbb{F}|} \leq \frac{q^2}{2^{2\lambda}},$$

where we used that $|\mathbb{F}| \geq k2^{2\lambda}$. Asymptotically, this is negligible as $q$ is polynomial, and concretely, this corresponds to $\lambda$ bits of security. □