# On the Pitfalls of Modeling Individual Knowledge

Wojciech Ciszewski
University of Warsaw
Warsaw, Poland
IDEAS Institute
Warsaw, Poland
w.ciszewski2@uw.edu.pl

Stefan Dziembowski
University of Warsaw
Warsaw, Poland
IDEAS Institute
Warsaw, Poland
stefan.dziembowski@crypto.edu.pl

Tomasz Lizurej
NASK National Research Institute
Warsaw, Poland
tomasz.lizurej@crypto.edu.pl

Marcin Mielniczuk
University of Warsaw
Warsaw, Poland
m.mielniczuk@uw.edu.pl

## Abstract

The concept of *knowledge* has been central in cryptography, especially within cryptographic proof systems. Traditionally, research in this area considers an abstract *prover* defending a claim that it knows a message $M$. Recently, a stronger concept—termed "individual" (Dziembowski et al., CRYPTO'23) or "complete" (Kelkar et al., CCS'24) knowledge—has emerged. This notion ensures the prover *physically stores* $M$ on a machine that it controls. As we argue in the paper, this concept also appears in earlier work on "nonoutsourceable puzzles" (Miller et al., CCS'15), which implicitly assumes that quickly performing complex computation on a string $M$ implies storing it on a single machine. In this line of work, the authors typically rely on algorithms whose computation requires a massive number of queries to a hash function H.

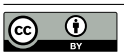
In this paper, we highlight a subtle issue in the modeling used in some of these papers: specifically, the assumption that H can be modeled as an *atomic* random oracle on long messages. Unfortunately, this does not correspond well to how hash functions are constructed in practice. For example, the real-world hash functions (e.g., Merkle-Damgård or sponge-based) allow partial evaluation on long inputs, violating this assumption. Another example is the hashing used in Bitcoin mining, which permits similar precomputation. This undermines some protocols relying on individual knowledge. We demonstrate practical attacks against Miller et al.'s and Kelkar et al.'s schemes based on this observation, and discuss secure alternatives. Our alternative constructions avoid reliance on the random oracle behavior of hash functions on long messages and provide security in the individual cryptography model of Dziembowski et al. (CRYPTO'23).

## CCS Concepts

• **Security and privacy**;

## Keywords

nonoutsourceable puzzles, hash functions, individual knowledge



This work is licensed under a Creative Commons Attribution 4.0 International License.
*CCS '26, The Hague, Netherlands*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832752

**ACM Reference Format:**
Wojciech Ciszewski, Stefan Dziembowski, Tomasz Lizurej, and Marcin Mielniczuk. 2026. On the Pitfalls of Modeling Individual Knowledge. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3830454.3832752

## 1 Introduction

The challenge of *proving knowledge* has been a topic of significant interest in cryptography, starting from the seminal work of Goldwasser, Micali, and Rackoff [23]. Generally speaking, in this line of work, the goal is to design protocols that allow a *prover* to convince a *verifier* that it *knows* a specific message $M$. In the traditional approach, the prover is considered an abstract entity, and the standard definitions do not guarantee that $M$ is actually present in the memory of a single machine in a way that makes it accessible to its owner. For example, it is perfectly fine for the prover to be emulated by a group of machines, where different machines hold parts of the message $M$, or even a share of $M$ using a secret sharing scheme [41]. It is also acceptable for the prover to have $M$ hidden within a trusted execution environment (TEE, see, e.g., [45]).

Perhaps surprisingly, some recent works have considered an intriguing stronger notion, which guarantees that the prover has physical access to data $M$, even in the presence of malicious provers that can either distribute their computation across multiple machines using secure multiparty computation (MPC) protocols (see, e.g., [14]) or use TEEs. This concept has been named *proofs of individual* [19] (or *complete* [29]) *knowledge*. Applications of this stronger notion include deniable messaging, electronic voting, and prevention of account sharing.

On a high level, the solution of [19, 29] is based on requiring the prover to compute a function $f$ (on input $M$) that is designed in such a way that the prover must perform the computation of $f$ *individually*, i.e., they must have free access to $M$ stored in their memory. More concretely, the function $f$ is built using a hash function h, modeled as a random oracle [5], and the protocol requires the prover to compute a massive number of hashes h within some time limit. This limit is set in such a way that it is infeasible to compute all the hashes without having at least some of their inputs in the memory of a single machine. This should hold both against a set of adversaries that "emulate" a single-machine prover by running an MPC protocol, and against those that do it using the TEEs. In the

first case, the assumption is justified by the fact that computing a hash in MPCs is significantly slower than computing it on a single machine. In the second case, the assumption corresponds to the fact that no TEE-equipped processor can compete in the efficiency of massive parallel hash computations with hardware explicitly designed for this purpose. Hence, in the latter case, the honest prover is assumed to use specialized hardware (e.g., ASICs) to compute the hashes. The idea of [29] is to use Bitcoin mining rigs for this purpose.

As we explain below, similar flavors of knowledge assumptions (although without considering attacks that use MPCs or TEEs) have been implicitly used in the past in the line of work on "nonoutsourceable puzzles" initiated by Miller et al. [35], which are *proof-of-work* (PoW) protocols [17] designed to discourage the creation of mining pools in cryptocurrencies. The main idea of [35] is to design a PoW mining protocol that discourages creating *mining pools*. Recall that in a Bitcoin mining pool (see, e.g., [38]), the *pool operator* outsources the mining work to the *pool members* (the "workers"), who are paid for their work. The nonoutsourceable puzzles protocol of [35] is designed in such a way that every pool member can "steal" the correct puzzle solution that they found. This mechanism discourages mining pool creation, as no pool operator can be guaranteed to receive the solutions from its members.

The main idea of the protocol of [35] is as follows. Their puzzle is designed in such a way that, to find a solution, the prover must know a random nonce $r$ (see Fig. 6 in Sec. 4 for details). On the other hand, knowing $r$ suffices to steal the solution from the pool operator. Note that breaking the nonoutsourceability of such a scheme would correspond to computing the solution without having the entire $r$ in the memory of the worker's machine. The subtle point is that, in order to outsource the computation, both the pool operator and the worker may collaborate to prevent the worker from having the entire $r$ in its memory, while still allowing it to perform the mining work. In particular, they may apply arbitrary strategies to split the computation in such a way that stealing the solution is not possible. This is similar to the situation in [19, 29], where, as described above, the malicious prover is interested in not having the entire input $M$ in its memory, while still being able to convince the verifier that it knows $M$.

While in this context MPCs and TEEs are not very relevant (due to the slowdown caused by MPCs and the inefficiency of TEEs for massive parallel hashing), other strategies may be applied to split the knowledge of certain strings between the worker and the operator. Hence, similarly to what is done in [19, 29], the protocol of [35] is very sensitive to the way "knowing" a particular string is modeled. In particular, as we show in Sec. 4.3, the parties can exploit an iterative structure of the hash function to split the knowledge of $r$ between them, in such a way that the worker can still perform the mining work, but cannot steal the solution.

## 1.1 Our contribution

The constructions in [29, 35] rely on hash functions modeled as random oracles [5]. Our key observation is that the modeling used there is problematic if the hash function $\mathsf{H}$ is constructed in an *iterative* way, or in some other way that allows partial pre-computation. In particular, the construction of [35] is not secure when instantiated with several standard hash functions, and the construction of [29] is not as secure as claimed when instantiated with the Bitcoin mining hash function (double SHA-256). More concretely, our contribution is as follows.

**(A)** We present an attack on the protocol of Kelkar et al. [29]. Specifically, we show that their protocol is vulnerable to an attack that exploits the hashing structure used in Bitcoin mining. This attack does not rely on the iterative structure of the hash function, but rather on the fact that the so-called *Bitcoin block header* consists of a field that is a Merkle root of the transactions included in the block, and can be precomputed separately. As above, we prove formally that this attack works. We also implement this attack on existing hardware. Our experiments show that the attack is feasible and relatively cheap, especially when the algorithm of [29] is used as a protection against TEE attacks, and is based on affordable, legacy Bitcoin mining hardware (see Tab. 1). **(B)** We present a real-life attack on the nonoutsourceable puzzles protocol of [35] when instantiated with several popular hash functions (SHA-256 and SHA-512). The attack is based on the iterative structure of the hash function. More concretely, we exploit the fact that standard hash functions are typically constructed using iterative methods such as Merkle-Damgård or sponge paradigms (see Sec. 2.1). A hash function $\mathsf{H}$ constructed this way can be viewed as a mode of operation for the *fixed-length function* $\mathsf{h}$ of fixed input length (called a *compression function* in the case of Merkle-Damgård and a *permutation function* in sponge). This means that if $\mathsf{H}$ is applied to a longer input $M$, then $\mathsf{H}$ can be precomputed on $M$'s prefix in such a way that the result of this precomputation partially hides information about this prefix. We exploit this property to split the knowledge of certain strings between the pool operator and the worker. We formally prove that the attack works. **(C)** We outline provably secure countermeasures to the attack from points **(A)** and **(B)**. The proofs are given in the "individual cryptography" model of [19].

As of now, the protocols that we break remain academic proposals (both published at ACM CCS), and to our knowledge, they have not been deployed in real life. Our results serve as cautionary examples, illustrating the dangers of using the oracle model in a way that is not compatible with typical implementations of hash functions, in the context of modeling individual knowledge. Our work also illustrates the suitability of the *distributed adversarial model* of [19] to study the security of protocols involving multiple mutually distrustful yet cooperating parties.

## 1.2 Related Work

In this section, we describe the related work, focusing first on the nonoutsourceable puzzles protocol and on the proofs of individual knowledge.

*Nonoutsourceable puzzles.* As mentioned above, the notion of nonoutsourceable puzzles was introduced by Miller et al. in [35]. Subsequently, several papers have been published that build on the idea of nonoutsourceable puzzles (e.g. [26, 47, 48]). The construction of [35] was also described in numerous survey papers [3, 42, 46] and books [37, 39], and was taught at a Coursera course

on cryptography.[1] A heuristically secure alternative construction of nonoutsourceable puzzles was proposed in [15]. An attack on the nonoutsourceable puzzles protocol of [35] was presented in [12]. It is, however, based on the assumption that the underlying blockchain supports smart contracts, and requires the pool members to put large deposits aside (that would be confiscated if they steal the solution). In contrast, our attack does not require any smart contracts and directly concerns the primitive of nonoutsourceable puzzles.

*Proofs of individual/complete knowledge.* *Proofs of individual knowledge* were introduced in [19], and independently, under the name of *proofs of complete knowledge*, in [29]. As highlighted above, the goal of this primitive is to allow a prover to convince a verifier that it knows a message $M$ with an additional guarantee that $M$ must be known entirely by a single entity. The main difference between the two papers is that [19] is more focused on the theoretical aspects of the primitive, while [29] is more focused on the practical applications. The advantage of the approach of [19] is that $M$ can be arbitrarily long, while the approach of [29] is limited to short messages. On the other hand—unlike [19]—the approach of [29] is designed to be compatible with the existing hardware for massive hash computations, more concretely, the *ASICs* (Application-Specific Integrated Circuits) that are used in Bitcoin mining. Recently, papers that build on the notion of individual/complete knowledge have been published in [2, 21].

*Other related work.* The individual cryptography model was also used in the context of secret sharing [8, 20] and threshold encryption [13]. Knowledge possession was also studied in the context of proofs of retrievability [9], proofs of replication [22], and proofs of space [18]. These primitives are, however, not directly related to individual knowledge, as they do not prevent the prover from splitting the message between multiple machines (see also [19, 29]). Weaknesses of the random oracle model have been studied before, e.g., in [4, 10, 30, 31]. These works, however, do not consider practical attacks on the protocols for individual (or complete) knowledge. On the theoretical side, a long and vital line of work on *indifferentiability* [32] analyzes random-oracle modeling of hash functions, taking into account their iterative structure. These papers, however, do not consider the individual knowledge model and, hence, are not directly related to our work.

### 1.3 Organization of the paper

The paper is organized as follows. In Sec. 2, we provide the necessary background and introduce some notation. In Sec. 3, we show an attack on the proof of complete knowledge of [29] and discuss possible countermeasures. In Sec. 4, we present our attack on the constructions of nonoutsourceable puzzles and outline countermeasures to it.

## 2 Preliminaries

We use standard cryptographic notation throughout this paper. For completeness, we review it in App. B. Other preliminaries are presented in the following subsections.

[1] A course by Arvind Narayanan, see https://www.coursera.org/learn/cryptocurrency.

### 2.1 Hash functions

As highlighted above, we rely on the iterative structure of modern hash functions (i.e., the Merkle-Damgård [16, 34] or sponge [6] paradigms). In this approach, used in all popular hash functions, including the widely-used SHA-family functions, the message is processed in fixed-size blocks, and the hash function is built by iterating a fixed-length *compression* or *permutation* function. Below, we briefly recall how iterative hash functions are constructed from fixed-length functions. Our presentation captures both the Merkle-Damgård and the sponge paradigms. To cover both paradigms, and to be able to discuss for which concrete hash functions our attacks apply, we need a relatively general definition of an iterative hash function and three parameters (denoted $\alpha$, $\beta$, and $\lambda$). Let $\mathsf{h}\colon \{0,1\}^{\alpha+\beta} \to \{0,1\}^{\alpha}$ be a *fixed-length function* (i.e., a compression [16, 34] or a permutation [6] function). We construct an *iterative* hash function $\mathsf{H}\colon \{0,1\}^{*} \to \{0,1\}^{\lambda}$ from $\mathsf{h}$ as follows. Define recursively a function $\mathsf{G}\colon \{0,1\}^{\alpha} \times \left(\{0,1\}^{\beta}\right)^{*} \to \{0,1\}^{\alpha}$ as follows (below $v \in \{0,1\}^{\alpha}$ and each $m_j \in \{0,1\}^{\beta}$):

- let $\mathsf{G}(v, \epsilon) := v$,
- for $m = (m_1 \,\|\, \cdots \,\|\, m_n)$, let $\mathsf{G}(v, m) := \mathsf{h}(\mathsf{G}(v, m_1 \,\|\, \cdots \,\|\, m_{n-1}) \,\|\, m_n)$.

Each value $\mathsf{G}(v, m_1 \,\|\, \cdots \,\|\, m_i)$ is called the *(intermediate) state* of the computation, and $\alpha$ is called the *state size*. Let $IV \in \{0,1\}^{\alpha}$ be a fixed string called an *initialization vector*. The function $\mathsf{H}$ takes as input a binary string $m \in \{0,1\}^{*}$ and then performs the following steps:

(1) Pad $m$ with a string $\mathsf{padding}_{|m|}$ so that the resulting string $m' = m \,\|\, \mathsf{padding}_{|m|}$ has length that is a multiple of $\beta$. This has to be done in a prefix-free way, e.g., by $\mathsf{padding}_{|m|}$ having the form $10^{*}$ concatenated with a binary representation of the length of $m$ in bits.
(2) Write $m' = (m_1 \,\|\, \cdots \,\|\, m_k)$, where each $m_i$ has length $\beta$.
(3) Output $\mathsf{out}(\mathsf{G}(IV, m_1 \,\|\, \cdots \,\|\, m_k))$, where out is some fixed function that maps $\{0,1\}^{\alpha}$ to $\{0,1\}^{\lambda}$.

Frequently, in hash functions based on the Merkle-Damgård paradigm $\alpha = \lambda$, and out is the identity function, i.e., $\mathsf{out}(x) = x$ for all $x \in \{0,1\}^{\lambda}$. In the case of the more modern sponge constructions, $\alpha$ does not need to be equal to $\lambda$, and out can be a much more complex function. The iterative structure allows one to *precompute* the value of $\mathsf{H}$ on a prefix of a long input $m$. More precisely, consider a message $m$ (of length greater than $\beta$), and write $m = m^0 \,\|\, m^1$, where $m^0$ is of length that is a multiple of $\beta$. Then the value of $\mathsf{H}(m)$ can be computed as $\mathsf{H}(m) = \mathsf{postcomp}_{\mathsf{H}}^{|m|}(\mathsf{precomp}_{\mathsf{H}}(m^0), m^1)$, where

$$\mathsf{precomp}_{\mathsf{H}}(m^0) \;=\; \mathsf{G}(IV, m^0) \tag{1}$$

$$\mathsf{postcomp}_{\mathsf{H}}^{len}(x, m^1) \;=\; \mathsf{out}(\mathsf{G}(x, m^1 \,\|\, \mathsf{padding}_{len})) \tag{2}$$

(in the above, *len* is an integer denoting the length of $m$).

We also rely on Merkle trees. Since these are a standard primitive, we move their description to App. C and only briefly present the notation. $\mathsf{Merkle}_{\mathsf{F}}$ denotes a Merkle hash based on a hash function $\mathsf{F}$. For a tree $T$, $mroot(T)$ denotes its root, and $\mathsf{MProof}(T, i)$ denotes the Merkle proof for the $i$th leaf of $T$. Moreover, $\mathsf{update}(T, i, leaf)$ denotes the Merkle tree obtained from $T$ by replacing its $i$th leaf with *leaf* and recomputing the tree, and $siblings(T, i)$ denotes the

sequence of sibling nodes on the path from the $i$th leaf to the root of $T$ (see App. C for more details).

## 2.2 Distributed adversarial model

Let us now recall the *distributed adversarial model* of [19, 20]. Here, a protocol is attacked by a *distributed* adversary $\mathcal{A} := (\mathcal{A}_1, \ldots, \mathcal{A}_a)$, i.e., a collection of PPT interactive Turing machines called *subadversaries*. Intuitively, the subadversaries represent distinct entities that coordinate an attack, while not necessarily trusting one another; in particular, each may also try to learn the others' secrets for its own advantage. Throughout the protocol execution, the subadversaries may communicate without restriction.

The adversary $\mathcal{A}$ may follow any strategy. In particular, mutual distrust does not prevent them from jointly performing secure multiparty computation (MPC) to pursue a shared objective (namely, breaking the protocol) while keeping their private inputs hidden. Since MPC can securely emulate any polynomial-time functionality, the subadversaries can, in principle, realize any polynomial-time attack strategy via MPC. The central point of [19, 20] is that MPC is typically far more costly than local ("plain") computation. Motivated by this gap, they formalize a model that distinguishes MPC-based work from plain work and use it to design protocols that stay secure against distributed adversaries whose ability to perform MPC is quantitatively limited.

Their constructions rely on a fixed-length function $\mathsf{h}$ (see Sec. 2.1) and model the overhead of secure computation via a *multi-party random oracle*. Concretely, $\mathsf{h}$ is idealized as a random oracle $\Omega_{\mathsf{h}}^{\mathrm{MPC}}$ (implementing a random function $\mathsf{h}$) that supports two types of access: *fast* and *slow* queries. Fast queries are unlimited, whereas slow queries are capped by a parameter $\sigma$ (the *slow query bound*). This reflects the asymmetry between evaluating hashes locally and evaluating them inside MPC.

Fast queries coincide with the standard random-oracle interface: each subadversary can query $\Omega_{\mathsf{h}}^{\mathrm{MPC}}$ on any input of its choice and learns both the full input $x$ and the full output $\mathsf{h}(x)$. Slow queries instead emulate an MPC-style hash evaluation. To issue such a query, each subadversary $\mathcal{A}_i$ sends $(qid, x_i)$ to $\Omega_{\mathsf{h}}^{\mathrm{MPC}}$, where $qid \in \{0,1\}^*$ is a query identifier. Once $\Omega_{\mathsf{h}}^{\mathrm{MPC}}$ has received the messages for the same $qid$ from all subadversaries, it computes $y = \mathsf{h}(x_1 \oplus \cdots \oplus x_a)$ and returns an XOR-sharing of $y$, i.e., it samples $(y_1, \ldots, y_a)$ with $y = y_1 \oplus \cdots \oplus y_a$ and delivers $y_i$ to $\mathcal{A}_i$. For a discussion motivating this interface, we refer the reader to the previous works, particularly Sec. 3.1 of [8].

As in the standard random oracle model, multi-party random oracles admit an *extractability* notion. Let the *global hash transcript* $\mathcal{T}$ be the list of all queries made to $\Omega_{\mathsf{h}}^{\mathrm{MPC}}$. For each subadversary $\mathcal{A}_i$, let its *local hash transcript* $\mathcal{T}_i$ be the list of queries to $\Omega_{\mathsf{h}}^{\mathrm{MPC}}$ made by $\mathcal{A}_i$. Finally, the slow-query restriction is meaningful only when the adversary has limited time to evaluate hashes via MPC. Indeed, if the adversary could spend arbitrary polynomial time *before* the protocol begins, it could precompute an unbounded number of slow queries. To account for this, the execution of the protocols in this model is split into two consecutive epochs: a *precomputation* epoch and an *online* epoch. During the online epoch, at most $\sigma$ slow queries are answered; any further slow queries are answered with $\perp$. During precomputation, the number of slow queries is limited only by the adversary's overall running time.

# 3 Proofs of Complete Knowledge – an attack and a countermeasure

As highlighted in Sec. 1, the aim of proofs of complete knowledge (PoCK) of [29] is to ensure that a prover *actually knows* a witness $w$—as opposed to $w$ being divided between a group of parties, secret-shared, or encumbered in a TEE enclave. This is captured by the notion of *forced revelation*: after a successful interaction with the verifier, the prover should be able to extract the witness $w$ given only their own execution trace. Note that this extraction should be a process that the prover could actually perform, and should *not* be an abstract cryptographic extractor built on thought-experiments, such as forking or rewinding.[2] The authors of [29] focus on achieving *zero-knowledge* PoCK (ZKPoCK), in which the interaction with the verifier does not leak any information about the witness $w$.

## 3.1 The ASIC-ZKPoCK scheme of Kelkar et al.

The key technique used by PoCK to achieve forced revelation is to require that the proving process involve interaction with external hardware, which can be eavesdropped on by the prover. The authors of [29] present two variants of their scheme. In our paper, we are interested in the variant called ASIC-ZKPoCK, which uses an ASIC (Application-Specific Integrated Circuit) as the external hardware. In the ASIC-ZKPoCK scheme of [29], this external hardware is a Bitcoin mining rig which performs a brute-force search to find values whose hash starts with many zeros. The *hash rate* (i.e., the per-second rate at which the hashes are computed) achieved by the mining rigs far exceeds that achieved by standard hardware. The protocol should be designed in such a way that the only feasible way to find a valid proof within some timeframe $\theta$ is to leverage the mining rig's capabilities. Therefore, this computation cannot be performed by an MPC protocol or a TEE enclave.

Since the authors of [29] are interested in constructing *zero-knowledge* PoCKs, they build their construction on top of a *sigma protocol*. Recall that a sigma protocol (see, e.g., [28]) is a public-coin 3-round zero-knowledge protocol between a prover and a verifier. More formally, let $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation. The prover $\mathsf{P}^\Sigma$ in a sigma protocol consists of two algorithms $\mathsf{P}_1^\Sigma$ and $\mathsf{P}_2^\Sigma$. Algorithm $\mathsf{P}_1^\Sigma(x, w) \to (a, st)$: takes as input the *statement* $x$ and the *witness* $w$ (such that $(x, w) \in \mathcal{R}$) and outputs the first message $a$ of the protocol and some state $st$, and algorithm $\mathsf{P}_2^\Sigma(st, c, x, w) \to s$: takes as input the state $st$, a random challenge $c \leftarrow\!\!\$\ C$ (where $C$ is the *challenge space*) from the verifier and outputs the final message $s$.

The verifier $\mathsf{V}^\Sigma(x, (a, c, s))$ takes as input the statement $x$ and the sigma protocol transcript $(a, c, s)$ and outputs true if the transcript is *valid* for the statement $x$ (and false, otherwise). Every sigma protocol satisfies completeness and special honest-verifier zero-knowledge properties (see, e.g., [28] for details). The property of sigma protocols that is particularly important for us is *special soundness*: given two accepting transcripts $(a, c, s)$ and $(a, c', s')$ with the same first message $a$ but different challenges $c \neq c'$, it is

[2] For example, the well-known knowledge extractor for Schnorr's protocol requires "rewinding" of the machine of the prover.

possible to efficiently compute ("extract") the witness $w$ such that $(x, w) \in \mathcal{R}$. The authors of [29] exploit this property to achieve forced revelation. On a high level, their idea is to design a scheme in which the prover P must use the mining rig to compute the sigma protocol responses for the same $a$ and at least two different challenges $c$ and $c'$. Since they assume that the prover P can eavesdrop on all communication with the mining rig, this causes the prover P to learn two accepting transcripts with the same first message $a$ but different challenges $c \neq c'$. Consequently, the prover P can use special soundness to extract the witness $w$.

A classic example of a sigma protocol is Schnorr's identification protocol [40]. In this case, the statement $x$ is a public key $g^w$ for some generator $g$ of a group $\mathcal{G}$ of prime order $q$, and the witness $w$ is the corresponding secret key. The prover's first message $a$ is a commitment of the form $a = g^u$ for some random $u \leftarrow\!\!\$\ \mathbb{Z}_q$, and the prover stores $st = u$. The verifier's challenge $c$ is an integer from $C = \mathbb{Z}_q$, and the prover's final message is $s = u + c \cdot w \mod q$. The verifier checks the validity of the transcript $(a, c, s)$ by verifying that $g^s = a \cdot x^c$.

The construction of [29] works generically for any sigma protocol[3] and is depicted in Fig. 3. We describe it in more detail shortly; for a moment, let us only highlight that the prover P and the verifier V interact in $n$ sequential, timed rounds. In each round $i$, the prover runs $\mathsf{P}_1^\Sigma$ to compute $(a, st)$, sends $a$ to the verifier, and retains the state $st$. The verifier responds with a random nonce $\rho$ and waits for a response $\pi$ (encoding the output of $\mathsf{P}_2^\Sigma$) within a fixed time limit $\theta$. If the prover fails to respond in time, the verifier rejects. Otherwise, the verifier decodes the sigma protocol response $s_i$ from $\pi$ and checks its validity by verifying that $\mathsf{V}^\Sigma(x, (a, c_i, s_i)) = \mathsf{true}$. Finally, the verifier accepts only if at least $y$ out of $n$ responses $\pi$ satisfy the proof-of-work condition, i.e., their hash values are below the required threshold. The requirement that it suffices to have $y$ out of $n$ valid responses comes from the fact that Bitcoin mining is a probabilistic process, and there is no guarantee that a valid proof of work can always be found within time $\theta$.

*3.1.1 Mining rig design.* Let us first recall the relevant elements of the structure of a Bitcoin block header and the mining procedure (these are illustrated in Figs. 1 and 2). The *block header* is a result $(B \| \nu)$ of a concatenation of a *block header prefix* $B \in \{0,1\}^{608}$ and a *nonce* $\nu \in \{0,1\}^{32}$. The block header prefix $B$ contains several fields. The one that is of particular importance for us is a field called the *Merkle root*, denoted $B.mroot$. It is a 256-bit string that is the root of a Merkle tree built over the sequence of hashes of Bitcoin transactions included in the block (see Sec. 2.1 for details on Merkle trees). The Merkle tree is computed using a SHA-256 hash function [44]. SHA-256 (denoted as H) is an iterative, Merkle-Damgård hash function built using a compression function $\mathsf{h}\colon \{0,1\}^{256+512} \to \{0,1\}^{256}$. Also, each transaction (before being included in the Merkle tree) is hashed twice using SHA-256.

Historically, to mine a new block, the miner would take a fixed block header prefix $B$ and search for a nonce value $\nu$ such that the block header $(B \| \nu)$ double-hashes to a value below a specific target. More precisely, the miner searched for a nonce $\nu$ such that $\mathsf{H}(\mathsf{H}(B \| \nu)) < 2^{256-d}$, where $d$ is a *difficulty parameter*. Unfortunately, the nonce field is only 32 bits, providing just $2^{32}$ possible values, which is insufficient for modern mining operations. For this reason, the mining process has evolved to allow modification of other parts of the block header. Technically, this is done by placing another nonce, called the *extranonce*,[4] in the so-called *coinbase transaction cbase*. This transaction (in Bitcoin, used to pay the miner's reward) is the first leaf of the Merkle tree that is used to compute the Merkle root $B.mroot$. To avoid changing the semantics of *cbase*, the extranonce is placed in one of the redundant fields of *cbase*. Let $cbase[\mathsf{enonce} := \xi]$ denote the coinbase transaction with extranonce $\xi$. We will write $cbase[\mathsf{enonce} := \bot]$ to denote *cbase* with the extranonce set to some default value (say: $\mathsf{enonce} := (0, \ldots, 0)$). Bitcoin mining is predominantly implemented in silicon as an ASIC (Application-Specific Integrated Circuit). Therefore, it is useful to identify the data needed by the ASIC to compute a $B$ with a given extranonce $\xi$. Note that $B$ is a function of:

(1) all the fields of the block header prefix $B$ except the Merkle root $B.mroot$, and
(2) the information needed to compute the Merkle root $B.mroot$ with the extranonce $\xi$ placed in the coinbase transaction *cbase*. Note that the Merkle root $B.mroot$ can be computed from: (a) the extranonce $\xi$, (b) the rest of the coinbase transaction $cbase[\mathsf{enonce} := \bot]$, and (c) the siblings on the Merkle branch from the root to the first leaf (i.e. $siblings(T, 1)$, see Sec. 2.1).

All the data above, except the extranonce $\xi$ (i.e. data in Points 1, 2b, and 2c), is sent to the ASIC in a *mining context* $C$. Let $C[\mathsf{enonce} := \xi]$ denote the block header prefix obtained from $C$ by computing the Merkle root from the coinbase transaction $cbase[\mathsf{enonce} := \xi]$ and $siblings(T, 1)$. To summarize: to mine a new block, the miner first constructs a context $C$ and sends it to the ASIC. The ASIC searches for a pair $(\nu, \xi)$ such that the block header $(C[\mathsf{enonce} := \xi], \nu)$ hashes to a value below the target $d$.

*3.1.2 Implementation details of the ASIC-ZKPoCK scheme of [29].* We now describe the ASIC-ZKPoCK scheme of [29] (depicted in detail in Fig. 3). Recall that in this protocol, the prover P interacts with the verifier V to prove in zero-knowledge the knowledge of a witness $w$ for a statement $x$ with respect to a relation $\mathcal{R}$. The prover P has access to an ASIC mining rig that performs Bitcoin mining. As explained above, the protocol is supposed to be designed in such a way that every prover (even a malicious one) has to be able to reconstruct $w$ just by eavesdropping on its own communication with the mining rig. The protocol is built on top of a sigma protocol $\mathsf{P} = (\mathsf{P}^\Sigma, \mathsf{V}^\Sigma)$ for $\mathcal{R}$ with challenge space $C$ (e.g., $C = \mathbb{Z}_q$ in the case of the Schnorr protocol). Let $\mathsf{F}\colon \{0,1\}^* \to C$ be a hash function modeled as a random oracle.

The main idea of [29] is to encode the sigma-protocol challenge-response pair $(c, s)$ into a block header template that the prover passes to the ASIC mining rig. On the technical level, this is done as follows. We extend the concept of the context $C$ used in Bitcoin mining to permit encoding of the sigma-protocol response $s$. This is done similarly to how the extranonce $\xi$ is encoded into $B$, i.e., it is placed in the coinbase transaction, which, as a consequence,

[3] Technically, their ASIC-ZKPoCK requires that the final message $s$ output by $\mathsf{P}_2^\Sigma$ is *computationally unique* given $a$ and $c$. This is, however, irrelevant to our attack.

[4] Technically, the extranonce is broken into *extranonce1* and *extranonce2*, but this is irrelevant to this discussion.

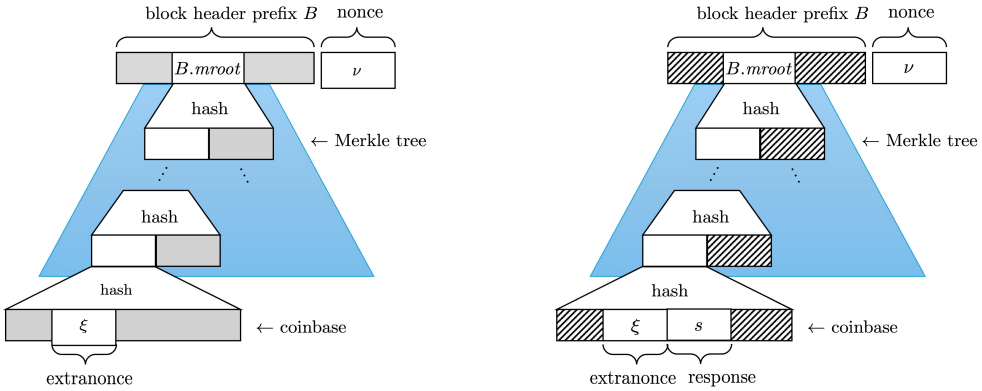



**Figure 1: An illustration of the Bitcoin block header structure. Left: the context $C$ (sent to the ASIC miner) is indicated in gray; right: the augmented context $\widehat{C}$ is shown in striped pattern.**

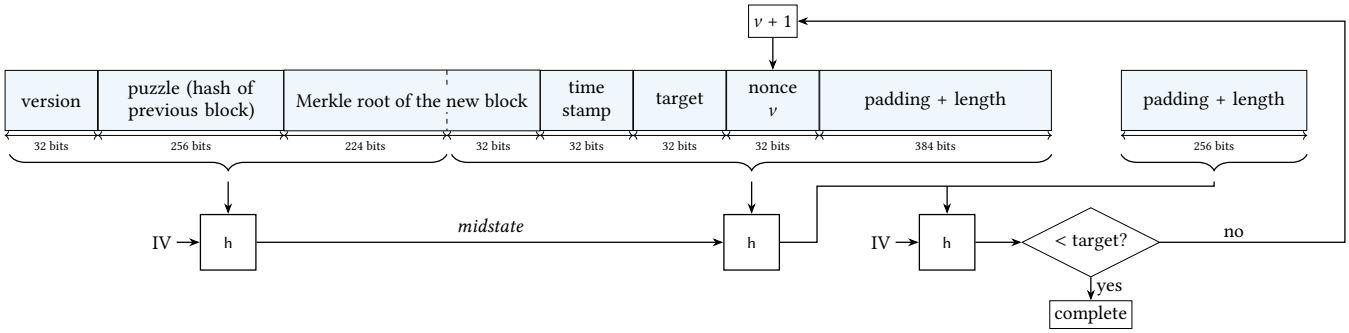



**Figure 2: Overview of the Bitcoin mining process. The block header is 640 bits long. During mining, the first 512 bits are preprocessed with h to produce the *midstate*; the remaining 128 bits are then padded and hashed together with the midstate, and the result is again hashed using h to produce the final hash (notice that the Bitcoin hash uses *double* SHA-256 hashing).**

modifies the Merkle root. Let $\widehat{C}$ denote this augmented context. Let $C_s := \widehat{C}[\texttt{response} := s]$ denote the context $\widehat{C}$ with the sigma-protocol response $s$ encoded in it.

The prover P samples the augmented context $\widehat{C}$ randomly (the protocol works regardless of how $\widehat{C}$ is sampled, as long as it is unlikely to repeat). Then, the prover computes the challenge $c := \mathsf{F}(\rho \| \widehat{C})$ (where $\rho$ is a random nonce sampled by the verifier, needed to avoid precomputation attacks). The prover then computes the sigma-protocol response $s := \mathsf{P}_2^{\Sigma}(st, c, x, w)$ and places it in the augmented context $\widehat{C}$ to obtain context $C_s := \widehat{C}[\texttt{response} := s]$. The prover P then sends $C_s$ to the ASIC mining rig. The mining rig then looks for a nonce pair $(\xi^*, \nu^*) \in \mathcal{S}$ (where $\mathcal{S}$ is a set of *allowed* nonce pairs whose role will be explained later) that solves the proof of work for the context $C_s$. If the ASIC finds such a nonce pair $(\nu, \xi)$, then the prover P sends it together with $C_s$ and $s$ to the verifier. The verifier then checks if all the work performed by the prover is valid. This check, in particular, involves recovering $c$ from $C_s$. Note that for a fixed $a$, the verifier learns only one pair $(c, s)$. Hence, by the ZK property of the sigma protocol, the verifier learns nothing about the witness $w$.

To ensure that the prover can only compute a response using a mining rig, the time the prover is given to find such a nonce pair $(\nu, \xi)$ is limited. This limit is denoted as $\theta$. The set $\mathcal{S}$ of allowed nonce pairs and the difficulty parameter $d$ are chosen in such a way that a valid nonce pair $(\nu, \xi)$ is (1) likely to be found in time $\theta$ when an ASIC is used for the search, but (2) *un*likely to be found for the first context $C_s$ that the prover tries.[5] Therefore, to find a valid proof of work within time $\theta$, the prover P must try many different contexts $C_s$, necessarily resulting in a different challenge $c$ and a different sigma-protocol response $s$. On the other hand, since $a$ is sent to the verifier at the beginning of a round, it must remain the same. Crucially, the authors of [29] assume that the only way the prover can use the ASIC mining rig to find a valid nonce pair $(\nu, \xi)$ is by sending it the context $C_s$ (as we later show in Sec. 3.2, this assumption does not hold in practice; this observation is the basis of our attack). Therefore, by eavesdropping on its own communication with the ASIC mining rig, the prover P learns many different valid sigma-protocol responses $s$ for the same first message $a$ but different challenges $c$. Hence, it can compute the witness $w$ using special soundness. Since, with a small but noticeable probability, the ASIC

[5] More precisely, to guarantee (1) the parameter $d$ has to be calibrated to the hash rate of the ASIC, while to guarantee (2) the set $\mathcal{S}$ has to be relatively small.

mining rig may find a valid nonce pair $(v, \xi)$ for the first context $C_s$ that the rig receives, the protocol is repeated for $n$ sequential rounds to ensure that with high probability at least two different contexts $C_s$ are sent to the ASIC mining rig. To tolerate the fact that the prover may not find a valid nonce pair $(v, \xi)$ within time $\theta$, we only require that $y$ out of $n$ rounds need to be successful.

**ASIC-ZKPoCK scheme of Kelkar et al.**

(1) For each round $i \in \{1, \ldots, n\}$:
  (a) The prover lets $(a, st) \leftarrow \mathsf{P}_1^{\Sigma}(x, w)$ and sends $a$ to V.
  (b) The verifier samples a nonce $\rho$ and sends it to P.
  (c) *Prover* P*:*
    (i) Choose $\widehat{C}$ randomly from some distribution.
    (ii) Let $c := \mathsf{F}(\rho \,\|\, \widehat{C})$.
    (iii) Compute the response $s := \mathsf{P}_2^{\Sigma}(st, c, x, w)$.
    (iv) Send the context $C_s := \widehat{C}[\texttt{response} := s]$ to the ASIC.
  (d) *ASIC:*
    (i) Look for a nonce pair $(\xi^*, v^*) \in \mathcal{S}$ such that $\mathsf{H}(\mathsf{H}(C_s[\texttt{enonce} := \xi^*] \,\|\, v^*)) < 2^{256-d}$.
    (ii) Return $(\xi^*, v^*)$ if found.
  (e) *Prover* P*:*
    (i) If the ASIC returned a nonce pair $(\xi^*, v^*)$, send the *round solution* $(\widehat{C}, s, \xi^*, v^*)$ to the verifier and proceed to the next round.
    (ii) Else:
      (A) If the deadline $\theta$ is approaching, then finish the round without submitting a proof.
      (B) Otherwise, sample a fresh $C_s$ (Step 1c).

(2) *Verifier V:* For each round solution $(\widehat{C}, s, \xi^*, v^*)$ received from the prover in Step 1(e)i, perform the following checks:
  (a) Check if it arrived before the round's time limit $\theta$.
  (b) Check if $(\xi^*, v^*) \in \mathcal{S}$ and if the solution meets difficulty $d$, i.e., if $\mathsf{H}(\mathsf{H}(C_s[\texttt{enonce} := \xi^*] \,\|\, v^*)) < 2^{256-d}$, where $C_s := \widehat{C}[\texttt{response} := s]$.
  (c) Recompute $c := \mathsf{F}(\rho \,\|\, \widehat{C})$ from the received $\widehat{C}$.
  (d) Check if $\mathsf{V}^{\Sigma}(x, (a, c, s)) = \texttt{true}$.

Accept iff *at least* $y$ rounds pass all checks.

**Figure 3: The ASIC-ZKPoCK scheme of [29].**

## 3.2 Our attack

We now show how MPC/TEE-based computation can be used in an adversarial way to break the forced revelation guarantee of the ASIC-ZKPoCK protocol. To this end, we construct a malicious prover $\mathsf{P}^*$ who proves knowledge of a witness $w$ for a statement $x$, even though it does not *individually* (or "completely") know $w$. This could be the case if $w$ is shared by $\mathsf{P}^*$ with other parties using secret sharing, and $\mathsf{P}^*$ only holds a share of $w$. Alternatively, $\mathsf{P}^*$ could hold $w$ hidden in a TEE enclave. In both cases, $\mathsf{P}^*$ can compute functions $f$ (relative to a hash function H) of $w$ using secure multiparty computation (MPC) or TEE-based computation. Following the conventions from *individual cryptography*, any hash evaluation by $f$ will be called *slow*.[6] Of course, we need to assume that if computing $f$ involves hashing, then it cannot be performed at the same speed as the mining rig. In our attack, the number of slow hash evaluations is small.

The high-level idea of our attack (for the detailed description, see Fig. 4) is similar to the attack on the nonoutsourceable puzzle from Sec. 4.3, since it exploits the fact that some values that are hashed together can be computed in two steps, hiding some parts of the input from the entity that sees that intermediate hash value only. As highlighted above, in the security argument of [29], the authors crucially rely on the assumption that the mining rig can only be used in a black-box way to compute the hash of the *entire* mining context $C_s$. In particular, they assume that the communication between the prover and the mining rig involves sending $s$. However, since $s$ is placed in a leaf of a Merkle tree, the prover can do the following: (1) Using slow hash evaluations, compute the Merkle root $z$ of the tree containing $s$, (2) Place $z$ in the block header prefix, resulting in the block header prefix $B$, and (3) Send only $B$ to the ASIC.

Since the extranonce $\xi$ is also placed in the coinbase transaction, the ASIC is not able to plant it there using only the Merkle root $z$. Thus, $\xi$ has to be fixed together with $s$ when computing $z$. Therefore, in our attack, the ASIC only searches for a nonce $v$, which translates to $2^{32}$ possible values. As we discuss in Sec. 3.3, for some ASICs, such reprogramming is feasible in practice. Before we do that, we prove that this attack breaks the security of the ASIC-ZKPoCK protocol of [29], i.e., that the values received by the ASIC do not reveal any information about the witness $w$. To be more formal, we need to assume that the underlying sigma protocol does not permit the reconstruction of the witness $w$ from a single valid transcript $(a, c, s)$. Technically, this does not follow from the zero-knowledge property of the sigma protocol, since it could be the case that one could compute the witness $w$ directly from the statement $x$. Therefore, we restrict our attention to *hard* relations $\mathcal{R}$, defined below.

*Definition 1 (Hard relation).* A binary relation $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ is called *hard* if there exists an instance generator algorithm Gen such that:

- For every security parameter $\kappa$, the pair $(x, w) \leftarrow \mathsf{Gen}(1^\kappa)$ satisfies $(x, w) \in \mathcal{R}$.
- On average, it is computationally infeasible to compute a witness for $x$, that is, for every PPT algorithm $\mathcal{A}$, it holds that

$$\Pr\left[(x, w') \in \mathcal{R} \;\middle|\; \begin{array}{c}(x, w) \leftarrow \mathsf{Gen}(1^\kappa);\\ w' \leftarrow \mathcal{A}(1^\kappa, x)\end{array}\right] \leq \mathsf{negl}(\kappa).$$

An example of a hard relation $\mathcal{R}$ is the discrete logarithm relation used, e.g., in the Schnorr protocol. Because hard relations are defined asymptotically, to instantiate them in practice, one needs to choose a security parameter $\kappa$. Therefore, we formulate the following lemma in an asymptotic way as well (assuming that the ASIC-ZKPoCK protocol is run for security parameter $\kappa$, denoting the hash output length).

[6] More precisely, in [8, 19, 20] the authors model the fact that a small number of hashes can be computed without any machine knowing their inputs (using MPCs or TEEs) by assuming that the adversary has a limited budget of so-called *slow oracle queries* that can be computed on shared inputs.

**Adversarial mining procedure for ASIC-ZKPoCK of Kelkar et al.**

For each round $i \in \{1, \ldots, n\}$:

(1) The prover $\mathsf{P}^*$ samples $(a, st) \leftarrow\!\$\ \mathsf{P}_1^{\Sigma}(x, w)$, sends $a$ to $V$, and receives nonce $\rho$ from $V$.
(2) *Prover $P^*$:*
  (a) Choose augmented context $\widehat{C}$ at random.
  (b) Let $c := \mathsf{F}(\rho \parallel \widehat{C})$.
  (c) Sample extranonce $\xi^*$ at random.
  (d) Execute the following using secret computation with shared (or embedded in TEE) witness $w$:

  **Code executed using secret computation**

  (i) Compute the response $s := \mathsf{P}_2^{\Sigma}(st, c, x, w)$.
  (ii) Using $s$ and $\xi^*$, compute the Merkle root $z$ of the coinbase transaction with response $s$ and extranonce $\xi^*$ placed in it.
  (iii) Output $z$.

  (e) Use $z$ to compute the block header template $B = C_s[\mathsf{enonce} := \xi^*]$ without knowing $s$.
  (f) Send $B$ to the ASIC.
(3) *ASIC:*
  (a) Look for a nonce $v^* \in \{0,1\}^{32}$ such that $(\xi^*, v^*) \in \mathcal{S}$ and $\mathsf{H}(\mathsf{H}(B \parallel v^*)) < 2^{256-d}$. Return $v^*$ if found.
(4) *Prover $\mathsf{P}^*$:*
  (a) If the ASIC returned $v^*$, send the *round solution* $(\widehat{C}, s, \xi^*, v^*)$ to V and proceed to the next round.
  (b) Else:
    (i) If the deadline $\theta$ is approaching, then finish the round without submitting a proof.
    (ii) Otherwise, sample a fresh $C_s$ (Step 2a).

**Figure 4: Adversarial mining attack for the ASIC-ZKPoCK protocol of [29].**

Lemma 2. *Let $1^\kappa$ be the security parameter. Let* Gen *be an instance generator for a hard relation $\mathcal{R}$. Consider an ASIC-ZKPoCK protocol from Fig. 3 for $\mathcal{R}$ built using a sigma protocol $\mathsf{P} = (\mathsf{P}^{\Sigma}, \mathsf{V}^{\Sigma})$ with challenge space $C$ and hash function $\mathsf{F}\colon \{0,1\}^* \to C$ modeled as a random oracle. Let $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\kappa$ be a hash function modeled as a random oracle. Let $(x, w)$ be generated as $(x, w) \leftarrow \mathsf{Gen}(1^\kappa)$.*

*Consider an execution of the ASIC-ZKPoCK protocol in which the prover $\mathsf{P}^*$ follows the adversarial mining procedure from Fig. 4. Let $\{B_j\}_{j=1}^N$ be the sequence of all block header prefixes sent by $\mathsf{P}^*$ to the ASIC during the execution (in Step 2f). Then for any poly-time eavesdropper $\mathcal{E}$ that takes as input $a, x, \rho$, and $\{B_j\}_{j=1}^N$, the probability that $\mathcal{E}$ outputs $w$ is negligible in $\kappa$.*

Note that in the lemma statement, we model the hash function H as a random oracle, even though it works on inputs of arbitrary length, and hence it can be subject to precomputation attacks. This does not affect the relevance of the lemma, since it shows an *attack*, not a security proof.

Proof sketch. Let Bad be the event that either

- $\mathsf{P}^*$ queries the random oracle H on a coinbase transaction containing the witness $w$, or
- $\mathcal{E}$ outputs $w$.

Clearly, before the execution of the protocol starts, Bad does not occur. Moreover, as long as Bad did not occur, the probability that it occurs in the next step of the algorithm is negligible in $\kappa$. This is because (a) the coinbase transaction is hashed using a random oracle, and consequently the $B_j$'s give no information about $w$, and (b) Gen is an instance generator for the hard relation $\mathcal{R}$. Therefore, the probability of Bad is negligible in $\kappa$, and hence the probability that $\mathcal{E}$ outputs $w$ is negligible in $\kappa$. □

## 3.3 Implementation and discussion on the attack parameters

As described above, our attack uses the ASIC miner to search only through the nonce space, while the extranonce needs to be set externally to the miner. This raises the question of whether such an attack is practical, i.e., whether the miner can achieve a high hash rate when searching only through the nonce space, and what bandwidth is needed to feed the miner with new block header prefixes. Tab. 1 summarizes the resources needed to mount the attack on the two mining rigs mentioned in [29], i.e., Antminer S19 Pro Hydro (with a hash rate of 154 TH/s) and Antminer S9 (13 TH/s). We performed experiments to estimate the feasibility of achieving these rates. Our experiments were performed on a Bitmain BM1397 ASIC chip, which requires the block header prefix to be provided externally. To obtain this chip, we purchased a Bitaxe Max miner (that contains it) for approximately USD 70. According to our benchmarks, the chip achieves a hash rate of around 225 GH/s.[7] This hash rate includes the time needed to transfer the prefix to the chip.[8]

Tab. 1 includes information on the number and costs of BM1397 chips required to match the hash rates of the two mining rigs. The practicality of our attack depends on whether we use ZKPoCK as a countermeasure against attacks by TEE or by MPC.

*3.3.1 TEE attacks.* When ZKPoCK is used as a countermeasure against TEE-based attacks (which is the primary focus of [29]), the eavesdropper $\mathcal{E}$ is assumed to be able to run on a powerful CPU within a TEE enclave. The 225 GH/s hash rate achieved by our ASIC implementation is around 3,350 times faster than the hash rate of a state-of-the-art CPU estimated in [29, p. 11] to be around $2^{26}$ H/s. Since the computation of a new block header prefix requires a small number of hash computations, a CPU can compute new prefixes fast enough to keep up with the miner's hash rate, which is true even when the CPU computation is done within SGX. This refutes the assumption made in [29, p. 16] that there is no "practical resource" that prevents eavesdropping by $\mathcal{E}$ and works significantly faster than a CPU. In more concrete terms, since the attack can be parallelized, it translates to an attack with 685 BM1397 chips (for

[7] The ∼ 30% slowdown compared to the advertised 330 GH/s hash rate of Bitaxe Max is because in our attack, we cannot use the AsicBoost technology [27].
[8] On the technical level, we do not actually transfer the block header prefix to the chip; instead, we send to it a midstate in the SHA-256 compression function. This does not affect the logic of our attack.

| | (a) hash-rate | (b) time in microseconds to replace the block header prefix | (c) number of BM1397 chips needed for the attack | (d) estimated cost of the attack in USD (USD 70 × the number of BM1397 chips) |
|---|---|---|---|---|
| Antminer S19 Pro Hydro | 154 TH/s | 28 | 685 | 47,950 |
| Antminer S9 | 13 TH/s | 330 | 58 | 4,060 |

**Table 1: Estimates for the practicality of our attack on ASIC-ZKPoCK executed on Antminer S19 Pro Hydro (mentioned in App. C.3 of [29]) and on Antminer S9 (used for the experiments in [29]). The table contains information about (a) the hash rate of the given miner, (b) the time needed to replace the block header prefix to match the hash rate, (c) the number of BM1397 chips needed to match the hash rate, and (d) the cost needed to mount our attack using these rigs. The estimates are based on the data from the top table, and the cost of a single BM1397 chip is 70 USD (the approximate price we paid). As explained in Sec. 3.3, these estimates rely on our experiments that demonstrate that the BM1397 chip achieved a hash rate of 225 GH/s (see also Footnote 7).**

Antminer S19 Pro Hydro) or 58 BM1397 chips (for Antminer S9). The cost of our attack is below USD 50,000 and below USD 5,000, respectively (see Tab. 1). This is a very conservative estimate, as a real-life attacker could obtain the chips at a lower price when bought in bulk, or use more efficient mining chips.

*3.3.2 MPC attacks.* Launching an attack using MPC is more challenging. As shown in Tab. 1, even for Antminer S9, one would need to recompute around three block header prefixes every millisecond to keep up with the miner's hash rate. Since each such computation requires three computations of the SHA-256 compression function, this would lead to high costs (also in terms of communication) even in the 2-party passive security setting. Still, given the current state of the art for MPC implementations of SHA-256, it does not seem entirely out of reach in the future. For Antminer S19 Pro Hydro, the requirements are even more challenging and probably out of reach for a long time.

## 3.4 Our countermeasure

Depending on the use case, there are several countermeasures that enable implementing a protocol that is immune to the attacks described in Sec. 3.2. One option is to use the zero-knowledge proof of individual knowledge (zkPIK) protocol from [19]. However, the aim of the PoCK construction [29] is to protect against malicious use of TEEs (such as SGX) using *dedicated* hardware. In contrast, the construction of [19] is not compatible with the hardware available on the market.

Perhaps the most straightforward countermeasure is to modify the ZKPoCK protocol [29] to include the sigma-response $s$ in a different part of the block header. Since Bitcoin mining relies on double SHA-256 hashing of the block header, we need to be careful to avoid precomputation attacks (see Sec. 4 below). To be more specific, consider the structure of the block header depicted in Fig. 2. If, e.g., the response $s$ is placed at the very beginning of the block header, then the cheating prover could precompute the midstate (i.e., the output of the first hash function in Fig. 2) and use it to mine by changing only the nonce, similarly to the attack described in Sec. 3.2.

One possibility would be to include $s$ in the timestamp field of the block header. This, however, is problematic, as 32 bits are not sufficient to include the entire response $s$ (which is 256 bits long). Furthermore, the naive solution of just splitting $s$ into 8 chunks of 32 bits each and proving their knowledge separately does not work because it does not rule out that each chunk was handled by a different subadversary.

Instead, our solution, which we call *Midstate-ZKPoCK*, places $s$ in the midstate, as it is exactly 256 bits long. As mentioned in Footnote 8, some mining ASICs (such as the BM1397) allow the midstate to be provided externally to the miner. Note that this limits the nonce space to 32 bits only, which can be worked around in the following way. Notice that the miner only needs the last 32 bits of the Merkle root used in the mining process. However, now that the midstate has been overwritten with $s$, the last 32 bits of the Merkle root do not convey any useful information and can be abused to serve as an extranonce, leading to a total of 64 bits of extranonce space. Similarly, since we do not need to produce a block that would *actually* be compatible with the Bitcoin network, we can also use the timestamp field as an extranonce, raising the total extranonce space to 96 bits.

*3.4.1 Countermeasure details.* We now provide more details on our countermeasure against the attacks from Sec. 3.2. We call our new construction *Midstate-ZKPoCK*. Formally, it replaces the notion of *context* with a much smaller *puzzle instance*. As described in Sec. 3.1.1, the original design involves sending the context (i.e., the block header together with the Merkle tree data). While our attacks exploited the design of the BM1397 miner, its design also allows us to provide a countermeasure. The main idea is to overwrite the midstate with the sigma protocol response $s$. To this end, the target is set appropriately according to the difficulty; the last 32 bits of the Merkle root and the timestamp are set to zero.

We argue that this design thwarts the attacks from Sec. 3.2 *regardless* of the miner hardware used, as long as the hash function is used in a black-box way (as necessitated by the Random Oracle Model). Indeed, even though the mining procedure continues to involve Merkle-Damgård hashing, the data passed to the ASIC is

the *only* source of entropy for the mining process, and hence the slow queries are useless.

To provide a formal model, in place of the forced revelation property of [29], we require that there exists an extractor Ext that, given access to the *local* hash transcript (cf. Sec. 2.2) of a subadversary $\mathcal{A}_i$ for some value of $i$, is able to extract the response witness. This extractor is necessarily *straight-line*. To see why this model excludes the original vulnerable scheme, notice that all the hash queries depicted in Fig. 1 would be present in the *global* hash transcript $\mathcal{T}$. However, as our attacks in Sec. 3.2 show, the hash queries corresponding to the computation of the midstates (constituting less than a $1/2^{31}$ fraction of the queries) would not be present in any *local* transcript, given that they would be computed jointly using slow queries.

In contrast, in our Midstate-PoCK scheme, the global hash transcript contains queries of two kinds: namely, corresponding to the inner and the outer hash of the double-SHA-256. The outer queries take the output of h and (fixed a priori) padding, and give no information to any party who computes them. This implies that the only queries that could be worth computing as slow queries are the inner hash queries, which constitute 50% of all hash queries and are totally symmetric.

Following the notation of [29], let $Q_{\mathsf{CPU}}$ be a bound on the hash rate that can be achieved by a CPU, and let $Q_{\mathsf{ASIC}}$ be the ASIC hash rate. In each round of the protocol, the adversary can make at most $\sigma = \theta \cdot Q_{\mathsf{CPU}}$ slow hash queries and at most $\phi = \theta \cdot mQ_{\mathsf{ASIC}}$ fast hash queries, where $m = \Theta(1)$ is the number of CPUs that the adversary can use. Let $\beta = 2^{32}$ be the size of the nonce space.

Lemma 3. *Suppose that $Q_{\mathsf{CPU}} \ll Q_{\mathsf{ASIC}}$. For every value of $\theta \gg 0$ there exists a choice of parameters $y$ and $n$ such that the Midstate-ZKPoCK protocol from Fig. 5 satisfies the following properties:*

- ***Completeness:*** *An honest prover with a fast query budget $\phi$ convinces the verifier with overwhelming probability.*
- ***Individual knowledge:*** *Let $(x, w) \in \mathcal{R}$ be an instance of a hard relation. Suppose that a distributed adversary $\mathsf{P}$ that convinces the verifier $\mathsf{V}$ to accept with probability at least $\epsilon$. Then, there exists an extractor $\mathsf{Ext}$ such that, for some subadversary $\mathcal{A}_i$ with its local hash transcript $\mathcal{T}_i$, the output $\mathsf{Ext}(x, \mathcal{T}_i)$ equals $w$ with probability at least $\epsilon - \mathsf{negl}(\kappa)$.*
- ***Zero-knowledge.*** *The verifier learns no information about the witness $w$ beyond the fact that the prover* individually *knows $w$ such that $(x, w) \in \mathcal{R}$. This property is implied by simulatability of the underlying transcript of the communication between the prover and the verifier.*

Proof sketch. The proof is almost identical to that of [29], and the crucial difference is in the *model*. Therefore, we only provide a sketch of the proof and refer the reader to [29] for the details.

*Completeness.* Using [29, Fact 6.1, Lemma 6.2], we have that as long as $y < np_{\mathsf{ASIC}}$ for $p_{\mathsf{ASIC}} = (1 - (1 - 1/2^d)^\phi)$, the honest prover manages to find a valid proof in at least $y$ instances with overwhelming probability, as long as $n = \Theta(\lambda)$ and $y/n = \Theta(1)$.

*Individual knowledge.* We analyze the probability that the adversary manages to solve the proof of work using at most $\sigma$ slow queries and at most $\beta$ fast queries in a single round. This happens with probability $p_{\mathcal{A}} = 1 - (1 - 1/2^d)^{\beta+\sigma}$, and as long as $y > np_{\mathcal{A}}$, a malicious prover cannot convince the verifier to accept.

*Parameter regime.* Combining the two, we obtain that it suffices to choose the ratio $y/n$ such that $p_{\mathcal{A}} < y/n < p_{\mathsf{ASIC}}$. Expanding, we obtain that:

$$1 - (1 - 1/2^d)^{\beta+\sigma} < y/n < 1 - (1 - 1/2^d)^{\phi},$$

which is easily satisfiable given that $\beta \ll \sigma \ll \phi$.

*Zero-knowledge.* The simulatability of the transcript of the communication between the prover and the verifier is achieved by leveraging a simulator that simulates the sigma-protocol transcript and then programs the random oracle on the challenge. □

# 4 Nonoutsourceable puzzles – attacks and countermeasures

We start this section by recalling the definition and construction of nonoutsourceable puzzles from [35]. Next, we present our attacks on the construction of [35] and propose our countermeasures. As explained in Sec. 1.2, the goal of nonoutsourceable puzzles is to prevent the pool operator from outsourcing mining work to the pool members by ensuring that pool members can "steal" the solution to any puzzle they have solved. We now describe this notion in more detail, using the framework from [35], slightly adapted to our purposes. The puzzle systems are parametrized by a hash function, and the bulk of work in solving the puzzle consists of computing this hash function on different inputs (this work is called *mining*, and the party performing it is called a *miner*). To stay consistent with the rest of our paper, we model the *work time* in terms of the number of calls to the (underlying) fixed-length function h modeled as an oracle $\Omega_{\mathsf{h}}$.[9] Note that, compared to Sec. 3, we drop the MPC label in the $\Omega_{\mathsf{h}}^{\mathsf{MPC}}$ symbol. This is because considering MPC attacks is not relevant for nonoutsourceable puzzles since the worker can freely share any secret information it possesses and ask the pool to perform computation over it. The model from Sec. 2.2 still applies, except that we consider only the *fast* queries to the oracle. We will often use the symbol H, which, as described in Sec. 2.1, denotes the hash function constructed from the fixed-length function h.

## 4.1 The definition of Miller et al.

Miller et al. [35] start with the following definition (see Def. 1 in [35]). A *scratch-off puzzle* consists of three algorithms: $\mathcal{G}$, Work, and Verify. Algorithm $\mathcal{G}$, on input a security parameter $1^\lambda$, generates a *puzzle instance* $puz \in \{0,1\}^*$ (or simply "puzzle") of length $\lambda$. This is an unpredictable random string, e.g., the hash of the previous block in a blockchain, and $\mathcal{G}$ is an abstraction of a puzzle generation process. Algorithm Work takes a puzzle instance $puz$, a payload $m \in \{0,1\}^*$, a parameter $t$ that upper bounds the algorithm work time, and outputs a *ticket* if the worker was able to solve the puzzle using $t$ calls to the oracle $\Omega_{\mathsf{h}}$ (or $\perp$, otherwise). The probability that $\mathsf{Work}(puz, m, t) \neq \perp$ depends on a difficulty parameter $d$ and the work time parameter $t$. Finally, algorithm Verify takes as input a puzzle instance $puz$, a payload $m$, and a *ticket*, and outputs 1 if the *ticket* is *valid* for the puzzle instance and payload, or 0 otherwise.

[9] Miller et al. model it in terms of abstract "units of time", and assume that a single "puzzle scratch" takes a fixed amount of time.

(1) For each round $i \in \{1, \ldots, n\}$:
  (a) The prover samples $(a, st) \leftarrow\!\$\ \mathsf{P}_1^{\Sigma}(x, w)$ and sends $a$ to V.
  (b) The verifier samples a nonce $\rho$ and sends it to P.
  (c) *Prover P:* For seed = 0, 1, 2, ... :
    (i) Let $c := \mathsf{F}(\rho \,\|\, \text{seed})$.
    (ii) Compute the sigma-response $s := \mathsf{P}_2^{\Sigma}(st, c, x, w)$.
    (iii) Send to the ASIC the puzzle instance $(s, \text{target})$.
  (d) *ASIC:*
    (i) Look for a nonce $v^*$ and extranonce $v'$ such that $\mathsf{H}(\text{postcomp}_{\mathsf{H}}^{640}(s, v' \,\|\, \text{target} \,\|\, v^*)) < 2^{256-d}$. Return $(v^*, v')$ if found; otherwise return "no hit".
  (e) *Prover P:*
    (i) If the ASIC returned a hit $(v^*, v')$, send the *round solution* $(s, \text{seed}, v^*, v')$ to the verifier and proceed to the next round.
    (ii) Else:
      (A) If the deadline $\theta$ is approaching, then finish the round without submitting a proof.
      (B) Otherwise, try another seed (Step 1c).
(2) *Verifier V:* For each round solution $sol = (s, \text{seed}, v^*, v')$ received from the prover in Step 1(e)i, perform the following checks:
  (a) Check if it arrived before the round's time limit $\theta$.
  (b) Check if the solution meets difficulty $d$, i.e., if $\mathsf{H}(\text{postcomp}_{\mathsf{H}}^{640}(s, v' \,\|\, \text{target} \,\|\, v^*)) < 2^{256-d}$.
  (c) Recompute $c := \mathsf{F}(\rho \,\|\, \text{seed})$.
  (d) Check if $\mathsf{V}^{\Sigma}(x, (a, c, s)) = \texttt{true}$.

Accept iff *at least* $y$ rounds pass all checks.

**Figure 5: The Midstate-ZKPoCK scheme.**

We require that a scratch-off puzzle satisfy the following properties. First, it must be *correct*, meaning that for any puzzle instance *puz*, payload $m$, and *ticket* $:= \mathsf{Work}(puz, m, t)$ if *ticket* $\neq \perp$, then $\mathsf{Verify}(puz, m, ticket) = 1$. Second, it must be $\zeta$-*feasible*, meaning that algorithm Work with work time at most $t$ will output a ticket with probability at least $\zeta$ (where $\zeta \in [0, 1]$ is a function of $d$ and $t$). Third, it has to be $\mu$-*non-transferable*[10] (for a constant $\mu \in [0, 1]$), meaning that, for a random puzzle *puz*, any adversary with work time at most $\mu \cdot t$ cannot output a valid $(m, ticket)$ pair with probability greater than $\zeta(d, t) + \mathsf{negl}(\lambda)$, even if the adversary can see a polynomial number of valid tickets on messages $m_1, \ldots, m_n$ of its choice (where each $m_i \neq m$). This requirement is needed to prevent stealing the tickets once they are broadcast to the network (see Def. 2 in [35] for the formal definition of non-transferability).

To define nonoutsourceability, Miller et al. [35] introduce the notion of an "outsourcing protocol". Let $t_{\mathsf{W}}$ and $t_{\mathsf{Op}}$ be the number of oracle calls to $\Omega_{\mathsf{h}}$ that the worker and the pool operator can make, respectively. Moreover, let $t_{\mathsf{E}}$ be an "effective time" parameter whose role is explained in a moment. Formally, a $(t_{\mathsf{W}}, t_{\mathsf{Op}}, t_{\mathsf{E}})$-*outsourcing protocol* for a scratch-off puzzle $(\mathcal{G}, \mathsf{Work}, \mathsf{Verify})$ is a pair of machines (W, Op), where W is a *worker*, and Op is a *pool operator* such that the following holds. Consider a game in which the pool operator Op is given a puzzle instance $puz \leftarrow\!\$\ \mathcal{G}(1^{\lambda})$, and the worker W is given no input. The parties interact, and at the end, the pool operator outputs a *ticket* and a payload $m$ (or $\perp$). It is required that the *success probability*, i.e., the probability that the pool operator outputs a valid ticket given a puzzle instance *puz*, is at least $\zeta(d, t_{\mathsf{E}})$ (where $\zeta$ is the feasibility parameter of the scratch-off puzzle, see above). Informally, the parameter $t_{\mathsf{E}}$ captures the effective work time by the protocol participants (i.e., the amount of work time that a "solo miner" would need to work to achieve the same success probability as the outsourcing protocol). To ensure that the protocol is useful in practice, $t_{\mathsf{E}}$ should be much larger than the amount of oracle calls made by the pool operator $t_{\mathsf{Op}}$. Of course, due to the $\mu$-non-transferability requirement, $t_{\mathsf{W}} + t_{\mathsf{Op}}$ must be at least $\mu \cdot t_{\mathsf{E}}$.

A scratch-off puzzle $(\mathcal{G}, \mathsf{Work}, \mathsf{Verify})$ is nonoutsourceable if there exists a malicious worker W* that can steal the solution to the puzzle. Formally, Miller et al. [35] define the nonoutsourceability of a scratch-off puzzle as follows. A scratch-off puzzle $(\mathcal{G}, \mathsf{Work}, \mathsf{Verify})$ is $(t_{\mathsf{W}}, t_{\mathsf{Op}}, t_{\mathsf{E}}, \pi)$-*nonoutsourceable*[11] if for every outsourcing protocol (W, Op) with parameters $t_{\mathsf{E}}, t_{\mathsf{Op}}$ and $t_{\mathsf{W}}$, there exists a malicious worker W*, working in time $t_{\mathsf{W}}$, that can output a valid ticket for a random message $m^*$ with probability at least $\pi \cdot \zeta(d, t_{\mathsf{E}})$ (after interacting with Op on random puzzle instance $puz^* \leftarrow\!\$\ \mathcal{G}(1^{\lambda})$). To prevent the pool operator from distinguishing between the honest and malicious worker, we require that the execution of the protocol (W*, Op) is indistinguishable from the execution of (W, Op) (from the point of view of Op). More precisely, for every $1^{\lambda}$, let $\mathsf{view}^*(1^{\lambda})$ denote the view of the pool operator Op in the execution of the protocol (W*, Op), and let $\mathsf{view}(1^{\lambda})$ denote the view of the pool operator in the execution of (W, Op). Then, these two views must be computationally indistinguishable, i.e., $\mathsf{view}^* \overset{\mathsf{c}}{\approx} \mathsf{view}$.

## 4.2 The scheme of Miller et al.

The scheme of Miller et al. [35] is depicted in Fig. 6. It is copied mainly from [36], with only minor notation changes. We denote the number of scratch attempts by $\widehat{t}$, instead of $t$ as in [36]. This

[10] The authors of [35] also require that Work can be parallelized. We skip this requirement, as it is not relevant to our paper. They also state another requirement called "incompressibility" that is less general than non-transferability. For simplicity, we merge both requirements into a single one.

[11] The definition in [35] has two additional parameters: $\alpha$ and $p_S$. We omit them, since they are not relevant to our paper.

is done to avoid confusion with the work time parameter $t$ used in our definition of outsourcing protocols.[12] It is chosen to be the maximal number such that the total number of hash function calls made by Work is at most $t$. The relation between $\widehat{t}$ and $t$ is discussed in Sec. 4.4. The scheme in Fig. 6 uses additional parameters $L, d, q$, and $q'$ whose roles are as follows. The parameter $L$ is the number of leaves of a Merkle tree built by the Work algorithm, the parameter $d$ sets the puzzle difficulty, and the parameters $q$ and $q'$ are the numbers of leaves selected during computation of the Work algorithm in different steps of the procedure.

The algorithm Work consists of three main phases. In the first phase ("Construct Merkle Tree", see Step 1), the worker sets up a Merkle tree with a root label *digest* from a list of leaves with randomly-selected values in the leaves. In the second ("Scratch") phase (see Step 2), the worker essentially runs mining on the leaves of the Merkle tree to obtain a PoW solution. The mining is done by randomly trying different values $r \leftarrow\!\!\$\ \{0,1\}^\lambda$. The PoW solution is bound to the root of the Merkle tree (*digest*), and the solution ticket includes a random value confirming the validity of the solution. This phase uses an Encode function that encodes a set of Merkle branches into a single binary string. The function Encode of [36] simply represents the set of branches as a concatenation of the branches (the concrete encoding of branches is not specified, and is irrelevant for our attack). In the third ("Sign payload") phase (see Step 3), the worker signs an arbitrary message which is again bound to the *digest*. The ticket for the whole procedure includes both the PoW solution and the signature of the message.

The main idea of Miller et al. [35] is that, in the scheme, a valid ticket contains the PoW solution as well as the signature of the message, and both of these are bound to the same Merkle tree root *digest*. As the authors claim, it is infeasible to find (with high probability) a valid PoW solution without knowing the majority of the leaves of the Merkle tree and the random value $r$ confirming the validity of the solution. At the same time, having the solution in hand also implies the possibility of signing an arbitrary message. Hence, the scheme seemingly is nonoutsourceable. The authors of [35] also provide a security proof for the scheme in the full version of their work [36]. Crucially, their proof assumes that the hash function H is modeled as a random oracle, even when applied to long messages. This appears in their proof in Definition 6 in [36], where the authors define "good scratch attempt" by referring to random-oracle calls that take the entire string as input. The rest of their proof relies on this definition, and hence on this assumption. In particular, it is implicitly assumed that if the oracle computed H on some input $x$, then either the worker or the operator must "know" $x$ entirely. As highlighted before, this assumption is not satisfied in practice, as hash functions are typically constructed using iterative methods (see Sec. 2.1) such as Merkle-Damgård [16, 34] or sponge [6] paradigms. This key observation is the main starting point of our attack that we present in the next section.

## 4.3 Our attack

We are now ready to present our attack on the nonoutsourceable puzzles of [35]. Following [35], we assume that the puzzle length

[12] In [36] the "unit of work" is the scratch attempt, while in our paper it is the hash query.

**The original nonoutsourceable scheme of [35]**

- Work$(puz, m, t)$:
  (1) **Construct Merkle tree.** Sample $L$ random strings $leaf_1, \ldots, leaf_L \leftarrow \{0,1\}^\lambda$, and then construct a Merkle tree $T := \mathsf{Merkle}_{\mathsf{H}}(leaf_1, \ldots, leaf_L)$ over the $leaf_i$'s using H as the hash function. Let $digest := mroot(T)$ denote the value in the root of this tree.
  (2) **Scratch.** Repeat the following *scratch attempts* as long as the budget for hash queries allows for it, i.e., so that the total number of queries to H is at most $t$ minus the number of queries needed in Steps 1 and 3 (denote the number of scratch attempts by $\widehat{t}$):[a]
    (a) Draw a random nonce, $r \leftarrow \{0,1\}^\lambda$.
    (b) Compute $\eta := \mathsf{H}(puz \| r \| digest)$, and use the value $\eta$ to select $q$ distinct leaves from the tree.
    (c) Let $B_1, B_2, \ldots, B_q$ denote the Merkle proofs (see Sec. 2.1) for the selected leaves.
    (d) Compute $\sigma_\eta := \mathsf{Encode}(B_1, \ldots, B_q)$ where $B_1, \ldots, B_q$ are in the sorted order.
    (e) If $\mathsf{H}(puz \parallel r \parallel \sigma_\eta) < 2^{\lambda-d}$ then record the solution pair $(r, \sigma_\eta)$ and goto "Sign payload".
  (3) **Sign payload.** If no solution was found, return $\perp$. Otherwise, sign the payload $m$ as follows:
    (a) Compute $\eta' := \mathsf{H}(puz \parallel m \parallel digest)$, and use the value $\eta'$ to select a set of $4q'$ distinct leaves from the tree such that these leaves are not contained in $\sigma_\eta$. From these, choose an arbitrary subset of $q'$ distinct leaves. Collect the corresponding branches for these $q'$ leaves, denoted $B_1, B_2, \ldots, B_{q'}$.
    (b) Let $\sigma'_\eta := \mathsf{Encode}(B_1, \ldots, B_{q'})$ in sorted order of $i$.
  (4) Return ticket $:= (digest, r, \sigma_\eta, \sigma'_\eta)$.
- Verify$(puz, m, ticket)$ :
  (1) Parse ticket $:= (digest, r, \sigma_\eta, \sigma'_\eta)$.
  (2) Compute $\eta := \mathsf{H}(puz \parallel r \parallel digest)$ and $\eta' := \mathsf{H}(puz \parallel m \parallel digest)$.
  (3) Verify that $\sigma_\eta$ and $\sigma'_\eta$ contain leaves selected by $\eta$ and $\eta'$ respectively.
  (4) Verify that $\sigma_\eta$ and $\sigma'_\eta$ contain valid Merkle paths with respect to *digest*.
  (5) Verify that $\mathsf{H}(puz \parallel r \parallel \sigma_\eta) < 2^{\lambda-d}$.

[a] See Sec. 4.4 for an estimation of $\widehat{t}$ in terms of $t$.

**Figure 6: The original scheme of [35] from the extended version of their paper [36].**

equals $\lambda$. A detailed description of $(\mathsf{W}, \mathsf{Op})$ appears in Fig. 7. Let us start with its informal description. The original construction of [35] expects the worker W to mine over the nonce $r$. However, nothing prevents the parties from making $r$ fixed, and letting the worker mine by sampling fresh leaves in the Merkle tree. Of course, each change of the leaves results in a fresh value of *digest*, and hence the value of $\mathsf{H}(puz \parallel r \parallel \sigma_\eta)$ changes as well, so the probability of finding a valid solution remains the same in each scratch attempt. Crucially, for a large class of parameters (see Lem. 4), the worker

can perform this mining without knowing the full nonce $r$. To see why, notice that the Work algorithm computes hashes involving $r$ only in the form $\mathsf{H}(puz \parallel r \parallel _)$, where $puz$ is the puzzle (see Steps (2b) and (2e)). The iterative construction of the hash function allows the operator Op in our attack to precompute the part depending on $r$ with only a single query, so that the worker W can postcompute the rest of the hash. Since the knowledge of $r$ is needed to produce the ticket, this prevents the worker from stealing it. Hence, the mining can be outsourced. For the detailed argument, see the proof of Lem. 4 (it appears in Sec. 4.4).

Lemma 4. *Assume that the hash function* $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ *is constructed iteratively using a fixed-length function* $\mathsf{h}\colon \{0,1\}^{\lambda+\beta} \to \{0,1\}^\lambda$ *that is one-way. Consider the scheme from Fig. 6 and assume that (A1)* $\lambda \le \beta$ *(A2)* $2\lambda \ge \beta$*, and (A3)* $2^{\lambda-\beta} = \mathsf{negl}(\lambda)$*, where* $\beta$ *is the block size of the hash function and* $\lambda$ *denotes a security parameter. Then, for every* $t$*, there exists a* $(t_{\mathsf{W}}, t_{\mathsf{Op}}, t)$*-outsourcing protocol* $(\mathsf{W}, \mathsf{Op})$ *with* $t_{\mathsf{Op}} = 1$ *and* $t_{\mathsf{W}} = t + \widehat{t} \cdot \log_2 L$*, where* $\widehat{t}$ *is the number of scratch attempts in Step 2 of the algorithm* Work *(see also Sec. 4.4 for an estimation of* $\widehat{t}$ *as a function of* $t$*).*

## 4.4 Proof of Lem. 4 (sketch)

An informal description of the outsourcing protocol appears at the beginning of Sec. 4.3. Going into more detail, the outsourcing protocol $(\mathsf{W}, \mathsf{Op})$ works as follows (see also Fig. 6). The operator samples the complete nonce and splits $r$ into two parts: $r_{\mathsf{Op}}$ and $r_{\mathsf{W}}$. The operator then precomputes $x := \mathsf{precomp}_{\mathsf{H}}(puz \parallel r_{\mathsf{Op}})$. Note that, due to Assumption A2, the length of $puz \parallel r$ is at least as large as the block length $\beta$, so this precomputation is possible. The operator then sends the precomputed value $x$ and the worker's portion $r_{\mathsf{W}}$ to the worker W. Crucially, the precomputation $x$ hides the operator's secret $r_{\mathsf{Op}}$ while still enabling the worker to evaluate hash functions of the form $\mathsf{H}(puz \parallel r \parallel \cdot)$ using postcomputation. Hence, in the "Sign payload" phase (Step 3), the worker cannot compute the ticket, as it does not know the entire $r$ (cf. Step 3). Of course, this only works if the "$r_{\mathsf{Op}}$" part of the nonce is long enough that, if chosen randomly, it is infeasible for the worker to guess. Let $\lambda_{\mathsf{W}} := |r_{\mathsf{W}}|$ and $\lambda_{\mathsf{Op}} := |r_{\mathsf{Op}}|$. We now have the following:

$$\begin{aligned} \lambda_{\mathsf{Op}} &= \lambda_{\mathsf{Op}} + \lambda_{\mathsf{W}} - \lambda_{\mathsf{W}} && (3)\\ &= \lambda_{\mathsf{Op}} + \lambda_{\mathsf{W}} - \lambda_{\mathsf{W}} - (\lambda + \lambda_{\mathsf{Op}}) \pmod{\beta} && (4)\\ &= -\lambda \pmod{\beta}, && (5) \end{aligned}$$

Here, in Eqs. (3) and (5), we applied simple addition and subtraction, and in Eq. (4) we used the fact that $\lambda + \lambda_{\mathsf{Op}}$ is a multiple of $\beta$. Now, by Assumption A1, Eq. (5) implies that $\lambda_{\mathsf{Op}} = \beta - \lambda$. Finally, by Assumption A3, we have that $2^{-\lambda_{\mathsf{Op}}} = 2^{\lambda-\beta} = \mathsf{negl}(\lambda)$. This implies that the worker cannot guess $r_{\mathsf{Op}}$ except with negligible probability.

Since knowing $r$ is required to produce a valid ticket in the "sign payload" phase that produces $\sigma'_\eta$, the worker cannot therefore construct a valid ticket on its own, while it is still able to do meaningful work for the pool operator. Only the operator, who knows the complete nonce $r = r_{\mathsf{Op}} \parallel r_{\mathsf{W}}$, can perform this final step and generate the complete ticket $(digest, r, \sigma_\eta, \sigma'_\eta)$. This prevents solution theft while enabling computational outsourcing. Let us now analyze the work time of the operator and the worker. The operator Op makes only a single query to the hash function h to perform the precomputation $x := \mathsf{precomp}_{\mathsf{H}}(puz \parallel r_{\mathsf{Op}})$, which takes one query to the fixed-length function h. To estimate the work time of the worker, observe (see Fig. 7) that we let the worker sample $L-1$ leaves *once*, and then replace the last leaf with a fresh one. We do it for efficiency reasons, as changing only one leaf allows for

**Outsourcing protocol** $(\mathsf{W}, \mathsf{Op})$

- The operator Op takes as input a puzzle $puz$ and a random message $m$ and performs the following steps:
  (1) Sample random $r = (r_{\mathsf{Op}} \parallel r_{\mathsf{W}}) \xleftarrow{\$} \{0,1\}^\lambda$, where $|r_{\mathsf{Op}}| = \lambda_{\mathsf{Op}}$ and $|r_{\mathsf{W}}| = \lambda_{\mathsf{W}}$.
  (2) Perform the precomputation of h on $(puz \parallel r_{\mathsf{Op}})$ (see Eq. (1)), i.e., let $x := \mathsf{precomp}_{\mathsf{H}}(puz \parallel r_{\mathsf{Op}})$.
  (3) Send $(x, r_{\mathsf{W}})$ to the worker W.
- The worker W receives $(x, r_{\mathsf{W}})$ and performs the following steps
  (1) **Construct Merkle tree.** Sample $L-1$ random strings $leaf_1, \ldots, leaf_{L-1} \leftarrow \{0,1\}^\lambda$, and then construct a Merkle tree $T := \mathsf{Merkle}_{\mathsf{H}}(leaf_1, \ldots, leaf_{L-1}, 0^\lambda)$.
  (2) **Scratch.** Repeat the following $\widehat{t}$ times (where $\widehat{t}$ is as in the original protocol, see Step 2 of Fig. 6):
    (a) Draw a random leaf $leaf_L \leftarrow \{0,1\}^\lambda$ and update the Merkle tree $T$ by replacing the last leaf with $leaf_L$, i.e., let $T' := \mathsf{update}(T, L, leaf_L)$ (see Sec. 2.1). Let $digest := mroot(T')$.
    (b) Compute $\eta := \mathsf{H}(puz \| r \| digest)$, by postcomputing (see Eq. (2)) the value $x$ received from the operator Op, i.e., let $\eta := \mathsf{postcomp}^{3\lambda}_{\mathsf{H}}(x, r_{\mathsf{W}} \parallel digest)$ ("$3\lambda$" corresponds to the length of $puz \parallel r \parallel digest$) and use the value $\eta$ to select $q$ distinct leaves from the tree exactly as in Step (2b) of the original scheme
    (c) Let $B_1, B_2, \ldots, B_q$ denote the Merkle proofs for the selected leaves *(as in Step (2c) of the original scheme).*
    (d) Compute $\sigma_\eta := \mathsf{Encode}(B_1, \ldots, B_q)$ in sorted order of $i$ *(as in Step (2d) of the original scheme).*
    (e) Compute $y = \mathsf{H}(puz \parallel r \parallel \sigma_\eta)$ by postcomputing the value $x$ received from the operator Op, i.e., let $y := \mathsf{postcomp}^{len}_{\mathsf{H}}(x, r_{\mathsf{W}} \parallel \sigma_\eta)$ (here, "$len$" denotes the length of $puz \parallel r \parallel \sigma_\eta$). If $y < 2^{\lambda-d}$ then send $(leaf_1, \ldots, leaf_L)$ to the operator Op.
- The pool operator Op receives $(leaf_1, \ldots, leaf_L)$ from the worker W and performs the following steps:
  (1) Verify that computation was done correctly by the worker W, i.e., that the leaves $(leaf_1, \ldots, leaf_L)$ form a Merkle tree for the $digest$, that $\eta = \mathsf{H}(puz \parallel r \parallel digest)$ determines the set $\sigma_\eta$, and that $\mathsf{H}(puz \parallel r \parallel \sigma_\eta) < 2^{\lambda-d}$.
  (2) Perform the "sign payload" step of the original scheme (Steps (3a) and (3b) on Fig. 6) on message $m$ to get $\sigma'_\eta$.
  (3) Return $ticket := (digest, r, \sigma_\eta, \sigma'_\eta)$.

**Figure 7: Our attack on the scheme of Miller et al. The hash function** $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ **is constructed iteratively using a fixed-length function** $\mathsf{h}\colon \{0,1\}^{\lambda+\beta} \to \{0,1\}^\lambda$**. Parameters are discussed in Sec. 4.**

| | $\lambda$ | $\beta$ | $\lambda - \beta$ |
|---|---|---|---|
| SHA-256 | 256 | 512 | -256 |
| SHA-512 | 512 | 1024 | -512 |
| SHA3-512 | 512 | 576 | -64 |

**Table 2: Comparison of SHA-256, SHA-512, and SHA3-512: security parameter $\lambda$ (equal to the hash output size), block size $\beta$, and the difference $\lambda - \beta$.**

computation of *digest* in $\log_2 L$ oracle calls by simply recomputing one path with the update procedure (see Sec. 2.1). Since we have $\widehat{t}$ scratch attempts, the total overhead is $\widehat{t} \cdot \log_2 L$. This results in $t_{\mathsf{W}} = t + \widehat{t} \cdot \log_2 L$ hash queries. This finishes the proof. □

Let us now present its informal corollary, which shows that the original scheme of [35] is not nonoutsourceable even for a very minimal parameter choice (the "effective work" is essentially equal to the work of the worker alone).

Corollary 5 (informal). *Under the assumptions of Lem. 4, the scheme from Fig. 6 is not* $(t_{\mathsf{W}}, 1, t)$*-nonoutsourceable for* $t_{\mathsf{W}} \approx t$.

The proof of the corollary appears below. Due to the assumptions of Lem. 4, the attack does not work for every possible choice of H. However, as we argue below, it works if H is any of the following hash functions: SHA-256, SHA-512, and, with a small adaptation and in a slightly weaker sense, SHA3-512 (see, e.g., [43]). Tab. 2 summarizes the relevant parameters of these functions. It can be easily seen that Assumptions A1 and A2 of the lemma are satisfied for all three functions. Moreover, for SHA-256 and SHA-512, their fixed-length function h is assumed to be one-way and Assumption A3 is satisfied in practice with a considerable margin. For SHA3-512, however, h is not one way: it first XORs the next input block into the 1600-bit sponge internal state, and then applies the reversible Keccak permutation. In particular, since the initial state is fixed, the first iteration is completely reversible. Fortunately, in our attack this reversibility can be circumvented: instead of sending $(x, r_{\mathsf{W}})$ to the worker, the operator sends $x' = x \oplus (r_{\mathsf{W}} \,\|\, 0^{1600-\lambda_{\mathsf{W}}})$, essentially partially applying h. The worker can later complete the h invocation using *digest* and postcompute. Recovering $r_{\mathsf{Op}}$ from $x'$ amounts to solving a constrained-input constrained-output (CICO) problem for Keccak (see [7, Sec. 8.2.4]). Under the ideal-permutation heuristic for Keccak-$f[1600]$, motivated by the hermetic sponge strategy [7, Sec. 8.1.1], recovering $r_{\mathsf{Op}}$ with constant success probability requires on the order of $2^{\beta-\lambda}$ classical permutation evaluations. Assumption A3 for SHA3-512 is on the edge, as in this case $\lambda - \beta = -64$, and $2^{-64}$ may not be "negligible enough" for some very powerful adversaries (although it is hard to imagine malicious workers who would be interested in investing so much effort in stealing the puzzle solution). We also note that in practice, the solution of [35] still provides a certain degree of protection against outsourcing. This is because the malicious worker W in our attack can still steal the solution *after* it learns the operator's part $r_{\mathsf{Op}}$ of the nonce. This would be, in particular, the case if the operator Op posts the solution publicly by announcing a new block on the blockchain.

Proof of Cor. 5. We will now show that in our attack, the worker's work time is only slightly larger than the work time $t$ of a solo miner. Let us first estimate the value of the parameter $\widehat{t}$ as a function of the work time parameter $t$. This is needed, since, unlike in the original paper [35], we count the work time as the number of calls to the fixed-length function h (see Sec. 2.1), rather than the number of "scratch attempts" (i.e., the number of iterations of the loop in Step 2 of the algorithm Work). Next, we estimate the number of calls to the fixed-length function h made by the algorithm Work in Fig. 6, as a function of $\widehat{t}$. We will use this estimation later to determine the value of the parameter $\widehat{t}$ as a function of the work time parameter $t$. Constructing a Merkle tree with $L$ leaves (in Step 1) requires $L - 1$ calls to H. Since each such call requires at most two calls to h, constructing the Merkle tree requires at most

$$2(L-1) \tag{6}$$

calls to h. Each scratch attempt requires: (1) recomputing $\eta$ (one call to H, and hence to h) and (2) computing $\mathsf{H}(puz \,\|\, r \,\|\, \sigma_\eta)$, which takes approximately $q \cdot \log_2 L$ calls to h (since $\sigma_\eta$ contains $q$ Merkle branches, each of length $\log_2 L$). Since we have $\widehat{t}$ scratch attempts, the total number of calls to h made by the algorithm Work in the "Scratch" phase (Step 2) is approximately

$$\widehat{t} \cdot (1 + q \cdot \log_2 L). \tag{7}$$

Finally, the "Sign payload" phase (in Step 3) requires approximately one call to H (and hence two calls to h, assuming that $|m| = \lambda$) for computing $\eta'$. Summing this with (6) and (7), we conclude that the total number of calls to h made by Work is approximately

$$2(L-1) + \widehat{t} \cdot (1 + q \cdot \log_2 L) + 2 = 2L + \widehat{t} \cdot (1 + q \cdot \log_2 L). \tag{8}$$

To ensure that the total number of calls to h is at most $t$, we set $\widehat{t}$ to be the maximal integer such that $2L + \widehat{t} \cdot (1 + q \cdot \log_2 L) \le t$, which gives

$$\widehat{t} \approx \left\lfloor \frac{t - 2L}{1 + q \cdot \log_2 L} \right\rfloor. \tag{9}$$

Recall (cf. Lem. 4) that the work $t_{\mathsf{W}}$ of the worker in the outsourcing protocol is equal to $t + \widehat{t} \cdot \log_2 L$. Since $L$ is typically much smaller than $t$, and $q \gg 1$ (in [36], it is set to 10), by Eq. (9), $t_{\mathsf{W}}$ is approximately equal to $t$. □

## 4.5 Our countermeasure

In this section we present a scheme that avoids the attack of Sec. 4.3. To build intuition behind the scheme, let us first explain the main reason why the standard Bitcoin mining procedure *is* outsourceable. In Bitcoin mining, the proof of work is tied to the block contents: mining and signing are coupled. Slightly simplifying things (see Fig. 2 for the details), the mining process consists of finding a nonce $\nu$ for a fixed puzzle $puz$ and block contents $B$ such that the resulting hash $\mathsf{H}(puz, B, \nu)$ is below a target value. This makes the mining process outsourceable: the operator can simply give the worker the block contents and let it find a valid nonce $\nu$. Later, the operator can use the nonce $\nu$ to "sign" the block $B$ and publish it. One of the main insights of [35] is to decouple these two steps. A naive approach would let the worker find a nonce $\nu$ such that $\mathsf{H}(puz \,\|\, \nu)$ starts with $d$ zeros, and then use $\nu$ as a witness to permit signing. This would, however, violate the non-transferability requirement

of nonoutsourceable puzzles: anyone who sees $(v, B)$ can then sign a different block $B' \neq B$.

Our approach is to enforce non-transferability by a different route. During signing, the worker proves *knowledge* of a valid solution $v$ to the puzzle *puz* without revealing it. The proof is bound to the block contents $B$ and the puzzle *puz*, so that it cannot be used to sign a different block $B' \neq B$. This requirement is formalized by the notion of a *signature of knowledge* (SoK) [11]. Informally, an SoK is a signature scheme defined with respect to an NP-relation $\mathcal{R}$ that allows a signer to sign a message $B$ with respect to an NP-statement $\varphi$, provided that the signer knows a witness $w$ such that $(\varphi, w) \in \mathcal{R}$. As noted by [24], SoKs are essentially equivalent to simulation-extractable non-interactive zero-knowledge proofs (SE-NIZKs). In our setting, signing the ticket means proving knowledge of $v$ such that $\mathsf{h}(puz \,\|\, v)$ starts with $d$ zeros; doing so directly would require a NIZK *relative to a random oracle*, which creates a modeling issue. More precisely, the relation with respect to which we want to prove knowledge is $\mathcal{R}_{\mathsf{h}}^{puz} = \{((puz, Y),\ v) :\ \mathsf{h}(puz \,\|\, v) = Y\}$. Since computing $\mathcal{R}_{\mathsf{h}}^{puz}$ requires evaluating h, the relation $\mathcal{R}_{\mathsf{h}}^{puz}$ is not an NP-relation in the usual sense.

We solve this issue by introducing a new oracle $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$ that implements a SoK. For the sake of generality we define $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$ for any relation $\mathcal{R}_{\mathsf{h}}$ whose evaluation requires calling $\Omega_{\mathsf{h}}$. This oracle is presented in Fig. 8a. Readers familiar with the UC literature may recognize $\Omega_{\mathsf{SoK}}$ as a deliberately stripped-down cousin of the ideal NIZK functionality $\mathcal{F}_{\mathsf{NIZK}}$ of [25]; we avoid the use of the UC formalism for simplicity. The main difference is that $\Omega_{\mathsf{SoK}}$'s Sign interface samples the signature $\sigma$ uniformly at random instead of letting it be chosen by the adversary, which is a stronger property than soundness. However, as shown below, the candidate scheme [1] actually satisfies this property.

*4.5.1 Instantiation of $\Omega_{SoK}$.* As noted by [24], an SoK can be obtained from a SE-NIZK by augmenting the witness with a tag, i.e., $x' = (x, tag)$, where *tag* is, for example, a collision-resistant hash of the message. We take $\Sigma$ to be the simulation-extractable variant of Groth16 [1]. It is easy to see that the simulator provided in [1] outputs proofs that are uniform given the verification equation holds. One might worry that the signature of knowledge for the relation $\mathcal{R}_{\mathsf{h}}^{puz}$ involves computing a SE-NIZK proof for a SNARK-*unfriendly* function h (e.g., SHA-256). To demonstrate that this solution remains practical, we benchmarked the generation time of the SoK using the candidate scheme [1]. The benchmark was run on a Dell Latitude 7440 equipped with an Intel Core i7-1365U CPU and 16 GB of RAM. Our implementation is available at https://github.com/marmistrz/sha256-sok-groth16, and the concrete results are presented in Tab. 3.

**Table 3: Runtime for $\mathsf{h}$ = SHA-256 with $|v| = |puz| = |tag|$ = 16 B, average over 25 samples.**

| Curve | Setup | Avg prove | Avg verify |
|---|---|---|---|
| BLS12-381 | 427.5 ms | 218.3 ms | 2.26 ms |
| BLS12-377 | 416.5 ms | 208.4 ms | 2.64 ms |
| BN254 | 257.2 ms | 150.0 ms | 1.51 ms |

**Oracle $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$**

This oracle has direct access to the oracle $\Omega_{\mathsf{h}}$, and is parametrized by a relation $\mathcal{R}_{\mathsf{h}}$ relative to an oracle $\Omega_{\mathsf{h}}$.

- Upon receiving a query $(\textsc{Sign}, m, x, w)$ from a subadversary $\mathcal{A}_i$, use $\Omega_{\mathsf{h}}$ to determine if $(x, w) \in \mathcal{R}_{\mathsf{h}}$; if so, then sample $\sigma \leftarrow\!\!\$\ \{0,1\}^{\kappa}$, record $(\sigma, m, x)$, and output $\sigma$ to $\mathcal{A}_i$; otherwise output $\perp$ to $\mathcal{A}_i$.
- Upon receiving a query $(\textsc{Verify}, m, x, \sigma)$ from a subadversary $\mathcal{A}_i$, check if $(\sigma, m, x)$ was recorded; if so, then output 1 to $\mathcal{A}_i$, and 0 otherwise.

**(a) A "fast-query" oracle $\Omega_{\mathsf{SoK}}$ with access to $\Omega_{\mathsf{h}}$.**

**Oracle $\Omega_{\mathsf{SoK}}^{\mathsf{MPC}}[\mathcal{R}_{\mathsf{h}}]$**

This oracle can be viewed as a wrapper around a "fast-query" oracle $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$, presented above.

- Upon receiving a query $(\textsc{Sign}, m, x, w)$ or $(\textsc{Verify}, m, x, \sigma)$ from some $\mathcal{A}_i$, forward it to $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$ and send back the response to $\mathcal{A}_i$.
- After receiving queries $(\text{slow}, qid, \textsc{Sign}, m_i, x_i, w_i)$ or $(\text{slow}, qid, \textsc{Verify}, m_i, x_i, \sigma_i)$ from all adversaries $\mathcal{A}_i$, reconstruct the inputs $m, x, w$ (resp. $m, x, \sigma$), forward the query $(\textsc{Sign}, m, x, w)$ (resp. $(\textsc{Verify}, m, x, \sigma)$) to $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$, and send back to each $\mathcal{A}_i$ a share of the response.

**(b) A "slow-query" oracle $\Omega_{\mathsf{SoK}}^{\mathsf{MPC}}$**

**Our scheme $(\mathcal{G}_{\mathsf{SoK}}, \mathsf{Work}_{\mathsf{SoK}}, \mathsf{Verify}_{\mathsf{SoK}})$**

- $\mathcal{G}_{\mathsf{SoK}}(1^{\lambda})$: return a random string $puz \leftarrow\!\!\$\ \{0,1\}^{\lambda}$.
- $\mathsf{Work}_{\mathsf{SoK}}(puz, m, t)$:
  1. **Scratch**. For $i = 1, \ldots, t$, check if $\mathsf{h}(puz \| i)$ starts with $d$ zeros. If so, let $v = i$ and go to the next step. If no such $i$ is found, return $\perp$.
  2. **Sign**. Let $Y = \mathsf{h}(puz \| v)$. Send $(\textsc{Sign}, m, (puz, Y), v)$ to $\Omega_{\mathsf{SoK}}\left[\mathcal{R}_{\mathsf{h}}^{puz}\right]$ and obtain a response $\sigma$.
  3. Return $ticket := (Y, \sigma)$.
- $\mathsf{Verify}_{\mathsf{SoK}}(puz, m, (Y, \sigma))$: If $Y$ does not start with $d$ zeros, return 0. Otherwise, send $(\textsc{Verify}, m, (puz, Y), \sigma)$ to $\Omega_{\mathsf{SoK}}\left[\mathcal{R}_{\mathsf{h}}^{puz}\right]$ and return the response.

**(c) Our nonoutsourceable scheme $(\mathcal{G}_{\mathsf{SoK}}, \mathsf{Work}_{\mathsf{SoK}}, \mathsf{Verify}_{\mathsf{SoK}})$.**

**Figure 8: Oracles and scheme for our SoK-based nonoutsourceable construction.**

*4.5.2 Extended cost model.* To be as general as possible, we allow the adversary to compute the SoK in a distributed manner, using an MPC protocol. To model this, we introduce a new oracle $\Omega_{\mathsf{SoK}}^{\mathsf{MPC}}[\mathcal{R}_{\mathsf{h}}]$ that allows the adversary to compute the SoK in a distributed manner, while hiding the witness from each individual party. In the formal model, we extend the query budgets $t_{\mathsf{W}}, t_{\mathsf{Op}}$ to account for the "slow" (MPC-mediated) queries of $\Omega_{\mathsf{h}}^{\mathsf{MPC}}$ and $\Omega_{\mathsf{SoK}}^{\mathsf{MPC}}[\mathcal{R}_{\mathsf{h}}]$ at a

very conservative rate. For the former, we assume that each slow query costs 1 unit of both $t_{\mathsf{W}}$ and $t_{\mathsf{Op}}$, i.e., the same as a fast query but charged to *both* parties. The honest solo-mining algorithm $\mathsf{Work}_{\mathsf{SoK}}$ (and hence the baseline defining $\zeta(d,\cdot)$) only ever uses fast queries. An outsourcing protocol may additionally use up to $\sigma$ *slow* queries—each either an MPC-computed hash check or an MPC-computed SoK (Groth16) proof—and, again, conservatively, we charge every slow query to $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$ as 1 unit against *both* $t_{\mathsf{W}}$ and $t_{\mathsf{Op}}$. In particular, a slow query is never cheaper than simply having one party issue the corresponding fast query, so a rational protocol only resorts to it when hiding the witness is worth the doubled cost. In the regime of interest ($t_{\mathsf{Op}} \ll t_{\mathsf{W}}$), this in particular implies that the slow query budget $\sigma$ is much smaller than the fast query budget.

*4.5.3 Our scheme and its security.* Our nonoutsourceable scheme $(\mathcal{G}_{\mathsf{SoK}}, \mathsf{Work}_{\mathsf{SoK}}, \mathsf{Verify}_{\mathsf{SoK}})$ is presented in Fig. 8c. The idea behind it is straightforward: the worker first solves the puzzle *puz* by finding a nonce $\nu$ such that $\mathsf{h}(puz \,\|\, \nu)$ starts with $d$ zeros, and then uses the oracle $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$ to sign the block contents $B$ with respect to the statement $(puz, Y)$, where $Y = \mathsf{h}(puz \,\|\, \nu)$.

We only prove the nonoutsourceability of our scheme, as non-transferability is immediate from the $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$ formalism, and incompressibility follows immediately from the standard properties of the random oracle h. Let $q_{\mathsf{W}}$ (resp. $q_{\mathsf{Op}}$) denote the number of *fast* Scratch queries $\mathsf{h}(puz \,\|\, i)$ issued directly by W (resp. Op) during an execution, and let $s \le \sigma$ denote the number of slow queries that were actually used. By definition of the budgets, $q_{\mathsf{W}} + s \le t_{\mathsf{W}}, \quad q_{\mathsf{Op}} + s \le t_{\mathsf{Op}}$. Crucially, slow queries hide the actual input and the output from each individual party, meaning that a "traitor worker" can only rely on its own fast queries to steal the puzzle, i.e., extract a winning $(puz, \nu)$. The lemma below shows that slow queries are not useful to the distributed adversary in this regime—intuitively, the operator would just be better off mining on its own instead. Since $q_{\mathsf{W}} + s \le t_{\mathsf{W}}$ and $q_{\mathsf{Op}} + s \le t_{\mathsf{Op}}$ with $q_{\mathsf{W}}, q_{\mathsf{Op}} \ge 0$, we always have $s \le \min(t_{\mathsf{W}}, t_{\mathsf{Op}}) = t_{\mathsf{Op}}$ in the regime of interest $t_{\mathsf{Op}} < t_{\mathsf{W}}$; by Lem. 6, whenever $t_{\mathsf{Op}} \ll t_{\mathsf{E}}$ this gives $s \le t_{\mathsf{Op}} \ll t_{\mathsf{E}} - t_{\mathsf{Op}} \le q_{\mathsf{W}}$, i.e. any slow queries used are dominated by the worker's own forced fast queries. This is why $\sigma$ is not tracked as a separate parameter in Lem. 8.

Lemma 6 (Query accounting is robust to $\sigma$). *In any $(t_{\mathsf{W}}, t_{\mathsf{Op}}, t_{\mathsf{E}})$-outsourcing protocol using the extended oracle above with slow-query budget $\sigma$, if Op's success probability is at least $\zeta(d, t_{\mathsf{E}})$ and $t_{\mathsf{Op}} < t_{\mathsf{W}}$, then the number $q_{\mathsf{W}}$ of fast Scratch queries made by W satisfies*

$$q_{\mathsf{W}} \;\ge\; t_{\mathsf{E}} - t_{\mathsf{Op}},$$

*regardless of how many slow queries $s \le \sigma$ are used.*

Proof. Every fast or slow query constitutes one independent Bernoulli($\epsilon$) trial toward the target success probability, where $\epsilon = 2^{-d}$. Since h is a random oracle, achieving success probability $\ge \zeta(d, t_{\mathsf{E}}) = 1 - (1-\epsilon)^{t_{\mathsf{E}}}$ requires

$$q_{\mathsf{W}} + q_{\mathsf{Op}} + s \;\ge\; t_{\mathsf{E}}.$$

From the operator's budget constraint, $q_{\mathsf{Op}} \le t_{\mathsf{Op}} - s$. Substituting,

$$q_{\mathsf{W}} \;\ge\; t_{\mathsf{E}} - q_{\mathsf{Op}} - s \;\ge\; t_{\mathsf{E}} - (t_{\mathsf{Op}} - s) - s \;=\; t_{\mathsf{E}} - t_{\mathsf{Op}},$$

which is independent of $s$ (and hence of $\sigma$). □

We state the following standard calculus lemma.

Lemma 7 (Concavity). *Let $\epsilon = 2^{-d}$, so $\zeta(d,x) = 1 - (1-\epsilon)^x$. Then $x \mapsto \zeta(d,x)$ is concave with $\zeta(d,0) = 0$, and for any $0 \le q \le t_{\mathsf{E}}$,*

$$\zeta(d,q) \;\ge\; \frac{q}{t_{\mathsf{E}}} \cdot \zeta(d, t_{\mathsf{E}}).$$

We are now ready to prove nonoutsourceability of our modified scheme. As noted after Lem. 6, slow queries are already limited by $t_{\mathsf{Op}}$, which is why we do not use an explicit slow query bound.

Lemma 8. *Let $(\mathcal{G}_{\mathsf{SoK}}, \mathsf{Work}_{\mathsf{SoK}}, \mathsf{Verify}_{\mathsf{SoK}})$ be the scratch-off puzzle of Fig. 8c. Let $t_{\mathsf{E}} > t_{\mathsf{Op}}$, and let $\mathcal{A} = (\mathsf{W}, \mathsf{Op})$ be a $(t_{\mathsf{W}}, t_{\mathsf{Op}}, t_{\mathsf{E}})$-outsourcing protocol, where $\mathcal{A}$ has access to both $\Omega_{\mathsf{h}}^{MPC}$ and $\Omega_{SoK}^{MPC}[\mathcal{R}_{\mathsf{h}}]$. Then, there exists a malicious worker $\mathsf{W}^*$, working in time $t_{\mathsf{W}}$, such that $(\mathsf{W}^*, \mathsf{Op})$ is perfectly indistinguishable from $(\mathsf{W}, \mathsf{Op})$ from Op's point of view, and $\mathsf{W}^*$ outputs a valid ticket for a message $m^*$ of its own choosing with probability at least $\pi \cdot \zeta(d, t_{\mathsf{E}})$, where $\pi = 1 - \frac{t_{\mathsf{Op}}}{t_{\mathsf{E}}} \le \frac{t_{\mathsf{W}}}{t_{\mathsf{W}} + t_{\mathsf{Op}}}$. Consequently, $(\mathcal{G}_{\mathsf{SoK}}, \mathsf{Work}_{\mathsf{SoK}}, \mathsf{Verify}_{\mathsf{SoK}})$ is $(t_{\mathsf{W}}, t_{\mathsf{Op}}, t_{\mathsf{E}}, \pi)$-nonoutsourceable. Moreover, if $\mathcal{A}$ is $\gamma$-effective (i.e. $t_{\mathsf{Op}} \le \gamma t_{\mathsf{E}}$ for some constant $0 < \gamma < 1$), then $\frac{t_{\mathsf{W}}}{t_{\mathsf{W}} + t_{\mathsf{Op}}} \ge \pi \ge 1 - \gamma$ is a positive constant independent of $t_{\mathsf{W}}$, $t_{\mathsf{Op}}$, and $t_{\mathsf{E}}$.*

Proof. **Construction of $\mathsf{W}^*$.** Let $\mathsf{W}^*$ run *identically* to W, except that it remembers its $\Omega_{\mathsf{h}}$ query transcript. We claim that such $\mathsf{W}^*$ can output a valid ticket for an arbitrary message $m^*$ with probability at least $\pi \cdot \zeta(d, t_E)$. Clearly, $\mathsf{W}^*$ works in the same time as W and the deviation is covert, i.e., $\mathsf{view}^* \equiv \mathsf{view}$.

**Bounding the success probability.** By Lem. 6, $\mathsf{W}^*$'s own fast-query count satisfies $q_{\mathsf{W}} \ge t_{\mathsf{E}} - t_{\mathsf{Op}}$ irrespective of how the protocol split its remaining work between the operator and slow queries. Each such query is an independent Bernoulli($\epsilon$) trial, so the probability that $\mathcal{T}_{\mathsf{W}^*}$ contains a winning $(puz, \nu)$ is at least $\zeta(d, q_{\mathsf{W}}) \ge \zeta(d,\ t_{\mathsf{E}} - t_{\mathsf{Op}})$. By Lem. 7 (with $q = t_{\mathsf{E}} - t_{\mathsf{Op}} \ge 0$),

$$\zeta(d,\ t_{\mathsf{E}} - t_{\mathsf{Op}}) \;\ge\; \frac{t_{\mathsf{E}} - t_{\mathsf{Op}}}{t_{\mathsf{E}}} \cdot \zeta(d, t_{\mathsf{E}}) \;=\; \left(1 - \frac{t_{\mathsf{Op}}}{t_{\mathsf{E}}}\right) \zeta(d, t_{\mathsf{E}}).$$

Whenever $\mathcal{T}_{\mathsf{W}^*}$ contains such a pair, $\mathsf{W}^*$ succeeds in producing a valid ticket for its own $m^*$, giving

$$\Pr[\mathsf{W}^* \text{ steals}] \ge \left(1 - \frac{t_{\mathsf{Op}}}{t_{\mathsf{E}}}\right) \zeta(d, t_{\mathsf{E}}),$$

i.e. $\pi = 1 - t_{\mathsf{Op}}/t_{\mathsf{E}}$.

By Lem. 6, we have $t_{\mathsf{W}} \ge q_{\mathsf{W}} \ge t_{\mathsf{E}} - t_{\mathsf{Op}}$, so $1 - t_{\mathsf{Op}}/t_{\mathsf{E}} \le t_{\mathsf{W}}/(t_{\mathsf{W}} + t_{\mathsf{Op}})$. Finally, $t_{\mathsf{Op}} \le \gamma t_{\mathsf{E}}$ directly implies $\pi = 1 - t_{\mathsf{Op}}/t_{\mathsf{E}} \ge 1 - \gamma$. □

## 5 Conclusion

We presented practical attacks on two schemes relying on the individual/complete knowledge model [29, 35]. Our attacks exploit the structure of the hash functions (and Bitcoin mining algorithm) used in these schemes. We also showed modifications of the original schemes that avoid the attacks. We conclude that the initial intuitions of the authors of [29, 35] were correct, but to be secure, they needed a more careful modeling of hash functions in the individual knowledge setting.

## Acknowledgments

We would like to thank the anonymous ACM CCS reviewers for their valuable comments and suggestions, which helped improve the quality of this paper. In particular, we thank one of the reviewers for pointing out weaknesses in the parameters of our original construction in Sec. 4.5, which led to the alternative construction that is currently presented. This work has been partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 885666, PROCONTRA). This paper was edited for grammar, spelling, and light style polishing using ChatGPT, GitHub Copilot, and Grammarly.

## References

[1] Oussama Amine, Karim Baghery, Zaira Pindado, and Carla Ràfols. 2024. Simulation extractable versions of Groth's zk-SNARK revisited. *International Journal of Information Security* 23, 1 (2024), 431–445.

[2] James Austgen, Andrés Fábrega, Mahimna Kelkar, Dani Vilardell, Sarah Allen, Kushal Babel, Jay Yu, and Ari Juels. 2025. Liquefaction: Privately Liquefying Blockchain Assets. In *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, Marina Blanton, William Enck, and Cristina Nita-Rotaru (Eds.). IEEE, 1493–1511. https://doi.org/10.1109/SP61157.2025.00156

[3] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the Age of Blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*. ACM, 183–198. https://doi.org/10.1145/3318041.3355458

[4] Mihir Bellare, Alexandra Boldyreva, and Adriana Palacio. 2004. An Uninstantiable Random-Oracle-Model Scheme for a Hybrid-Encryption Problem. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3027)*, Christian Cachin and Jan Camenisch (Eds.). Springer, 171–188. https://doi.org/10.1007/978-3-540-24676-3_11

[5] Mihir Bellare and Phillip Rogaway. 1993. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, Virginia, USA) *(CCS '93)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/168588.168596

[6] G.M. Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. 2007. Sponge Functions. *ECRYPT Hash Workshop 2007* (01 2007).

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2011. Cryptographic Sponge Functions. https://keccak.team/files/CSF-0.1.pdf Version 0.1.

[8] Jan Bormet, Stefan Dziembowski, Sebastian Faust, Tomasz Lizurej, and Marcin Mielniczuk. 2025. Strong Secret Sharing with Snitching. In *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 16003)*, Yael Tauman Kalai and Seny F. Kamara (Eds.). Springer, 295–327. https://doi.org/10.1007/978-3-032-01884-7_10

[9] Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. Proofs of retrievability: theory and implementation. In *Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009, Chicago, IL, USA, November 13, 2009*, Radu Sion and Dawn Song (Eds.). ACM, 43–54. https://doi.org/10.1145/1655008.1655015

[10] Ran Canetti, Oded Goldreich, and Shai Halevi. 2004. The random oracle methodology, revisited. *J. ACM* 51, 4 (2004), 557–594. https://doi.org/10.1145/1008731.1008734

[11] Melissa Chase and Anna Lysyanskaya. 2006. On Signatures of Knowledge. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4117)*, Cynthia Dwork (Ed.). Springer, 78–96. https://doi.org/10.1007/11818175_5

[12] Alexander Chepurnoy and Amitabh Saxena. 2020. Bypassing Non-Outsourceable Proof-of-Work Schemes Using Collateralized Smart Contracts. In *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12063)*, Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala (Eds.). Springer, 423–435. https://doi.org/10.1007/978-3-030-54455-3_30

[13] James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, Arup Mondal, and Esra Yeniaras. 2024. Detecting Rogue Decryption in (Threshold) Encryption via Self-Incriminating Proofs. *IACR Cryptol. ePrint Arch.* (2024), 794. https://eprint.iacr.org/2024/794

[14] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. 2015. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press.

[15] Philip Daian, Ittay Eyal, Ari Juels, and Emin Gün Sirer. 2017. (Short Paper) PieceWork: Generalized Outsourcing Control for Proofs of Work. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10323)*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer, 182–190. https://doi.org/10.1007/978-3-319-70278-0_11

[16] Ivan Bjerre Damgård. 1990. A Design Principle for Hash Functions. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer New York, New York, NY, 416–427.

[17] Cynthia Dwork and Moni Naor. 1992. Pricing via Processing or Combatting Junk Mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 740)*, Ernest F. Brickell (Ed.). Springer, 139–147. https://doi.org/10.1007/3-540-48071-4_10

[18] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. 2015. Proofs of Space. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9216)*, Rosario Gennaro and Matthew Robshaw (Eds.). Springer, 585–605. https://doi.org/10.1007/978-3-662-48000-7_29

[19] Stefan Dziembowski, Sebastian Faust, and Tomasz Lizurej. 2023. Individual Cryptography. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14082)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, 547–579. https://doi.org/10.1007/978-3-031-38545-2_18

[20] Stefan Dziembowski, Sebastian Faust, Tomasz Lizurej, and Marcin Mielniczuk. 2024. Secret Sharing with Snitching. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 840–853. https://doi.org/10.1145/3658644.3690296

[21] Andrés Fábrega, Amy Zhao, Jay Yu, James Austgen, Sarah Allen, Kushal Babel, Mahimna Kelkar, and Ari Juels. 2025. Voting-Bloc Entropy: A New Metric for DAO Decentralization. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 1299–1318. https://www.usenix.org/conference/usenixsecurity25/presentation/fabrega-entropy

[22] Ben Fisch. 2019. Tight Proofs of Space and Replication. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11477)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, 324–348. https://doi.org/10.1007/978-3-030-17656-3_12

[23] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.* 18, 1 (1989), 186–208. https://doi.org/10.1137/0218012

[24] Jens Groth and Mary Maller. 2017. Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs. In *Advances in Cryptology – CRYPTO 2017*. Springer, 581–612. https://doi.org/10.1007/978-3-319-63715-0_20

[25] Jens Groth, Rafail Ostrovsky, and Amit Sahai. 2006. Perfect Non-Interactive Zero-Knowledge for NP. In *Advances in Cryptology – EUROCRYPT 2006 (Lecture Notes in Computer Science, Vol. 4004)*. Springer, 339–358. https://doi.org/10.1007/11761679_21

[26] Runchao Han, Haoyu Lin, and Jiangshan Yu. 2020. VRF-Based Mining Simple Non-outsourceable Cryptocurrency Mining. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2020 International Workshops, DPM 2020 and CBT 2020, Guildford, UK, September 17-18, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12484)*. Springer, 287–304. https://doi.org/10.1007/978-3-030-66172-4_19

[27] Timo Hanke. 2016. AsicBoost - A Speedup for Bitcoin Mining. *CoRR* abs/1604.00575 (2016). arXiv:1604.00575

[28] Carmit Hazay and Yehuda Lindell. 2010. *Sigma Protocols and Efficient Zero-Knowledge*. Springer Berlin Heidelberg, Berlin, Heidelberg, 147–175. https://doi.org/10.1007/978-3-642-14303-8_6

[29] Mahimna Kelkar, Kushal Babel, Philip Daian, James Austgen, Vitalik Buterin, and Ari Juels. 2024. Complete Knowledge: Preventing Encumbrance of Cryptographic Secrets. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 2415–2429. https://doi.org/10.1145/3658644.3690273

[30] Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov. 2025. How to Prove False Statements: Practical Attacks on Fiat-Shamir. *IACR Cryptol. ePrint Arch.* (2025), 118. https://eprint.iacr.org/2025/118

[31] Gaëtan Leurent and Phong Q. Nguyen. 2009. How Risky Is the Random-Oracle Model? In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5677)*, Shai Halevi (Ed.). Springer, 445–464. https://doi.org/10.1007/978-3-642-03356-8_26

[32] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. 2004. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2951)*, Moni Naor (Ed.). Springer, 21–39. https://doi.org/10.1007/978-3-540-24638-1_2

[33] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 293)*, Carl Pomerance (Ed.). Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[34] Ralph C. Merkle. 1990. A Certified Digital Signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer New York, New York, NY, 218–238.

[35] Andrew Miller, Ahmed E. Kosba, Jonathan Katz, and Elaine Shi. 2015. Nonoutsourceable Scratch-Off Puzzles to Discourage Bitcoin Mining Coalitions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 680–691. https://doi.org/10.1145/2810103.2813621

[36] Andrew Miller, Ahmed E. Kosba, Jonathan Katz, and Elaine Shi. 2015. Nonoutsourceable Scratch-Off Puzzles to Discourage Bitcoin Mining Coalitions. Full version of [35] available at https://soc1024.ece.illinois.edu/nonoutsourceable_full.pdf.

[37] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction.* Princeton University Press, USA.

[38] Meni Rosenfeld. 2011. Analysis of Bitcoin Pooled Mining Reward Systems. *CoRR* abs/1112.4980 (2011). https://doi.org/10.48550/arxiv.1112.4980

[39] S. Ruj, S.S. Kanhere, and M. Conti. 2024. *Blockchains: A Handbook on Fundamentals, Platforms and Applications.* Springer International Publishing.

[40] Claus-Peter Schnorr. 1989. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 435)*, Gilles Brassard (Ed.). Springer, 239–252. https://doi.org/10.1007/0-387-34805-0_22

[41] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[42] Wenbo Wang, Dinh Thai Hoang, Peizhao Hu, Zehui Xiong, Dusit Niyato, Ping Wang, Yonggang Wen, and Dong In Kim. 2019. A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks. *IEEE Access* 7 (2019), 22328–22370. https://doi.org/10.1109/ACCESS.2019.2896108

[43] Wikipedia contributors. 2025. Secure Hash Algorithms — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Secure_Hash_Algorithms&oldid=1310574193. [Online; accessed 9-October-2025].

[44] Wikipedia contributors. 2025. SHA-2 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=SHA-2&oldid=1303317086. [Online; accessed 9-August-2025].

[45] Wikipedia contributors. 2025. Trusted execution environment — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Trusted_execution_environment&oldid=1292922440. [Online; accessed 4-June-2025].

[46] Jie Xu, Cong Wang, and Xiaohua Jia. 2023. A Survey of Blockchain Consensus Protocols. *ACM Comput. Surv.* 55, 13s (2023), 278:1–278:35. https://doi.org/10.1145/3579845

[47] Gongxian Zeng, Zhengan Huang, Yu Wang, Xin Mu, and Cheng Zhang. 2023. Strongly nonoutsourceable scratch-off puzzles in blockchain. *Soft Comput.* 27, 17 (2023), 11941–11960. https://doi.org/10.1007/S00500-023-08753-1

[48] Gongxian Zeng, Siu-Ming Yiu, Jun Zhang, Hiroki Kuzuno, and Man Ho Au. 2017. A Nonoutsourceable Puzzle Under GHOST Rule. In *15th Annual Conference on Privacy, Security and Trust, PST 2017, Calgary, AB, Canada, August 28-30, 2017.* IEEE Computer Society, 35–43. https://doi.org/10.1109/PST.2017.00015


# A Open science

We provide all necessary artifacts to replicate our results and build upon our work: (1) a proof-of-concept implementation of the Groth16-based signature of knowledge for SHA-256 instantiating $\Omega_{\mathsf{SoK}}[\mathcal{R}_{\mathsf{h}}]$: github.com/marmistrz/sha256-sok-groth16, (2) a proof-of-concept implementation demonstrating midstate injection into a Bitcoin miner, including modified firmware for the Bitaxe Max platform available at: github.com/wciszewski/ESP-Miner-midstate/tree/midstate_override, (3) a simple modified Stratum server used for testing and demonstration: github.com/wciszewski/rust-stratum-pool/tree/midstate_override. Our work builds on Bitaxe Max, an open-source Bitcoin miner, with original firmware available at: github.com/bitaxeorg/bitaxemax. Our modifications enable the miner to operate using externally precomputed intermediate hashes (midstates), whereas the default firmware computes and batches them in groups of four before passing them to the BM1397 ASIC.

# B Standard notation

The symbol $\leftarrow\!\!\$$ denotes uniform sampling from a set, i.e., for a finite set $\mathcal{X}$, we write $x \leftarrow\!\!\$\ \mathcal{X}$ to indicate that $x$ is chosen uniformly at random from $\mathcal{X}$. For a security parameter $\lambda \in \mathbb{N}$, we write $\mathrm{negl}(\lambda)$ to denote a negligible function in $\lambda$. The symbol $\stackrel{c}{\approx}$ denotes computational indistinguishability between two probability distributions or random variables, meaning that no probabilistic polynomial-time algorithm can distinguish between them with non-negligible advantage.

# C Merkle trees

Let us also recall the classic *Merkle tree* construction [33] that is used to commit to a set of values and later prove the membership of a value in the committed set. Let $L$ be a power of 2 and let $\lambda \in \mathbb{N}$ be some parameter. Let $\mathsf{F}$ denote a function $\mathsf{F}\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$. Here $\mathsf{F}$ can be a hash function restricted to inputs of length $2\lambda$, or simply a compression function (see Sec. 2.1). We define the function $\mathsf{Merkle}_{\mathsf{F}}$, which on input a list of $L$ values $(leaf_1, \ldots, leaf_L) \in (\{0,1\}^{\lambda})^L$ outputs a labeled binary tree $T$ (called a *Merkle tree*) with $L$ leaves, where the label of the $i$th leaf is equal to $leaf_i$, and each internal node is labeled with the value of $\mathsf{F}$ applied to the concatenation of the labels of its two children. Moreover, let $mroot(T)$ denote the label of the root of Merkle tree $T$, and for $i = 1, \ldots, L$, let $\mathsf{MProof}(T, i)$ denote the Merkle proof for the $i$th leaf of a tree $T$, i.e., the siblings of all nodes on the branch of $T$ going from this leaf to the root. Merkle trees can also be easily updated in time logarithmic in the number of leaves. For a Merkle tree $T$ as above, let $\mathsf{update}(T, i, leaf)$ denote the Merkle tree $T'$ that is obtained from $T$ by replacing the $i$th leaf with $leaf$ and recomputing the tree. Note that for a fixed leaf index $i$, the value of $mroot(T)$ depends on the value of $leaf_i$ and on the labels of the sibling nodes on the path from the $i$th leaf to the root (denote this sequence as $siblings(T, i)$). Therefore, given $\mathsf{MProof}(T, i)$, the update operation can be performed in time $O(\log L)$.