How to Prove Work: With Time or Memory

XIANGYU SU1, MARIO LARANGEIRA1,2, AND KEISUKE TANAKA1,3

1Tokyo Institute of Technology, Tokyo 152-8550, Japan
2Input Output Hong Kong (IOHK), Hong Kong
3Corresponding author: Xiangyu Su (su.x.ab@m.titech.ac.jp)

This work was supported by the Input Output Cryptocurrency Collaborative Research Chair through the Input Output Hong Kong (IOHK), through the Japan Science and Technology Agency (JST) CREST under Grant JPMJCR14D6, through the JST OPERA, through the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP16H01705 and Grant JP17H01695, through the JST CREST Grant Number JPMJCR2113, through the JSPS KAKENHI JP21H04879 and JP21K11882.

ABSTRACT Proposed by Dwork and Naor (Crypto’ 92) as an anti-spam technique, proof-of-work is attracting more attention with the boom of cryptocurrencies. A proof-of-work scheme involves two types of participants, i.e., provers and verifiers. Provers are asked to solve a computational puzzle, and verifiers need to check the solution’s correctness. A widely adopted hash-based construction achieves an optimal gap in computational complexity between provers and verifiers. However, in industry, proof-of-work is done by highly dedicated hardware, e.g., “ASIC”, which is not generally accessible, let alone the high energy consumption rates. In this work, we turn our eyes back to the original meaning of “proof of work”. Under a trusted setting, we propose a framework and its constructions based on computationally hard problems and the unified definition of hard cryptographic primitives by Biryukov and Perrin (Asiacrypt’ 17). The new framework enables us to have a proof-of-work scheme with time-hardness or memory-hardness while cutting down power consumption and reducing the impact of dedicated hardware.

INDEX TERMS Blockchain, moderate hard primitives, proof-of-work.

I. INTRODUCTION

A. BACKGROUND AND MOTIVATION

The proof-of-work (PoW) scheme was introduced by Dwork and Naor in the early 1990s [1] to combat junk mail. The idea is simple: Senders are given a function that requires enough computing power to evaluate. The evaluation gains the sender access to the e-mail service and is regarded as a “proof of work”. A malicious sender intends to send large amounts of junk e-mail. The behavior costs the sender a significant portion of its computing power during the evaluation. Thus, the scheme can mitigate frivolous use. Almost 20 years later, the work of Bitcoin by Satoshi Nakamoto [2] utilized a PoW scheme to mitigate the “Sybil Attack” in a peer-to-peer network. An attacker who creates numerous pseudonymous identities to subvert the network needs significant advantages in computing power. By assuming the upper bounds of the attacker’s computing power, the PoW scheme rules out such an attack. The innovative work refueled the interest of the community in researching PoW-related schemes.

Participants in a PoW scheme are provers and verifiers. Provers are given a puzzle instance (puzzle, T) and output a solution solution. T indicates the computational complexity (difficulty) of puzzle. Verifiers check the (puzzle, solution) pair’s correctness concerning the corresponding difficulty T. The puzzle should be moderately difficult, i.e., the puzzle costs neither too much computing power nor too little.

A widely adopted construction is based on hash functions. We consider a hash function hash that takes in binary strings of arbitrary length and outputs binary strings of length $N \in \mathbb{N}$, i.e., $\text{hash} : \{0, 1\}^* \rightarrow \{0, 1\}^N$. With a puzzle instance $x \in \{0, 1\}^*$ and a target value $T$, provers need to find a solution $y \in \{0, 1\}^*$, such that $\text{hash}(x, y) < T$. For example, concretely in the Bitcoin system, the hash value should begin with at least $T$ zeros. A prover needs $2^T$ times of iterative hash evaluation to find a valid solution in expectation. Thus, when the scheme is implemented in a large-scale scenario, the numerous hash evaluations require massive energy consumption. Another problem comes from the dedicated hardware of computing hash functions. Such hardware, e.g., the “ASIC”, is relatively expensive and can outperform regular CPUs. Normal users may be excluded from participating in the system. Thus, leading the system to unfairness and centralization. Also, the hardware is only capable of particular types of hash functions and will lose its value when no more hashes are needed. Finally, the security is also questionable. As pointed out by Ball et al. [3]: the security is based on the belief that concrete...
hash functions, say SHA-256, behave unpredictably, which has only heuristically provable guarantees.

These problems highlight the need for more research on alternative PoW frameworks and constructions. For frameworks, considerably many results have been proposed, including proof-of-stake [4], proof-of-space [5], and other paradigms [6]. However, to the best of our knowledge, only a handful of constructions based on computational assumptions are known [3]. There are two main barriers:

1) If we use a security parameter comparable to the hash-based construction, the puzzle can be overwhelmingly difficult;

2) To fine-tune the difficulty may require readjusting the security parameters of the underlying hard problems.

In this work, we propose a PoW scheme that circumvents these problems at the expense of a trusted puzzle generation phase. Our construction may find its applications in hybrid settings (a consortium blockchain) where a trusted third party, e.g., a company providing puzzle generation services.

B. OUR APPROACH AND CONTRIBUTION

The straightforward idea is to use trapdoor functions. We push the argument as follows. To find the trapdoor of a certain trapdoor function is comparable to inverting the function in computational complexity (evidence can be found in [7], [8]), and the function is trivially invertible if the trapdoor is given. Thus, we design a moderately hard approach to offer the trapdoor to provers. Precisely, with the unified framework of hard cryptographic primitives by Biryukov and Perrin [9], we can ask provers to evaluate a hard function before obtaining the trapdoor. The unified framework brings us even more advantages since the definition has three axes: time, memory, and code. In this work, we focus on the former two and propose:

- A new framework for PoW schemes called “advice-based PoW” and its weakened variant, the “advice-based PoW with trusted generators”;
- A generic construction for the advice-based PoW with trusted generators, based on one-way trapdoor functions and asymmetrically hard functions [9];
- Two concrete constructions for the weakened scheme, which can achieve provably time or memory hardness.

We model our work in the random-access machine model. Like the original PoW scheme, the advice-based PoW scheme is associated with puzzle generation, solving, and verification algorithms. The puzzle generation should be efficient. The complexity of solving should match the designed difficulty, whereas the verification complexity should be at most logarithmic of the solving. Intuitively, the puzzle consists of an instance of a computationally hard problem (trapdoor function) and a pre-image of an asymmetrically hard function. The solving algorithm evaluates the hard function to obtain a piece of advice and uses the advice to invert the given instance. We require a trusted generator to guarantee the advice contains information of the trapdoor.

Our framework provides a generic transformation from asymmetrically hard functions to a PoW scheme. Thus, it fills the gap between cryptographically hard primitives and practical PoW schemes. We present the construction based on one-way trapdoor functions and asymmetrically hard functions under a trusted setting. We require the instance of the one-way trapdoor function that can be generated from a given trapdoor. The RSA problem satisfies such a property. Thus, we also show two concrete constructions utilizing the RSA problem. One is based on the RSW time-lock puzzle [10] (RSW puzzle) for the time-hard construction; the other one is based on the DIODON function [9] for the memory-hard construction, respectively. In the end, we introduce a distributed puzzle generation protocol from multiparty computation [11]. The distributed process can discard the trusted generators from the weakened scheme. We argue that iterative squaring (of the RSW puzzle) is relatively easy so that dedicated hardware will not surpass standard CPUs vastly. We demonstrate the proofs for the time-hard construction as the other follows similarly.

C. RELATED WORK

We investigate constructions of the time-hard and memory-hard PoW schemes. For time-hardness, we have the time-lock puzzle scheme [10] and verifiable delay functions [12], while for memory-hardness, we have the memory-hard PoW scheme [13], [14] and the proof-of-space scheme [5].

1) TIME-LOCK PUZZLE AND VERIFIABLE DELAY FUNCTION

The time-lock puzzle, introduced by Rivest, Shamir, and Wagner [10], is already a semi-PoW in advance, lacking only the efficient public verifiability. By adding a knowledge proof protocol, Boneh et al. [12] enhanced the time-lock puzzle scheme as the verifiable delay functions, providing efficient and public to the verification of the time-lock puzzle. Built on top of the time-lock puzzle, constructions by Woswoloski [15] and Pietrzak [16] rely heavily on the structure of concrete time-lock puzzles, leaving the lack of flexibility. Projects, e.g., IOTA, are considering to take verifiable delay functions as candidates for replacing PoW schemes [17].

2) MEMORY-HARD PoW SCHEMES AND PROOF-OF-SPACE

Aiming to reduce ASICs’ impact, Biryukov and Khovratovich [13], proposed the memory-hard PoW schemes. They took the memory-hard password hashing functions as basic building blocks. The construction also has the drawback that there is only heuristic provability. Several attacks are presented by Coelho et al. in [14], causing concerns on the security. As an alternative, the proof-of-space scheme also considers memory-hardness. However, an adversary can convince verifiers with valid proof from honest provers because it is not unique. Thus, the proof-of-space scheme seems improper for building a memory-hard PoW scheme.
II. PRELIMINARIES

We use λ for a security parameter. PPT denotes probabilistic polynomial time. The function negl(λ) is negligible of λ, if for every positive integer c, there exists a large enough λ, such that negl(λ) < λ−c holds. Given a set 𝒳, x ∈ uniform sample from 𝒳; while for an algorithm Alg, x ← Alg denotes that x is assigned to the outputs of Alg on fresh randomness. For an index set, [k] = {0, 1, . . . , k − 1}. The table of notations can be found in Table 1.

We also introduce terms, including resource, hardness, and difficulty, with given or renewed uses in this paper.

A. RESOURCE

Regardless of time-wise or memory-wise, the unified measurement of computational complexity is about the resource to be consumed. Thus under the term of resource, given R = (ρ, u), the resource requirements of a computational task, ρ = {Time, Memory} denotes the resource type for evaluating the task, and u denotes the desired amount of resource units for completing the task.

Although u outlines the desired computational complexity, we intend to analyze it in a more fine-grained way. Thus we define a δ-function, regarding the possible lower bounds for the task.

Definition 1 (δ-Function): For any amount of resource units u, there exists 0 < ε ≤ 1, such that δ(u) is lower bounded by uε.

The δ-function outlines the loss of the required resource: a clever participant may complete the task with less than u units of resource, but its consumption must be larger than δ(u) as long as the scheme is secure. Thus a larger ε, i.e., closer to 1, provides a better loss rate for evaluating the task and better security.

B. HARDNESS AND DIFFICULTY

We clarify these easily confused terms before proceeding to formal definitions.

- **Hardness** of hard functions is the resource cost of evaluating a hard function, with time or memory units;
- **Difficulty** of PoW puzzles is the total cost of solving a PoW puzzle, with time and memory units.

In later sections, we consider hardness with respect to time and memory, but handle the difficulty with a unified notation: T(λ, u), given R = (ρ, u). In general, a PoW puzzle is T(λ, u)-difficult,

- for ρ = Time, if solving the puzzle takes no less than T(λ, δ(u)) time;
- for ρ = Memory, if solving the puzzle takes no less than T(λ) time and δ(u) memory units.

Remark 1 (Size Restriction of u): u must be sub-exponential of λ asymptotically. Any algorithm with a longer run time, e.g., exponential of λ, enables provers to crack the underlying hard problem without going through the designed routine. On the other hand, u should be well-chosen to provide the desired moderate hardness.

III. DEFINITIONS OF FRAMEWORKS

In this section, we present the advice-based framework for the PoW scheme and its weakened variant, which requires a trusted puzzle generation phase. The formal definitions for syntax will be unified, i.e., without specifying time-hardness or memory-hardness. However, in terms of security, time-memory trade-offs are always the case for memory-hard cryptographic primitives. Thus, we give specific definitions of security concerning time and memory separately.

A. ADVICE-BASED PoW DEFINITIONS

The advice-based PoW framework involves a tuple of algorithms (Setup, PGen, EvalSolve, Verify). Setup generates public parameters that determine domains according to a security parameter; PGen generates a puzzle with an auxiliary input. The auxiliary input serves as the alternative approach for providing trapdoors; EvalSolve consists of two sub-algorithms Eval and Solve, where Eval outputs a piece of advice for Solve to solve the puzzle with a solution; Verify verifies the correctness of the puzzle and solution concerning the resource.

Definition 2 (Advice-Based PoW): The tuple of algorithms (Setup, PGen, EvalSolve, Verify) works as follows:

- Setup(1λ, R) → pp. On input a security parameter λ and resource R = (ρ, u) with type ρ and units amount u, Setup outputs a public parameter pp, which determines the puzzle and solution domain;
- PGen(pp) → (puz, aux). On input the public parameter pp, PGen outputs a puzzle puz for advice-based PoW and a puzzle-auxiliary aux for evaluating the advice. The computational complexity for solving (puz, aux) follows the given R in Setup;
- EvalSolve(pp, puz, aux) → solution. On inputs, the two sub-algorithms Eval and Solve of EvalSolve are:
  - Eval(pp, aux) → advice. Eval takes in pp and aux, produces a piece of advice advice;
  - Solve(pp, puz, advice) → solution. With obtained advice from Eval, Solve solves puz and outputs the solution solution;
- Verify(pp, puz, solution) → {0/1}. Verify is a deterministic algorithm. It checks the correctness for the pair (puz, solution), accepts with “1” and “0” otherwise.

Remark 2 (Inherent Difficulty): Instead of in PGen’s outputs, we regard the difficulty T(λ, u) as an inherent property of (puz, aux) generated with parameters pp and R = (ρ, u), and T(λ, u) is only used for security analysis.

We weaken the advice-based PoW by adding a trapdoor known to the puzzle generator and discard the trapdoor after the puzzle generation phase.

Definition 3 (Advice-Based PoW With Trusted Generators): The difference to Definition 2 lies in the Setup and PGen algorithm:
• Setup\((t^\lambda, R) \rightarrow (pp, td)\), where td is an additional secret trapdoor for the puzzle generator;

• PGen\((pp, td) \rightarrow (puz, aux)\). A trusted generator runs KGen with td from the Setup algorithm and generates \((puz, aux)\);

• Eval\((\cdot)\) and Verify work identically as in Definition 2.

In order for a cryptographic primitive to be useful, it must be correct: for the advice-based PoW scheme, the Setup provides sufficiently possible for a properly generated \(puz, aux\), the probability \(puz\) solution with negligible probability. Precisely, for any adversary that can only find a valid piece of advice using fewer resource units than \(\delta(u)\) with non-negligible probability, hardness means that no adversary can trade its memory with its time. Whereas in the soundness game, the adversary only obtains \(\delta(u)\) memory units, the probability of \(A\) winning the hardness game is negligible of \(\delta \lambda\) and the probability is taken over PGen, \(A\) and Verify’s randomness.

As shown by Hellman [18], formalization of memory-hardness is difficult because one can trade its memory with computation time, which leads to the violation of memory-hardness. Thus, instead of only using memory, we measure both time and memory costs in our definition.

Definition 7 \((t, \delta)-Memory-Hard\): An advice-based PoW with trusted generators scheme is \((t, \delta)-memory-hard\), if for any algorithm \(A = (A_0, A_1)\), where \(A_0\) runs in \(O(\text{poly}(\lambda, u))\) time and \(A_1\) runs in \(\delta(u)\) time, the probability of \(A\) winning the hardness game is negligible of \(\delta \lambda\) and the probability is taken over PGen, \(A\) and Verify’s randomness.

Remark 3 (Achievable Soundness): We consider the range of \(t\) such that we can have meaningful soundness. Recall the restriction on resource consumption \(u\) in Section II, bounded above by the sub-exponential of \(\lambda\). Directly, \(t\) should be larger than \(u\) or \(T(\lambda)\) in time-hard or memory-hard scheme. Moreover, \(t\) should be strictly less than exponential of \(\lambda\), i.e., \(O(2^{\lambda/\epsilon})\) with sufficiently large \(c\) since no algorithm running in \(t\)-time can solve the underlying hard problem, which is \(puz\) in our case. \(t \ll O(2^{\lambda/\epsilon})\) guarantees that provers cannot crack \(puz\) for an acceptable solution with brute force.

2) HARDNESS

Soundness captures the impossibility of solving the puzzle without a valid piece of advice. Hardness means that no adversary can obtain such advice using fewer resource units than \(\delta(u)\) with non-negligible probability. Hardness, as previously mentioned, has time-hardness and memory-hardness defined separately in this section. We first define a unified hardness game. Consider the following game in Figure 1 between a two-stage adversary \(A = (A_0, A_1)\) and a challenger \(CH\):

The adversary can make at most \(q\) queries of \(aux\) to the challenger. It is said to win the hardness game if the challenger outputs “1” with non-negligible probability, and \(aux^*\) is not in the adversary’s query set \(\{aux\}_q\).

Definition 6 \((\delta)-Time-Hard\): An advice-based PoW with trusted generators scheme is \(\delta\)-time-hard, if for any algorithm \(A = (A_0, A_1)\), where \(A_0\) runs in \(O(\text{poly}(\lambda, u))\) time and \(A_1\) runs in \(\delta(u)\) time, the probability of \(A\) winning the hardness game is negligible of \(\delta\lambda\) and the probability is taken over PGen, \(A\) and Verify’s randomness.

3) DIFFICULTY

In the difficulty game, we provide the adversary a full power with \(puz\) and the proper \(aux^*\). Whereas in the soundness definition, the adversary only obtains \(puz\); in the hardness game, the adversary only has \(aux^*\). The adversary can make \(q\) queries of \(aux\) to the challenger for \(\{\text{advice}\}_q\). We give the following game-based definition for the difficulty in Figure 2.

The adversary is said to win the difficulty game if the challenger outputs “1” with non-negligible probability and \(aux^* \not\in \{aux\}_q \wedge \text{Eval}(u, aux^*) \not\in \{\text{advice}\}_q\). Notice that, the adversary can process \(puz\) in advance in the difficulty game.

Definition 8 \((T(\lambda, u))-Difficult\): An advice-based PoW with trusted generators scheme is \((T(\lambda, u))-difficult\), if for any algorithm \(A = (A_0, A_1)\), where \(A_0\) runs in \(T(\lambda, \delta(u))\) time \((T(\lambda)\) time and \(\delta(u)\) memory units) the probability of \(A\)
TABLE 1. Notations in this paper.

| Notation       | Meaning                                                      |
|----------------|--------------------------------------------------------------|
| $\lambda$     | A security parameter                                        |
| PPT            | Probabilistic polynomial time                                |
| neg($\lambda$) | A negligible function in lambda                              |
| $x \leftarrow \mathcal{X}$ | An element $x$ is sampled from a finite set $\mathcal{X}$ uniformly at random |
| $x \leftarrow \text{Alg}$ | $x$ is assigned to a (PPT) algorithm $\text{Alg}'$’s output |
| $[k] = \{0, 1, \ldots, k - 1\}$ | An index set from 0 to $k - 1$ |

winning the difficulty game is negligible of $\lambda$ and is taken over $\text{PGen}$, $\mathcal{A}$ and Verify’s randomness.

Difficulty cannot be derived directly from soundness and hardness, as there may be shortcuts when given $\text{puz}$ and $\text{aux}$ simultaneously. That is, $\text{aux}$ may leak information for solving $\text{puz}$ before the valid advice. We rule out such an attack in concrete constructions and present a more detailed analysis in Section V.

IV. BUILDING BLOCKS
Here we introduce the building blocks, i.e., asymmetrically hard function family [9], denoted by $\text{AHF}$; and one-way trapdoor function family, denoted by $\text{TDP}$.

A. ASYMMETRICALLY HARD FUNCTIONS
We formalize the implied definitions from [9] for further use and show the candidate constructions. We denote an asymmetrically hard function family as $\text{AHF}$, and it involves a four-tuple of algorithms (Gen, Sample, Eval, Asy):

- $\text{Gen}(1^\lambda, R) \rightarrow (i, \text{td})$. On inputs $\lambda$ and $R = (\rho, u)$, $\text{Gen}$ outputs an index $i$ for $f_i \in \text{AHF}$ and the corresponding trapdoor $\text{td}$;
- $\text{Sample}(1^\lambda, i) \rightarrow x$. On inputs $\lambda$ and the index $i$, $\text{Sample}$ samples $x \leftarrow \mathcal{X}$, where $\mathcal{X}$ is the domain of $f_i$;
- $\text{Eval}(1^\lambda, i, x) \rightarrow y$. Eval evaluates $f_i(x)$;
- $\text{Asy}(1^\lambda, i, \text{td}, x) \rightarrow y'$. Asy evaluates $f_i(x)$ with a trapdoor $\text{td}$.

Taken together, these algorithms should satisfy the following properties, omitting $1^\lambda$ below.

**Definition 9** ($\langle R, \delta \rangle$-Asymmetrically Hard Function Family): $\text{AHF}$ is an $\langle R, \delta \rangle$-asymmetrically hard function family for $R = (\rho, u)$ and $\delta = \delta(u)$, if for any $(i, \text{td}) \leftarrow \text{Gen}(R)$, the following holds for $f_i \in \text{AHF}$.

- **Correctness**: For all $x \leftarrow \text{Sample}(i)$,
  \[
  \Pr[\text{Eval}(i, x) = f_i(x)] = 1,
  \]
  where the probability is taken over $\text{Eval}$’s randomness.

- **Hardness**: $f_i$ is $\delta(u)$-hard ($(t(u), \delta(u))$-hard, respectively when the resource is memory). That is, for any adversary $\mathcal{A}$ without $\text{td}$, samples $x \leftarrow \text{Sample}(i)$ and runs with less than $\delta(u)$-resource units of $\rho = \text{Time}$ or $(t(u), \delta(u))$-resource units of $\rho = \text{Memory}$,
  \[
  \Pr[\mathcal{A}(i, x) = f_i(x)] < \text{negl}(\lambda),
  \]
  where the probability is taken over $\mathcal{A}$’s randomness.

- **Asymmetrical hardness**: For all $x \leftarrow \text{Sample}(i)$, $\text{Asy}$ runs with $O(\lambda)$-resource units of $\rho$ and satisfies correctness:
  \[
  \Pr[\text{Asy}(i, x) = f_i(x)] = 1,
  \]
  where the probability is taken over $\text{Asy}$’s randomness.

B. TYPICAL ONE-WAY TRAPDOOR FUNCTIONS
We refine the textbook definition of the one-way trapdoor functions to fit in our generic constructions. Concretely, the one-way trapdoor function family generator takes a trapdoor as input and outputs an instance of the corresponding one-way trapdoor function, or $\bot$ if the instance does not exist. We also adopt a similar syntax used for the asymmetrically hard functions. We denote a typical one-way trapdoor function family as $\text{TDP}$, and it involves a four-tuple of algorithms (Gen, Sample, Eval, Invert):

- $\text{Gen}(1^\lambda, \text{td}) \rightarrow i$. On input $\lambda$ and a trapdoor $\text{td}$, if $\text{td}$ corresponds to an $f_i \in \text{TDP}$, $\text{Gen}$ outputs the index $i$ of $f_i$, otherwise $\text{Gen}$ outputs $\bot$;
- $\text{Sample}(1^\lambda, i) \rightarrow x$. On inputs $\lambda$ and the index $i$, $\text{Sample}$ samples $x \leftarrow \mathcal{X}$, where $\mathcal{X}$ is the domain of $f_i$;
- $\text{Eval}(1^\lambda, i, x) \rightarrow y$. Eval evaluates $f_i(x)$;
- $\text{Invert}(1^\lambda, i, \text{td}, y) \rightarrow x'$. Invert inverts $y = f_i(x)$ for $x$ with a trapdoor $\text{td}$.

The algorithms should satisfy the following properties, omit also $1^\lambda$ below, and for algorithms with no input after this omission, we omit parentheses, e.g., $\text{Gen}$ instead of $\text{Gen}(\cdot)$.

**Definition 10** (One-Way Trapdoor Function Family): $\text{TDP}$ is an one-way trapdoor function family, if for any $i \leftarrow \text{Gen}(R, \text{td})$, the following holds for $f_i \in \text{TDP}$.

- **Correctness**: For all $x \leftarrow \text{Sample}(i)$,
  \[
  \Pr[\text{Invert}(i, \text{td}, \text{Eval}(i, x)) = x] = 1,
  \]
  where the probability is taken over $\text{Invert}$’s randomness;

- **One-wayness**: For $y \leftarrow \text{Eval}(i, x)$ on all $x$, any adversary $\mathcal{A}$ without $\text{td}$, running in $\text{poly}(\lambda)$ time,
  \[
  \Pr[\mathcal{A}(i, y) = x] < \text{negl}(\lambda),
  \]
  where the probability is taken over $\mathcal{A}$’s randomness.

C. ASYMMETRICALLY HARD FUNCTION CANDIDATES
Asymmetrically hard functions are defined over two resource types, time and memory. We show the candidate constructions: the RSW puzzle and the DIONDON function. Then
prove that they are time-hard and memory-hard asymmetrically hard function families, respectively.

1) THE RSW TIME-LOCK PUZZLE

The four-tuple of algorithms in the RSW puzzle scheme (Gen, Sample, Eval, Asy) is as follows:

- In Gen(1^\lambda, R), interpret R = (Time, u). Sample a larger integer N = pq, where p,q are two prime numbers and |p| = |q| = \lambda. Set outputs as (i, td) \in ((N, u), \phi(N) = (p - 1)(q - 1));
- In Sample(1^\lambda, i), interpret i = (N, u). Sample x \leftarrow \mathbb{Z}_N^*, and set outputs as x;
- In Eval(1^\lambda, i, x), interpret i = (N, u). Compute y = x^{2^u} \mod N and set outputs as y;
- In Asy(1^\lambda, i, td, x), interpret i = (N, u) and td = \phi(N). Compute y' = x^{2^u} \mod \phi(N) \mod N and set outputs as y'.

We formalize the following assumption which was implicitly mentioned in [12].

Assumption 1 (RSW Time-Lock Assumption): Given ((N, u), x), the outputs of honestly executed Gen and Sample, any adversary A runs in \delta(u)-time,

Pr[A((N, u), x) = x^{2^u} \mod N] < negl(\lambda),

where the probability is taken over A's randomness.

By definition, the RSW puzzle scheme is correct and asymmetrically hard. Deriving from Assumption 1 directly, the RSW puzzle scheme is \delta(u)-hard. Thus we have the following lemma:

Lemma 1: An RSW puzzle scheme is a time-hard asymmetrically hard function family if the RSW time-lock assumption holds.

The proof is straightforward. Thus, we skip it and discuss the DIODON function next.

2) THE DIODON FUNCTION

The DIODON function scheme [9] puts forward the idea of the RSW puzzle, creating a list that stores the results of RSW puzzles. Memory-hardness comes from the inability to delete the list, which is achieved by randomly and iteratively computing hash functions over the list.

Informally, the four-tuple of algorithms in the DIODON function scheme (Gen, Sample, Eval, Asy) is:

- Gen runs in the similar manner as in the RSW puzzle scheme, but outputs index with two additional parameters, i \in \{N, (k, l, u)\}, where k is for iterative squarings and l is for iterative hashings;
- Sample works the same and outputs a preimage x;
- Eval takes in (N, (k, l, u)) and x, evaluates u RSW puzzles and stores the results in a list \mathcal{V} = \{V_i\}_{i \in [u]}; where \mathcal{V}_0 = x and for \mathcal{V}_i = x^{2^i} \mod N. Starting from \mathcal{V}_{u-1}, it computes an index j = \mathcal{V}_{u-1} \mod u, and hashes over \mathcal{V}_{u-1} and \mathcal{V}_j, it then iterates l times with the result from previous hash as input and sets the last result as output y;
- Asy takes in (N, (k, l, u)), \phi(N) and x. Instead of storing the results of RSW puzzles, it computes every RSW puzzle involved in the hash function with index i by \mathcal{V}_i = x^{2^i} \mod \phi(N) \mod N. After hashing for l times, it outputs the last result as y'.

Given an RSA group \mathbb{Z}_N^*, parameters k,l,u, asymmetrical key \phi(N) and a preimage x, Eval and Asy from [9] of the DIODON function scheme go as follows:

For the DIODON function, we have the following lemma.

Lemma 2: A DIODON function scheme is a memory-hard asymmetrically hard function.

Proof: Correctness of the DIODON function scheme is by definition. Learning from Definition 5, we modify \delta(u)-hardness to (t(k, l, u), \delta(u))-hardness, where t(k, l, u) represents the time complexity and \delta(u) denotes the memory units consumed. Without the knowledge of \phi(N), by [19], the DIODON function scheme is "optimally linearly memory-hard", which means i.e., t(k, l, u) \times \delta(u) is constant, i.e., Eval either stores the whole list \mathcal{V} or saves memory for a factor f but pay the same factor in time. Finally,
Algorithm 1 The DIODON Evaluation Eval
Input: \((N, (k, l, u)) \) and \(x\);
Output: \(y\).
1: \(V_0 = x\)
2: for all \(i \in \{1, 2, \ldots, u - 1\} \) do
3: \(V_i = V_{i-1} \mod N\)
4: end for
5: temp = \(V_{u-1}\)
6: for all \(i \in \{0, 1, \ldots, l - 1\} \) do
7: \(j = \text{temp} \mod u\)
8: temp = \(H(\text{temp}, V_j)\)
9: end for
10: return \(y = \text{temp}\)

Algorithm 2 The DIODON Asymmetry Asy
Input: \((N, (k, l, u)), \phi(N) \) and \(x\);
Output: \(y'\).
1: \(e = 2^{k(u-1)} \mod \phi(N)\)
2: \(\text{temp} = x^e \mod N\)
3: for all \(i \in \{0, 1, \ldots, l - 1\} \) do
4: \(j = \text{temp} \mod u\)
5: \(e_j = 2^{kj} \mod \phi(N)\)
6: \(\text{temp} = H(\text{temp}, (x^e) \mod N)\)
7: end for
8: return \(y = \text{temp}\)

Asy stores nothing but runs in comparable time with Eval and outputs the same result. Given all above, the DIODON function scheme is a memory-hard asymmetrically hard function.

Remark 4 (Parameterizability): As a summary of this section, we extract “parameterizability” from \(\delta(u)\)-hardness in Definition 9. An asymmetrically hard function is parameterizable if the hardness is tunable without adjusting security parameters. Adjusting the security parameter changes the generation’s initial inputs, affecting algorithms afterward, which is not desirable in practice. Thankfully, \(\delta(u)\)-hardness (or \(t(k, l, u), \delta(u)\)-hardness respectively) is satisfiable in both the RSW puzzle and the DIODON function, thus parameterizable for both schemes. Henceforth, we specify the parameters by denoting them as \(u\)-RSW puzzle and \((k, l, u)\)-DSODON function.

V. CONSTRUCTIONS

We start from a generic construction for the advice-based PoW with trusted generators scheme and instantiate with concrete building blocks from the previous section: the RSW puzzle and the DIODON function for time-hard and memory-hard constructions, respectively. The construction is accomplished by the RSA problem representing the typical one-way trapdoor function. We provide proof that matches the definitions in Section III. Finally, we introduce a multiparty computation protocol (MPC) to distribute the puzzle generation.

A. GENERIC CONSTRUCTION

The intuition behind the generic construction is that, in the trusted generation phase, \(\text{PGen}\) uses a secret trapdoor \(td\) to generate advice and solution before knowing the corresponding aux and \(\text{puz}\). The process from aux to advice evaluates an asymmetrically hard function, and the process from \(\text{puz}\) to \(\text{solution}\) is to invert one-way trapdoor functions. We adopt the notations from object-oriented programming: by \(\text{AHF}\) and \(\text{TDP}\), denoting the asymmetrically hard function family and one-way trapdoor function family, respectively. We omit \(1^k\) below, and for algorithms with no input after this omission, we omit parentheses, e.g., \(\text{Gen}\) instead of \(\text{Gen}(\cdot)\).

Construction 1 (Generic Construction): The triple of algorithms (\(\text{PGen}, \text{Eval}, \text{Solve}\)) of the advice-based PoW with trusted generators scheme works as follows:

- A trusted third party runs \(\text{Setup}(\lambda, R)\):
  - Run \(\text{AHF.Gen}(R) \rightarrow (f_{\text{ahf}}, td)\), such that \(f_{\text{ahf}}\) is an \(R\)-hard asymmetrically hard function;
  - Sample \(a \leftarrow \text{AHF.Sample}(f_{\text{ahf}})\), such that \(a \in \mathcal{X}_{\tau}\), which is in the domain of \(f_{\text{ahf}}\);
  - Compute \(\text{AHF.Asy}(f_{\text{ahf}}, td, a) \rightarrow td_{\tau}\), such that \(td_{\tau} = f_{\text{ahf}}(a)\);
  - Run \(\text{TDP.Gen}(td) \rightarrow g_{\text{tdp}}\) for a one-way trapdoor function \(g_{\text{tdp}}\) that has trapdoor \(td_{\tau}\);
  - Set outputs: \(pp = (f_{\text{ahf}}, g_{\text{tdp}})\) and \(td = (td_{\tau}, td_\lambda)\).

- The trusted third party runs \(\text{PGen}(pp, td)\):
  - Sample \(x \leftarrow \text{TDP.Sample}(g_{\text{tdp}})\), such that \(x \leftarrow \mathcal{X}_{\tau}\), where \(\mathcal{X}_{\tau}\) is domain of \(g_{\text{tdp}}\);
  - Compute \(y \leftarrow \text{TDP.Eval}(g_{\text{tdp}}, x)\) as the instance of the puzzle;
  - Set outputs: \(\text{puz} = y\) and \(\text{aux} = a\).

- Provers run \(\text{Eval} \rightarrow (pp, \text{puz}, \text{aux})\):
  - Phrase \(pp = (f_{\text{ahf}}, g_{\text{tdp}})\), \(\text{puz} = y\) and \(\text{aux} = a\).
  - Run advice \(\leftarrow \text{AHF.Eval}(f_{\text{ahf}}, a)\) for the piece of advice;
  - Run \(x \leftarrow \text{TDP.Invert}(g_{\text{tdp}}, \text{advice}, y)\) for the solution candidate;
  - Set outputs as \(\text{solution} = x\).

- Any party can run \(\text{Verify}(pp, \text{puz}, \text{solution})\) as verifiers:
  - Phrase \(pp = (f_{\text{ahf}}, g_{\text{tdp}})\), \(\text{puz} = y\) and \(\text{solution} = x\);
  - Output 1, if \(g_{\text{tdp}}(x) = y\) holds, and 0 otherwise.

B. CONCRETE CONSTRUCTIONS

We show the transformation from the RSA problem and the RSW puzzle or the DIODON function to an advice-based PoW with trusted generators scheme with time or memory-hardness. We take the RSA problem as the underlying computationally hard problem. The RSW puzzle and the DIODON function are taken as the alternative solving process. Here, we slightly manipulate the generic construction. An additional property (see the remark below) in the RSA problem enables a simpler puzzle generation.

Construction 2 (Time-Hard Construction): Given a security parameter \(\lambda\), a hash function \(\text{hash} : \mathbb{Z}_N \rightarrow \mathbb{Z}_N^k\):
• In Setup$(1^\lambda, R)$, phrase $R = (\text{Time}, u)$. Sample two prime numbers $p, q$ of length $\lambda$, compute $N = pq$ and $\phi(N) = (p-1)(q-1)$. Sample $a \leftarrow \mathbb{Z}_N^*$, compute $a_u = a^{2^x} \mod \phi(N)$ mod $N$ and hash it for $d = \text{hash}(a_u)$. Set outputs: $pp = N$ and $td = (\phi(N), d)$.

• In PGen$(pp, td)$, phrase $pp = N$ and $td = (\phi(N), d)$. Compute $e$, such that $e \cdot d \equiv 1 \mod \phi(N)$. Sample $y \leftarrow \mathbb{Z}_N^*$. Set outputs: $puz = (N, e, y)$ and $aux = (N, a)$.

• In EvalSolve$(pp, puz, aux)$, phrase $pp = N$, $puz = (N, e, y)$ and $aux = (N, a)$; set solution $x'$. Output 1 if $x'^e = y \mod N$ holds, and 0 otherwise.

Construction 3 (Memory-Hard Construction): Except for the additional parameters in the DIODON function, i.e., $k$ and $I$ for iterative squaring and hashing, all remains the same.

• In Setup$(1^\lambda, R)$, phrase $R = (\text{Memory}, u)$. Sample two prime numbers $p, q$ of length $\lambda$, compute $N = pq$ and $\phi(N) = (p-1)(q-1)$. Sample $a \leftarrow \mathbb{Z}_N^*$, evaluate an $(k, l, u)$-DIODON function on $a$ with $\phi(N) = (p-1)(q-1)$ without consuming memory units, then hash it for $d$. Set outputs: $pp = N$ and $td = (\phi(N), d)$.

• PGen, EvalSolve, Verify remains the same.

Remark 5 (RSA as Typical One-Way Trapdoor Function): Notice that in Construction 1, the domain of asymmetrically hard functions $\mathcal{X}_t$ is not necessary to be the same as the domain of one-way trapdoor functions $\mathcal{X}_g$. Moreover, we clarify that the generic construction works due to modifying the one-way trapdoor function families. We enable the TDP.Gen algorithm to take in a trapdoor (td, in the generic case or $d$ in the RSA case) and output the corresponding instance (e in the RSA case) of the one-way trapdoor function. The RSA problem satisfies this property. Thus, it enables our concrete constructions to be practical. Moreover, it provides an efficient puzzle generation given $e \cdot d \equiv 1 \mod \phi(N)$. However, if we provide users both $e$ and $d$ of an RSA problem, we open the door for users to compute $\phi(N)$, which leads the Setup algorithm of the advice-based schemes to be not reusable. We list the constructions with the reusable generation phase as future work.

1) EFFICIENCY

In practical implementations, parameters should be chosen wisely to balance efficiency and security. Start from an observation on lower bounds of solving the RSA problem, given the general number field sieve algorithm [20], considered to be one of the fastest algorithms for factoring large integers. Although we do not know if solving an RSA problem is as hard as factoring $[7], [8]$. 

Assumption 2: Given an RSA group $\mathbb{Z}_N^*$ with $N = pq$, $|l| = |q| = \lambda$, and an instance of RSA problem $(e, y)$, for any algorithm Alg without $p, q$ runs in $\omega(2^{\lambda/\epsilon})$ time, the probability of producing an $x'$, such that $x'^e = y \mod N$ is negligible. Where $\epsilon$ is a properly large coefficient.

Assumption 2 draws the line for soundness, thus we adjust the parameters, $u$ and $T$ with a precise relation: $T(\lambda, u) \ll 2^{\lambda/\epsilon}$. With this in mind, the efficiency of the concrete constructions is as follows:

• Setup and PGen sample two elements from $\mathbb{Z}_N^*$, compute a RSW puzzle via asymmetrical evaluation, a hash function, and an RSA evaluation. The cost is determined by two exponentiations in $\mathbb{Z}_N^*$, thus in $\tilde{O}(\lambda)^4$.

• EvalSolve computes an RSW puzzle via regular evaluation, a hash function, and an RSA inversion with the inverse key. The cost is determined by evaluating the RSW puzzle, which is $u$, thus matching the designed difficulty $T(\lambda, u)$.

• Verify checks the correctness of an RSA problem with one exponentiation in $\mathbb{Z}_N^*$. Thus the cost is in $\tilde{O}(\lambda)$.

2) PROOFS

Without loss of generality, we give the proof of soundness and hardness of the time-hard construction, under Assumption 2 and Assumption 1. Finally, for difficulty, we prove it with soundness, hardness, and a collision and preimage-resistant hash. Thus, we complete the arguments from Section III.

Theorem 1: Assuming the existence of a collision and preimage-resistant hash function hash, Construction 2 of advice-based PoW with trusted generators scheme with time-hardness satisfies the following properties:

• Correctness. By Construction 2;

• Soundness. Under Assumption 2;

• Time-hardness. Under Assumption 1;

• Difficulty. By soundness, time-hardness and the properties of the hash function hash.

Proof: Correctness is obvious due to the construction. We prove soundness, time-hardness, and difficulty as follows:

a: SOUNDNESS

Recall the generation of $puz$ and $aux$ from Setup and PGen, the uniform sampling of $a$ and $hash$ guarantees the uniform distribution of $d \in \mathbb{Z}_N^*$, thus the uniformity of $e$. Moreover, $x$ is also uniformly sampled from $\mathbb{Z}_N^*$, thus the uniformity of $y$. If an adversary intends to produce an acceptable solution $x'$ with an arbitrary piece of advice $\alpha'$, which by collision-resistance of hash, Pr$[\alpha' = d] < \text{neg}(\lambda)$. To break soundness is to invert the RSA problem $(N, e, y)$ with no other information. By Assumption 2, even the adversary runs in much longer than $T(\lambda, u)$ but less than $2^{\lambda/\epsilon}$ cannot invert $(N, e, y)$, thus cannot produce an acceptable solution $x'$.

b: TIME-HARDNESS

Given an RSA problem $(N, e, y)$, the adversary queries $\{aux\}_q$ under its choice and obtains $\{\text{advice}\}_q$ from the challenger. Such a pre-processing procedure comes from the
adversary may see polynomial many (aux, advice) pairs from previous evaluations. To break the time-hardness, the adversary needs to produce a valid advice’ on the given aux*. However, as assumed in Assumption 1, the probability that one can evaluate an RSW puzzle plainly with less than δ(u)-time is negligible. Moreover, hash guarantees that collisions happen among {advice}q and advicee with negligible probability. Thus time-hardness holds for Construction 2.

c: DIFFICULTY
Recall our arguments in Section III. The “advice unpredictability”, i.e., one cannot predict advice before fully evaluating on aux, is vital for the construction to be difficult. Otherwise, the leaked information of advice may accelerate the solving procedure of puz. With a hash function hash, being collision-resistant and preimage-resistant, any adversary tries to guess any bits in advice from aux is equivalently breaking the preimage resistance of the hash function. Consequently, the difficulty relies on the hardness, i.e., if there exists an adversary who breaks the difficulty game in Figure 2, we can construct an adversary that breaks the hardness game in Figure 1. To prove this, we consider the following game:

1) The difficulty game, where puz ← PGen is given to the adversary at first step with corresponded R and pp;
2) An intermediate game, where puz is an arbitrary piece of the puzzle such that independent with PGen, R and pp;
3) The hard game, where only R and pp are sent to the adversary. We regard such situation as puz =⊥.

The probability loss between each game is negligible of λ. Thus difficulty holds as long as hardness holds.

C. DISTRIBUTED PUZZLE GENERATION
Inspired by Kopp et al. [21], we briefly introduce a novel method, i.e., the MPC protocol, to distribute the trusted generation phase in construction 2. Thus enhance the construction to conform to our Definition 2.

An MPC protocol allows a set of mutually distrusting parties to collaboratively compute a function with private input values. For example, given n parties holding their secret si respectively and intending to evaluate some f(s1, · · · , sn), the MPC protocol secures that participants learn nothing but their own inputs and the solution f(s1, · · · , sn).

In our case, we require a semi-honest secure MPC protocol, i.e., adversaries try to gather as much information about the underlying inputs and intermediate outputs but honestly follow the protocol. As in Kopp et al.’s work, we use the secret-sharing MPC protocol, e.g., [11] as a foundation. For this, we define an MPC protocol MPC involving a triple of algorithms (Setup, Share, Reveal), where:

- (pp) ← Setup(1λ, n) is a randomized algorithm. It takes a security parameter λ and the number of participants n as input and outputs the public parameters pp for the system;
- (s) ← Share(pp, s) shares the secret s among the n participants using a secret sharing scheme such that each of the n participants with index i receives one share.
- We use (s) to denote the vector of the shares of s. Note that the Share algorithm can be executed by any party with access to the pp;
- s ← Reveal(pp, (s)) reconstructs the value s from the shares (s). The MPC participants send their shares to an external party who executes the Reveal algorithm and learn the clear value s.

Due to the existence of arbitrarily computable functions MPC protocols, by abuse of notation, we apply computable functions on secret shares to denote the computation of the secret shares of the result of the function, i.e., f((s1), · · · , (sn)) by f((s1), · · · , (sn)). The clear value f(s1, · · · , sn) is not revealed by this operation. Note that pp may be needed for this computation, but is omitted to simplify our notation.

We show the modification on PGen algorithm in Construction 2 with the early defined MPC protocol as follows.

- Each party choose a secret random value ai ← ZN and distributes the secret shares with Share(MPC, pp, ai);
- All parties compute (a) = ∑ (ai) using MPC;
- Globally sample y ← ZN;
- All parties distribute (2Ny mod φ(N));
- All parties compute (sk) = hash((a)12Ny mod φ(N));
- All parties compute the reverse (e) such that (e) · (sk) ≡ 1 mod φ(N);
- Reveal the puzzle by e ← Reveal(MPC, pp, (e)) and a ← Reveal(MPC, pp, (a));
- Return the PoW puzzle (N, ((e, y), a)).

The transformation of the generic and memory-hard constructions will be left as future work.

VI. CONCLUSION
This paper proposes an advice-based framework (with trusted generators) for the PoW scheme. It presents a generic construction for the weakened variant based on the computationally hard assumption resisting dedicated hardware. Our construction is based on one-way trapdoor functions and asymmetrically hard functions. We prove security for a concrete construction based on RSA assumption and the RSW puzzle. Finally, we substitute the trusted generator from the weakened scheme with a distributed puzzle generation protocol. Therefore, our framework and constructions circumvent the problems caused by the hash function-based PoW construction.

ACKNOWLEDGMENT
An earlier version of this paper was presented at the 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).

REFERENCES
[1] C. Dwork and M. Naor, “Pricing via processing or combating junk mail,” in Proc. 12th Annu. Int. Cryptol. Conf. Adv. Cryptol. (Lecture Notes in Computer Science), vol. 740, E. F. Brickell, Ed., Santa Barbara, CA, USA. Berlin, Germany: Springer, Aug. 1992, pp. 139–147, doi: 10.1007/3-540-48071-4_10.
[2] S. Nakamoto. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. [Online]. Available: http://bitcoin.org/bitcoin.pdf
[3] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan, “Proofs of work from worst-case assumptions,” in *Proc. 38th Ann. Int. Cryptol. Conf. Adv. Cryptol.* (Lecture Notes in Computer Science), vol. 10991, H. Shacham and A. Boldyreva, Eds., Santa Barbara, CA, USA, Cham, Switzerland: Springer, Aug. 2018, pp. 789–819, doi: 10.1007/978-3-319-96884-1_26.

[4] A. Kayaç, A. Russell, B. David, and R. Oliynyk, “Ouroboros: A provably secure proof-of-stake blockchain protocol,” in *Proc. 37th Ann. Int. Cryptol. Conf. Adv. Cryptol.* (Lecture Notes in Computer Science), vol. 10401, J. Katz and H. Shacham, Eds., Santa Barbara, CA, USA, Cham, Switzerland: Springer, Aug. 2017, pp. 357–388, doi: 10.1007/978-3-319-63688-7_12.

[5] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, “Proofs of space,” in *Proc. 35th Ann. Cryptol. Conf. Adv. Cryptol.* (Lecture Notes in Computer Science), vol. 9216, R. Gennaro and M. Robshaw, Eds., Santa Barbara, CA, USA, Berlin, Germany: Springer, Aug. 2015, pp. 585–605, doi: 10.1007/978-3-662-48000-7_29.

[6] Atieniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song, “Provable data possession at untrusted stores,” in *Proc. Conf. Comput. Commun. Secur.*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., Alexandria, VA, USA, Oct. 2007, pp. 598–609, doi: 10.1145/1315245.1315318.

[7] D. Boneh and R. Venkatesan, “Breaking RSA may not be equivalent to factoring,” in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn. Adv. Cryptol.* (Lecture Notes in Computer Science), vol. 1403, K. Nyberg, Ed., Espoo, Finland. Berlin, Germany: Springer, May/Jun. 1998, pp. 59–71, doi: 10.1007/BFb0054417.

[8] D. R. L. Brown, “Breaking RSA may be as difficult as factoring,” *J. Cryptol.*, vol. 29, no. 1, pp. 220–241, 2016, doi: 10.1007/s00145-014-9202-9.

[9] A. Biryukov and L. Perrin, “Symmetrically and asymmetrically hard cryptography,” in *Proc. 23rd Int. Conf. Theory Appl. Cryptol. Inf. Secur. Adv. Cryptol.* (Lecture Notes in Computer Science), vol. 10626, T. Takagi and T. Peyrin, Eds., Hong Kong. Cham, Switzerland: Springer, Dec. 2017, pp. 417–445, doi: 10.1007/978-3-319-70700-6_15.

[10] R. Rivest, A. Shamir, and D. A. Wagner, “Time-lock puzzles and timed-release crypto,” Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep., 1996.

[11] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, “Multiparty computation from somewhat homomorphic encryption,” in *Proc. 32nd Ann. Cryptol. Conf. Adv. Cryptol.* (Lecture Notes in Computer Science), vol. 7417, R. Safavi-Naini and R. Canetti, Eds., Santa Barbara, CA, USA, Cham, Switzerland: Springer, Aug. 2012, pp. 643–662, doi: 10.1007/978-3-642-32009-5_38.

[12] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. “Verifiable delay functions,” in *Proc. 38th Ann. Int. Cryptol. Conf. Adv. Cryptol.* (Lecture Notes in Computer Science), vol. 10991, H. Shacham and A. Boldyreva, Eds., Santa Barbara, CA, USA, Cham, Switzerland: Springer, Aug. 2018, pp. 757–788, doi: 10.1007/978-3-319-96884-1_25.

[13] A. Biryukov and D. Khovratovich, “Egalitarian computing,” in *Proc. 25th Secur. Symp.*, T. Holz and S. Savage, Eds., Austin, TX, USA, Aug. 2016, pp. 315–326. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/biryukov.

[14] F. Coelho, A. Larroche, and B. Colin, “Itsuku: A memory-hardened password release crypto,” Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep., 1996.

[15] MARIO LARANGEIRA received the degree in computer engineering from the Technological Institute of Aeronautics (ITA), in 2004, and the M.S. and Ph.D. degrees from the Tokyo Institute of Technology, in 2008 and 2013, respectively. He is currently a Cryptographer working as a Research Fellow at Input Output Hong Kong (IOHK), and Specially Appointed Associate Professor in security, cryptography and blockchain systems with the Department of Mathematical and Computing Science, Tokyo Institute of Technology. His research interests include published works on the area of blockchain-based systems, public-key cryptographic schemes, secure multiparty computation protocols, and security models for provable security.

[16] KEISUKE TANAKA received the B.E. degree in computer science from Yamanashi University, in 1992, and the M.S. and Ph.D. degrees from the Japan Advanced Institute of Science and Technology, in 1994 and 1997, respectively. He is currently a Professor with the Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. Before joining the Tokyo Institute of Technology, he was a Research Engineer with the NTT Information Sharing Platform Laboratories.

[17] XIANGYU SU received the bachelor’s degree from the Department of Information Theory and Data Science, Nankai University, China, in 2016, and the master’s degree from the Department of Mathematical and Computing Science, Tokyo Institute of Technology, Japan, in 2019, where he is currently pursuing the Ph.D. degree.