How to Prove Work: With Time or Memory (Extended Abstract)

Xiangyu Su
Tokyo Institute of Technology
Tokyo, Japan
su.x.ab@m.titech.ac.jp

Mario Larangeira
IOHK, Tokyo Institute of Technology
Tokyo, Japan
mario@c.titech.ac.jp
mario.larangeira@iohk.io

Keisuke Tanaka
Tokyo Institute of Technology
Tokyo, Japan
keisuke@is.titech.ac.jp

***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 kinds of participants, provers and verifiers. Provers intend to solve a puzzle with a solution, while verifiers are in charge of checking the puzzle's correctness and solution pair. The 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 on the original meaning of "proof of work". Under a trusted setting, our proposed framework and its constructions are 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 hardwares.**

***Index Terms*—Blockchain, Proof-of-Work, Hard Primitives**

I. Introduction

When introduced by Dwork and Naor in the early 1990s [1], the proof-of-work (PoW) scheme was to combat junk mail. The idea is simple: Senders evaluate a moderately hard function as "proof of work" to gain access to an e-mail service. A sender is malicious if it intends to send large amounts of junk e-mail. Such malicious senders have to donate a significant portion of their computing power during the evaluation. Thus the scheme mitigates frivolous use. Almost 20 years later, the work of Bitcoin by Satoshi Nakamoto [2] used a similar idea, *i.e.*, the PoW scheme, to mitigate the "Sybil Attack" in the 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 attacks. The innovative work refueled the interest of the community in researching PoW-related schemes.

A. Background and Motivations

Participants in a PoW scheme are provers and verifiers. Provers solve a given puzzle instance (puzzle, T) with a solution $\mathsf{solution}$, where T is the difficulty of puzzle. By difficulty, we mean computational complexity. Verifiers check the $(\mathsf{puzzle}, \mathsf{solution})$ pair's correctness, according to the corresponding difficulty T. The puzzle should be moderately hard, *i.e.*, the puzzle costs neither too much computing power nor too little.

A widely adopted construction is base on hash functions: Upon receiving a puzzle and a target value for difficulty, provers try to find a solution. The hash of the $(\mathsf{puzzle}, \mathsf{solution})$ pair should be less than the target value T. For example, in the Bitcoin system, the hash value should begin with at least T zeros.

In general, the hash-based construction requires massive energy consumption in large-scale implementation due to the massive number of executions required of the hash function. Another issue comes from the hardware dedicated to computing hashes, which sacrifices the fairness and the decentralization of the currency systems for financial rewards. Since such hardware, *e.g.*, the "ASIC" is relatively expensive and produces much more hashing power than the regular CPUs, which leads to the centralization of hashing power to specific groups who can afford the hardware. Moreover, the hardware is highly dedicated, 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 issues highlight the need for more research on alternatives for 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]. The two main barriers are:

1) The puzzle can be overwhelmingly difficult given a similar security parameter from the hash function based construction;
2) It is infeasible to fine-tune the difficulty without readjusting the security parameters of the underlying hard problems.

This work was supported by the Input Output Cryptocurrency Collaborative Research Chair funded by IOHK.

Our work mitigates the difficulty of the computationally hard problems with an alternative solving process while maintaining the difficulty of being moderate and tunable. We achieve this at the expense of involving a trusted puzzle generation phase. Our construction may find its applications in hybrid settings where a trusted third party, *e.g.*, a company which maintains a consortium blockchain, can provide puzzle generation services.

B. Our Approaches

The straightforward idea is to use trapdoor functions. We push the argument as follows. For a secure trapdoor function, to find the trapdoor is comparable in computational complexity to inverting the function, while the function is trivially invertible if the trapdoor is given. Henceforth, we design a moderately hard approach to offer the trapdoor. Precisely, the unified framework by Biryukov and Perrin [7] of hard cryptographic primitives gives us a way to ask provers to evaluate a hard function before obtaining the trapdoor. The unified framework brings even more advantages since the definition is along three axes in [7]: 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 [7];
- Two concrete constructions, which achieve time or memory hardness for the PoW schemes provably.

Our work is in the random-access machine model. The framework provides a generic transformation from asymmetrically hard functions [7] to PoW schemes. Thus it fills the gap between cryptographically hard functions and practical PoW schemes. Similar to the original PoW scheme. An advice-based PoW scheme associates with an efficient puzzle generation, a solving algorithm that matches the computing power to a designed difficulty, and an efficient verification that should be at most logarithmic in the solving's computation complexity. We novelly handle the puzzle by an instance of computationally hard problems and a preimage for hard functions. The solving algorithm first evaluates the hard problem to obtain a piece of advice, *e.g.*, the trapdoor of a trapdoor function, and finally, solve the puzzle with the advice. However, there is no guarantee of the relation between the advice and the trapdoor without a trusted puzzle generation phase in practical constructions. Thus, we include a trusted generator in the framework for providing correct puzzle generation services.

For the contributions, we present a generic construction based on a special kind of one-way trapdoor functions and asymmetrically hard functions with an efficient generation under a trusted setting. We require the one-way trapdoor function can be generated from a given trapdoor, *e.g.*, the RSA problem. Henceforth, by instantiating it with the RSA problem, we show two concrete constructions based on the RSW time-lock puzzle [8] (RSW puzzle) and the DIODON function [7] for advice-based PoW with trusted generators scheme with time and memory-hardness, respectively. We argue that iterative squaring (of the RSW puzzle) is relatively easy so that dedicated hardware will not surpass standard CPUs vastly. We only demonstrate the proofs and implementation for the time-hard construction due to the page limitation. The other follows similarly.

C. Related Work

We investigate the constructions of the proof-of-work scheme with time-hardness or memory-hardness. For time-hardness, we have the time-lock puzzle scheme [8] and verifiable delay functions [9], while for memory-hardness, we have the memory-hard PoW scheme [10], [11] and the proof-of-space scheme [5].

a) Time-lock puzzle and verifiable delay functions.: The time-lock puzzle, introduced by Rivest, Shamir, and Wagner [8], is already a semi-PoW in advance, lacking only the efficient public verifiability. By adding a knowledge proof protocol, Boneh et al. [9] 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 Wesolowski [12] and Pietrzak [13] 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 [14].

b) Memory-hard PoW schemes and proof-of-space schemes.: Aiming to reduce ASICs' impact, Biryukov and Khovratovich [10], 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 [11], 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 and 1^λ for clarifying the length of inputs. PPT denotes probablistic polynomial time. The function $\mathsf{negl}(\lambda)$ is negligible of λ, if for every positive integer c, there exists a large enough λ, such that $\mathsf{negl}(\lambda) < \lambda^{-c}$ holds. Given a set $\mathcal{X}$, $x \xleftarrow{\$} \mathcal{X}$ means x is randomly and uniformly sampled from $\mathcal{X}$; while for an algorithm Alg, $x \leftarrow \mathsf{Alg}$ denotes that x is assigned to the outputs of Alg on fresh randomness. For an index set, $[k] = \{0, 1, \ldots, k-1\}$.

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

$\mathsf{R} = (\rho, u)$, the resource requirements of a computational task, $\rho = \{\text{Time}, \text{Memory}\}$ 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 $0 < \epsilon \leq 1$, such that $\delta(u)$ is lower bounded by u^{ϵ}.

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 $\delta(u)$ 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: $T(\lambda, u)$, given $\mathsf{R} = (\rho, u)$. In general, a PoW puzzle is $T(\lambda, u)$-difficult,

- for $\rho =$ Time, if solving the puzzle takes no less than $T(\lambda, \delta(u))$ time;
- for $\rho =$ Memory, if solving the puzzle takes no less than $T(\lambda)$ time and $\delta(u)$ memory units.

Remark (Size Restriction of u): u must be subexponential 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 show a weakened variant, which requires a trusted puzzle generation phase, the advice-based PoW with trusted generators. 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 $(\mathsf{Setup}, \mathsf{PGen}, \mathsf{EvalSolve}, \mathsf{Verify})$. 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 $(\mathsf{Setup}, \mathsf{PGen}, \mathsf{EvalSolve}, \mathsf{Verify})$ works as follows:

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

Remark (Inherent Difficulty): Instead of in PGen's outputs, we regard the difficulty $T(\lambda, u)$ as an inherent property of $(\mathsf{puz}, \mathsf{aux})$ generated with parameters pp and $\mathsf{R} = (\rho, u)$, and $T(\lambda, u)$ is only used for security analysis.

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

Definition 3 (Advice-Based PoW with Trusted Generators): The difference to Definition 2 lies in the Setup and PGen algorithm:

- $\mathsf{Setup}(1^{\lambda}, \mathsf{R}) \to (\mathsf{pp}, \mathsf{td})$, where td is an additional secret trapdoor for the puzzle generator;
- $\mathsf{PGen}(\mathsf{pp}, \mathsf{td}) \to (\mathsf{puz}, \mathsf{aux})$. A trusted generator runs KGen with td from the Setup algorithm and generates $(\mathsf{puz}, \mathsf{aux})$;
- EvalSolve 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 solution output from EvalSolve should be accepted by Verify with sufficient possibility for a properly generated $(\mathsf{puz}, \mathsf{aux})$.

Definition 4 (Correctness): An advice-based PoW scheme is correct, if Setup and PGen are properly executed, if $\mathsf{EvalSolve}(\mathsf{pp}, \mathsf{puz}, \mathsf{aux}) \to \mathsf{solution}$,

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \mathsf{puz}, \mathsf{solution}) = 0\right] < \mathsf{negl}(\lambda).$$

B. Security

The primary security of a proof-of-work scheme is the difficulty, *i.e.*, given a pair of $(\mathsf{puz}, \mathsf{aux})$ with difficulty $T(\lambda, u)$

of resource type ρ, except only negligible probability, an adversary can obtain an acceptable solution with fewer resource units.

We introduce two fundamental properties: soundness and hardness, focusing on puz and aux (Solve and Eval), respectively. However, soundness and hardness are insufficient to achieve difficulty because the intermediate steps of the $eval$ algorithm may leak information for the Solve algorithm. Thus one can obtain a correct solution without the designed advice. Henceforth, we require a third property called "advice unpredictability". We show this property concerning concrete constructions in Section V.

1) Soundness: Recall the structure of Solve, soundness relies on the fact that without a piece of valid advice, provers even run in a super-polynomial time of λ can only find a valid solution with negligible probability. Precisely, for any adversary $\mathcal{A}$, running in time $t(\lambda)$ that is larger than $T(\lambda, u)$, on input only the puzzle puz (aux is hidden to the adversary), the probability $\mathcal{A}(\mathsf{pp}, \mathsf{puz}) \to \mathsf{solution}'$ such that $\mathsf{Verify}(\mathsf{pp}, \mathsf{puz}, \mathsf{solution}') = 1$ is negligible of λ. The definition of soundness is unified because we focus on the run time for an adversary to produce an acceptable solution.

Definition 5 (t-Sound): An advice-based PoW with trusted generators scheme is t-sound, if the following holds for any adversary $\mathcal{A}$ runs less than time t:

$$\Pr[\mathsf{Verify}(\mathsf{pp}, \mathsf{puz}, \mathsf{solution}') = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{R})$, $(\mathsf{puz}, \mathsf{aux}) \leftarrow \mathsf{PGen}(\mathsf{pp})$ and $\mathsf{solution}' \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{puz})$. The probability takes over randomness of PGen and $\mathcal{A}$.

Remark (Achieveable 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 λ. 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 λ, *i.e.*, $O(2^{\lambda/c})$ with sufficiently large c since no algorithm running in t-time can solve the underlying hard problem, which is puz in our case. $t << O(2^{\lambda/c})$ guarantees that provers cannot crack puz for an acceptable solution with brute force.

2) Hardness: Soundness captures the impossibility for solving the puzzle without a valid piece of advice, and hardness means that no prover 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 $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ and a challenger $\mathcal{CH}$:

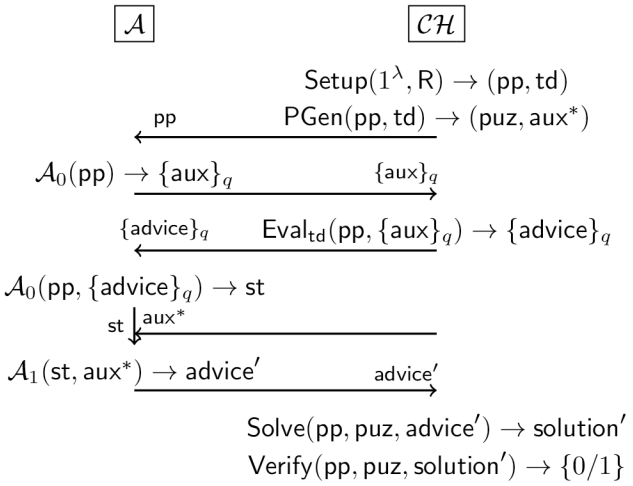

Fig. 1: Hardness Game

The adversary 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 $\{\mathsf{aux}\}_q$, where q is the maximum number of the adversary's query times.

Definition 6 (δ-Time-Hard): An advice-based PoW with trusted generation scheme is δ-time-hard, if for any algorithm $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, where $\mathcal{A}_0$ runs in $O(\mathsf{poly}(\lambda, u))$ time and $\mathcal{A}_1$ runs in $\delta(u)$ time, the probability of $\mathcal{A}$ winning the hardness game is negligible of λ and the probablity is taken over PGen, $\mathcal{A}$ and Verify's randomness.

As shown by Hellman [15], 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, δ)-Memory-Hard): An advice-based PoW with trusted generation scheme is (t, δ)-memory-hard, if for any algorithm $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, where $\mathcal{A}_0$ has running time $O(\mathsf{poly}(\lambda, u))$ and $\mathcal{A}_1$ runs in time $t(\lambda)$ using up to $\delta(u)$ memory units, the probability of $\mathcal{A}$ winning the hardness game is negligible of λ and is taken over PGen, $\mathcal{A}$ and Verify's randomness.

3) Difficulty: In the difficulty game, we provide the adversary a full power with puz and the proper aux^*. While in the soundness definition, the adversary only obtains puz; in the hardness game, the adversary only has aux^*. Similar to the hardness game, the adversary can query $\{\mathsf{aux}\}_q$ to the challenger for $\{\mathsf{advice}\}_q$. We give the following game-based definition for the difficulty in Figure 2.

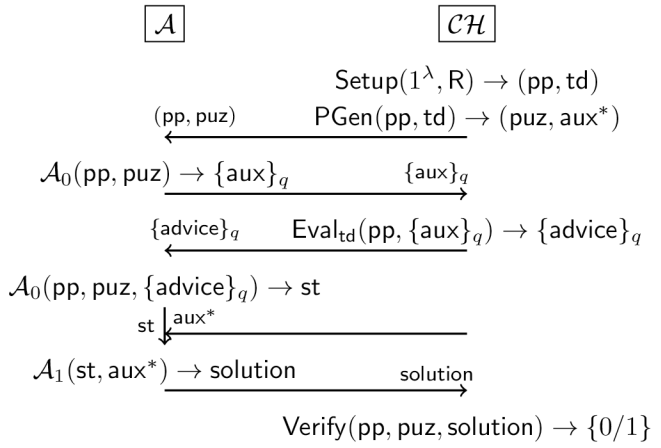

Fig. 2: Difficulty Game

The adversary is said to win the difficulty game if the challenger outputs "1" with non-negligible probability and $\mathsf{aux}^* \notin \{\mathsf{aux}\}_q \wedge \mathsf{Eval}(u, \mathsf{aux}^*) \notin \{\mathsf{advice}\}_q$. Notice that, in the difficulty game, the adversary can process puz in advance.

Definition 8 ($T(\lambda, u)$-Difficult): An advice-based PoW with trusted generation scheme is $T(\lambda, u)$-difficult, if for any algorithm $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, where $\mathcal{A}$ runs in $T(\lambda, \delta(u))$ time ($T(\lambda)$ time and $\delta(u)$ memory units) the probability of $\mathcal{A}$ winning the difficulty game is negligible of λ and is taken over PGen, $\mathcal{A}$ and Verify's randomness.

Intuitively, soundness indicates that if an adversary is provided with an arbitrary advice' under its choices with the given puz, it is infeasible to compute an acceptable solution within super-polynomial time. Hardness indicates that even if the adversary can query on auxs under its choices, it cannot produce a piece of valid advice on the very aux^* without donating enough computing power (resource).

IV. Building Blocks

Here we introduce the building blocks for constructions, starting from asymmetrically hard function family [7], denoted by AHF; and one-way trapdoor function family, denoted by TDP.

A. Asymmetrically Hard Functions

We formalize the implied definitions from [7] for further use and show the candidate constructions. An asymmetrically hard function family involves a four-tuple of algorithms $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Asy})$:

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

Taken together, these algorihtms should satisfy the following properties, omitting 1^λ below.

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

- Correctness: For all $x \leftarrow \mathsf{Sample}(i)$,
$$\Pr[\mathsf{Eval}(i, x) = f_i(x)] = 1,$$
where the probability is taken over 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 td, samples $x \leftarrow \mathsf{Sample}(i)$ and runs with less than $\delta(u)$-resource units of $\rho = \text{Time}$ or $(t(u), \delta(u))$-resource units of $\rho = \text{Memory}$,
$$\Pr[\mathcal{A}(i, x) = f_i(x)] < \mathsf{negl}(\lambda),$$
where the probability is taken over $\mathcal{A}$'s randomness.
- Asymmetrical hardness: For all $x \leftarrow \mathsf{Sample}(i)$, Asy runs with $O(\lambda)$-resource units of ρ and satisfies correctness:
$$\Pr[\mathsf{Asy}(i, x) = f_i(x)] = 1,$$
where the probability is taken over Asy's randomness.

B. Special One-Way Trapdoor Functions

We refine the textbook definition of the one-way trapdoor functions to fit in our generic constructions. More concretely, the generator of the one-way trapdoor function family takes a trapdoor as input and outputs the corresponding a one-way trapdoor function or $\perp$ if there exists no such a function. We also adapt a similar syntax used for the asymmetrically hard functions. A special one-way trapdoor function family involves a four-tuple of algorithms $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Invert})$:

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

Similarly, these algorithms should satisfy the following properties, omit also 1^λ below, and for algorithms with no input after this omission, we omit parentheses, *e.g.*, Gen instead of $\mathsf{Gen}()$.

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

- Correctness: For all $x \leftarrow \mathsf{Sample}(i)$,
$$\Pr[\mathsf{Invert}(i, \mathsf{td}, \mathsf{Eval}(i, x)) = x] = 1,$$
where the probability is taken over Invert's randomness.
- One-wayness: For $y \leftarrow \mathsf{Eval}(i, x)$ on all $x \leftarrow \mathsf{Sample}(i)$, any adversary $\mathcal{A}$ without td, running in $\mathsf{poly}(\lambda)$ time,
$$\Pr[\mathcal{A}(i, y) = x] < \mathsf{negl}(\lambda),$$
where the probability is taken over $\mathcal{A}$'s randomness.

C. Candidates of Asymmetrically Hard Functions

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

1) RSW time-lock puzzle.: The four-tuple of algorithms in the RSW puzzle scheme $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Asy})$ is as follows:

- In $\mathsf{Gen}(1^\lambda, \mathsf{R})$, interpret $\mathsf{R} = (\mathsf{Time}, u)$. Sample a larger integer $N = pq$, where p,q are two prime numbers and $|p| = |q| = \lambda$. Set outputs as $(i, \mathsf{td}) \stackrel{\Delta}{=} ((N, u), \phi(N) = (p-1)(q-1))$;
- In $\mathsf{Sample}(1^\lambda, i)$, interpret $i = (N, u)$. Sample $x \leftarrow \mathbb{Z}_N^*$ and set outputs as x;
- In $\mathsf{Eval}(1^\lambda, i, x)$, interpret $i = (N, u)$. Compute $y = x^{2^u} \mod N$ and set outputs as y;
- In $\mathsf{Asy}(1^\lambda, i, \mathsf{td}, x)$, interpret $i = (N, u)$ and $\mathsf{td} = \phi(N)$. 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 [9].

Assumption 4.1 (RSW Time-Lock Assumption): Given $((N, u), x)$, the outputs of honestly executed Gen and Sample, any adversary $\mathcal{A}$ runs in $\delta(u)$-time,

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

where the probability is taken over $\mathcal{A}$'s randomness.

By definition, the RSW puzzle scheme is correct and asymmetrically hard. Deriving from Assumption 4.1 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) DIODON function.: The DIODON function scheme [7] puts forward the idea of the RSW puzzle, creating a list that stores the results of evaluating RSW puzzles. Memory-hardness comes from the inability to delete the list, achieved by computing hash functions over the list randomly and iteratively.

Informally, the four-tuple of algorithms in the DIODON function scheme $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Asy})$:

- Gen runs in the similar manner as in the RSW puzzle scheme, but outputs index with two additional parameters, $i \stackrel{\Delta}{=} (N, (k, l, u))$, 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 $V = \{V_i\}_{i \in [u]}$, where $V_0 = x$ and for $i \in \{1, 2, \ldots, u-1\}$, $V_i = x^{2^{(k \cdot i)}} \mod N$. Notice that $V_i = V_{i-1}^{2^k} \mod N$. Starting from V_{u-1}, it computes an index $j = V_{u-1} \mod u$, and hashes over V_{u-1} and 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 $x^{2^{(k \cdot i)} \mod \phi(N)} \mod N$. 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 [7] of the DIODON function scheme go as follows:

Algorithm 1 The DIODON Evaluation Eval

Input: $(N, (k, l, u))$ and x;
Output: y.

$V_0 = x$
for all $i \in \{1, 2, \ldots, u-1\}$ **do**
    $V_i = V_{i-1}^{2^k} \mod N$
end for
$\mathsf{temp} = V_{u-1}$
for all $i \in \{0, 1, \ldots, l-1\}$ **do**
    $j = \mathsf{temp} \mod u$
    $\mathsf{temp} = H(\mathsf{temp}, V_j)$
end for
return $y = \mathsf{temp}$

Algorithm 2 The DIODON Asymmetry Asy

Input: $(N, (k, l, u))$, $\phi(N)$ and x;
Output: y'.

$e = 2^{k(u-1)} \mod \phi(N)$
$\mathsf{temp} = x^e \mod N$
for all $i \in \{0, 1, \ldots, l-1\}$ **do**
    $j = \mathsf{temp} \mod u$
    $e_j = 2^{kj} \mod \phi(N)$
    $\mathsf{temp} = H(\mathsf{temp}, (x^{e_j} \mod N))$
end for
return $y = \mathsf{temp}$

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 7, 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 [16], 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 V or saves memory for a factor f but pay the same factor in time. Finally, 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 (Parameterizability): As a summary of this section, we extract "parameterizability" from $\delta(u)$-hardness in Defintion 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)-DIODON 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 special one-way trapdoor function. We show a sketch of the proof that matches the definitions in Section III.

A. Generic Construction

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

Construction 1 (Generic Construction): The triple of algorithms $(\mathsf{PGen},\mathsf{EvalSolve},\mathsf{Verify})$ of the advice-based PoW with trusted generators scheme works as follows:

- A trusted third party runs $\mathsf{Setup}(1^\lambda,\mathsf{R})$:
  - Run $\mathsf{AHF.Gen}(\mathsf{R}) \to (f_{\mathsf{ahf}},\mathsf{td}_f)$, such that f_{ahf} is an R-hard asymmetrically hard function;
  - Sample $a \leftarrow \mathsf{AHF.Sample}(f_{\mathsf{ahf}})$, such that $a \in \mathcal{X}_f$, which is in the domain of f_{ahf};
  - Compute $\mathsf{AHF.Asy}(f_{\mathsf{ahf}},\mathsf{td}_f,a) \to \mathsf{td}_g$, such that $\mathsf{td}_g = f_{\mathsf{ahf}}(a)$;
  - Run $\mathsf{TDP.Gen}(\mathsf{td}_g) \to g_{\mathsf{tdp}}$ for a one-way trapdoor function g_{tdp} that has trapdoor td_g;
  - Set outputs: $\mathsf{pp} = (f_{\mathsf{ahf}}, g_{\mathsf{tdp}})$ and $\mathsf{td} = (\mathsf{td}_f, \mathsf{td}_g)$
- The trusted third party runs $\mathsf{PGen}(\mathsf{pp},\mathsf{td})$:
  - Sample $x \leftarrow \mathsf{TDP.Sample}(g_{\mathsf{tdp}})$, such that $x \leftarrow \mathcal{X}_g$, where $\mathcal{X}_g$ is domain of g_{tdp};
  - Compute $y \leftarrow \mathsf{TDP.Eval}(g_{\mathsf{tdp}}, x)$ as the intance of the puzzle;
  - Set outputs: $\mathsf{puz} = y$ and $\mathsf{aux} = a$.
- Provers run $\mathsf{EvalSolve}(\mathsf{pp},\mathsf{puz},\mathsf{aux})$:
  - Phrase $\mathsf{pp} = (f_{\mathsf{ahf}}, g_{\mathsf{tdp}})$, $\mathsf{puz} = y$ and $\mathsf{aux} = a$.
  - Run $\mathsf{advice} \leftarrow \mathsf{AHF.Eval}(f_{\mathsf{ahf}}, a)$ for the piece of advice;
  - Run $x \leftarrow \mathsf{TDP.Invert}(g_{\mathsf{tdp}},\mathsf{advice},y) \to$ for the solution candidate;
  - Set outputs as $\mathsf{solution} = x$.
- Any party can run $\mathsf{Verify}(\mathsf{pp},\mathsf{puz},\mathsf{solution})$ as verifiers:
  - Phrase $\mathsf{pp} = (f_{\mathsf{ahf}}, g_{\mathsf{tdp}})$, $\mathsf{puz} = y$ and $\mathsf{solution} = x$.
  - Output 1, if $g_{\mathsf{tdp}}(x) = y$ holds, and 0 otherwise.

B. Concrete Constructions

We demonstrate a 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, and the RSW puzzle and the DIODON function as the alternative solving process (w). In the concrete constructions, we slightly manipulate the generic construction. An additional property (see remark below) in the RSA problem enables us to have a simpler puzzle generation.

Construction 2 (Time-Hard Construction): Given a security parameter λ, a hash function $\mathsf{hash} : \mathbb{Z}_N \to \mathbb{Z}_N^*$:

- In $\mathsf{Setup}(1^\lambda,\mathsf{R})$, phrase $\mathsf{R} = (\mathrm{Time}, u)$. Sample two prime numbers p, q of length λ, compute $N = pq$ and $\phi(N) = (p-1)(q-1)$. Sample $a \xleftarrow{\$} \mathbb{Z}_N^*$, compute $a_u = a^{2^u \bmod \phi(N)} \bmod N$ and hash it for $d = \mathsf{hash}(a_u)$. Set outputs: $\mathsf{pp} = N$ and $\mathsf{td} = (\phi(N), d)$;
- In $\mathsf{PGen}(\mathsf{pp},\mathsf{td})$, phrase $\mathsf{pp} = N$ and $\mathsf{td} = (\phi(N), d)$. Compute e, such that $e \cdot d \equiv 1 \mod \phi(N)$. Sample $y \xleftarrow{\$} \mathbb{Z}_N^*$. Set outputs: $\mathsf{puz} = (N, e, y)$ and $\mathsf{aux} = (N, a)$;
- In $\mathsf{EvalSolve}(\mathsf{pp},\mathsf{puz},\mathsf{aux})$, phrase $\mathsf{pp} = N$, $\mathsf{puz} = (N, e, y)$ and $\mathsf{aux} = (N, a)$. Evaluate the u-RSW puzzle on a and hash it for advice, *i.e.*, $\mathsf{advice} \leftarrow \mathsf{hash}(a^{2^u} \bmod N)$. Compute the solution candidate with $x' = y^{\mathsf{advice}} \bmod N$. Set outputs as $\mathsf{solution} = x'$;
- In $\mathsf{Verify}(\mathsf{pp},\mathsf{puz},\mathsf{solution})$, phrase $\mathsf{pp} = N$, $\mathsf{puz} = (N, e, y)$ and $\mathsf{solution} = x'$. Output 1, if $x'^e = y \mod N$ holds, and 0 otherwise.

Construction 3 (Memory-Hard Construction): Regardless of the additional parameters in the DIODON function: *i.e.*, k for iterative squaring and l for iterative hashing, the principle remains the same.

- In $\mathsf{Setup}(1^\lambda,\mathsf{R})$, phrase $\mathsf{R} = (\mathrm{Memory}, u)$. Sample two prime numbers p, q of length λ, compute $N = pq$ and $\phi(N) = (p-1)(q-1)$. Sample $a \xleftarrow{\$} \mathbb{Z}_N^*$, 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: $\mathsf{pp} = N$ and $\mathsf{td} = (\phi(N), d)$;
- $\mathsf{PGen}, \mathsf{EvalSolve}, \mathsf{Verify}$ remains the same.

Remark (RSA as Special One-Way Trapdoor Function): Notice that in Construction 1, the domain of asymmetrically hard functions $\mathcal{X}_f$ 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 the modification on the

one-way trapdoor function families, enabling the TDP.Gen algorithm to take in a trapdoor (td_g in the generic case or d in the concrete cases) and output the corresponding one-way trapdoor function g_{tdp}. Although general one-way trapdoor functions may not guarantee this, the RSA problem satisfies such a 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, by providing users both e and d of an RSA problem, we open the door of computing $\phi(N)$ for the users, thus 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 [17], considered to be one of the fastest algorithms for factoring large integers. Although solving an RSA problem may not be as hard as factoring [18].

Assumption 5.1: Given an RSA group $\mathbb{Z}_N^*$ with $N = pq, |p| = |q| = \lambda$, and an instance of RSA problem (e, y), for any algorithm Alg without p,q runs in $\omega(2^{\lambda/c})$ time, the probability of producing an x', such that $x'^e = y \mod N$ is negligible. Where c is a properly large coefficient.

Assumption 5.1 draws the line for soundness, thus we adjust the parameters, u and T with a precise relation: $T(\lambda, u) \ll 2^{\lambda/c}$. With this in mind, the efficiency of the concrete constructions is as follows:

- Setup and PGen sample two elements from $\mathbb{Z}_N^*$, compute an 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 $\tilde{O}(\lambda)$[1].
- 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 5.1 and Assumption 4.1. For difficulty, we prove it with soundness, hardness, and a collision and preimage-resistant hash. Thus completing 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 5.1.
- **Time-hardness.** Under Assumption 4.1.
- **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$'$ with an arbitrary piece of advice$'$, which by collision-resistance of hash, $\Pr[\mathsf{advice}' = d] < \mathsf{negl}(\lambda)$. To break soundness is to invert the RSA problem (N, e, y) with no other information. By Assumption 5.1, even the adversary runs in much longer than $T(\lambda, u)$ but less than $2^{\lambda/c}$ cannot invert (N, e, y), thus cannot produce an acceptable solution$'$.

b) Time-hardness.: Given an RSA problem (N, e, y), the adversary queries $\{\mathsf{aux}\}_q$ under its choice and obtains $\{\mathsf{advice}\}_q$ from the challenger. Such a pre-processing procedure comes from the fact that the adversary can see polynomial many $(\mathsf{aux}, \mathsf{advice})$ 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 4.1, the probability that one can evaluate an RSW puzzle plainly with less than $\delta(u)$-time is negligible. Moreover, hash guarantees that collisions happen among $\{\mathsf{advice}\}_q$ and advice* 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 tuple of games:

1) The difficulty game, where puz $\leftarrow$ 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 is sent to the adversary. We regard such situation as puz $=\perp$.

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

[1] $\tilde{O}(\lambda) = O(\lambda \log^k(\lambda))$ for some k.

REFERENCES

[1] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," 1993, pp. 139–147.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, http://bitcoin.org/bitcoin.pdf.

[3] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan, "Proofs of work from worst-case assumptions," 2018, pp. 789–819.

[4] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," 2017, pp. 357–388.
[5] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, "Proofs of space," 2015, pp. 585–605.
[6] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. Song, "Provable data possession at untrusted stores," 2007, pp. 598–609.
[7] A. Biryukov and L. Perrin, "Symmetrically and asymmetrically hard cryptography," 2017, pp. 417–445.
[8] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," Cambridge, MA, USA, Tech. Rep., 1996.
[9] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," 2018, pp. 757–788.
[10] A. Biryukov and D. Khovratovich, "Egalitarian computing," 2016, pp. 315–326.
[11] F. Coelho, A. Larroche, and B. Colin, "Itsuku: a memory-hardened proof-of-work scheme," Cryptology ePrint Archive, Report 2017/1168, 2017, https://eprint.iacr.org/2017/1168.
[12] B. Wesolowski, "Efficient verifiable delay functions," 2019, pp. 379–407.
[13] K. Pietrzak, "Simple verifiable delay functions," 2019, pp. 60:1–60:15.
[14] S. Popov, H. Moog, D. Camargo, A. Capossele, V. Dimitrov, A. Gal, A. Greve, B. Kusmierz, S. Mueller, A. Penzkofer *et al.*, "The coordicide," 2020.
[15] M. E. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Trans. Information Theory*, vol. 26, no. 4, pp. 401–406, 1980. [Online]. Available: https://doi.org/10.1109/TIT.1980.1056220
[16] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro, "Scrypt is maximally memory-hard," 2017, pp. 33–62.
[17] C. Pomerance, "A tale of two sieves," *Notices of the AMS*, vol. 43, no. 12, pp. 1473–1485, December 1996.
[18] D. Boneh and R. Venkatesan, "Breaking rsa may be easier than factoring," 1998, pp. 59–71.