PURED: A unified framework for resource-hard functions

Alex Biryukov[1,2] and Marius Lombard-Platet[1]

[1] DCS, University of Luxembourg, Esch-sur-Alzette, Luxembourg
[2] SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
first.last@uni.lu

Abstract. Algorithm hardness can be described by 5 categories: hardness in computation, in sequential computation, in memory, in energy consumption (or bandwidth), in code size. Similarly, hardness can be a concern for solving or for verifying, depending on the context, and can depend on a secret trapdoor or be universally hard. Two main lines of research investigated such problems: cryptographic puzzles, that gained popularity thanks to blockchain consensus systems (where solving must be moderately hard, and verification either public or private), and white box cryptography (where solving must be hard without knowledge of the secret key). In this work, we improve upon the classification framework proposed by Biryukov and Perrin in Asiacypt 2017 and offer a united hardness framework, PURED, that can be used for measuring all these kinds of hardness, both in solving and verifying. We also propose three new constructions that fill gaps previously uncovered by the literature (namely, trapdoor proof of CMC, trapdoor proof of code, and a hard challenge in sequential time trapdoored in verification), and analyse their hardness in the PURED framework.

Keywords: puzzle cryptography · white-box cryptography · memory hardness · VDF · trapdoor problems

1 Introduction

In some specific situations, it might be preferable that algorithms are inefficient: for instance, RSA assumes that factoring is hard in computation. Hard problems can be used to slow down attackers or honest parties (the problems are then usually called puzzles). Such examples include code protection [30], resistance against password cracking [43], or blockchain consensus [11, 33]. A brief summary of hardnesses in the literature can be found in Table 1. A more exhaustive survey has been carried in [3], but did not tackle how to measure the puzzle hardness.

Because of the variety of such applications and the resources considered, research on the general topic is quite scattered and not always consistent with each other. Our goal is twofold: expand existing literature on a generic hardness framework, and give a unified measure of hardness. We also along the way illustrate our framework with new constructions that fill gaps in the literature, and can be of individual interest.

Hardness type		Resource				
Solving	Verification	SeqTime	CPU	Mem	BW	Code
Moderate/Hard	Moderate/Hard	See Section 3.2				
	Trapdoored	Iterated modular squaring in group of unknown order	Any non-NP problem	Argon [14], BalloonHash [23], scrypt [40]	Argon [14], scrypt [40]	Lookup table of random numbers
	Easy	SeqTime challenge (see Section 6)	Encryption of NP-complete witness, of proof of work	*[unknown]*	*[unknown]*	*[unknown]*
		Proof of sequential work [39, 28], trapdoorless VDF or in trustless group of unknown order [22, 31] or in MPC-generated RSA groups [26, 49]	NP-complete problems; Proof-of-Work [32]; witness or PoW encryption[35, 38]	Proof-of-Space [33]; Equihash, ethash [16, 34]; MTP/Itsuku [15, 27]	*[unknown]*	Proof of storage [37]
Trapdoored	Moderate/Hard	See Section 3.1				
	Trapdoored	RSA time-lock [45]	Encrypted PoK of a secret key or of factoring, Skipper [17]	Diodon [17]	*[unknown]*	SPACE, ASASA, lookup on a large trapdoored S-box [21, 20, 13]
	Easy	Trapdoor VDF [51, 41, 29]	EUF-CMA signatures, Schnorr protocol [47], proof of factoring [42], signatures of knowledge [25]	Trapdoor proof of CMC (See Section 5), memory lock (see Section 3.3)	Bandwidth lock (see Section 3.3)	HSig-BigLUT (see Section 4)
Easy	Moderate/Hard	See Section 3.1				
	Trapdoored	*[unknown]*	Encrypt 0 (with IND-CPA scheme)	*[unknown]*	*[unknown]*	*[unknown]*
	Easy	Return 0	Return 0	Return 0	Return 0	Return 0

Table 1. Panorama of existing resource-hard problems. Our constructions and reductions are in grey cells. We also discuss about easy verification in Section 3.4

Our contributions We present the following results:

- A formal hardness framework, PURED, uniting hardness over different kinds of resources, such as CPU, memory, code, sequential time, as well as over different issues: solving and verification, and with or without trapdoors.
- Generic reductions between different hardness classes.
- We propose three constructions, with hardness proved in our PURED model:
  - a trapdoored (in solving) proof of code, that cannot be implemented by an algorithm much smaller than the honest one, unless a secret trapdoor is known. On the other hand, verification only requires a few lines of code;
  - a trapdoor (in solving) proof of CMC that requires a significant amount of memory, unless a secret trapdoor is known, while verification is always easy;
  - a trapdoored (in verification) challenge of sequential time. This problem necessarily takes time to solve, but can be fast to verify if one knows the secret trapdoor.

Outline In Section 2, we introduce the problem and define our PURED framework. Then, in Section 3 we offer reductions between different hardness classes. Sections 4 to 6 propose three new constructions, respectively a trapdoor proof of CMC, a trapdoor solving proof of code, and a SeqTime challenge, along with their proofs of hardness in the PURED framework. We conclude in Section 7.

2 General Resource-Hardness Framework

2.1 Resources

We here give a quick panorama of the resources we consider in this paper.

CPU Computation complexity is defined by the number of operations (which can be counted as clock cycles, multiplications, exponentiations...) required to evaluate a program. CPU does not account for parallelization.

SeqTime is defined by the minimal sequential time (measured in cycles, multiplications...) for solving a problem. While CPU can be linked to circuit size, SeqTime can be linked to circuit depth and measures the minimal number of consecutive steps to compute a function problems are non-parallelisable and do not depend on the number of processors, and have been introduced notably for time lock puzzles [45], then for verifiable delay functions [22, 41, 51]. Typical examples include iterated hashing and iterated squaring.

Mem In many cases, a memory intensive algorithm can be rewritten to use very little memory, at the expense of an increased computation time. Time-memory product [40], then cumulative memory complexity (CMC, [8]) addressed this issue. CMC (further refined in [5]) counts the sum of memory requirements at each step of the execution, and does not depend on parallelization. In this paper, we use the CMC memory-hardness framework, which is notably the framework in which scrypt hardness has been proven [6].

Code Code complexity is the minimum size of a binary implementing a given algorithm. Code hardness has been used notably for whitebox cryptography [30, 20, 21, 13], and applies to programs that cannot be efficiently compressed. Code complexity usually relies on lookup tables given to the user at generation time. If the solver/verifier can generate these tables on the fly, then the problem becomes Mem hard rather than code hard.

BW Bandwidth complexity has been introduced to count the number of off-chip memory accesses, which is invariant between CPUs and ASICs, thus offering a hardware-agnostic complexity. Energy complexity [7, 4], then bandwidth complexity [44, 18], lower-bound the energy cost. Bandwidth complexity increases with the square root of the CMC [18], and does not vary with parallelization.

Other complexities On top of the aforementionned complexities, other complexities have been proposed [48, 1, 9, 50, 2]. However, either they have been superseded by one of the previous complexities, or either their usage remains minimal.

2.2 Resource-Hardness Game

Problem class The high level idea is that a problem class is specifically crafted so that it requires hardness in resource R, i.e. it requires at least u units of R to solve (resp. verify) an instance (resp. a solution), for any u.

This gives us the following definition of a problem class.

Definition 1. *A problem class instance is a tuple* $(\mathcal{P}_u^\lambda, \mathsf{Solve}, \mathsf{Verify})$*, with* $\mathsf{Solve} : \mathcal{P}_u^\lambda \times \{0,1\}^* \to \{0,1\}^*$, $\mathsf{Verify} : \{0,1\}^* \to \{0,1\}$ *the reference algorithm implementations for solving and verifying the problem, and* λ *is a security parameter.* Solve *is a (potentially) nondeterministic algorithm that takes as input a problem instance* $P \in \mathcal{P}_u^\lambda$*, and a random tape*[3] *z:* $\mathsf{Solve}(P, z)$*.*

Furthermore, Solve *and* Verify *implementations must be sound, i.e. any output of* Solve *is valid for* Verify*:* $\forall P \in \mathcal{P}_u^\lambda, \forall z \in \{0,1\}^*, \mathsf{Verify}(P, \mathsf{Solve}(P, z)) = 1$*.*

A problem class $\mathfrak{C}^*$ *is an algorithm* $\mathsf{Generate}(u, \lambda)$ *that is easy to evaluate and generates a problem class instance* $\mathfrak{C}$ *where* $\mathfrak{C} = (\mathcal{P}_u^\lambda, \mathsf{Solve}, \mathsf{Verify})$*. Furthermore, in case of a solving (resp. verification) trapdoor, it also privately discloses the trapdoor* t_S *(resp.* t_V*) to privileged parties.*

Note that generation is trustless if and only if no secret trapdoor are created. We also assume that solving (or verifying) a problem from a problem class instance does not yield any advantage for solving (or verifying) another problem from another instance of the same problem class[4].

Finally, while we model the probabilistic nature of solving algorithm, probabilistic verification algorithms are left out of scope of this work.

[3] In this paper, we often omit the random tape and write $s \xleftarrow{\$} \mathsf{Solve}(P)$ when information on z is irrelevant, and $s \leftarrow \mathsf{Solve}(P, z)$ otherwise.

[4] For instance, consider the problem of inverting a hash for the hash function $H(\cdot, r)$ where r is a random string for each problem class instance. For a given r, inverting two hashes is cheaper than double the cost of inverting one hash. However, in the case of inverting one hash of $H(\cdot, r)$ and one hash of $H(\cdot, r')$, this is not true.

Hardness game The hardness (either in solving or verifying) of a problem class is evaluated through the following protocol. For Verify, since using the probability of being correct on a random input yields the 0 function as an overwhelmingly perfect verification algorithm for hard to solve problems, we rather use a variant of Youden's J statistic [52], where 0 indicates no better performance than random, and ± 1 a perfectly good (or bad) performance.

Definition 2 (Algorithm advantage). *The advantage* Adv^s *of an algorithm* $\mathcal{A}$ *on solving a problem class* $\mathfrak{C}^*$ *is* $\mathsf{Adv}^s(\mathcal{A}) = Pr_{P,z,\mathfrak{C}}\left(\mathit{Verify}(P, \mathcal{A}(P, z)) = 1\right)$*, where the randomness is taken over* $P \in \mathcal{P}_u^\lambda$*, the random tape* z *of* $\mathcal{A}$ *and the generation of* $\mathfrak{C} = (\mathcal{P}_u^\lambda, \mathit{Solve}, \mathit{Verify}) \xleftarrow{\$} \mathfrak{C}^*.\mathit{Generate}(u, \lambda)$*.*

The advantage Adv_D^v *of an algorithm* $\mathcal{A}$ *on verifying a problem class* $\mathfrak{C}^*$ *over a domain* D *is*

$$\mathsf{Adv}_D^v(\mathcal{A}) = Pr_{P,s,\mathfrak{C}}(\mathcal{A}(P, s) = 1 | \mathit{Verify}(P, s) = 1) + Pr_{P,s,\mathfrak{C}}(\mathcal{A}(P, s) = 0 | \mathit{Verify}(P, s) = 0) - 1$$

where the randomness is taken over $P \in \mathcal{P}_u^\lambda$*,* $s \in D$*, and the generation of* $\mathfrak{C} = (\mathcal{P}_u^\lambda, \mathit{Solve}, \mathit{Verify}) \xleftarrow{\$} \mathfrak{C}^*.\mathit{Generate}(u, \lambda)$*.*

Remark 1. For the verifying game, the advantage is domain dependent as there might be strings that are obvious non solutions. Depending on the problem class, determining if the challenge belongs to the set of possible solutions might be a difficult problem in itself, thus the requirement to specify the domain of the verification advantage.

Hard problem class While Solve and Verify are already defined in the previous section, they might not be the algorithms chosen by an attacker, who might want to compromise accuracy in exchange of a more efficient resource usage.

We note the consumption of resource R with the function $\mathsf{Res}(\cdot)$: an algorithm $\mathcal{A}$ using u units of resource R on entry P is noted $\mathsf{Res}_R(\mathcal{A}(P)) = u$. The resource R will be omitted if the context allows.

Our definition relies on the hardness game defined earlier, however we use probabilities rather than an interactive game. Our definition extends the hardness framework used in [6, 28], and generalizes other similar approaches [17, 12]. A high-level understanding of our definition is: *If a problem class is hard, then for any algorithm solving (or verifying) a problem instance, the probability this algorithm uses less than some amount of resource is minimal.*

Definition 3 (PURED resource hardness). *Let* u *be a nonnegative number,* λ *be a security parameter, and* $\mathfrak{C} = (\mathcal{P}_u^\lambda, \mathit{Solve}, \mathit{Verify})$ *a problem class.*

- *We say that the problem class* $\mathfrak{C}^*$ *is a* $(p, u, R, \delta, \epsilon)$*-solving hard problem (with* ε *a function of* p*,* u *and* δ*), if*

$$\forall \mathcal{A}, \mathsf{Adv}^s(\mathcal{A}) \geq p \Rightarrow Pr_{P,z,\mathfrak{C}}\left[\mathit{Res}(\mathcal{A}(P, z)) < \delta u\right] < \epsilon + \mathsf{negl}(\lambda)$$

where the probability is taken over the sampling of $P \in \mathcal{P}_u^\lambda$*, the random tape* z*, and the generation of* $\mathfrak{C} = (\mathcal{P}_u^\lambda, \mathit{Solve}, \mathit{Verify}) \xleftarrow{\$} \mathfrak{C}^*.\mathit{Generate}(u, \lambda)$*, and* negl *denotes a function negligible in* λ*.*

– *Similarly,* $\mathfrak{C}^*$ *is* $(p, u, R, \delta, \epsilon)$ *verifying hard problem over* D *if:*

$$\forall \mathcal{A}, \mathsf{Adv}_D^v(\mathcal{A}) \geq p \Rightarrow Pr_{P,s,\mathfrak{C}}\left[\mathit{Res}(\mathcal{A}(P,s)) < \delta u\right] < \epsilon + \mathsf{negl}(\lambda)$$

Where the probability is taken over the sampling of $P \in \mathcal{P}_u^\lambda, s \in D$ *and the generation of* $\mathfrak{C} = (\mathcal{P}_u^\lambda, \mathit{Solve}, \mathit{Verify}) \xleftarrow{\$} \mathfrak{C}^*.\mathit{Generate}(u, \lambda)$.

Definition 4 (PURED for trapdoor hardness). *Similarly, a problem class is trapdoored if it is at most* $(p, \log u, R, \delta, \epsilon)$ *solving hard for people knowing the trapdoor, but* $(p, u, R, \delta, \epsilon)$ *hard otherwise. More precisely,*

– *The* $(p, u, R, \delta, \epsilon)$ *solving hard problem class* $\mathfrak{C}^*$ *is said to be trapdoored if there exists* $\mathcal{A}$ *such that*

$$\mathsf{Adv}^s(\mathcal{A}(t_\mathfrak{C}, \cdot)) \geq p \wedge Pr_{P,z,\mathfrak{C}}\left[\mathit{Res}(\mathcal{A}(t_\mathfrak{C}, P, z)) < \delta \log u\right] \geq \epsilon + \mathsf{negl}(\lambda)$$

With $t_\mathfrak{C}$ *the solving trapdoor generated by* $\mathfrak{C}^*.\mathit{Generate}$*, see Definition 1.*

– *The* $(p, u, R, \delta, \epsilon)$ *verifying hard over a domain* D *problem class* $\mathfrak{C}^*$ *is said to be trapdoored if there exists* $\mathcal{A}$ *such that*

$$\mathsf{Adv}_D^v(\mathcal{A}(t_\mathfrak{C}, \cdot)) \geq p \wedge Pr_{P,s,\mathfrak{C}}\left[\mathit{Res}(\mathcal{A}(t_\mathfrak{C}, P, s)) < \delta \log u\right] \geq \epsilon + \mathsf{negl}(\lambda)$$

A moderately hard problem is one that is hard, for some parameters deemed acceptable. An easy problem can be solved within a small amount of resource usage.

Remark 2. Following [28], u and δ are two different parameters instead of just one, as u indicates the estimated hardness of the problem, with a typical $\varepsilon - \delta$ approach.

One can show that proof of work of difficulty d (i.e. given s find x so that $H(x||s)$ starts with d zeroes) is $(p, p2^d, \mathsf{CPU}, \delta, \delta)$ solving hard in the ROM.

Similarly, according to [6] the function scrypt [40] is $(p, \frac{M^2 n}{25}, \mathsf{Mem}, 1 - \frac{100 \log M}{n}, 1 - p + 0.08M^6 2^{-n} + 2^{-M/30})$ solving hard in the parallel random oracle model, where M and n are (integer) parameters.

Finally, in the literature, for CPU related problems, the notion of hardness is sometimes defined by (t, p) security, which means that any attacker running in time t cannot succeed with probability more than p. Thus, in our framework a (t, p) secure problem class is $(p, t, \mathsf{CPU}, \delta, \mathbb{1}_{\delta \geq 1})$ hard, where $\mathbb{1}$ is the indicator function.

We assume that ϵ is increasing in δ, and decreasing in p.

As noted previously, this definition is only hard *on average*. We only define systematic hardness on problems relying on a function that can be approached as a random oracle.

Definition 5 (systematic hardness). *Let* u *be a nonnegative number,* λ *be a security parameter, and* $\mathfrak{C}_h^*$ *a problem class that relies on a function* h*, modeled as a random variable.*

We say that the problem class $\mathfrak{C}_h^*$ is a systematic $(p, u, R, \delta, \epsilon)$ solving hard problem, if

$$\forall \mathcal{A}, \forall P \in \mathcal{P}_u^\lambda, Pr_h\left[\mathsf{Verify}(P, s) = 1\right] \geq p \Rightarrow Pr_h\left[\mathsf{Res}(\mathcal{A}(P)) < \delta u\right] < \epsilon + \mathsf{negl}(\lambda)$$

Where randomness is taken over the choice of h in the random oracle model. The notion of systematic verifying hardness is similarly defined.

2.3 Bounded adversaries

An important remark is that the current definition of our PURED framework leaves the possibility of an unbounded adversary in Code and SeqTime problems, which obviously cause issues if the problem relies on an underlying problem, assumed computationally intractable (e.g. factoring a RSA modulus for Wesolowki VDF [51]). Hence, we additionally request, for Code and SeqTime, to specify the adversary computational bound. Note that for CPU and Mem, this bound is already natively included in our framework.

For instance, we can say that a problem class is $(p, u, R, \delta, \varepsilon, s)$ solving hard if it is $(p, u, R, \delta, \varepsilon)$ solving hard, for any PPT adversary bounded by 2^s operations (which include oracle calls or table lookups).

Our constructions in the PURED framework We give the hardness of the constructions we present below in Sections 4 to 6. The different parameters are described in the associated sections.

- HSig-BigLUT$_u^{\lambda,\gamma}$ (Section 4)
  - Systematically $(p, u, \mathsf{Code}, \delta, \mathbb{1}_{p=0}\mathbb{1}_{\delta<1})$ trapdoor solving hard against a 2^λ bounded adversary, with $u = \gamma L - \lambda \approx \lambda(L-1)$
  - Verification complexity of $O(\lambda)$ in Code
- Trapdoor proof of CMC $\mathsf{TdPoCMC}_{M,L,k}^\lambda$ (Section 5)
  - $((pq)^k, \frac{qML\lambda}{25}, \mathsf{Mem}, 1 - \frac{100 \log M}{\lambda}, 1 - p + \mathsf{negl}(M) + \mathsf{negl}(\lambda))$ trapdoor solving hard
  - Verification complexity at most $O\left(k^2 \lambda \log \frac{L}{k}(\lambda + \log L)\right)$ in Mem
- SeqTime Challenge$_{T,w,t}^\lambda$ (Section 6), against an adversary that is 2^λ bounded, and makes at most 2^q calls to the oracle:
  - Systematically $(p, 2T-1, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq 1})$ solving hard
  - Systematically $(p, 2T-1, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq \left(1 - p - \frac{\log_2(T) w}{2^w - 2q - 1}\right)^{1/n}})$ solving hard if the solver knows the trapdoor
  - $(p, 2T-1, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq 1})$ trapdoored verifying hard over $\{0,1\}^w$

3 Problem class reductions

In this section, we show generic reductions from one difficulty class to another. In all these theorems, we do not specify the computational bound of the adversary (when needed), since it always transfers from the old class to the new one (even though better bounds might exist in the new class), thus can be omitted for simplicity.

3.1 Leveraging trapdoored solving hard into verifying hard

We show how to obtain a problem class hard in verification from a problem class that is trapdoored in solving. For our proof, we need what we call a *deterministic* problem class, which means that for any problem instance P, there is only one solution s, which implies that Solve is deterministic.

Theorem 1. *Let $\mathfrak{C}^*, \mathfrak{D}^*$ be problem classes, let H be a hash function of codomain $\{0,1\}^n$. If $\mathfrak{C}^*$ is $(p, u, R, \delta, \epsilon)$ trapdoored solving hard and deterministic, then there exists a problem class $\mathfrak{E}^*$ that is $(p - \frac{1-p}{2^n}, u, R, \delta, \epsilon)$ verifying hard in the random oracle model on the domain $\{\mathfrak{C}, P, c : \mathfrak{C} = (\mathcal{P}_u^\lambda, \_, \_) \xleftarrow{\$} \mathfrak{C}^*.\mathsf{Generate}(u, \lambda); P \xleftarrow{\$} \mathcal{P}_u^\lambda; c \in \{0,1\}^n\}$.*

The case when $\mathfrak{D}^*$ is not deterministic is left for future work.

Proof. The idea of this proof is that the solver will generate on the fly their own trapdoor problem class, then encode the solution inside that class. For instance, for a CPU hard problem, instead of encrypting the result with the verifier's public key, the solver XORs the solution with a random message encrypted using a public key generated on the spot. We now give a formal proof of this idea for any kind of resource hardness.

Let $\mathfrak{D}^*$ be a problem class of unspecified hardness, and $\mathfrak{D} = (\mathcal{Q}_u^\lambda, \mathsf{Solve}_\mathfrak{D}, \mathsf{Verify}_\mathfrak{D})$ be an instance of $\mathfrak{D}^*$. Let $\mathfrak{C}^*$ be a $(p, u, R, \delta, \epsilon)$ trapdoor solving problem class.

Our goal is to create a problem class that is $(p, u, R, \delta, \epsilon)$ verifying hard.

Let $\mathfrak{E}^*$ be a problem class such that $\mathsf{Generate}(u, \lambda)$ returns a tuple $\mathfrak{E} = (\mathcal{Q}_u^\lambda, \mathsf{Solve}_\mathfrak{E}, \mathsf{Verify}_\mathfrak{E})$, with the following properties.

- Upon a problem entry $Q \in \mathcal{Q}_u^\lambda$, $\mathsf{Solve}_\mathfrak{E}(Q)$ will run $s \leftarrow \mathsf{Solve}_\mathfrak{D}(Q)$, then generate an instance $\mathfrak{C} = (\mathcal{P}_u^\lambda, \mathsf{Solve}_\mathfrak{C}, \mathsf{Verify}_\mathfrak{C})$ of $\mathfrak{C}^*$ with solving trapdoor $t_\mathfrak{C}$ (note that $\mathsf{Solve}_\mathfrak{E}(Q)$ knows $t_\mathfrak{C}$). Then, they sample a random P from $\mathcal{P}_u^\lambda$ and run the trapdoored function $s' \leftarrow \mathsf{Solve}_\mathfrak{C}(t_\mathfrak{C}, P)$. Finally, $\mathsf{Solve}_\mathfrak{E}(Q)$ returns $(\mathfrak{C}, P, s \oplus H(s'))$, where H is a preimage resistant hash function of codomain size higher than the size of s.
- Upon a possible solution $(\mathfrak{C}, P, c)$, $\mathsf{Verify}_\mathfrak{E}$ is defined by $\mathsf{Verify}_\mathfrak{E}(Q, (\mathfrak{C}, P, c)) = \mathsf{Verify}_\mathfrak{D}(Q, c \oplus H(\mathsf{Solve}_\mathfrak{C}(P)))$.

We now prove that $\mathfrak{C}^*$ is verifying hard over the set $\{\mathfrak{C}, P, c : \mathfrak{C} = (\mathcal{P}_u^\lambda, \_, \_) \xleftarrow{\$} \mathfrak{C}^*.\mathsf{Generate}(u, \lambda); P \xleftarrow{\$} \mathcal{P}_u^\lambda; c \in \{0,1\}^n\}$.

Let $\mathcal{V}$ be a PPT algorithm, with a non-null verifying advantage p.

In order to run $\mathcal{V}$ on input $(\mathfrak{C}, P, c)$, the adversary can either find a preimage of c in the random oracle model), or treat it as a random variable (assuming the random oracle model).

- If the $\mathcal{V}$ treats c as a random variable, the distributions $\mathcal{V}(Q, (\mathfrak{C}, P, c)))$ and $\mathcal{V}(Q, (\mathfrak{C}, P, r))$ are indistinguishable, so no matter the input, $\mathcal{V}$ will answer randomly, with a given probability q. Thus, $\mathcal{V}$ verification advantage is $\mathsf{Adv}_s(\mathcal{V}) = q + (1-q) - 1 = 0$, which contradicts the hypothesis that $\mathcal{V}$ has advantage $p > 0$.

– If $\mathcal{V}$ computes a preimage of c, given that we operate in the random oracle model, this means that $\mathcal{V}$ actually computes the value $\mathsf{Solve}_{\mathfrak{C}}(P)$, so the probability for $\mathcal{V}$ to use less than δu resources is less than ϵ. We still must assess $\mathcal{V}$ advantage. Let $(\mathfrak{C}, P, c)$ be a valid answer. If $\mathsf{Solve}_{\mathfrak{C}}(P)$ answers correctly with probability p, then $\mathcal{V}$ validates the input correctly with probability p in the best case[5]. Now, if $(\mathfrak{C}, P, c)$ is not a valid answer, there is the possibility that $\mathsf{Solve}_{\mathfrak{C}}$ does not answer correctly (with probability $1-p$) and that the wrong answer of Solve actually gives a valid on c (with probability $\frac{1}{2^n}$). Hence, on an invalid answer, $\mathcal{V}$ will answer correctly with probability $1 - \frac{1-p}{2^n}$, hence $\mathcal{V}$ has advantage $p - \frac{1-p}{2^n}$, hence the result.

Furthermore, we observe that $\mathfrak{E}^*$ is not much harder to solve than $\mathfrak{D}^*$.

Note that the same result does not immediately transfer for creating easy verification problems: adding the solving trapdoor in the public information at generation might be problematic if it is equal to the solving trapdoor, as is the case for the RSA time-lock [45].

3.2 Leveraging solving hard to verification hard

We process similarly here. As noted in our table, there exist problems which are assumed to be hard both in solving and verification, mostly because the verification algorithm consists of the solving algorithm. However, these problems might lack a formal proof on the verification hardness, given that verification was not considered during their design.

Let $\mathfrak{C}^*$ be a deterministic problem class $(p, u, R, \delta, \epsilon)$ hard in solving. We show how to create a new problem class that is close to $\mathfrak{C}^*$, safe for the fact that it is now also hard in verification.

Theorem 2. *Let $\mathfrak{C}^*$ be a problem class. If $\mathfrak{C}^*$ is $(p, u, R, \delta, \epsilon)$ solving hard, then there exists a problem class $\mathfrak{E}^*$ that is $(p - \frac{1-p}{2^n}, u, R, \delta, \epsilon)$ verifying hard on a specific domain. Furthermore,*

– *If $R = \mathsf{CPU}$, then $\mathfrak{E}^*$ is $(p, u, R, \delta, \epsilon')$ solving hard, with $\epsilon'(p, \delta) = \epsilon(\sqrt{p}, \delta)$.*
– *If $R = \mathsf{SeqTime}$, $R = \mathsf{Mem}$, $R = \mathsf{Code}$ or $R = \mathsf{BW}$, then $\mathfrak{E}^*$ is $(p, 2u, R, \delta, \epsilon'')$ solving hard, with $\epsilon''(p, \delta) = \max\limits_{\substack{0 \leq p_1 \leq p \\ 0 \leq \delta_1 \leq 2\delta}} \epsilon(p_1, \delta_1)\epsilon(p/p_1, 2\delta - \delta)$.*

Proof. We use essentially the same construction as in Theorem 1, and thus will reuse the same notations, but this time there is no trapdoor involvedhence we can take $\mathfrak{D}^* = \mathfrak{C}^*$ (we name $\mathfrak{C}_1$ the instance from $\mathfrak{C}^*$, and $\mathfrak{C}_2$ the instance from $\mathfrak{D}^*$), and thus verifying a problem in $\mathfrak{E}^*$ requires solving one problem in $\mathfrak{C}^*$ (which is $(p, u, R, \delta, \epsilon)$ solving hard), and verifying one problem in $\mathfrak{C}^*$. Thus $\mathfrak{E}^*$ is at least $(p, u, R, \delta, \epsilon)$ verification hard.

For solving hardness, in order to solve a problem in $\mathfrak{E}^*$, an algorithm $\mathcal{A}$ must solve two instances P_1, P_2 of $\mathfrak{C}_1, \mathfrak{C}_2$, respectively.

[5] This is the best case since the only bits of information $\mathcal{V}$ disposes of are from $\mathsf{Solve}_{\mathfrak{C}}(P)$. As we saw previously answering randomly gives a null advantage

- If $R = \mathsf{SeqTime}$, the two instances can be solved in parallel.
- If $R = \mathsf{CPU}$ or $R = \mathsf{Code}$, because the two problem class instances are different, solving one does not offer any advantage in solving the other (see the remarks below Definition 1) so there is no other possibility than dedicate twice the amount of computation (or code).
- Similarly, if $R = \mathsf{Mem}$ (resp. $R = \mathsf{BW}$), the two problems can be solved one after the other, or in parallel, and both methods lead to a doubled cost.

Let us now consider an algorithm $\mathcal{A}$ that solves instances $P \in \mathcal{P}$ for the problem class instance $\mathfrak{E}$ with probability q. In order to solve P, $\mathcal{A}$ must solve two problems from $\mathfrak{C}^*$ (but of different instances), namely P_1 and P_2. On average, let us assume that $\mathcal{A}$ solves P_1 (resp. P_2) with probability p_1 (resp. p_2), with $p_1 p_2 = q$.

Case where $R = \mathsf{CPU}$. Both problems P_1 and P_2 can be solved in parallel. Thus, because $\mathfrak{C}^*$ is $(p, u, R, \delta, \epsilon)$ solving hard, we get that $\Pr[\mathsf{Res}(\mathcal{A}_1(P_1)) < \delta u] < \epsilon(p_1, \delta)$ where $\mathcal{A}_1$ is the part of $\mathcal{A}$ dedicated to solving P_1. Similar formula applies to P_2.

Because solving P uses less than δu resources of R if and only if both solving P_1 and solving P_2 require less than δu of resources each, we get $\Pr[\mathsf{Res}(\mathcal{A}(P)) < \delta u] = \Pr[\mathsf{Res}(\mathcal{A}_1(P_1)) < \delta u] \Pr[\mathsf{Res}(\mathcal{A}_2(P_2)) < \delta u] < \epsilon(p_1, \delta)\epsilon(q/p_1, \delta)$.

Because ϵ is decreasing in p, the right hand side is lower than $1\epsilon(\sqrt{q}, \delta)$, no matter the values of p_1 and p_2. Hence, $\mathfrak{E}^*$ is $(p, u, R, \delta, \epsilon')$ solving hard, with $\epsilon'(p, \delta) = \epsilon(\sqrt{p}, \delta)$.

Case where $R = \mathsf{SeqTime}$ or $R = \mathsf{Mem}$ or $R = \mathsf{Code}$. In this case, the algorithm $\mathcal{A}$ must not allot more than $2\delta u$ units of R *cumulated* to $\mathcal{A}_1$ and $\mathcal{A}_2$.

Let $\mathcal{A}$ be an algorithm solving $\mathfrak{E}$, with probability q. Using the same notations as before, let us assume that $\mathcal{A}_1$ solves P_1 with probability p_1, and $\mathcal{A}_2$ solves P_2 with probability $p_2 = q/p_1$. If $\delta_1 u$ units are devoted to solving P_1, then $\mathcal{A}$ can devote up to $(2\delta - \delta_1)u$ units to P_2, so that the total amount of resources devoted does not exceed δ.

Hence, $\mathcal{A}$ succeeds solving P with probability $p_1 p_2 = q$, and $\Pr[\mathsf{Res}(\mathcal{A}(P)) < 2\delta u] < \epsilon(p_1, \delta_1)\epsilon(q/p_1, 2\delta - \delta_1)$, from which we derive our upper bound.

3.3 Leveraging trapdoored solving hard and trapdoored verification to easy verification

From a deterministic problem class that is trapdoored solving, it is trivial to create a problem class that is easy in verifying: it suffices, at generation, to publish the solving trapdoor. However, this also breaks the solving complexity, thus reducing the interest of doing so.

However, we can design one-time problems with problems that are deterministic (i.e. only one possible solution) and trapdoored in solving. As a matter of fact, this approach had been used by Rivest for his time capsule challenge [46].

Algorithm 1 Resource-lock Solve and Verify algorithm
Inputs: $\mathfrak{C}$ is a trapdoored solving hard class problem of trapdoor t, H is a second preimage resistant hash function whose outputs are more than $|t|$ bits, $c = H(\mathfrak{C}.\mathsf{Solve}(m)) \oplus t$, $m \in \mathfrak{C}.\mathcal{P}$

function SOLVE(m, c, $\mathfrak{C}$, H)
  $s \leftarrow \mathfrak{C}.\mathsf{Solve}(m)$
  return $H(s) \oplus c$
end function

function VERIFY($(m, c, \mathfrak{C}, H), v$)
  $\triangleright$ $(m, c, \mathfrak{C}, H)$ is a problem instance, v is a response
  return $H(\mathfrak{C}.\mathsf{Solve}(m, v)) \oplus c \stackrel{?}{=} v$
end function

In this section, we simply generalize the concept with Algorithm 1, and give its security in our framework.

With this construction, we get the following result.

Theorem 3. *Let $\mathfrak{C}$ be a problem class that is $(p, u, R, \delta, \epsilon)$ solving trapdoored hard, of solving trapdoor t. Then the resource-lock problem class of Algorithm 1 defined on top of $\mathfrak{C}$ is publicly verifiable and $(p, u, R, \delta, \epsilon)$ solving hard for at most one instance.*

Proof. We simply show that the problem class $\mathcal{D}$ defined with Algorithm 1 on top of $\mathfrak{C}$ offers the claimed security, since the soundness is immediate. It is also immediate that verification is easy, and that once one instance has been solved and published (hence once the solving trapdoor has been published), the solving hardness disappears.

Let us explore the solving strategies for a solver of the resource-lock problem class that does not dispose of the solving trapdoor. For Verify to accept a solution, either the solver finds the trapdoor, either a preimage on H, or the solution to the instance of $\mathfrak{C}$. Finding the trapdoor is, by hypothesis, assumed to be infeasible with probability higher than $\mathsf{negl}(\lambda)$. Similarly, finding a preimage of $H(\mathfrak{C}.\mathsf{Solve}(m, v))$ is equally as hard as finding the trapdoor (since the hash codomain has the same size as the trapdoor), and hence negligible in λ. Finally, solving $\mathfrak{C}$ without the trapdoor is $(p, u, R, \delta, \epsilon)$ hard, which concludes the proof.

3.4 Leveraging any problem class to easy verification

In this section, we briefly describe how proofs of computation allow to transform problem classes into similar problem classes, but with easy verification.

Proofs of computation have been pioneered in [10], and much refined since then.Proofs of computation are often derived from zero-knowledge proofs [36].

Because of the generality of the construction, one can build succinct zero-knowledge proofs for any language in NP. Because of usual requirements on zk-proofs, checking a proof cannot take much CPU (or SeqTime), and because of the succinctness, Mem usage is low as well. Furthermore, the code of these constructions is also relatively small.

4 HSig-BigLUT: Code, systematic trapdoored-hard solving, easy verification problem class

4.1 Primer on homomorphic signature and the BFKW scheme

It is obvious that a homomorphic signature scheme cannot be forgery resistant, but the definition can be adapted as follows

Definition 6 (Homomorphic EUF-KMA security). *Let Sig = (Gen, Sign, Verify) be a signature scheme over a vector space $\mathbb{F}$. Sig is Homomorphic EUF-KMA secure if the advantage of any PPT adversary $\mathcal{A}$, who knows the signatures of some messages $\{\mathbf{m}\}$, in forging a valid signature (outside of the span of $\{\mathbf{m}\}$) is negligible:*

$$Pr\left[\begin{array}{l|l} \mathbf{m}^* \notin \mathsf{Span}(\{\mathbf{m}\}) \wedge & (\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda); \{\mathbf{m}\} \xleftarrow{\$} \mathbb{F}; \mathsf{Span}(\{\mathbf{m}\}) \subsetneq \mathbb{F} \\ \mathsf{Verify}(\mathsf{pk},\mathbf{m}^*,\sigma^*) = 1 & (\mathbf{m}^*,\sigma^*) \xleftarrow{\$} \mathcal{A}\left(\mathsf{pk}, (m, \mathsf{Sign}(\mathsf{sk},m))_{m\in\{\mathbf{m}\}}\right) \end{array}\right] < \mathsf{negl}(\lambda)$$

Where $\mathsf{Sign}(\mathsf{sk},\{\mathbf{m}\})$ is the set of signatures of the messages of $\{\mathbf{m}\}$, and $\mathsf{Span}(\{\mathbf{m}\})$ the vector space generated by the vectors of $\{\mathbf{m}\}$.

In this paper, we rely on the BFKW scheme [24], which allows for signing m vectors of $\mathbb{F}_p^n$, with m known in advance. The BFWK scheme is proven to be homomorphically EUF-KMA resistant under co-CDH in the ROM [24, Theorem 6], assuming that $\frac{1}{p} = \mathsf{negl}(\lambda)$.

For a keypair $(\mathsf{pk},\mathsf{sk})$, two scalars β_1,β_2, two vectors $\mathbf{v}_1,\mathbf{v}_2$ and their respective signatures σ_1,σ_2, we have $\mathsf{BFKW}_m.\mathsf{Verify}(\mathsf{pk},\beta_1\mathbf{v}_1+\beta_2\mathbf{v}_2,\sigma_1^{\beta_1}\sigma_2^{\beta_2}) = 1$. The scheme relies on $m+1$ public parameters (generators of a finite group), which can be stored compactly by storing the seed that generated them. For more details, please refer to Appendix A.2 or [24].

4.2 HSig-BigLUT construction

We now show how to leverage the BFKW signature scheme in a code-hard solving problem with public verification.

Let us consider a lookup table LUT consisting with L entries. Each entry $LUT[i]$ is of the form $LUT[i] = \mathsf{BFKW}_L.\mathsf{Sign}(\mathbf{e}_i)$, where $\mathbf{e}_i$ is the i-th element of the canonical base of $\mathbb{F}_p^L$.

Our problem instances are of the form $x \in \{0,1\}^*$, which is hashed via a hash function H of codomain $(\mathbb{F}_p^*)^L$. The solver has to return a linear combination of all the signatures from the LUT, where the coefficients are taken from $H(x)$. In other words the solver must return $\mathsf{BFKW}_L.\mathsf{Sign}\left(\sum_i H(x)[i]\cdot\mathbf{e}_i\right)$. Verification of the challenge is a simple signature verification.

We thus formalize the algorithms in Algorithm 2.

Algorithm 2 HSig-BigLUT$^{\lambda}_{u}$ Generate, Solve and Verify algorithms
Parameter: h is a hash function such that $h(x)$ returns L values of $\mathbb{F}_p^*$

function GENERATE(u, λ)
  $p \leftarrow$ prime of λ bits
  $L \leftarrow \lfloor \frac{u}{\lambda} \rfloor$
  $\mathsf{pk}, \mathsf{sk} \leftarrow$ instance of BFKW with a random id id, and for signing over the canonical basis $(\mathbf{e}_i)_{1 \leq i \leq L}$ of $\mathbb{F}_p^L$
  $LUT \leftarrow$ empty table of L entries
  for $i \in \{1, \ldots, L\}$ **do**
    $LUT[i] \leftarrow \mathsf{BFKW_L.Sign}(\mathsf{sk}, \mathbf{e}_i)$
  end for
  $\mathfrak{C} \leftarrow (\{0,1\}^*, \mathsf{Solve}^{LUT}, \mathsf{Verify})$
  return $(\mathfrak{C}, \mathsf{sk}, \perp)$
    ▷ solving trapdoor is sk
end function

function SOLVE$^{LUT}(x)$
  $s \leftarrow \prod_{i=1}^{L} LUT[i]^{h(x)[i]}$
  return s
end function

function VERIFY(x, s)
  $\mathbf{v} \leftarrow \sum_{i=1}^{L} h(x)[i] \cdot \mathbf{e}_i$
  return $\mathsf{BFKW_L.Verify}(\mathsf{pk}, \mathbf{v}, s)$
end function

Theorem 4. *Under the co-CDH in the ROM and assuming* $1/p = \mathsf{negl}(\lambda)$*, HSig-BigLUT consisting of* L *entries of size* γ *is systematically* $(p, \gamma L - \lambda, \mathsf{Code}, \delta, \mathbb{1}_{p=0}\mathbb{1}_{\delta<1})$*-trapdoor solving hard against an* 2^{λ} *bounded adversary, and verification can be done with probability* 1 *using* $O(\lambda)$ *resources in* Code*, where* γ *is the compressed size of the BFKW scheme (i.e., the size of signatures once compressed)*[6]*.*

Proof. Verification hardness. This part is trivial: for verification, one only needs the simple code written in Algorithm 2, which does not make any call to any lookup table. The solver only needs to store the public parameters, which essentially consist of $L+1$ generators of a finite group (and one bilinear map), which can be compactly stored by storing the random seed that generated them.

Solving hardness First, a solver with trapdoor solves in $O(\lambda)$ by signing the message with the private key. Let us now focus on the case where the solver does not have the trapdoor.

We first see that an attacker can safely remove λ/L bits to each entry and yet compute the solution to any input in 2^{λ} steps. Thus, we consider the base u to be $u = \gamma L - \lambda$. Let us now consider an attacker that removes even more bits. Since, for any input, the adversary must reconstruct all signatures of the LUT, then bruteforcing the solution to a problem instance will cost them more than 2^{λ} operations, which is impossible. Thus a success probability of $p > 0$ indicates that the adversary has at least u bits of Code complexity, thus the result.

[6] Since BFKW relies on elliptic curves, γ is expected to be close from λ

5 Trapdoor proof of CMC: Mem, trapdoored solving, easy verification problem class

5.1 A primer on Diodon [17]

Diodon (see Algorithm 3) is a memory trapdoored hard solving problem. [17] shows that without trapdoor, one must operate $\eta \times M$ squarings while storing $M \times \log_2(N)$ bits, while a user in possession of the trapdoor can instead use $L \times \log_2(N)$ squaring operations and $2\log_2(N)$ bits of memory. Its complexity is estimated around $\Omega(LM\log_2(N))$ in the CMC model [17, Appendix A].

There is no known efficient trapdoorless verification of Diodon.

Algorithm 3 Diodon Solve algorithm [17]
Parameters: public key $N = pq$ of λ bits, $\eta \in \mathbb{N}^*$, memory requirement M, length requirement L, problem instance $x \in \mathbb{Z}/N\mathbb{Z}$, H a hash function of λ bits

$V_0 \leftarrow x$ ▷ Expansion phase
for $i \in \{1, \ldots, M\}$ **do**
  $V_i \leftarrow V_{i-1}^{2^\eta} \mod N$
end for
$S_0 \leftarrow H(V_M)$ ▷ Hash chaining phase
for $i \in \{1, \ldots, L\}$ **do**
  $k_i \leftarrow S_{i-1} \mod M$
  $S_i \leftarrow H(S_{i-1}, V_{k_i})$
end for
return S_L

5.2 A primer on VDFs

Wesoloski VDF (Verifiable Delay Function, [51], also described in Appendix A.1) relies on iterated squaring in RSA groups, and is $(p, T, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq 1})$ trapdoor solving hard (assuming that factorization of the RSA modulus is computationnally intractable), with easy verification. More details are given in [51]. In this paper, for (y, π) a solution to a Wesolowski VDF problem instance, we often refer to π as the *Wesolowski proof of y*.

5.3 Trapdoor proof of CMC: the general idea

The main idea of our proof of CMC is to adapt Diodon so that the verifier can validate a proposed solution without spending a significant amount of memory. This is done by, at each step of the hash chaining phase, adding a VDF proving the validity of the value. These VDFs are then aggregated in a Merkle tree.

A high-level view of the protocol is described in Figure 1, and the full algorithm is described in Algorithm 4.

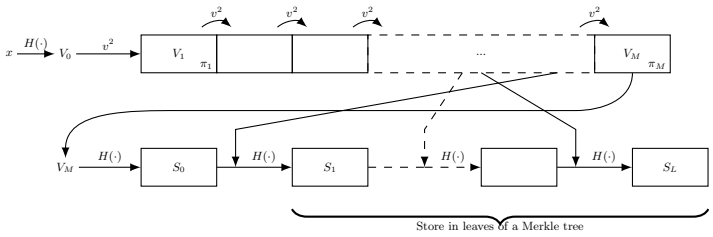

Fig. 1. A high-level view of the first steps (expansion and hash chaining phases) of the trapdoor proof of CMC. π_i is the Wesolowsky proof of the VDF computation of V_i. Then, we form a Merkle tree from the hash chain.

5.4 Trapdoored proof of CMC protocol

Summing it up, we get the Solve and Verify algorithms of our trapdoored proof of CMC in Algorithm 4.

Algorithm 4 Trapdoor proof of CMC $\mathsf{TdPoCMC}^{\lambda}_{M,L,k}$ Solve and Verify algorithms
Parameters: N a RSA module of λ bits, M a memory requirement, L a hashing chain length, k a number of openings

function SOLVE(x) ▷ $x \in \mathcal{P}^{\lambda}_u = \mathbb{Z}/N\mathbb{Z}$
  $V_0 = x$
  for $\in \{1, \ldots, M\}$ **do**
    $V_i \leftarrow V_{i-1}^2 \mod N$
  end for
  $s \leftarrow V_L$
  $S_0 \leftarrow H(V_L) \mod L$
  for $i \in \{1, \ldots, L\}$ **do**
    $k_i \leftarrow S_{i-1} \mod L$
    $\pi_i \leftarrow$ the Wesolowski proof of computation of V_{k_i}
    $S_i \leftarrow H(i, S_{i-1}, V_{k_i})$
  end for
  $r \leftarrow$ the root of a Merkle tree where the leaves are the tuples $\ell_i = (V_{k_i}, S_{i-1}, \pi_i)$
  $p \leftarrow$ Open k paths using the Fiat Shamir heuristic. For each opening that leads to leaf ℓ_i, also include ℓ_{i+1}
  return r, p
end function

function VERIFY($x, (r, p)$)
  for each opening o in p **do**
    $\ell := (V, S, \pi) \leftarrow$ the opening of the leaf, contained in o
    $i \leftarrow$ the index of ℓ in the tree
    $\ell' := (V', S', \pi') \leftarrow$ the following leaf in the tree, contained in o
    Check that (V, π) corresponds to a valid Wesolowski proof of the computation of $x^{2^S \mod L} \mod N$
    Check that the path from ℓ is valid and leads to the root r
    $S'' \leftarrow H(i, S, V)$
    Check that S'' is equal to S'
  end for
  If at least one of the check fails, return 0, otherwise return 1
end function

Theorem 5. *In a group where iterated squaring is* $(p, T, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq 1})$ *solving hard, the trapdoor proof of CMC problem class* $\mathsf{TdPoCMC}^{\lambda}_{M,L,k}$ *is* $((pq)^k, \frac{qML\lambda}{25}, \mathsf{Mem}, 1 - \frac{100 \log M}{\lambda}, 1 - p + \mathsf{negl}(M) + \mathsf{negl}(\lambda))$ *solving hard in the parallel ROM.*

Proof. Let us consider a solver that wishes to minimize their CMC, while having a good probability of passing verification. Looking at the security proof of scrypt, we see that the CMC of creating a (continuous) hash chain of size L/E, following an expansion phase of size M, with words of size λ, is around $\Omega(\frac{LM\lambda}{E})$. Because the hashes are index dependent, a chain cannot be reused at another place without immediately introducing a discontinuity.

Hence, computing the hash chain (with E discontinuities) will cost $\Omega(LM\lambda)$ to an attacker no matter what.

Furthermore, as discussed previously, the verifier, when asking for opening of leaf i, not only asks for S_i, V_i but also S_{i+1}, V_{i+1} so they can verify the chain has no discontinuity at step i. Let q be the proportion of steps correctly computed by the solver (without loss of generality, we can assume that other steps are computed at cost 0, and are always incorrect). Given results on scrypt, and notably the fact that scrypt is $(p, \frac{M^2\lambda}{25}, \mathsf{Mem}, 1 - \frac{100 \log M}{\lambda}, 1 - p + 0.08M^6 2^{-\lambda} + 2^{-M/20})$ solving hard (scrypt uses $L = M$), we can assume that the generation of a 2-chain is $(p, \frac{2M\lambda}{25}, \mathsf{Mem}, 1 - \frac{100 \log M}{\lambda}, 1 - p + \mathsf{negl}(M) + \mathsf{negl}(\lambda))$ hard. The probability that k openings do not reveal a cheater is then of $(pq)^k$. Thus, our protocol with k openings is at least $((pq)^k, \frac{ML\lambda}{25}, \mathsf{Mem}, 1 - \frac{100 \log M}{\lambda}, 1 - p + \mathsf{negl}(M) + \mathsf{negl}(\lambda))$ hard, hence the claim.

We now give the verification complexity of the trapdoor proof of CMC. Most importantly, the verification cost does not depend on M, which allows verification to be much smaller than solving when $L \ll M$.

Theorem 6. *The trapdoor proof of CMC problem class* $\mathsf{TdPoCMC}^{\lambda}_{M,L,k}$ *can be verified in a CMC of at most* $O\left(k^2 \lambda \log \frac{L}{k}(\lambda + \log L)\right)$.

Proof. A verifier receives k openings of the Merkle tree (plus one root value that can be neglected). An opening consists of:

- One path down the Merkle tree ($\log L$ nodes of size λ each)
- Two leaves, the leaf ℓ_i and the following leaf ℓ_{i+1}. We have $\ell_i = (V_{k_i}, S_{i-1}, \pi_i)$ hence one leaf is $\lambda + \lambda + 2\lambda = 4\lambda$ bits.

Verifying the VDF takes $O(\lambda)$ operations, verifying the tree opening takes $O(\log L)$ time, hence k verifications take $O(k(\lambda + \log L))$ time.

An opening consists of one node each of depth $1, 2, \ldots, L-1$, and two nodes of depth L. Thus, with k openings, there are at most 2^j nodes of depth $j+1$ for $j + 1 \leq \log(k)$. On the other hand, we cannot assume that there will be any collision on the nodes of depth higher than $\log k$. In total, we thus have k nodes of depth lower than $\log k$, and $k(\log L - \log k)$ nodes of higher depth, thus a higher bound of $k(\log L + 1 - \log k)$ nodes. Summing it, the proof uses $k(\log L + 1 - \log k)$ nodes of size λ and $2k$ leaves of size 4λ each, hence a memory requirement of at most $O\left(k\lambda \log \frac{L}{k}\right)$.

Multiplying the time requirements with the memory requirements, we get a CMC of at most $O\left(k^2 \lambda \log \frac{L}{k}(\lambda + \log L)\right)$.

6 SeqTime challenge: SeqTime systematic hard solving and trapdoored hard verifying problem class

6.1 A primer on proofs of sequential work

Proofs of sequential work can be seen as VDFs for which there exist cheating strategies, hence while it can be proved that the cheater must spend at least some time to solve the problem, the solution is not unique.

While other constructions exist (see for instance [39]) we hereby rely on the proof of sequential work $POSW_{n,w,t}$ described in [28], which is proven to be systematically $(p, 2^{n+1}-1, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq \left(1-p-\frac{nw}{2^{w-2q-1}}\right)^{1/t}})$ solving hard in the parallel ROM against a 2^q bounded adversary (in oracle calls), where n, w, t are various parameters. See [28] for more details, or Appendix A.3.

6.2 Our construction

The construction we propose in this section, SeqTime challenge, consists of solving two puzzles, one being trapdoored hard and the other one being trapdoorless in SeqTime. Both puzzles are solved in parallel, then we use a construction similar to the lock puzzles (see Section 3.3).

Let x be the solution to the iterated squaring problem, and y, z the solution to the POSW instance, along with its proof. We require that the hash size is bigger than y, i.e. $|h(x)| \geq |y|$. The algorithms for Solve and Verify are summarized in Algorithm 5.

Algorithm 5 $\mathsf{SeqTime}$ Challenge$^{\lambda}_{T,w,q,t}$ Solve and Verify algorithm
Parameters: A semiprime N of λ bits, a time parameter T (being a power of 2), a hash function H_x of codomain size 2^x

function SOLVE(a, b)
  $s, o \leftarrow \mathrm{POSW}_{\log_2 T,w,t}.\mathsf{Solve}(a)$
  $y \leftarrow b^{2^{2T-1}} \mod N$ ▷ s and y are computed in parallel
  $h \leftarrow H_{|o|}(y)$
  return $(s, o \oplus h)$
end function

function VERIFY$((a, b), (s, c))$
  $y \leftarrow b^{2^{2T-1}} \mod N$
  $o \leftarrow H_{|c|}(y) \oplus c$
  return $\mathrm{POSW}_{\log_2 T,w,t}.\mathsf{Verify}(a, (s, o))$
end function

Theorem 7. *Let $T, w \in \mathbb{N}, H$ he a hash function of codomain $\{0,1\}^w$.*

Let us use a group where, against a 2^λ bounded adversary, iterated squaring is systematically $(p, T, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq 1})$. Against a 2^λ bounded adversary with at most 2^q oracle calls, the SeqTime challenge is systematically $(p, 2T-1, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq 1})$ solving hard in the parallel ROM. If the solver knows the trapdoor, then the problem class is systematically $(p, 2T-1, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq \left(1-p-\frac{\log_2(T)w}{2^{w-2q-1}}\right)^{1/n}})$ solving hard in the random oracle model.

The $\mathsf{SeqTime}$ *challenge is also* $(p, 2T-1, \mathsf{SeqTime}, \delta, \mathbb{1}_{\delta \geq 1})$ *trapdoored verifying hard on* $\{0,1\}^w$.

Proof. **Solving hardness.** On instance (a, b), a honest solver will run $\mathrm{POSW}_{\log_2 T,w,t}.\mathsf{Solve}(a)$ and compute $b^{2^{2T-1}} \mod N$ in parallel. Computing $b^{2^{2T-1}} \mod N$ requires at least T consecutive steps to have a nonnegligible chance of success, so there is no incentive for the solver to find a solution to the POSW in less than T consecutive steps, which then gives a success probability of 1.

Moreover, in order to get a valid solution from an instance a, b, the solver has to output (s, c) such that $c \oplus (b^{2^{2T-1}} \mod N)$ is a valid opening for the POSW DAG tree of challenge a. Let us assume that the solver does not compute the value $b^{2^{2T-1}} \mod N$. Because at least one bit of y defined as $y \leftarrow b^{2^{2T-1}} \mod N$ can be seen as random (see [19]), the value $H_{|c|}(y)$ is random in the random oracle model, hence $c \oplus H_{|c|}(y)$ is random. Because the string $c \oplus H_{|c|}(y)$ is supposed to be a valid opening, it must contain at least the two hashes that lead to the root label, i.e. contain the labels l_1, l_2 such that $H(x, \varepsilon, l_1, l_2) = s$. Hence the random string $c \oplus H_{|c|}(y)$ must contain a preimage of s, which is negligible in the random oracle model. Hence, we conclude that the only strategy leading to a valid solution with nonnegligible probability must include a valid computation of $b^{2^{2T-1}} \mod N$, at which point the adversary has no interest in deviating from the honest execution of the protocol.

However, we note that the solver in possession of the trapdoor can compute $b^{2^{2T-1}} \mod N$ using the trapdoor, and is only limited by the solving complexity of the POSW.

Verification hardness. Let us assume that there exists an algorithm $\mathcal{A}$ such that there exists $p > 0$ with $\mathsf{Adv}_v(\mathcal{A}) > p$. If $\mathcal{A}$ takes less than $2T-1$ steps, they cannot compute $b^{2^{2T-1}} \mod N$, hence, using the same argument as above, must verify the POSW with nothing but a random string. We conclude that if $p > 0$ (i.e. if $\mathcal{A}$ has any advantage better than random), then it is impossible to succeed in less than $2T-1$ consecutive steps, hence the complexity.

7 Conclusion and future work

In this paper, we presented the PURED framework, uniting the hardness models over different resources, and explored its properties. We also introduced three new problem classes, with a proven hardness.

Another interesting consideration to add would be to see how one can embed trustless problem generation in the framework.

Finally, we observe that in the literature, many schemes that are trapdoored hard to solve and verify use the same trapdoor for solving and verifying. It would be interesting to investigate on schemes relying on different trapdoors for both solving and verifying, and explore how this particularity could be embedded into the PURED framework.

8 Acknowledgements

The authors would like to thank Aleksei Udovenko for his suggestions on how to improve the HSig-BigLUT performance.

A Related constructions

A.1 Wesolowski's VDF [51]

The description of Wesolowski algorithm is summarised in Algorithm 6.

Algorithm 6 Wesolowski VDF Solve and Verify algorithm [51]
Parameters : G a group of unknown order, $g \in G$, $T \in \mathbb{N}$, H_{prime} hashes to a prime of 2λ bits, bin outputs a binary representation

function SOLVE(g, T)
  $y \leftarrow g^{2^T} \mod N$
  $\pi \leftarrow g^{\lfloor 2^T/\ell \rfloor} \mod N$
end function

function VERIFY$((g, T), (y, \pi))$
  $\ell \leftarrow H_{\text{prime}}(\texttt{bin}(g)||\texttt{bin}(y))$
  $r \leftarrow$ remainder of 2^T divided by ℓ
  return $\pi^\ell g^r \stackrel{?}{=} y$
end function

A.2 BFKW scheme [24]

Given a security parameter λ, a number m of vectors from $\mathbb{F}_p^n$ with $p > 2^\lambda$, the BFKW schemes generates the following (public) parameters:

Two groups $\mathbb{G}, \mathbb{G}_T$ of prime order p, a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, random generators[7] $g_1, \ldots, g_n, h$ of $\mathbb{G}$, a hash function H that maps to $\mathbb{G}$, and id is a public nonce to prevent signature reuse in a different setup[8], as well as vectors $\mathbf{v}_1, \ldots, \mathbf{v}_m$ to be signed.

The secret key is $\mathsf{sk} \stackrel{\$}{\leftarrow} \mathbb{Z}_p$. The public key is $\mathsf{pk} \leftarrow h^{\mathsf{sk}}$. $\mathsf{BFKW}_m.\mathsf{Sign}$ and $\mathsf{BFKW}_m.\mathsf{Verify}$ algorithms are described in Algorithm 7. It is assumed that the decomposition in the base $(\mathbf{v}_1, \ldots, \mathbf{v}_m)$ is an easy task. Furthermore, for any coefficients β_1, β_2, and any two signature pairs $(\mathbf{m}_1, \sigma_1), (\mathbf{m}_2, \sigma_2)$, the signature of the message $\beta_1\mathbf{m}_1 + \beta_2\mathbf{m}_2$ is $\mathsf{BFKW.Combine}(\mathsf{pk}, (\{\beta_1, \sigma_1\}, \{\beta_2, \sigma_2\}) = \sigma_1^{\beta_1}\sigma_2^{\beta_2}$.

A.3 Proofs of successive work [28]

Let $n \in \mathbb{N}$, we create a complete binary tree (V, E) of depth n. G_n^{POSW} is defined as follows: $G_n^{\text{POSW}} = (V, E \cup E')$, where $(u, v) \in E'$ if and only if v is a leaf

[7] For code compactness, a user might prefer to store the random seed leading to these generators.

[8] In our context, the nonce prevents a cheating solver from using twice the same lookup table for two different code-hard instances.

Algorithm 7 BFKW$_m$ Sign and Verify algorithms over the m-dimensional vector space $\mathsf{Span}(\mathfrak{B}) \subset \mathbb{F}_p^n$, with $\mathfrak{B} = (\mathbf{v}_1, \ldots, \mathbf{v}_m)$ [24]

function BFKW$_m$.SIGN($\mathsf{sk}, \mathbf{w}$)
Decompose w in $\mathfrak{B}$: $w = \sum_{i=1}^{m} \alpha_i \mathbf{v}_i$
return $\left(\prod_{i=1}^{m} H(id, i)^{\alpha_i} \prod_{j=1}^{n} g_j^{w_j} \right)^{\mathsf{sk}}$
▷ w_j is the j-th coordinate of $\mathbf{w}$
end function

function BFKW$_m$.VERIFY($\mathsf{pk}, \mathbf{w}, \sigma$)
Decompose w in $\mathfrak{B}$: $w = \sum_{i=1}^{m} \alpha_i \mathbf{v}_i$
return $e(h, \sigma) \stackrel{?}{=} e\left(\mathsf{pk}, \prod_{i=1}^{m} H(\mathsf{id}, i)^{\alpha_i} \prod_{j=1}^{n} g_j^{w_j} \right)$
end function

and u is a left sibling of a node belonging to the shortest path from v to the root. We also identify each node of depth n with a string, from 0^n to 1^n, in the lexicographic order from left to right (the root is identified with ε). As an example, we give G_4^{POSW} in Figure 2.

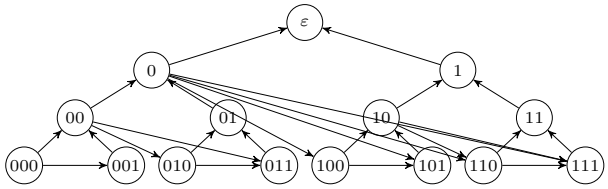

Fig. 2. Representation of the DAG G_4^{POSW}

The proof, for a challenge x, in computing the label of the root of the DAG, where the label of a node u having parents[9] $p_1, \ldots, p_k$ each of respective label $l_{p_1}, \ldots, l_{p_k}$ is given by $l_u = H(x, u, l_{p_1}, \ldots, l_{p_k})$, with H a hash function.

The POSW solving consists in labelling the root of the DAG. A node label depends on all its childrens nodes labels, thus a intuitively a solver must iteratively label each node. The solution to the problem instance consists of the root of the tree, along with a random opening of the tree à la Merkle: for a challenge leaf node u, the solver sends the labels of u along with the labels of all the siblings of the nodes on the path from u to the root ε. Then, the verifier checks that the labels do lead to the proposed solution.

Theorem 8 (from [28]). *The problem class $POSW_n$, using the DAG G_n^{POSW}, the hash function H of codomain $\{0,1\}^w$, is systematically $(p, \mathsf{SeqTime}, 2^{n+1} - 1, \delta, \mathbb{1}_{\delta \geq \left(1 - p - \frac{2nwq^2}{2^w}\right)^{1/n}})$ solving hard in the random oracle model, where q is the total number of (not necessarily sequential) queries to H.*

Moreover, there exists a verification algorithm that takes no more than n steps in $\mathsf{SeqTime}$.

[9] For an oriented graph $G = (V, E)$, u is a parent of v if $(u, v) \in E$.

References

1. Abadi, M., Burrows, M., Manasse, M., Wobber, T.: Moderately hard, memory-bound functions. ACM Trans. Internet Technol. **5**(2), 299–327 (may 2005)
2. Abliz, M., Znati, T.: A guided tour puzzle for denial of service prevention. In: 2009 Annual Computer Security Applications Conference. pp. 279–288 (2009)
3. Ali, I.M., Caprolu, M., Pietro, R.D.: Foundations, properties, and security applications of puzzles: A survey. ACM Comput. Surv. **53**(4), 72:1–72:38 (2020)
4. Alwen, J., Blocki, J.: Efficiently computing data-independent memory-hard functions. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016. pp. 241–271. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
5. Alwen, J., Blocki, J., Pietrzak, K.: Sustained space complexity. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II. Lecture Notes in Computer Science, vol. 10821, pp. 99–130. Springer (2018). https://doi.org/10.1007/978-3-319-78375-8_4, https://doi.org/10.1007/978-3-319-78375-8_4
6. Alwen, J., Chen, B., Pietrzak, K., Reyzin, L., Tessaro, S.: Scrypt is maximally memory-hard. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 33–62. Springer International Publishing, Cham (2017)
7. Alwen, J., Gazi, P., Kamath, C., Klein, K., Osang, G., Pietrzak, K., Reyzin, L., Rolinek, M., Rybar, M.: On the memory-hardness of data-independent password-hashing functions. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. p. 51–65. ASIACCS '18, Association for Computing Machinery, New York, NY, USA (2018)
8. Alwen, J., Serbinenko, V.: High parallel complexity graphs and memory-hard functions. In: Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing. p. 595–603. STOC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2746539.2746622, https://doi.org/10.1145/2746539.2746622
9. Ateniese, G., Chen, L., Francati, D., Papadopoulos, D., Tang, Q.: Verifiable capacity-bound functions: A new primitive from kolmogorov complexity (revisiting space-based security in the adaptive setting). Cryptology ePrint Archive, Paper 2021/162 (2021), https://eprint.iacr.org/2021/162, https://eprint.iacr.org/2021/162
10. Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: Koutsougeras, C., Vitter, J.S. (eds.) Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA. pp. 21–31. ACM (1991). https://doi.org/10.1145/103418.103428, https://doi.org/10.1145/103418.103428
11. Back, A.: Hashcash - a denial of service counter-measure (2002)
12. Baldimtsi, F., Kiayias, A., Zacharias, T., Zhang, B.: Indistinguishable proofs of work or knowledge. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 902–933. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
13. Biryukov, A., Bouillaguet, C., Khovratovich, D.: Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key (extended abstract). In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology – ASIACRYPT 2014. pp. 63–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

14. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New generation of memory-hard functions for password hashing and other applications. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 292–302 (2016). https://doi.org/10.1109/EuroSP.2016.31
15. Biryukov, A., Khovratovich, D.: Egalitarian computing (MTP 1.2). CoRR **abs/1606.03588** (2016), http://arxiv.org/abs/1606.03588
16. Biryukov, A., Khovratovich, D.: Equihash: Asymmetric proof-of-work based on the generalized birthday problem. Ledger **2**, 1–30 (Apr 2017). https://doi.org/10.5195/ledger.2017.48, https://ledger.pitt.edu/ojs/ledger/article/view/48
17. Biryukov, A., Perrin, L.: Symmetrically and asymmetrically hard cryptography. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part III. Lecture Notes in Computer Science, vol. 10626, pp. 417–445. Springer (2017). https://doi.org/10.1007/978-3-319-70700-6_15, https://doi.org/10.1007/978-3-319-70700-6_15
18. Blocki, J., Ren, L., Zhou, S.: Bandwidth-hard functions: Reductions and lower bounds. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 1820–1836. CCS '18, Association for Computing Machinery, New York, NY, USA (2018)
19. Blum, L., Blum, M., Shub, M.: Comparison of two pseudo-random number generators. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) Advances in Cryptology. pp. 61–78. Springer US, Boston, MA (1983)
20. Bogdanov, A., Isobe, T.: White-box cryptography revisited: Space-hard ciphers. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 1058–1069. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813699, https://doi.org/10.1145/2810103.2813699
21. Bogdanov, A., Isobe, T., Tischhauser, E.: Towards practical whitebox cryptography: Optimizing efficiency and space hardness. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 126–158. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
22. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 757–788. Springer International Publishing, Cham (2018)
23. Boneh, D., Corrigan-Gibbs, H., Schechter, S.: Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 220–248. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
24. Boneh, D., Freeman, D., Katz, J., Waters, B.: Signing a linear subspace: Signature schemes for network coding. In: Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5443, pp. 68–87. Springer (2009)
25. Chase, M., Lysyanskaya, A.: On signatures of knowledge. In: Dwork, C. (ed.) Advances in Cryptology - CRYPTO 2006. pp. 78–96. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
26. Chen, M., Doerner, J., Kondi, Y., Lee, E., Rosefield, S., Shelat, A., Cohen, R.: Multiparty generation of an rsa modulus. Journal of Cryptology **35**(2), 12 (Mar 2022)

27. Coelho, F., Larroche, A., Colin, B.: Itsuku: a memory-hardened proof-of-work scheme. IACR Cryptol. ePrint Arch. p. 1168 (2017), http://eprint.iacr.org/2017/1168
28. Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018. pp. 451–467. Springer International Publishing, Cham (2018)
29. De Feo, L., Masson, S., Petit, C., Sanso, A.: Verifiable delay functions from supersingular isogenies and pairings. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology – ASIACRYPT 2019. pp. 248–277. Springer International Publishing, Cham (2019)
30. Delerablée, C., Lepoint, T., Paillier, P., Rivain, M.: White-box security notions for symmetric encryption schemes. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) Selected Areas in Cryptography – SAC 2013. pp. 247–264. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
31. Dobson, S., Galbraith, S.D.: Trustless groups of unknown order with hyperelliptic curves. IACR Cryptol. ePrint Arch. p. 196 (2020), https://eprint.iacr.org/2020/196
32. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) Advances in Cryptology — CRYPTO' 92. pp. 139–147. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
33. Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: Gennaro, R., Robshaw, M. (eds.) Advances in Cryptology – CRYPTO 2015. pp. 585–605. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
34. Ethereum Community: ethash — ethereum wiki, https://eth.wiki/en/concepts/ethash/ethash
35. Garg, S., Gentry, C., Sahai, A., Waters, B.: Witness encryption and its applications. In: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing. p. 467–476. STOC '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2488608.2488667, https://doi.org/10.1145/2488608.2488667
36. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing. p. 291–304. STOC '85, Association for Computing Machinery, New York, NY, USA (1985)
37. Kamara, S.: Proofs of storage: Theory, constructions and applications. In: Muntean, T., Poulakis, D., Rolland, R. (eds.) Algebraic Informatics. pp. 7–8. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
38. Liu, J., Jager, T., Kakvi, S.A., Warinschi, B.: How to build time-lock encryption. Designs, Codes and Cryptography **86**(11), 2549–2586 (Nov 2018). https://doi.org/10.1007/s10623-018-0461-x, https://doi.org/10.1007/s10623-018-0461-x
39. Mahmoody, M., Moran, T., Vadhan, S.: Publicly verifiable proofs of sequential work. In: Proceedings of the 4th Conference on Innovations in Theoretical Computer Science. p. 373–388. ITCS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2422436.2422479, https://doi.org/10.1145/2422436.2422479
40. Percival, C.: Stronger key derivation via sequential memory-hard functions (2009)
41. Pietrzak, K.: Simple verifiable delay functions. In: Blum, A. (ed.) 10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA. LIPIcs, vol. 124, pp. 60:1–60:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITCS.2019.60, https://doi.org/10.4230/LIPIcs.ITCS.2019.60

42. Poupard, G., Stern, J.: Short proofs of knowledge for factoring. In: Imai, H., Zheng, Y. (eds.) Public Key Cryptography. pp. 147–166. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
43. Provos, N., Mazières, D.: A future-adaptive password scheme. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. p. 32. ATEC '99, USENIX Association, USA (1999)
44. Ren, L., Devadas, S.: Bandwidth hard functions for asic resistance. In: Kalai, Y., Reyzin, L. (eds.) Theory of Cryptography. pp. 466–492. Springer International Publishing, Cham (2017)
45. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep., Massachusetts Institute of Technology, USA (1996)
46. Rivest, R.L.: Description of the LCS35 time capsule crypto-puzzle. https://people.csail.mit.edu/rivest/lcs35-puzzle-description.txt (1999)
47. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) Advances in Cryptology — CRYPTO' 89 Proceedings. pp. 239–252. Springer New York, New York, NY (1990)
48. Thompson, C.D.: Area-time complexity for vlsi. In: Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing. p. 81–88. STOC '79, Association for Computing Machinery, New York, NY, USA (1979)
49. Vitto, G.: Factoring primes to factor moduli: Backdooring and distributed generation of semiprimes. IACR Cryptol. ePrint Arch. p. 1610 (2021), https://eprint.iacr.org/2021/1610
50. Walfish, M., Vutukuru, M., Balakrishnan, H., Karger, D., Shenker, S.: Ddos defense by offense. In: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. p. 303–314. SIGCOMM '06, Association for Computing Machinery, New York, NY, USA (2006)
51. Wesolowski, B.: Efficient verifiable delay functions. J. Cryptol. **33**(4), 2113–2147 (2020). https://doi.org/10.1007/s00145-020-09364-x, https://doi.org/10.1007/s00145-020-09364-x
52. Youden, W.J.: Index for rating diagnostic tests. Cancer **3**(1), 32–35 (Jan 1950)