Timed Secret Sharing

Anonymous Author(s)

1 INTRODUCTION

Secret sharing has been a promising tool in cryptographic schemes for decades. It allows a dealer to split a secret into some pieces of shares that carry no sensitive information on their own when being treated individually, but lead to the original secret when having a sufficient number of them together. In a threshold secret sharing scheme, which is the focus of this work, any number of parties that meet a specified threshold is considered to be sufficient for secret reconstruction. This design rational has been proven to be useful in a wide range of cryptographic applications where the system is vulnerable to a single point of failure.

Traditionally, the threshold secret sharing schemes have been defined and deployed without any notion of *time* explicitly involved. That is, once the dealer shares the secret, she goes offline and it is up to a threshold of shareholders to get together and recover the secret. However, there are applications where the dealer needs to ensure that the secret will not be reconstructed until some time in the future while she is not available to perform the sharing then. An illustrative example can be a type of *distributed* Dead Man's Switch [23], where the dealer wishes to share her secret to a set of switches to be opened only after passing some time (*e.g.*, after her death).

At first glance, the underlying honest majority assumption of the threshold secret sharing schemes seems to be addressing such delayed action if having a time hardcoded in the system, as the honest parties are supposed to follow the protocol. However, there are two clear reasons why this cannot provide a proper solution. First and foremost, such an assumption is indeed not reasonable when it comes to a matter of life and death like that of Dead Man's Switch. Here the dealer may wish to have a guarantee that nobody learns the secret whatsoever *without* making any assumption about the shareholders. Second, providing the shareholders with a consistent notion of time is challenging. Moreover, there are secret sharing-based applications where the shareholder(s) can perform the reconstruction if the received shares meet some condition. For instance, In STAR [20], which is a private threshold aggregation reporting scheme, if a threshold of users (*i.e.*, dealers) with the *same* measurement (*i.e.*, secret) send their shares to the server (*i.e.*, shareholder) she can successfully perform the reconstruction; otherwise, no sensitive information will be leaked. It is easy to see once sharing is done by the users, the server can obtain the measurement if the condition meets, negating the usefulness of any implicit timing assumption.

We also consider having an upper time-bound, a time by then the secret should get reconstructed. For this, we can take the simple but common example of paper submission in a venue. When having the paper (*i.e.*, secret) shared, the reconstruction should take place before some deadline; otherwise, the reconstructor (*i.e.*, submission server) would not accept the submission.

This Paper. Motivated by what already mentioned, in this paper we introduce *timed secret sharing* (TSS) scheme, allowing a dealer to share a secret to a set of n shareholders such that it can only be reconstructed by a threshold t of shareholders no sooner than a lower time bound T_1, and be accepted by a reconstrcutor no later than an upper time bound T_2. To provide verifiability for the scheme, we propose *verifiable timed secret sharing* (VTSS), protecting against malicious dealer while sharing and malicious shareholders while reconstructing. We then take a step further and present *publicly verifiable timed secret sharing* (PVTSS) scheme, allowing public verification.

As side contributions, we introduce the idea of *secret sharing with additional shares*, providing robustness property in case less than a threshold of (honest) parties are available during the reconstruction period. This then leads us to our last scheme called *decrementing-threshold secret sharing* (DTSS), establishing an interesting trade-off between time and fault-tolerance in a threshold secret sharing scheme. Finally, we explore some concrete applications for our constructions.

Our Approaches. We base our constructions on time-based cryptographic primitives, in particular, *time-lock puzzles* (TLPs) [1, 40], and *verifiable delay functions* (VDFs) [7, 39, 49]. Basically, to realize the lower time bound T_1, the dealer makes the shares unavailable up to time T_1 by putting them into TLPs. This treatment has two consequences. First, no computationally bounded shareholders can learn their shares before T_1. So, even if all the shareholders collude they cannot recover the secret. Second, working through TLPs that resist parallel computing provides a consistent relative timing (*i.e.*, computational timing) for the participating parties, eliminating the need for a shared global clock. The first step to make the scheme publicly verifiable is to deploy a *publicly verifiable secret sharing* (PVSS) scheme [14, 42]. Moreover, we also need to protect against the possibility of distributing *malformed puzzles* by a dealer, *i.e.*, puzzles that either cannot be opened or contain invalid shares. It is problematic that such flawed action is detected by parties only after spending time and computational power on the process of obtaining/unlocking the shares. We get around this problem by using *linearly homomorphic time lock puzzles* (LHTLP)[16], which are not verifiable originally, but have been augmented with such property in the recent work of [33]. We then construct efficient sigma protocols, allowing us to simultaneously ensure the extractability of the puzzles and validity of the embedded share.

To realize the upper time bound T_2, we take two treatments. The first assumes an (honest) verifier/reconstructor who remains online and observes the protocol execution (*e.g.*, a participating party). This way, she can simply reject any share received after T_2. The second treatment deals with a situation where the verifier/reconstructor may go offline and not observe the protocol. This allows us to highlight the role of T_2 by making use of *short-lived proofs* (SLPs)

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–24

https://doi.org/XXXXXXX.XXXXXXX

[3], a recent advance in the literature that fastens the soundness of a proof system to the notion of time. At a high level, the idea behind SLPs is to enable forgeability by establishing a *compound* NP relation for a disjunction of two statements, one with a witness known to prover and the one expressing a VDF computation. So, such structure causes SLPs to guarantee soundness w.r.t. witness only if being received/verified before some time T; otherwise, it might be a forgery produced after computing the VDF. We utilize this tool to make the correct secret reconstruction dependent on the upper time bound T_2. We remark that the independency of the approach used to provide the lower and upper time bounds makes it possible to treat them separately. That is, we can think of having a *delayed secret sharing* that works just by locking shares in the distribution phase and doing the reconstruction phase as normal. Likewise, we can only define an upper time bound for secret reconstruction and ignore the share locking step in the distribution.

In threshold secret sharing schemes it is typically assumed that a threshold of shares is available at the reconstruction phase. In fact, this is necessary for the *correctness* of the scheme. We observe that it is possible to relax this assumption in a TSS scheme by having the dealer provide *additional time-locked shares*, offering *robustness* in case only *one* honest party remains online while maintaining security against an adversary controlling up to t parties. We then generalize this idea by having the dealer release each additional time-locked share periodically over time, while she goes offline after the sharing phase just like the normal secret sharing schemes. Multi-instance time-lock puzzle, a recent work by Abadi and Kiayias [1], allows us to implement this scheme efficiently. It turns out such design features *gradual* robustness and fault-tolerance reduction concurrently, so we call it decrementing-threshold secret sharing secret sharing. We notice that such fault-tolerance reduction over time can introduce an off-chain mechanism to break public goods game [6, 31] that is inherent in threshold secret sharing schemes, where only a threshold of shares is sufficient to reconstruction the secret. More precisely, the possibility of having an adversary obtain the secret ahead of others over time implicitly forces parties to step in and release their shares. The on-chain solutions attempt to solve this issue by rewarding the shareholders who publish theirs shares sooner.

2 RELATED WORK

There is a large body of works on the combination of computational timing and cryptographic primitives such as commitment [2, 9, 22, 36, 47], encryption [12, 18, 34], signature [5, 21, 26, 46], and more. The essence of almost all of these works is to enable the receiver(s) to forcefully open the locked object after a predefined amount of time by working through some computational operation. Boneh and Naor [9] put forth the notion of timed commitment where, before solving the puzzle, the receiver gets convinced that it is well-formed by running an interactive protocol with the prover. Recently, Manevich and Akavia [36] presented a primitive called attribute verifiable timed commitments (AVTC) as an enhancement on [9], enabling the sender to convince the receiver that the committed secret possesses some *arbitrary* attribute. They do so by augmenting the original timed commitment with an interactive zero-knowledge proof. Notice that, technically speaking we could make a black box use of AVTC in PVTSS to protect against a malicious dealer, however, the construction given in [36] is for generic attributes and therefore involves elaborated interactive operations and number theoretical assumptions which makes it unsuitable for the setting of secret sharing.

The recent work of [46] proposed practically efficient constructions for encapsulating a signature into a TLP, ensuring the receiver can *extract* the *valid* signature after carrying out sequential computation. At a high level, the sender secret shares the signature and embeds each share in a linearly homomorphic TLP [16]. Then, the sender and receiver run a cut-and-choose protocol for verifying the correctness of the puzzles. Moreover, to enable receive to compact all the pieces of time-locked signatures and solve one single puzzle, a range proof is used to guarantee that no overflow occurs. The idea of using cut-and-choose technique for TLPs to protect against a malicious sender was presented in [5]. With a focus on reducing the interaction in MPC protocols with limited-time secrecy, [2] proposed a gage time capsule (GaTC), allowing a sender to commit to a value that others can obtain it after putting a total computational cost which is parallelizable to let each solver claim a monetary reward in exchange for their work. The security guarantee of GaTC is similar to DTSS in the sense that over time it gradually decays, as the adversary can invest more and more computational resources. Doweck and Eyal [22] constructed a multi-party timed commitment (MPTC) that enables a group of parties to jointly commit to a secret to be opened by an aggregator later on via brute-force computation. The MPTC primitive of [22] does not allow one to prove any attribute for the committed secret, unlike the AVTC primitive. Moreover, the MPTC commitment opening is parallelizable, like the GaTC primitive.

A timed-release encryption (TRE) scheme is presented in [18], enabling anyone to encrypt a message under a public key where the information (*i.e.*, randomness) needed to derive the corresponding secret key is encoded into a TLP. So, anybody can decrypt the ciphertext after solving the puzzle. Deploying TLE of [18] in our PVTSS does not work as all the parties should be able to verify the validity of the encrypted shares before solving the puzzles. In addition, in PVTSS each party chooses its key pairs, while in TLE the puzzle generator does that. As performing sequential computations might be beyond the capacity of some users, [48] develops a system to allow users to outsource their tasks to some servers in a privacy-preserving manner. Very recently, the work of [45] constructed a TLP that supports unbounded batch-solving while enjoying a transparent setup and a puzzle size independent of the batch size. Although their construction is only of theoretical importance and does not have practical efficiency, it enables a party to solve many puzzle instances simultaneously at the cost of solving one puzzle (*i.e.*, batch solving). It is worth noting that such a setting is not applicable to our work as each party just needs to know its own share and solving other parties' puzzles gives her nothing but some ciphertext. One of the motivating reasons for TLP of [45] is to let a party solves the puzzles of others (via batch solving) in case a large number of parties abort. In Section 5, we solve such a liveness issue via a different technique.

3 PRELIMINARIES

3.1 Threat Model

We consider a standard synchronous network where each pair of parties in a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ is connected via an authenticated communication channel and each message is delivered within a time-bound. There is also a dealer D that takes the role of distributing the secret among participating parties. As common in the literature for verifiable secret sharing schemes, we assume the existence of broadcast channels. For publicly verifiable scheme, we assume the existence of an authenticated bulletin board which once a messages is posted to, it becomes available to everyone. We assume a static adversary that may corrupt up to t out of n parties, with $n = 2t + 1$ being the optimal fault tolerance in synchronous setting. The dealer D can also be corrupted. Static adversary chooses their corruption targets before the start of protocol execution. Corruption occurs in two forms of curious/semi-honest or malicious. In the former, the corrupted parties are assumed to follow the protocol but may try to learn some information by observing the protocol execution. In the latter, however, the corrupted parties are allowed to do any adversarial behavior of their choice. Apart from the aforementioned corruption models, the secret sharing scheme also tolerates a threshold of crash faults referring to the parties that follow the protocol but may stop working at some point and fail to send or receive any messages afterwards. The adversary's computational power is bounded with respect to a security parameter λ that gives her a negligible advantage in breaking the security of underlying primitives. Such algorithms are often known as probabilistic polynomial time (PPT). Finally, we denote by $[n]$ the set $\{1, \ldots, n\}$.

3.2 Secret Sharing

The core building block of our constructions is secret sharing scheme where allows a dealer D to distribute a secret s among a set of n parties $\mathcal{P} = \{P_1, \ldots, P_n\}$. The scheme typically consists of two main phases of *distribution* and *reconstruction*. In the former, the dealer sends each party their corresponding share, and in the latter, any proper set of parties can uniquely reconstruct the secret by pooling their received shares. A (t, n)[1] threshold secret sharing guarantees that the secret is reconstructed by any subset of at least $t+1$ shares (*i.e.*, correctness) while no information is revealed about secret by gathering any fewer shares (*i.e.*, t-security). In this work we focus on Shamir secret sharing [43] for proposing concrete constructions which is based on polynomial interpolation over a finite field of prime order. However, we present the definitions generically that capture any linear secret sharing.

Verifiable Secret Sharing (VSS). The basic (t, n) threshold secret sharing scheme of [43] only provides security against *passive* adversary, meaning that its security is guaranteed as long as the participating parties run the protocol as specified by the scheme. When considering malicious adversary, it is required to have dealer ensure parties about the validity of sharing and parties ensure the reconstructor about the validity of their submitted shares. To this end, verifiable secret sharing (VSS) schemes [24, 30] provided verifiability in secret sharing. Moreover, apart from satisfying the t-security condition likewise the basic setting, here the scheme requires an additional assumption of $n - t$-*robustness* that guarantees there is enough number of shares to reconstruct the secret in the case some parties refuse to take part and give back there shares.

Publicly Verifiable Secret Sharing (PVSS). To extend the scope of verifiability to the public, not only participating parties, publicly verifiable secret sharing (PVSS) schemes [14, 42] deploy some cryptographic primitives such as encryption and non-interactive zero-knowledge (NIZK) proofs to enable any external verifier to verify the distribution phase by the dealer and reconstructing phase by the parties. Such publicity in verification is of importance in some applications like randomness beacons [14].

3.3 Time-lock Puzzle

A time-lock puzzle (TLP) locks a secret such that it can be retrieved after a predefined amount of sequential computation.

Definition 1 (Time-lock Puzzle)**.** *A time-lock puzzle (TLP) consists of the following two algorithms:*

(1) $\mathsf{TLP.Gen}(1^\lambda, T, s) \to Z$, *a probabilistic algorithm that takes time parameter T and a secret s, and generates a puzzle Z.*
(2) $\mathsf{TLP.Solve}(T, Z) \to s$, *a deterministic algorithm that solves the puzzle Z and retrieves the secret s.*

A valid TLP scheme must satisfy the two requirements of *correctness* and *security*. The correctness ensures that the solution is indeed obtained if the protocol being executed as specified. The security ensures that no PPT adversary running in parallel obtains the solution within the time bound T, except with negligible probability.

Correctness [16]**.** A TLP scheme is correct if for all $\lambda \in \mathbb{N}$, all polynomials $T(\cdot)$ in λ, and all $s \in S_\lambda$, it holds that

$$\Pr\left[\mathsf{TLP.Solve}(T(\lambda), Z) \to s : \mathsf{TLP.Gen}(1^\lambda, T(\lambda), s) \to Z\right] = 1$$

Security [16]**.** A TLP scheme is secure with gap $\epsilon < 1$ if there exists a polynomial $\tilde{T}(\cdot)$ such that for all polynomials $T(\cdot) \geq \tilde{T}(\cdot)$ and every polynomial-size adversary $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in N}$ of depth $\leq T^\epsilon(\lambda)$, there exists a negligible function $\mu(\cdot)$, such that for all $\lambda \in \mathbb{N}$ and $s_0, s_1 \in \{0,1\}^\lambda$ it holds that

$$\Pr\left[\mathcal{A}(Z) \to b : \mathsf{TLP.Gen}(1^\lambda, T(\lambda), s_b) \to Z, b \xleftarrow{\$} \{0,1\}\right] \leq \frac{1}{2} + \mu(\lambda)$$

In particular, the seminal work of [40] introduced the notion of *encrypting to the future* using an RSA-based TLP. Loosely speaking, the sender encrypts a message m under a key k derived from the solution s to a puzzle Z. So, anybody can obtain m after running $\mathsf{TLP.Solve}(T, Z)$ and learning the key.

Homomorphic Time-lock Puzzle (HTLP). As in some other cryptographic primitives like encryption and commitment, time-lock puzzles can also support homomorphisim, an important property turned out to be useful in many applications like electronic voting. This was first investigated in [16] by utilizing a more sophisticated algebraic structure for the TLP motivated by Paillier encryption [38]. Homomorphic property enables one to combine many instances of TLP embedding a set of solutions $(s_1, \ldots, s_n)$ and turn them into

[1] The security threshold is often denoted by t, but here we use k to avoid confusion with the notion of time t.

one compact TLP embedding an arbitrary function $f(s_1, \ldots, s_n)$. Despite the feasibility of supporting arbitrary functions (*i.e.*, fully homomorphic), the efficient constructions proposed in [16] only support either addition or multiplication (*i.e.*, partially homomorphic).

Definition 2 (Homomorphic Time-Lock Puzzles [16]). *Let $C = \{C_\lambda\}_{\lambda\in\mathbb{N}}$ be a class of circuits and S_λ be a finite domain. A homomorphic time-lock puzzle (HTLP) with respect to C and with solution space S_λ is a tuple of algorithms* (HTLP.Setup, HTLP.Gen, HTLP.Solve, HTLP.Eval) *as follows.*

(1) Setup: $\mathsf{HTLP.Setup}(1^\lambda, T) \to pp$, *a probabilistic algorithm that takes a security parameter 1^λ and time parameter T, and generates public parameters pp.*
(2) Puzzle generation: $\mathsf{HTLP.Gen}(pp, s) \to Z$, *a probabilistic algorithm that takes public parameters pp and a secret $s \in S_\lambda$, and generates a puzzle Z.*
(3) Puzzle solving: $\mathsf{HTLP.Solve}(pp, Z) \to s$, *a deterministic algorithm that takes public parameters pp and puzzle Z, and retrieves a secret s.*
(4) Evaluation: $\mathsf{HTLP.Eval}(C, pp, Z_1, \ldots, Z_n) \to Z'$, *a probabilistic algorithm that takes a circuit $C \in C_\lambda$ and a set of n puzzles $(Z_1, \ldots, Z_n)$, and outputs a puzzle Z'.*

The aforementioned HTLP scheme must satisfy three properties including correctness, security, and compactness. As in this work we do not make use of the compactness property, we only provide the formal security definition and skip the the straightforward correctness definition.

Security [16]. A HTLP scheme (HTLP.Setup, HTLP.Gen, HTLP.Solve, HTLP.Eval) is secure with gap $\epsilon < 1$ if there exists a polynomial $\tilde{T}(\cdot)$ such that for all polynomials $T(\cdot) \geq \tilde{T}(\cdot)$ and every polynomial-size adversary $(\mathcal{A}_1, \mathcal{A}_2) = \{(\mathcal{A}_1, \mathcal{A}_2)_\lambda\}_{\lambda\in N}$ where the depth of $\mathcal{A}_2$ is bounded from above by $T^\epsilon(\lambda)$, there exists a negligible function $\mu(\cdot)$, such that for all $\lambda \in \mathbb{N}$ it holds that

$$\Pr\left[\mathcal{A}_2(pp, Z, \tau) \to b \;:\; \begin{array}{l} \mathcal{A}_1(1^\lambda) \to (\tau, s_0, s_1) \\ \mathsf{HTLP.Setup}(1^\lambda, T(\lambda)) \to pp \\ b \xleftarrow{\$} \{0,1\} \\ \mathsf{HTLP.Gen}(pp, s_b) \to Z \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

The puzzle is defined over group of unknown order and is of form $Z = (u, v)$, where $u = g^r$ and $v = h^{r.N}(1+N)^s$. One notable point regarding the construction is that a trusted setup assumption is needed to generate the public parameters $pp = (T, N, g, h)$, where N is a safe modulus [2] and $h = g^{2^T}$. Such a setup phase is responsible for generating the parameters as specified and keeping the random coins secret; otherwise, either the puzzle is not solvable or one can efficiently solve it in time $t \ll T$. Having said that, the authors in [16] point out this assumption can be removed if construction gets instantiated over class groups instead of an RSA group of unknown order. However, this comes at the cost of a higher computational overhead by the puzzle generator.

[2] A safe modulus is a product of two safe primes $P = 2p' + 1, Q = 2q' + 1$, where p' and q' are prime numbers.

Multi-instance Time-lock Puzzle (MTLP). Abadi and Kiayias [1] proposed a primitive called multi-instance time-lock puzzle (MTLP), allowing composing a puzzle's instances such that the solver solves the instances sequentially one after the other and obtains the solutions at different points in time, without needing to run parallel computations on them [3]. The primitive builds on the classical RSA-based puzzle, making it chained and publicly verifiable against an honest puzzle generator, i.e., assuming an honest puzzle generator, anybody can efficiently check the solver's claimed solution is valid.

Definition 3 (Multi-instance Time-lock Puzzle [1]). *A Multi-instance Time-lock Puzzle (MTLP) consists of the following five algorithms:*

(1) $\mathsf{MTLP.Setup}(1^\lambda, T, z) \to \{pk, sk, \vec{\mathbf{d}}\}$, *a probabilistic algorithm that takes security parameter λ, time parameter T, and the number of puzzles z, and outputs a key pair (pk, sk) and a secret witness vector $\vec{\boldsymbol{d}}$.*
(2) $\mathsf{MTLP.Gen}(\vec{\mathbf{m}}, pk, sk, \vec{\mathbf{d}}) \to \{\vec{\mathbf{o}}, \vec{\mathbf{h}}\}$, *a probabilistic algorithm that takes a message vector $\vec{\boldsymbol{m}}$, the public-private key (pk, sk), secret witness vector $\vec{\boldsymbol{d}}$, and outputs a puzzle vector $\vec{\boldsymbol{o}}$ and a commitment vector $\vec{\boldsymbol{h}}$.*
(3) $\mathsf{MTLP.Solve}(pk, \vec{\mathbf{o}}) \to \vec{\mathbf{s}}$, *a deterministic algorithm that takes the public key pk and the puzzle vector $\vec{\boldsymbol{o}}$, and outputs a solution vector $\vec{\boldsymbol{s}}$, where s_j is of form $m_j \,||\, d_j$.*
(4) $\mathsf{Prove}(pk, s_j) \to \pi_j$, *a deterministic algorithm that takes the public key pk and a solution s_j, and outputs a proof π_j.*
(5) $\mathsf{Verify}(pk, \pi_j, h_j) \to \{0, 1\}$, *a deterministic algorithm that takes the public key pk, proof π_j, and commitment h_j. If verification succeeds, it outputs 1, otherwise 0.*

The described MTLP scheme has three properties including correctness, security, and validity. We provide the formal security and validity definitions and skip the the straightforward correctness definition.

Security [1]. A multi-instance time-lock puzzle is secure if for all λ and T, any number of puzzle: $z \geq 1$, any j (where $1 \leq j \leq z$), any pair of randomised algorithm $\mathcal{A} : (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in time $O(poly(jT, \lambda))$ and $\mathcal{A}_2$ runs in time $\delta(jT) < jT$ using at most $\pi(T)$ parallel processors, there exists a negligible function $\mu(.)$, such that:

$$\Pr\left[\begin{array}{l} \mathcal{A}_2(pk, \ddot{o}, \tau) \to \ddot{a} \\ \text{s.t.} \\ \ddot{a} : (b_i, i) \\ m_{b_i,i} = m_{b_{j'},j} \end{array} \;:\; \begin{array}{l} \mathsf{MTLP.Setup}(1^\lambda, \Delta, z) \to (pk, sk, \vec{\mathbf{d}}) \\ \mathcal{A}_1(1^\lambda, pk, z) \to (\tau, \vec{\mathbf{m}}) \\ \forall j', 1 \leq j' \leq z : b_{j'} \xleftarrow{\$} \{0,1\} \\ \mathsf{MTLP.Gen}(\vec{\mathbf{m}}', pk, sk, \vec{\mathbf{d}}) \to \ddot{o} \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

3.4 Timed Commitment

An inherent limitation of the well-known time-lock puzzles such as [16, 40] is the lack of verifiability, meaning that the receiver cannot check the validity of the received puzzle unless after putting time and effort in solving it. To fill this gap, a timed commitment scheme [9] enables the receiver to ensure about the well-formedness of the puzzle before start solving it. The recent work of [36] augments the timed commitment of Boneh and Naor [9] with a zero knowledge proof, enabling the sender to prove any arbitrary attribute regarding

[3] If done naively, the solver has to deal with each puzzle separately, requiring substantial parallelisation.

the committed value. Intuitively, given a statement and the corresponding witness $(v; w) \in R$, they follow the MPC in the head framework [28] to construct a predicate $\mathcal{F} : \{0,1\}^* \rightarrow \{0,1\}$ that verifies the committed value (*i.e.*, witness) indeed satisfies the relation R with respect to the public statement v. To do so, they introduce attribute verifiable timed commitment (AVTC) primitive with the tuple of three algorithms (AVTC.Commit, AVTC.Open, AVTC.Solve). The first algorithm allows the sender to commit to a witness w while guaranteeing that $(v, w) \in R$. The second algorithm opens the commitment by the sender and the last one enables the receiver to open the commitment via sequential computation. Note that in the context of VTSS we only make a black box use of the first and the last algorithms.

3.5 Verifiable Delay Function

Another important time-based cryptographic primitive is verifiable delay function (VDF) [7, 39, 49]. Similar to TLP, VDF imposes some computational delay via evaluating a sequential function such that deploying parallelism does not lead to a significant advantage in computing the output in less than time T. Moreover, the VDF provides a unique output and a proof of correctness that can be verified efficiently by anyone. Here efficiency means at most poly-logarithmic run in the time and security parameter $poly(\log T, \lambda)$.

Definition 4 (Verifiable Delay Function)**.** *A verifiable delay function (VDF) consists of the following three algorithms:*

(1) $\mathsf{VDF.Setup}(1^\lambda, T) \rightarrow pp$, *a probabilistic algorithm that takes security parameter λ and time parameter T, and generates system parameters pp.*
(2) $\mathsf{VDF.Eval}(pp, x) \rightarrow \{y, \pi\}$ *a deterministic algorithm that given system parameters pp and a randomly chosen input x, computes a unique output y and a proof π.*
(3) $\mathsf{VDF.Verify}(pp, x, y, \pi) \rightarrow \{0,1\}$, *a deterministic algorithm that verifies y indeed is a correct evaluation of the x. If verification succeeds, the algorithm outputs 1, and otherwise 0.*

Intuitively, there are three security properties that a valid VDF should satisfy. There must be a run time constraint of $(1+\epsilon)T$ for a positive constant ϵ limiting the evaluation algorithm, called *ϵ-sequentially*. The VDF should have *sequentially*, meaning no adversary using parallel processors can successfully compute the output without executing the T sequential computation. Lastly, the VDF evaluation should be a function with *uniqueness* property. That is, the verification algorithm must accept only one output per input.

VDF constructions. Among variety of constructions, VDFs based on repeated squaring have gained more attention as they offer simple evaluation function that is more compatible to the hardware and provide better accuracy in terms of the time needed to perform the computation. The two concurrent works of [39, 49] suggest evaluating the function $y = x^{2^T}$ over a hidden-order group. Despite similarity in construction, they present two independent ways for proof generation. Particularly, the one proposed by Wesolowski [49] enjoys the luxury of having a constant size proof and verification cost. In addition, Wesolowski's construction can be instantiated over class groups of imaginary quadratic fields [11] where do not require trusted setup assumption.

3.6 Sigma protocols

A zero-knowledge protocol enables proving the validity of a claimed statement by the prover P to the verifier V without revealing any information further. While zero-knowledge protocols involve a rich body of settings and notions, we particulalry consider a well-known type of them called Sigma protocols where are useful building blocks in many cryptographic constructions. Let v denote an instance that is known to both parties and w denote a witness that is only known to the P. Let $R = \{(v; w)\} \in \mathcal{V} \times \mathcal{W}$ denote a relation containing the pairs of instances and corresponding witnesses. A Sigma protocol Σ on the $(v; w) \in R$ is an interactive protocol with three movements between P and V as follows.

(1) Announcement: $\Sigma.\mathsf{Ann}(v, w) \rightarrow a$, runs by P and outputs a message a to V.
(2) Challenge: $\Sigma.\mathsf{Cha}(v) \rightarrow c$, runs by V and outputs a message c to P.
(3) Response: $\Sigma.\mathsf{Res}(v, w, c) \rightarrow r$, runs by P and outputs a message r to V.
(4) Verify: $\Sigma.\mathsf{Ver}(v, a, c, r) \rightarrow \{0,1\}$, runs by V and outputs 1 if statement holds.

A Sigma protocol has three main properties including *completeness*, *knowledge soundness*, and *zero-knowledge*. Completeness guarantees the verifier gets convinced if parties follow the the protocol. Knowledge soundness states that a malicious prover P^* cannot convince the verifier of a statement without knowing its corresponding witness except with a negligible probability. This is formalised by considering an efficient algorithm called *extractor* to extract the witness given a pair of valid protocol transcripts with different challenges showing the computational infeasibility of having such pairs and therefore guaranteeing the knowledge of witness by P. The notion of zero-knowledge ensures that no information is leaked to the verifier regarding the witness. This is formalised by considering an efficient algorithm called *simulator* which given the instance v, and also the challenge c, outputs a simulated transcript that is indistinguishable from the transcript of the actual protocol execution. Note that this property only needs to hold against an *honest verifier* which seems to be a limitation of the description, but allows for having much more efficient constructions compared to generic models. The interactive protocol described above can be easily turned into a non-interactive variant using the Fiat-Shamir heuristic [25] in the random oracle model, making it publicly verifiable with no honest verifier assumption.

Zero knowledge Proof of Equality of Discrete Logarithm. Supporting different compositions makes Sigma protocols super powerful in solving problems where the prover wishes to convince the verifier not only of one fact, but also a combination of facts. One of the common Sigma protocols is discrete logarithm equality (DLEQ) proof. That is, for a tuple of publicly known values (g_1, x, g_2, y), where g_1, g_2 are random generators and x, y are two elements of the cyclic group $\mathbb{G}$ of order q, respectively, the DLEQ proof enables a prover P to prove to the verifier V that she knows a witness α such that $x = g_1^\alpha$ and $y = g_2^\alpha$. A DLEQ proof is in fact an AND-composition of two Sigma protocols for relation $R = \{(v_i; w) : v_i = g_i^w\}$ with the *same* witness and challenge. The following protocol is a Sigma

protocol for generating a DLEQ proof due to Chaum-Pedersen [15].

(1) P chooses a random element $u \leftarrow \mathbb{Z}_q$, computes $a_1 = g_1^u$ and $a_2 = g_2^u$, and sends them to the V.
(2) V sends back a randomly chosen challenge $c \leftarrow \mathbb{Z}_q$.
(3) P computes $r = u + c\alpha$ and sends it to V.
(4) V checks if both $g_1^r = a_1 x^c$ and $g_2^r = a_2 y^c$ hold.

Throughout the paper we use the non-interactive version of this protocol which produces a single message $\mathsf{DLEQ.P}(\alpha, g_1, x, g_2, y)$ as proof π verified via $\mathsf{DLEQ.V}(\pi, g_1, x, g_2, y)$. The challenge is computed by the prover as $c = H(x, y, a_1, a_2)$, where H is a cryptographic hash function modelled as random oracle.

3.7 Short-lived proofs

Very recently Arun et al. [3] put forth the notion of *short-lived proofs* (SLPs) where can be roughly defined as types of proofs with expiration such that their soundness will disappear after a period of time. The deniable nature of these proofs resembles the notion of designated verifier proofs [29], where limit the verification to only one entity leaving anybody else unsure of the validity of the claimed statement. This is achieved by generating a proof on the composition of two statements where the designated verifer is only aware of one of them, making the proof convincing for the other one. The authors in [3] follow this direction, augmenting it with the notion of time and public-verifiability. At a high level, an SLP is proof of an OR-composition $R \vee R_{VDF}$, where R is an arbitrary relation and R_{VDF} is a VDF evaluation relation. Interestingly, this proof is only convincing to the verifier for a determined period of time T as forging the proof is possible after performing some sequential computation for evaluating the VDF. Since anybody can come up with a VDF proof which provides efficient public verification, short-lived proofs are publicly verifiable, making them more versatile compared to designated verifier proofs.

Definition 5 (Short-lived Proof). *A short-lived proof scheme includes a tuple of following algorithms:*

(1) $\mathsf{SLP.Setup}(1^\lambda, T) \rightarrow pp$, *a probabilistic algorithm that takes security parameter λ and time parameter T, and generates public parameters pp used for the following algorithms.*
(2) $\mathsf{SLP.Gen}(pp, v, w, b) \rightarrow \pi$, *a probabilistic algorithm that takes a $(v; w) \in R$ and a random value b, and generates a proof π.*
(3) $\mathsf{SLP.Forge}(pp, v, b) \rightarrow \pi$, *a probabilistic algorithm that takes any instance v and a random value b, and generates a proof π.*
(4) $\mathsf{SLP.Verify}(pp, v, \pi, b) \rightarrow \{0, 1\}$, *a probabilistic algorithm verifying that π indeed is a valid short-lived proof of the instance v. If verification succeeds, the algorithm outputs 1, and otherwise 0.*

Note that the definition assumes there exists a *randomness beacon* which outputs an unpredictable value b periodically at certain times. There are various ways to implement such beacons including using a public blockchain [10], financial market [19], and more. As we will discuss later on, such assumption is necessary to eliminate the need for having a shared global clocks (*i.e.*, timetamping). So, as parties agree on initial point in time (implied by b), the proof π tied to b must have been observed before time T to be convincing, otherwise might be a forgery.

A short-lived proof must satisfy five properties including *completeness*, *forgeability*, *soundness*, *zero knowledge*, and *indistinguishability*. Completeness means if parties follow the protocol, the verifier should get convinced of the proof showing $(v; w) \in R$ in time less than T. The proof should be forgeable, enabling anyone running in time $(1 + \epsilon)T$ to generate a valid proof. Soundness states that it is computationally infeasible for a malicious prover P^* running with parallel processors to generate a convincing proof in time less than T, except with a negligible probability. Preserving the privacy of the witness w is implied by the zero knowledge property, formalizing by a simulator which outputs a transcript computationally indistinguishable from the one generated in the real execution of the protocol. And finally, a short-lived proof must be indistinguishable from a forgery, making it time-sensitive for the verifier to accept the proof only within its validity period. Indistinguishability means it should not be possible to distinguish if somebody has taken the witness w or (y, π) to generate the proof.

SLP Using Sigma protocols. Short-lived proofs can be instantiated both using generic (non-interactive) zero-knowledge proofs and efficient Sigma protocols. However, as shown in [3], making a Sigma protocol short-lived is rather tricky as it needs some modification in protocol for OR-composition to be secure according to SLP properties. In fact, the modification is done in such a way to let honest prover creates an SLP in a short time without needing to wait for time T to compute the VDF but forces the malicious prover to do the sequential computation, preventing her from computing a forgery before time T. More accurately, in an Or-composition the prover can convince the verifier even if she only knows the witness to one of the relations. To do so, the verifier lets the prover somehow cheat by using the simulator for the relation that she does not know the witness for. Thus, having one degree of freedom the prover chooses two sub-challenges c_1 and c_2 under the constraint that $c_1 + c_2 = c$. Note that the prover is free to fix one of them and compute the other one under the constraints. The observation made in [3] to let the honest prover quickly generate the short-lived proof is to involve the beacon b in the generation of the challenge. Therefore, an honest prover just needs to pre-compute the VDF on a random value b^* allowing her to use it when computing the forgery by freely setting one of the sub-challenges, say c_2, to $b^* \oplus b$ and letting $c_1 = c \oplus c_2$. A malicious prover, however, should compute the VDF on demand as she does not know a witness w for the relation R and c_1 gets fixed by the simulator, taking away the possibility of setting c_2 as specified. For completeness, the construction of a SLP for a Sigma protocol is presented in Figure 1.

As optimization, some alternative ways for generating a VDF solution by the honest prover instead of pre-computing a VDF from scrath have been proposed by Arun et al. that we refer the reader to [3] for more details.

4 TIMED SECRET SHARING

In a secret sharing scheme, after the dealer completes the distribution phase and all the parties receive their corresponding shares, a qualified set of parties is able to perform the reconstruction. Needless to say, this can occur at anytime as there is no notion of time explicitly involved in the definition. Making the system dependent on time, the reconstruction occurs within a determined time period,

$[T_1, T_2]$, where T_1 is the lower time bound and T_2 is the upper time bound. These time bounds might be enforced by the dealer itself or as part of the system requirements, or even a combination of these two. An important consideration, however, is that the dealer's *availability* should not change by making the scheme time-based, meaning that the dealer's role should finish after the distribution phase likewise in the original setting.

4.1 Timed Secret Sharing Definition

Here we give a formal definition for timed secret sharing (TSS). Our definition directly follows the original definition of a threshold secret scheme scheme augmented with the notion of time.

Definition 6 (Timed Secret Sharing). *A timed secret sharing (TSS) scheme is defined by the following algorithms in three main phases.*

(1) **Initialization:** The dealer D generates public parameters pp as follows.
   - Setup: $\mathsf{TSS.Setup}(1^\lambda, T_1) \to pp$, *on input security parameter λ and time parameter T_1, generates public parameters pp.*

(2) **Distribution:** The dealer D takes a secret $s \in S_\lambda$ and carries out the following algorithm.
   - Sharing: $\mathsf{TSS.Sharing}(pp, s) \to \{C_1, \ldots, C_n\}$, *on input a secret $s \in S_\lambda$, generates a locked share C_i with time parameter T_1 for each party P_i in the set $\mathcal{P}$.*

(3) **Reconstruction:** Each party P_i in the set $\mathcal{P}$ may perform the followings:
   - Recovering: $\mathsf{TSS.Recover}(pp, C_i) \to s_i$, *on input public parameters pp and locked share C_i, recovers the share s_i no sooner than T_1.*
   - Pooling: $\mathsf{TSS.Pool}(pp, \mathcal{S}, T_2) \to s$, *on input a set $\mathcal{S}$ of shares of size $> t$ before time T_2, outputs the secret s. Otherwise, abort $\perp$.*

A TSS scheme must satisfy correctness, privacy and security.

Correctness. A TSS satisfies correctness if for all secret $s \in S_\lambda$ it holds

$$\Pr\left[\mathsf{TSS.Pool}(pp, \mathcal{S}, T_2) \to s \;:\; \begin{array}{l}\mathsf{TSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{TSS.Sharing}(pp, s) \to \{C_i\}_{i\in[n]},\\ \mathsf{TSS.Recover}(pp, Z_i) \to s_i\end{array}\right] = 1$$

Privacy. A TSS satisfies privacy if for all parallel algorithms $\mathcal{A}$ whose running time is at most $t < T_1$ there exists a simulator Sim and a negligible function μ such that for all secret $s \in S_\lambda$ and $\forall\lambda \in \mathbb{N}$ it holds

$$\Pr\left[\mathcal{A}(pp, C_i) = 1 \;:\; \begin{array}{l}\mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{PVTSS.Sharing}(pp, s) \to \{C_i\}_{i\in[n]}\end{array}\right] -$$

$$\Pr\left[\mathcal{A}(pp, C_j) = 1 \;:\; \begin{array}{l}\mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{Sim}(pp) \to \{C_i\}_{j\in[n]}\end{array}\right] \le \mu(\lambda)$$

Security. A TSS satisfies security if an adversary controlling a subset of at most $\le t$ shares (*i.e.*, parties) $\mathcal{S}'$ learns no information about s in an information-theoretic sense. So, it must hold

$$\Pr\left[\mathsf{TSS.Pool}(pp, \mathcal{S}', T_2) \to s \;:\; \begin{array}{l}\mathsf{TSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{TSS.Sharing}(pp, s) \to \{Z_1\}_{i\in[n]},\\ \mathsf{TSS.Recover}(pp, Z_i) \to s_i\end{array}\right] \le \mu(\lambda)$$

4.2 Timed Secret Sharing Construction

In this section, we present an instantiation of TSS. For the sake of exposition, we base our construction on Shamir secret sharing as the most prevalent threshold secret sharing scheme. To establish a lower time bound T_1, we make use of time-lock puzzles as they allow the dealer to lock the shares into puzzles and enforce a computational delay for each party to recover their corresponding share. The characteristics of a time-lock puzzle guarantee that no party P_i can obtain her share s_i except by running through sequential computation parameterized by T_1. To establish an upper time bound T_2, we assume all parties, including the reconstructor, have access to a shared global clock, providing them approximately with the current time. So, the reconstructor does not accept the shares sent after T_2. We will relax the timing assumption later on.

(1) **Initialization:**
   - Setup: $\mathsf{TSS.Setup}(1^\lambda, T_1) \to pp$, the protocol works over $\mathbb{Z}_p$, where $p > n$. The dealer D runs $\mathsf{TLP.Setup}(1^\lambda, T_1)$ and publishes the system parameters pp.

(2) **Distribution:**
   - Sharing: $\mathsf{TSS.Sharing}(pp, s) \to \{Z_1, \ldots, Z_n\}$, the dealer D picks a secret $s \in Z_p$ to be shared among n parties. She samples a degree-t Shamir polynomial $f(\cdot)$ such that $f(0) = s$ and $f(i) = s_i$ for $i \in [n]$. She runs $\mathsf{TLP.Gen}(1^\lambda, T_1, s_i)$ to create puzzle Z_i with time parameter T_1, locking the share s_i for all $i \in [n]$. Finally, the dealer D privately sends each party P_i its corresponding puzzle Z_i.

(3) **Reconstruction:**
   - Recovering: $\mathsf{TSS.Recover}(pp, Z_i) \to s_i$, upon receiving the puzzle Z_i, party P_i starts solving it by running $\mathsf{TLP.Solve}(T_1, Z_i)$ to recover the share s_i. No party obtains their share sooner that time T_1 from receiving the puzzle.
   - Pooling: $\mathsf{TSS.Pool}(pp, \mathcal{S}, T_2) \to s$, upon having sufficient number of shares (*i.e.*, $\ge t + 1$) received before time T_2, the reconstrctor (a party in $\mathcal{P}$) reconstructs the secret s using Lagrange interpolation at $f(0)$ or aborts otherwise.

Theorem 1. *Let TLP be a secure time-lock puzzle. Then the described construction realizes the properties of TSS from Section 4.1.*

Proof. The correctness is straightforward. The privacy property follows directly from that of the underlying TLP which implies the indistinguishably of a puzzle produced by PVTSS.Sharing and the one produced by Sim. Since all the puzzles are communicated through private channels, no party can learn other's share after T_1. Finally, the security stems from the underlying threshold secret sharing, where a subset of shares $\mathcal{S}'$ of size $\le t$ reveals no information about the secret s. □

5 VERIFIABLE TIMED SECRET SHARING

So far we assumed all parties including dealer follow the protocol faithfully, providing passive security. Now, we want to take a step further and present a scheme with active security, where parties may misbehave and arbitrary deviate from the execution of the protocol. We need to protect against a dealer sending incorrect or even no shares during the distribution phase. Also, we need to

protect against a shareholder sending incorrect share during the reconstruction phase.

5.1 Verifiable Timed Secret Sharing Definition

Here we give a formal definition for verifiable timed secret sharing (VTSS). Our definition directly follows that of Feldman VSS [24] which is the basis for all the existing VSS schemes.

Definition 7 (Verifiable Timed Secret Sharing). *A verifiable timed secret sharing (TSS) scheme is defined by the following algorithms in three main phases.*

(1) **Initialization:**
   - Setup: $\mathsf{VTSS.Setup}(1^\lambda, T_1) \to pp$, *on input security parameter* λ *and time parameter* T_1, *generates public parameters* pp.

(2) **Distribution:**
   - Sharing: $\mathsf{VTSS.Sharing}(pp, s) \to \{C_i, \pi_i\}$, *on input public parameter* pp *and a secret* $s \in S_\lambda$, *generates locked share* C_i *with time parameter* T_1 *for each party* P_i *for* $i \in [n]$. *Moreover, it outputs a proof of validity* π_i *for each party's locked share* C_i.
   - Share verification: $\mathsf{VTSS.Verify}_1(pp, C_i, \pi_i) \to \{0,1\}$, *on input public parameters* pp, *locked share* C_i, *and proof* π_i, *each party* P_i *checks the validity of the her share. This includes verifying the published locked share* C_i *is well-formed (i.e., extractability) and contains a valid share of some secret* s *(i.e., verifiability). This algorithm is run by each party* P_i.
   - Complaint round: If a set of parties of size $\geq t+1$ complain about sharing, the dealer D is disqualified. Otherwise, the dealer reveals the corresponding locked shares with proofs by broadcasting $\{C_i, \pi_i\}$. If a proof does not verify (or dealer does not broadcast), the dealer is disqualified.

(3) **Reconstruction:**
   - Recovering: $\mathsf{VTSS.Recover}(pp, C_i) \to \{s_i\}$, *on input public parameters* pp *and locked share* C_i, *outputs a share* s_i. *This algorithm is run by each party* P_i.
   - Recovery verification: $\mathsf{VTSS.Verify}_2(pp, s_i, \pi_i) \to \{0,1\}$, *on input public parameters* pp, *a share* s_i, *and proof* π_i, *checks the validity of submitted share. This algorithm is run by the verifier* V.
   - Pooling: $\mathsf{VTSS.Pool}(pp, \mathcal{S} = \{s_i\}_{i\in\varphi}, T_2) \to s$, *on input public parameters* pp, *the set of indices* φ *of* $t+1$ *shares* s_i, *and time parameter* T_2, *outputs the secret* s. *This algorithm is run by the* V. *If the set* $\mathcal{S}$ *contains less than* $t+1$ *valid shares received before* T_2, *the verifier aborts.*

A VTSS scheme must satisfy correctness, soundness, privacy, and security.

Correctness. A VTSS satisfies correctness if for all secret $s \in S_\lambda$ and all $i \in [n]$ it holds

$$\Pr\left[\begin{array}{l|l} \mathsf{VTSS.Verify}_1(pp, C_i, \pi_i) = 1 & \mathsf{VTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{VTSS.Verify}_2(pp, s_i, \pi_i) = 1 & \mathsf{VTSS.Sharing}(pp, s) \to \{C_i, \pi_i\}, \\ \mathsf{VTSS.Pool}(pp, \mathcal{S}, T_2) \to s & \mathsf{VTSS.Recover}(pp, C_i) \to \{s_i\} \end{array}\right] = 1$$

Soundness. The extractability and verifiability of the shares together implies soundness. A VTSS scheme is sound if there exists a negligible function μ such that for all PPT adversaries $\mathcal{A}$ and all $\lambda \in \mathbb{N}$, it holds

$$\Pr\left[\begin{array}{l|l} b_1 = 1 \vee b_2 = 1 & \mathsf{VTSS.Setup}(1^\lambda, T_1) \to pp, \\ & \mathcal{A}(pp) \to (\{C_i, \pi_i\}_{i\in[n]}, s), \\ & b_1 := \mathsf{VTSS.Verify}_1(pp, C_i, \pi_i) \wedge \nexists s \text{ s.t.} \\ & \mathsf{VTSS.Sharing}(pp, s) \to \{C_i, \cdot\}, \\ & b_2 := \mathsf{VTSS.Verify}_2(pp, s_i, \pi_i) \wedge \nexists s_i \text{ s.t.} \\ & \mathsf{VTSS.Recover}(pp, C_i) \to \{s_i\} \end{array}\right] \leq \mu(\lambda)$$

Privacy. A PVTSS satisfies privacy if for all parallel algorithms $\mathcal{A}$ whose running time is at most $t < T_1$ there exists a simulator Sim and a negligible function μ such that for all secret $s \in S_\lambda$ and $\forall \lambda \in \mathbb{N}$ it holds

$$\Pr\left[\mathcal{A}(pp, \{C_i, \pi_i\}_{i\in[n]}) = 1 \;:\; \begin{array}{l} \mathsf{VTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{VTSS.Sharing}(pp, s) \;\to \{C_i, \pi_i\}_{i\in[n]} \end{array}\right] -$$

$$\Pr\left[\mathcal{A}(pp, \{C_i, \pi_i\}_{i\in[n]}) = 1 \;:\; \begin{array}{l} \mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{Sim}(pp) \to \{C_i, \pi_i\}_{i\in[n]} \end{array}\right] \leq \mu(\lambda)$$

Security. A VTSS satisfies security if there exists a negligible function μ such that for an adversary controlling a subset of at most $\leq t$ parties/shares $\mathcal{S}'$ it holds

$$\Pr\left[\mathsf{VTSS.Pool}(pp, \mathcal{S}', T_2) \to s \;:\; \begin{array}{l} \mathsf{VTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{VTSS.Sharing}(pp, s) \to \{C_i, \pi_i\}, \\ \mathsf{VTSS.Recover}(pp, C_i) \to \{s_i\} \end{array}\right] \leq \mu(\lambda)$$

6 PUBLICLY VERIFIABLE TIMED SECRET SHARING

So far we assumed all parties involved including the dealer follow the protocol faithfully, providing passive security. Now, we want to take a step further and present a scheme with active security, where parties may misbehave and arbitrary deviate from the execution of the protocol. We seek *public verifiaibility*, enabling anyone, even an external party, to verify the correctness of the different phases of the protocol. To this end, the first component to be replaced is a *publicly verifiable secret sharing* (PVSS) scheme where enforces parties to behave correctly by non-interactively proving the validity of the sent messages at phases of distribution and reconstruction. Moreover, we need to protect against a malicious dealer who may decide to publish *invalid locked shares*. It is problematic that such flawed action is detected by the parties only after spending a substantiate time and computational power on the process of obtaining/unlocking the shares. Thus, parties should be able to first check the validity of the locked shares and then start unlocking them. Apart from a huge waste of resources, such circumstance can make the system vulnerable to some type of denial-of-service (DoS) attacks, preventing a network of parties to carry out their intended operation.

A formal definition for PVTSS scheme is as below:

Definition 8 (Publicly Verifiable Timed Secret Sharing). *A publicly verifiable timed secret sharing (PVTSS) scheme is defined by the following algorithms in three main phases.*

(1) **Initialization:**
   - Setup: $\mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp$, *on input security parameter* λ *and time parameter* T_1, *generates system parameters* pp. *This involves parameters for the underlying PVSS and also for share locking mechanism chosen by the dealer* D. *Also, each*

party P_i announces a registered public key pk_i which its secret key sk_i is only known to them.

(2) **Distribution:**
- Sharing: $\mathsf{PVTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]}) \to \{C_1, \ldots, C_N, \pi_D\}$, *on input system parameter pp and a randomly selected secret $s \in S_\lambda$, generates locked encrypted share C_i with time parameter T_1 for each party P_i with respect to its corresponding public key pk_i for $i \in [n]$. Moreover, it outputs a proof π_D for the correctness of sharing. This algorithm is run by the dealer D*
- Share verification: $\mathsf{PVTSS.Verify}_1(pp, \{C_i\}_{i\in[n]}, \pi_D, \{pk_i\}_{i\in[n]}) \to \{0,1\}$, *on input system parameters pp, locked encrypted shares $\{C_i\}_{i\in[n]}$, and proof π_D, checks the validity of the sharing. This includes verifying all the published locked encrypted shares are well-formed (i.e., extractability) and are correct sharing of some secret s (i.e., verifiability). Note that this algorithm is run by any (external) verifier V.*

(3) **Reconstruction:**
- Recovering: $\mathsf{PVTSS.Recover}(pp, C_i, pk_i, sk_i) \to \{\tilde{s}_i, \pi_i\}$, *on input system parameters pp, locked encrypted share C_i, and the key pair (pk_i, sk_i), outputs a decrypted share $\tilde{s}_i$ together with proof π_i of valid decryption. This algorithm is run by each party P_i.*
- Recovery verification: $\mathsf{PVTSS.Verify}_2(pp, C_i, \tilde{s}_i, \pi_i) \to \{0,1\}$, *on input system parameters pp, locked encrypted share C_i, decrypted share $\tilde{s}_i$ and proof π_i, checks the validity of the decryption. This algorithm is run by any (external) verifier V.*
- Pooling: $\mathsf{PVTSS.Pool}(pp, \mathcal{S} = \{\tilde{s}_i\}_{i\in\varphi}, T_2) \to s$, *on input system parameters pp, the set of indices φ of $t+1$ decrypted shares $\tilde{s}_i$, and time parameter T_2, outputs the secret s. This algorithm is run by any verifier V. If the set $\mathcal{S}$ contains less than $t+1$ valid shares received before T_2, the verifier aborts.*

The given definition is based on the general model for PVSS presented in [14, 42]. Intuitively, the security requirements for a PVTSS are *soundness*, convincing any verifier that the correct shares are obtainable after time T_1, *privacy*, ensuring that no parallel algorithms running in time at most T_1 can obtain any of the (encrypted) share except with a negligible probability, and *security*, guaranteeing the secret remains hidden to an adversary corrupting at most t parties. A PVTSS scheme must satisfy the following properties.

Correctness. A PVTSS satisfies correctness if for all secret $s \in S_\lambda$ and all $i \in [n]$ it holds that

$$\Pr\left[\begin{array}{l} \mathsf{PVTSS.Verify}_1(pp, \{C_i\}_{i\in[n]}, \\ \pi_D, \{pk_i\}_{i\in[n]}) = 1 \\ \mathsf{PVTSS.Verify}_2(pp, C_i, \tilde{s}_i, \pi_i) = 1 \\ \mathsf{PVTSS.Pool}(pp, \mathcal{S}, T_2) \to s \end{array} : \begin{array}{l} \mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{PVTSS.Sharing}(pp, s, \\ \{pk_i\}_{i\in[n]}) \to \{\{C_i\}_{i\in[n]}, \pi_D\}, \\ \mathsf{PVTSS.Recover}(pp, C_i, \\ pk_i, sk_i) \to \{\tilde{s}_i, \pi_i\} \end{array}\right] = 1$$

Soundness. The extractability and verifiability of the shares together implies soundness. A PVTSS scheme is sound if there exists a negligible function μ such that for all PPT adversaries $\mathcal{A}$ and all $\lambda \in \mathbb{N}$, it holds that

$$\Pr\left[\begin{array}{l} b_1 = 1 \vee b_2 = 1 \end{array} : \begin{array}{l} \mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathcal{A}(pp) \to (\{C_i\}_{i\in[n]}, \pi_D, \tilde{s}, \pi), \\ b_1 := \mathsf{PVTSS.Verify}_1(pp, \{C_i\}_{i\in[n]}, \pi_D, \\ \{pk_i\}_{i\in[n]}) \wedge \nexists s \text{ s.t.} \\ \mathsf{PVTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]}) \\ \to \{\{C_i\}_{i\in[n]}, \cdot\}, \\ b_2 := \mathsf{PVTSS.Verify}_2(pp, C, \tilde{s}, \pi), \wedge \nexists sk \text{ s.t.} \\ \mathsf{PVTSS.Recover}(pp, C, pk, sk) \to \{\tilde{s}, \cdot\}, \end{array}\right] \leq \mu(\lambda)$$

Privacy. A PVTSS satisfies privacy if for all parallel algorithms $\mathcal{A}$ whose running time is at most $t < T_1$ there exists a simulator Sim and a negligible function μ such that for all secret $s \in S_\lambda$ and $\forall\lambda \in \mathbb{N}$ it holds that

$$\Pr\left[\mathcal{A}(pp, \{C_i\}_{i\in[n]}, \pi_D) = 1 \; : \; \begin{array}{l} \mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{PVTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]}) \\ \to \{\{C_i\}_{i\in[n]}, \pi_D\} \end{array}\right] -$$
$$\Pr\left[\mathcal{A}(pp, \{C_i\}_{i\in[n]}, \pi_D) = 1 \; : \; \begin{array}{l} \mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{Sim}(pp) \to (\{C_i\}_{i\in[n]}, \pi_D) \end{array}\right] \leq \mu(\lambda)$$

Security.[4] After T_1 (and before secret reconstruction), the public information together with the secret keys of any set of at most $t-1$ parties gives no information about the secret. A PVTSS satisfies security if (there exists a negligible function μ such that) for an adversary controlling a subset of at most $\leq t$ parties/shares $\mathcal{S}'$ it holds that

$$\Pr\left[\mathsf{TSS.Pool}(pp, \mathcal{S}', T_2) \to s \; : \; \begin{array}{l} \mathsf{PVTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{PVTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]}) \\ \to \{\{C_i\}_{i\in[n]}, \pi_D\}, \\ \mathsf{PVTSS.Recover}(pp, C_i, pk_i, sk_i) \\ \to \{\tilde{s}_i, \pi_i\} \end{array}\right] \leq \mu(\lambda)$$

An indistinguishability definition given in [27, 41] and adapted by [14] formalizes this.

Definition 9 (Indistinguishably of Secrets (Security))**.** *PVTSS is said to be secure if the any polynomial time adversary $\mathcal{A}$ corrupting at most $t-1$ parties has a negligible probability in the following game played against a challenger.*

(1) *Playing the role of a dealer, the challenger runs the Setup step of the PVTSS and sends all the public information to $\mathcal{A}$. Moreover, it creates the key-pairs for the honest parties and send the corresponding public keys to $\mathcal{A}$.*
(2) *$\mathcal{A}$ creates and sends the public keys of the corrupted parties to the challenger.*
(3) *The challenger randomly picks the values s and s' in the space of the secret. It then chooses $b \leftarrow \{0,1\}$ uniformly at random and runs the Sharing step of the protocol with s as secret. It sends $\mathcal{A}$ all public information generated in that phase together with s_b.*
(4) *$\mathcal{A}$ outputs a guess b'.*
The advantage of $\mathcal{A}$ is defined as $|\Pr[b = b] - 1/2|$.

It is worth mentioning that as in [14] the security requirement here does not capture any privacy for the secret after the reconstruction phase. In other words, the privacy of the secret matters as long as it is reconstructed by an eligible set of parties after which anybody (any external party) can learn the secret. Moreover, one

[4] This property is presented as IND1-Secrecy in [27, 41].

may notice that *before* time T_1 the privacy requirement implies the security. That is, even if the adversary corrupts *all* the parties involved in the protocol, she cannot learn any information about the secret.

6.1 PVTSS Construction

In what follows we propose a construction for PVTSS. Before presenting the protocol description in details, we elaborate on some components used in the transition from TSS to PVTSS.

SCRAPE PVSS.. The interesting work of [14] introduced the first PVSS scheme that achieves linear computational complexity for all steps including distribution of the secret, verification of the shares, and reconstruction of the secret. The proposed PVSS is an improvement over that of [42] enjoying the luxury of a new method for doing the verification regarding the equivalence of Shamir secret sharing and Reed Solomon error-correcting code [37]. In a nutshell, the protocol works as follows. The dealer D chooses a random secret $s \xleftarrow{\$} \mathbb{Z}_q$ and sets the group element of form $S = h^s$, splits s into shares $\{s_i\}_{i\in[n]}$ and computes the encrypted shares $\{\tilde{s}_i\}_{i\in[n]}$ using corresponding parties' public keys $\{pk_i\}_{i\in[n]}$. The dealer also publishes a set of commitments to shares $\{v_i\}_{i\in[n]}$, enabling anybody to check the validity of the ciphertexts (*i.e.*, encrypted shares correspond to the committed shares) and consistency of the shares (*i.e.*, shares are evaluations of the same polynomial). Upon receiving a threshold number of valid shares, anybody can use Lagrange interpolation in the exponent to reconstruct the secret S. SCRAPE PVSS is proposed in two versions, one in random oracle model under Decisional Diffie-Hellman (DDH) assumption and the other one in plain model under Decisional Bilinear Squaring (DBS) assumption [27]. We use the non-pairing variant of the protocol which offers *knowledge soundness*. This is vital for security reasons, ensuring that the secrets chosen by the adversary is independent form those of honest' parties.

Dealing with a malicious Dealer. Probably the most challenging part of the protocol design is to protect against a malicious dealer who may send malformed locked shares to parties. What complicates the matter is that even ensuring the puzzle is well-formed (*i.e.*, it is solvable/extractable) is not enough as it is possible that some (encrypted) locked share is not valid. In other words, extractability and verifiability with respect to some public information should be achieved *concurrently*. So, the dealer needs to simultaneously ensure parties (any verifier) that each locked encrypted share satisfies the correctness condition of the underlying PVSS (*i.e.*, its DLEQ is equal to that of the corresponding commitment). So, this takes away naively deploying the timed commitment primitive of [9] as it only guarantees the embedded message will be correctly unlocked after doing some sequential computation. Very Recently, Manevich and Akavia [36] introduced a primitive called attribute verifiable timed commitments (AVTC) that augments the basic version with verifiability for any arbitrary attribution. Although we can technically make a black box use of AVTC for our purpose, but it comes with two caveats. First, as the proposed construction aims for proving any arbitrary attribute for the committed value, it incurs a huge overhead both for the prover and the verifier by deploying the generic (interactive) zero-knowledge proofs, which is undesirable in our setting. Second, we would like to have leader goes offline after the dealing phase while timed commitment definition genuinely has a revealing phase where the sender should reveal what has been committed later on.

Our solution is based on having dealer blinds each encrypted shares $\tilde{s}_i$ using some randomness β_i, locks β_i using LHTLP, and publishes all the puzzles for $i \in [n]$ on a public bulletin board. Here, we make use of the result from [33] to enforce dealer to prove the puzzles containing β_i are well-formed. What remains to show is the blinded/locked encrypted shares C are valid with respect to the locked randomness β_i. We do it by modifying DLEQ proofs (Section 3.6) and constructing a Sigma protocol for a relation (language) describing AND-composition of *modified/blinded* DLEQ and valid LHTLP. Note that we may interchangeably use the term blinded/locked encrypted shares in the rest of the paper.

Here we do not need to make use of the homomorphic property of LHTLP as each party just needs to solve *one* puzzle to get her own share while the dealer (efficiently) generates N puzzles at the beginning. In fact, as pointed out earlier, solving other parties' puzzles has no advantage for some party who is capable of solving many puzzle instances within the same time period. This is because the (locked) shares are encrypted under each party's public key and can only be learnt using the corresponding secret key. Note that in [46] the existence of a user with such capability is problematic as it enables them to learn some sensitive information (*i.e.*, signature) prior to others and possibly use them in their favour. So, compacting all the puzzles into one puzzle mitigates the issue.

Proof of Exponentiation. As part of the proving that a sequential computation of the form $y = x^u$, where $u = 2^T$, was done correctly, Wesolowski [49] introduced a simple protocol for proof of exponentiation in a hidden-order group. The proof structure resembles that of Sigma protocol (without containing a *knowledge*) and enables a verifier to do the verification efficiently without needing to perform the whole exponentiation. We restate the protocol here as it is the basis of the validity proof for puzzle generation.

(1) V randomly picks l from the set of all 2λ-bit prime numbers denoted by $Primes(2\lambda)$ and sends it to P.
(2) P computes $u = ql + r$, where $q \in Z$ and $0 \leq r < l$ are two unique integers.
(3) P computes $\pi = x^q \bmod N$ and sends it to the V.
(4) V computes $r = u \bmod l$, and accepts if $y = x^r \pi^l \bmod N$.

The above protocol can be described as an honest-verifier non-interactive argument for the language

$$L_{\text{exp}} = \{(x, u, y) : y = x^u\}$$

Boneh et al. [8] showed this protocol can be extended to a proof of knowledge of exponent. As with other Sigma protocols, the above protocol can be made non-interactive via Fiat-Shamir heuristic.

LHTLP Validity Proof. Now we briefly describe the honest-verifier zero-knowledge argument of knowledge protocol presented in [33] to let a puzzle generator show the puzzle is well-formed. It builds upon the LHTLP of form $Z = (U, V)$ proving the knowledge of

randomness and solution used [5]. The protocol Π_{HTLPV} is described for the language

$$L_{\mathsf{LHTLPV}} = \{stm = (U, V, T, g, h) \mid \exists \mathsf{wit} = (s, r) : U = \pm g^r \wedge V = h^{r \cdot N}(1+N)^s\}$$

(1) P randomly chooses $x \xleftarrow{\$} [0, \lceil N/2 \rceil \cdot 2^{2\lambda}]$ and $t \xleftarrow{\$} \mathbb{Z}_N$. Then, computes and sends $a_1 \leftarrow g^x \bmod N$ and $a_2 \leftarrow h^{x \cdot N}(1+N)^t \bmod N^2$ to V.
(2) V randomly picks $c \xleftarrow{\$} [0, 2^\lambda]$ and sends it to P.
(3) P computes $\mu \leftarrow x + cr$, $\eta \leftarrow t + cs \bmod N$ and sends them to V.
(4) V accepts if
   - $\mu \in [0, \lceil N/2 \rceil \cdot 2^\lambda + \lceil N/2 \rceil \cdot 2^{2\lambda}]$
   - $g^\mu = a_1 U^c \bmod N$
   - $h^{\mu \cdot N}(1+N)^\eta = a_2 V^c \bmod N^2$

Theorem 2. *The protocol Π_{LHTLPV} is a public-coin honest-verifier zero-knowledge argument of knowledge corresponding to the language L_{LHTLPV}.*

Due to the space limitation, we avoid presenting the proof and refer the reader to ([33], Theorem 4). However, we provide some intuition on how the proofs work. The HVZK property can be shown using several indistinguishable hybrid arguments starting from the transcript of the real protocol execution and leading to the simulated one generated by the Simulator. Regarding soundness, the protocol obtains the witness-extended emulation property by tiding it up with breaking the strong RSA assumption. *to prove the security of the protocol under the strong RSA assumption via reduction*

Blinded DLEQ proof. Apart from showing that the puzzles are well-formed and can be opened after time T_1, the dealer also needs to convince the parties that the encrypted shares $\tilde{s}_i$ are correctly generated, meaning that they contain the same shares as the commitments v_i. However, here the privacy requirement of the PVTSS (Section 5.1) implies the encrypted shares $\tilde{s}_i$ to be available only after time T_1. We can resolve this by having dealer D initially publishes *blinded* encrypted shares $\tilde{s}_i f^{\beta_i}$ where f is a generator of the group and β_i is a randomly chosen value to blind each party's encrypted share for $i \in [n]$. Then, the dealer D locks the randomness β_i using an LHTLP and publishes the resulting puzzles $\{Z_i\}_{i\in[n]}$ together with the set of blinded encrypted shares $\{\tilde{s}_i f^{\beta_i}\}_{i\in[n]}$ and commitments $\{v_i\}_{i\in[n]}$ on the public bulletin board. By making a simple modification to the DLEQ protocol (Section 3.6) we can construct a Sigma protocol to show that the blinded values (encrypted shares) contain the same share as the commitments. The following is a protocol Π_{BDLEQ} (simultaneous knowledge of two secrets) for the language

$$L_{\mathsf{BDLEQ}} = \{stm = (g_1, x, g_2, g_3, y) \mid \exists \mathsf{wit} = (\alpha, \beta) : x = g_1^\alpha \wedge y = g_2^\alpha g_3^\beta\}$$

(1) P chooses two random elements $u_1, u_2 \xleftarrow{\$} \mathbb{Z}_q$, computes $a_1 = g_1^{u_1}$ and $a_2 = g_2^{u_1} g_3^{u_2}$, and sends them to V.
(2) V sends back a randomly chosen challenge $c \xleftarrow{\$} \mathbb{Z}_q$.
(3) P computes $r_1 = u_1 + c\alpha$ and $r_2 = u_2 + c\beta$ and sends them to V.
(4) V checks if both $g_1^{r_1} = a_1 x^c$ and $g_2^{r_1} g_3^{r_2} = a_2 y^c$ hold.

Theorem 3. *The protocol Π_{BDLEQ} is a public-coin honest-verifier zero-knowledge argument of knowledge corresponding to the language L_{BDLEQ}.*

Proof. We show that the Π_{BDLEQ} satisfies the properties of a Sigma protocol. Completeness clearly holds, as

$$g_1^{r_1} = g_1^{u_1 + c\alpha} = g_1^{u_1} g_1^{c\alpha} = a_1 x^c$$
$$g_2^{r_1} g_3^{r_2} = g_2^{u_1 + c\alpha} g_3^{u_2 + c\beta} = g_2^{u_1} g_3^{u_2} (g_2^{u_1} g_3^{u_2})^c = a_2 y^c$$

For special soundness, given two accepting transcripts $(a_1, a_2; c; r_1, r_2)$ and $(a_1, a_2; c'; r_1', r_2')$ the witness (α, β) can be found as follows

$$g_1^{r_1} = a_1 x^c, g_2^{r_1} g_3^{r_2} = a_2 y^c \quad ; \quad g_1^{r_1'} = a_1 x^{c'}, g_2^{r_1'} g_3^{r_2'} = a_2 y^{c'}$$

$$g_1^{r_1 - r_1'} = x^{c - c'} \Leftrightarrow x = g_1^{\frac{r_1 - r_1'}{c - c'}}$$

$$g_2^{r_1 - r_1'} g_3^{r_2 - r_2'} = y^{c - c'} \Leftrightarrow y = g_2^{\alpha} g_3^{\frac{r_2 - r_2'}{c - c'}}$$

Hence, the witness β can be found as $\beta = (r_2 - r_2')/(c - c')$ given the witness $\alpha = (r_1 - r_1')/(c - c')$.

Let c be a given challenge. Zero-knowledge property is implied by the fact that the the following two distributions, namely real protocol distribution and simulated distribution, are identically distributed.

$$\mathsf{Real} : \{(a_1, a_2; c; r_1, r_2) : u_1, u_2 \xleftarrow{\$} \mathbb{Z}_q, a_1 = g_1^{u_1}, a_2 = g_2^{u_1} g_3^{u_2}; r_1 = u_1 + c\alpha, r_2 = u_2 + c$$

$$\mathsf{Sim} : \{(a_1, a_2; c; r_1, r_2) : r_1, r_2 \xleftarrow{\$} \mathbb{Z}_q; a_1 = g_1^{r_1} x^{-c}, a_2 = g_2^{r_1} g_3^{r_2} y^{-c}\}$$

Note that the probability of occurring for each distribution is the same and equals $1/q^2$. □

Putting it all Together. Given above, we can form a Sigma protocol Π_{valid} for the AND-composition of the two aforementioned relations of R_{BDLEQ} and R_{HTLPV} which share a witness β. That is, the blinding factor for BDLEQ and solution for LHTLPV. Note that we assume both protocols use the common challenge, with same challenge space providing λ-bit security. So

$$R_{\mathsf{valid}} = \{(x, y, U, V; \alpha, \beta, r) \mid v = g_1^\alpha \wedge c = g_2^\alpha g_3^\beta \wedge U = \pm g^r \wedge V = h^{r \cdot N}(1+N)^\beta\}$$

Theorem 4. *The protocol Π_{valid} is a public-coin honest-verifier zero-knowledge argument of knowledge corresponding to the language L_{valid}.*

A proof of Theorem 4 is given in Appendix D.

Protocol Description. Now we describe the complete protocol Π_{PVTSS} where the dealer D, and all parties $\mathcal{P} = \{P_1, \ldots P_n\}$ are enforced to behave honestly by presenting proper proofs during the protocol execution. Any (external) verifier V can verify the protocol execution and finally obtain the secret.

(1) **Initialization:**
   - Setup: $\mathsf{PVTSS.Setup}(1^\lambda, T_1) \rightarrow pp$, the dealer D generates and publishes the system parameters pp. This involves parameters for the underlying PVSS including independently chosen generators $g_1, g_2, g_3 \in \mathbb{G}_q$, and parameters for underlying LHTLP acquired from $\mathsf{HTLP.Setup}(1^\lambda, T_1)$. Each party

[5] The protocol uses witness-extended emulation technique to prove security where the simulator running in polynomial time needs to both simulate the view of the prover and extract a witness. This can be thought of as a generalization of the special soundness property. For more details refer to [32, 33]

P_i announces a registered public key $pk_i = g_1^{sk_i}$ which its secret key sk_i is only known to them.

(2) **Distribution:**

– Sharing: $\mathsf{PVTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]}) \to \{C_1, \ldots, C_N, \pi_D\}$, the dealer D randomly chooses $s \xleftarrow{\$} \mathbb{Z}_q$ and defines the secret $S = g_1^s$ to be shared among n parties with public keys $\{pk_i\}_{i\in[n]}$. The dealer computes Shamir shares $f(i) = s_i$, commitments $v_i = g_2^{s_i}$, and encrypted shares $\hat{s}_i = pk_i^{s_i}$ for all $i \in [n]$ using a degree-t Shamir polynomial $f : \mathbb{Z}_q \to \mathbb{Z}_q$, where $f(0) = s$. The dealer blinds the encrypted shares $\{\hat{s}_i\}_{i\in[n]}$ using independent blinding factors β_i, resulting in $\{c_i\}_{i\in[n]}$, where $c_i = \hat{s}_i g_3^{\beta_i}$. She then locks every blinding factor β_i in a linear homomorphic time-lock puzzle Z_i with time parameter T_1 by running $\mathsf{HTLP.Gen}(pp, \beta_i)$. Let denote $C_i = \{c_i, Z_i\}$. She shows the extractability of puzzles and verifiability of shares by running Π_{valid} resulting in a proof $\pi_D =: (v_i, e, r_{1i}, r_{2i}, \mu_i, \eta_i)$ for $i \in [n]$. Finally, the dealer publishes the locked encrypted shares $\{C_i\}_{i\in[n]}$ and proof π_D.

– Share verification: $\mathsf{PVTSS.Verify}_1(pp, \{C_i\}_{i\in[n]}, \pi_D, \{pk_i\}_{i\in[n]}) \to \{0, 1\}$, the verifier V checks the validity of the sharing (*i.e.*, extractability of the puzzles and verifiability of the shares). She first checks the proof π_D is valid. Then, She validates consistency of the shares by sampling a code word $\mathbf{y}^\perp \in C^\perp$, where $\mathbf{y}^\perp = \{y_1^\perp, \ldots, y_n^\perp\}$, and checking if $\prod_{j=1}^n v_i^{y_i^\perp} = 1$.

(3) **Reconstruction:**

– Recovering: $\mathsf{PVTSS.Recover}(pp, C_i, pk_i, sk_i) \to \{\tilde{s}_i, \pi_i\}$, after checking the validity of sharing phase, any party P_i wishing to obtain their share unlocks the blinding factor β_i by running $\mathsf{HTLP.Solve}(pp, Z_i)$, and obtains their share $\tilde{s}_i$ after decypting $\hat{s}_i$ as $\tilde{s}_i = \hat{s}_j^{1/sk_i}$. Then, the party P_i reveals the share $\tilde{s}_i$ together with a proof $\pi_i =: \{\mathsf{DLEQ.P}(g_1, pk_i, \tilde{s}_i, \hat{s}_i), \beta_i\}$ of valid decryption.

– Recovery verification: $\mathsf{PVTSS.Verify}_2(pp, C_i, \tilde{s}_i, \pi_i) \to \{0, 1\}$, any (external) verifier V can check the validity of published share $\tilde{s}_i$ by running $\mathsf{DLEQ.V}(\pi_i, g_1, pk_i, \tilde{s}_i, \hat{s}_i)$. Note that having C_i, the verifier first obtains $\hat{s}_i$ using β_i.

– Pooling: $\mathsf{PVTSS.Pool}(pp, \mathcal{S} = \{\tilde{s}_i\}_{i\in\varphi}, T_2) \to s$, anyone wishing to reconstruct the secret checks the published proofs π_i are correct as described in Section. Upon having sufficient number of shares (*i.e.*, $\geq t + 1$) received before time T_2, denoted by $\mathcal{S}$, the secret $S = g_1^s$ is reconstructed using Lagrange interpolation in the exponent.

Theorem 5. *The above construction* Π_{PVTSS} realizes the definition of PVTSS from Section.

For a proof of Theorem 5 see Appendix F.

Proof. *The required properties follow directly from definitions of the underlying primitives. The completeness of t R is due to the completeness of R. t-Forgeability follows from the correctness property of the underlying VDF, ensuring that Forge can produce convincing forgeries in (1 +)t steps by running VDF.Eval(b) and using the output (y,) to satisfy the VDF half of the disjunction. The Indistinguishability and Zero-Knowledge properties both follow immediately from the zero-knowledge property of , preventing the adversary from knowing which half of the disjunction was satisfied and meaning an efficient simulator exists as required. The t-Soundness property relies on the t-Sequentiality of the VDF. The restrictions on algorithms A0, A1 in the t-Soundness definition are identical to those in the t-Sequentiality definition, meaning such algorithms will not be able to solve the VDF with non-negligible probability. This means that any adversary able to produce proofs must know a witness w for x, which the extractor for R can then efficiently extract*

□

Homomorphic PVTSS.. The PVSS scheme presented in [42] offers the homomorphic property allowing anyone to efficiently reconstruct a combined secret (*i.e.*, aggregation of independent secrets) while running the reconstruction mechanism for once. This is due to the additive homomorphic property of the underlying encryption used as part of the sharing that enables aggregating the encrypted shares for each party without having to first decrypt them individually. It is straightforward to extend such homomorphism to our proposed PVTSS construction due to the use of homomorphic time-lock puzzle.

6.2 On The Notion of Clock

We preferably seek to avoid using timestamp (*e.g.*, public blockchain) or centralized servers (*e.g.*, NTP) for addressing the timing of the protocol. We discuss how the setting of our focus which considers an (honest) verifier to check the steps of the protocol allows to ignore the availability of a shared global clock for the sake of detecting the lower and upper time bound. It turns out that having clocks that advance roughly at the same time is indeed required to remove the possibility of any dispute between the parties and verifier. In fact, clocks with inconsistent pace may lead to a wrong understanding of the ending time T_2, making the verifier reject even some of the honest parties' shares. To put it another way, having clocks with inconsistent pace makes it possible that some party sends their share after T_2 while assuming their share should get accepted by the verifier. This is problematic in scenarios where parties receive rewards in exchange for participating in the protocol and contributing with their shares [31].

Since the dealer publishes all the sharing information including locked encrypted shares on a public bulletin board, parties are supposed to start executing the protocol from a known point in time. So, there is no need to have a common knowledge of time (*i.e.*, shared global clock) to provide such consistency in pace. This allows us to consider relative timing and rely on the weaker and more realistic assumption of local clocks (initially desynchronized) which advance roughly at the same speed and can be realized using synchronizers [4]. We can even further mitigate this assumption via the notion of *computational timing*, where parties and any (external) verifier run a sequential computation to track the lower and upper time bound consistently. That is, when parties use computational timing (*e.g.*, sequential squaring), the verifier can always safely decide which shares have been sent in the due time. This is because of the (implicit) honest assumption for the verifier and the fact that the sequential squaring is consistent across all the parties and advances roughly at the same time.

Simultaneous communication of shares. A large body of literature on threshold secret sharing schemes including Shamir secret sharing and its variants consider simultaneous communication setting in which at least a threshold of parties communicate their shares *concurrently / at the same time* for reconstruction. Particularly, this assumption is crucial security wise in the SCRAPE PVSS [14], where the shares are published and the secret is reconstructed publicly. Such synchronicity is necessary to ensure that the adversary corrupting at most t parties cannot utilize of some revealed shares to reconstruct the secret even before honest parties do. Note that this is because, in the worst case, the adversary just requires one additional share to reconstruct the secret while each honest party requires t shares to do so. Therefore, to prevent the adversary to learn the secret sooner than the honest parties - which its consequences can be catastrophic in time-sensitive application - a threshold number of shares should be communicated concurrently.

Given above we notice that the implicit assumption made in SCRAPE PVSS (and other synchronous protocols) is the parties have access to a shared global clock (perfectly synchronised clocks). Moreover, this implies an agreement between (honest) parties regarding when to send the shares (Also, there is always enough number of honest parties available at the reconstruction time) . Observe that the use of computational timing in PVTSS mitigates the assumption of the shared global clock. Actually, this relaxation on assumption comes at no cost in transition from PVSS to PVTSS.

6.3 On The Use of Short-lived Proofs

To recall, the purpose of setting an upper bound is to ensure that the valid secret is reconstructed by the time T_2. So far we assumed the existence of an *online* verifier who monitors/observes the bulletin board/protocol over time to pick those shares/contributions published in the valid period. Now, an interesting question one may ask is whether we can eliminate such assumption on verifier and design the system such that the reconstruction automatically becomes under threat of failure after a specified period of time. We observe that, this can be achieved security-wise by making the validity of reconstructed secret dependent on the notion of time. That is, the secret is only valid if being reconstructed by T_2; otherwise, the verifier cannot *safely* use the shares publicly posted. This is because the adversary could have forged a *valid* proof on some *invalid* share without knowing/using the associated witness, resulting in reconstructing an invalid secret, *i.e.*, a secret different that the one shared by the dealer in the first place. In other words, we enforce an upper time bound by setting an expiration time on the validity of the shares such that the proofs associated with them lose their *soundness* after some time, making the proofs inconvincible for the verifier. Thus, this design rational intuitively offers a stronger guarantee for the upper bound as the immediate consequence of using forgeable proofs is that the secret reconstructed after T_2 is not usable. To establish this setting, we deploy the notion of short-lived (zero-knowledge) proofs [3] in our PVTSS construction.

Remark 1. Recently, there has been a number of works focusing on the notion of forgeability over time, particularly for developing short-lived signature or forward-forgeable signature [3, 44]. To the best of our knowledge, the work of [3] is the only one exploring the time-based forgeability in proof systems. This in turn enables us to make use of their scheme in our work to provide the upper time bound for PVTSS.

Remark 2. Note that as mentioned, here we relax our assumption regarding availability of a *time-tracker* verifier who observes the protocol over time. Due to the characteristic of short-lived proofs [3], the use of them is meaningful when the verifier/observer does not remain online during the validity period; otherwise, she can always reject the proofs sent afterwards, negating the forgeability property.

Setting an upper bound via short-lived proofs. We bring the *forgeability* property in our PVTSS construction by piggybacking on the *proof of decryptions* used as part of the Reconstruction phase, turning them into short-lived proofs where their expiration time matches the upper time bound T_2. As mentioned in Section 2.5, a DLEQ proof is a Sigma protocol for the relation $R_{DLEQ} = \{(v_1, v_2; w) : (v_1; w) \in R_{DL}, (v_2; w) \in R_{DL}\}$, given a Sigma protocol for proof of knowledge of discrete logarithm relation R_{DL}. As described in [3], a short-lived proof for any arbitrary relation R for which there exists a Sigma protocol can be efficiently constructed *by combining the original statement with a VDF-related statement in a disjunction.* For completeness, we present the short-lived proof for a relation R using pre-computed VDFs in Figure 1. In our protocol we make a black box use of short-lived DLEQ proof generation denoted by DLEQ.SLP and verification denoted by DLEQ.SLV. Notice that according to [3], having access to a *randomness beacon* which outputs random value b periodically is assumed for generation of short-lived proofs. We require that the beacon value b used to compute π_i is not known until the time T_1, with $T = T_2 - T_1$ being the time parameter for the underlying VDF. Given that, anyone verifying a proof before T_2 knows that it could have not been computed through forgery. We remark that, to deploy short-lived proofs we need to use the DDH-based version of SCRAP PVSS which its DLEQ proof comes with *knowledge soundness* property. This ensures that the secrets the adversary samples are independent of the secrets chosen by the honest nodes to use.

The algorithms for PVTSS scheme with short-lived proofs does not change in comparison with Definition 7. However, there is one point to highlight. In that definition, we only made use of T_2 as an argument for PVTSS.Pool run by the verifier, while T_1 is part of the system parameters pp generated by the dealer in initialization. However, when using short-lived proofs, we explicitly feed the upper time bound T_2 and a beacon value b in two algorithms, PVTSS.Recover and PVTSS.Verify$_2$. This is essentially due to necessity of the knowledge of time parameter $t = T_2 - T_1$ and b for proof generation and verification. Note that, as pointed out in [3], t does not need be hardcoded when PVTSS.Setup is run. This allows using VDFs with any time parameter $t' > t$, while still generating short-lived proofs with respect to time t.[6]

The properties of new scheme follow closely the ones described in Section 5.1 with some modification in soundness and security definitions as follows.

Protocol Description. As the protocol description is very much similar to that of presented in Section 4.2, we avoid a complete

[6]This means even if different provers use different time parameters $t' > t$ for their VDF evaluations, only those proofs observed before time t are convincing.

(1) Initialization: On input a random value b^*, computes $\mathsf{VDF.Eval}(pp, b^*) \rightarrow \{y^*, \pi^*_{VDF}\}$
(2) Proof generation: $\mathsf{SLP.Gen}(pp, v, w, b) \rightarrow \pi$,
- Compute $\Sigma.\mathsf{Announce}(v, w) \rightarrow a$
- Compute $c = H(v \parallel b \parallel a)$
- Set sub-challenge $c_2 = b^* \oplus b$
- Compute sub-challenge $c_1 = c \oplus c_2$
- Compute $\Sigma.\mathsf{Response}(v, w, a, c_1) \rightarrow r$
- Output $\pi =: \{a, c_1, r, c_2, y^*, \pi^*_{VDF}\}$

(3) Forgery: $\mathsf{SLP.Forge}(pp, v, b) \rightarrow \tilde{\pi}$,
- Compute $\Sigma.\mathsf{Simulator}(v) \rightarrow (\tilde{a}, \tilde{c_1}, \tilde{r})$
- Compute $c = H(v \parallel b \parallel \tilde{a})$
- Set sub-challenge $c_2 = c \oplus \tilde{c_1}$
- Compute $\mathsf{VDF.Eval}(pp, b \oplus c_2) \rightarrow \{y, \pi_{VDF}\}$
- Output $\tilde{\pi} =: \{\tilde{a}, \tilde{c_1}, \tilde{r}, c_2, y, \pi_{VDF}\}$

(4) Proof verification: $\mathsf{SLP.Verify}(pp, v, \pi/\tilde{\pi}, b) \rightarrow \{0, 1\}$
- Compute $c = H(v \parallel b \parallel a)$
- Accept if:
  - $c = c_1 \oplus c_2$
  - $\Sigma.\mathsf{Verify}(v, a, c_1, r) = 1$
  - $\mathsf{VDF.Verify}(pp, b \oplus c_2, y, \pi_{VDF}) = 1$

Figure 1: Short-lived proof for a relation $R = \{(v, w)\}$ using precomputed VDFs [3]

presentation of the steps and just point out the use of short-lived proofs in the new construction that we denote by $\Pi_{\mathsf{PVTSS}_{\mathsf{slp}}}$.

Reconstruction:

- Recovering: $\mathsf{PVTSS.Recover}(pp, C_i, pk_i, sk_i, b, T_2) \rightarrow \{\tilde{s}_i, \pi_i\}$, after checking the validity of sharing phase, any party P_i wishing to obtain their share, unlocks the blinding factor β_i by running $\mathsf{HTLP.Solve}(pp, Z_i)$, and obtain their share $\tilde{s}_i$ after decypting $\hat{s}_i$ as $\tilde{s}_i = \hat{s}_j^{1/sk_i}$. Then, the party P_i reveals the share $\tilde{s}_i$ together with a proof $\pi_i =: \{\mathsf{DLEQ.SLP}(sk_i, g_1, pk_i, \tilde{s}_i, \hat{s}_i), \beta_i\}$ of valid decryption. Note that DLEQ.SLP involves calling $\mathsf{SLP.Gen}(pp, v, w, b)$ as in Figure 1 for the relation $R_{DLEQ} = \{(g_1, pk_i, \tilde{s}_i, \hat{s}_i; sk_i)\}$ given a beacon value b publicly known no sooner than T_1.
- Recovery verification: $\mathsf{PVTSS.Verify}_2(pp, C_i, \tilde{s}_i, \pi_i, b, T_2) \rightarrow \{0, 1\}$, Any (external) verifier V can check the validity of published share $\tilde{s}_i$ via $\mathsf{DLEQ.SLV}(\pi_i, g_1, pk_i, \tilde{s}_i, \hat{s}_i)$. This involves calling SLP.Verify as in Figure 1. Note that having C_i, the verifier first obtains $\hat{s}_i$ using β_i.
- Pooling: $\mathsf{PVTSS.Pool}(pp, \mathcal{S} = \{\tilde{s}_i\}_{i \in \varphi}, T_2) \rightarrow s$, anyone wishing to reconstruct the secret checks the published proofs π_i are correct as described in Section. Upon having sufficient number of shares (*i.e.*, $\geq t + 1$) received before time T_2, denoted by $\mathcal{S}$, the secret $S = g_1^s$ is reconstructed using Lagrange interpolation in the exponent.

Observe that, the adversary can produce short-lived proofs for some (at most t) randomly selected shares without having to obtain the valid ones (*i.e.*, ones originally sent from dealer) by unlocking the corresponding puzzles. So, to take part in the reconstruction it just needs to start computing VDFs upon the availability of b at T_1. This stems from the fact that the shares $\{s_1, \ldots, s_n\}$, and the secret s are uniformly distributed. Thus, the properties of a short-lived proof including *forgeability* and *indistinguishability* guarantee that no verifier can distinguish an invalid share/proof from a valid one.

Theorem 6. *The construction* $\mathsf{PVTSS}_{\mathsf{slp}}$ *realizes the properties of PVTSS.*

Proof. We show $\mathsf{PVTSS}_{\mathsf{slp}}$ satisfies the security properties of PVTSS presented in Section 5.4.

□

Failure Probability. Using short-lived proofs, we can further analyse the probability of a reconstruction failure by an external verifier after T_2. Let t be the number of adversarial shares and n be the total number of shares publicly available. Given that the incorporation of even *one* invalid share results in an invalid reconstruction, the success probability can be computed as $p = \frac{p_1}{p_2}$, where $p_1 = \binom{n-t}{t+1}$ and $p_2 = \binom{n}{t+1}$. We can show that by a proper choice of the parameters n, t the reconstruction fails with overwhelming probability. Setting $t = \frac{n}{2} - 1$, we have $p \leq n2^{-(\frac{n}{2}+1)}$ which is a negligible value in λ for a choice of $n = \lambda$.[7]

6.4 On The Setup Phase

The PVTSS protocol presented in Section 5.2 requires a setup phase for generating the system parameters pp, in particular for the underlying TLP. As mentioned earlier, making a trusted setup assumption is necessary for the LHTLP of [16] as it does not have a transparent setup and the functionality of the primitive depends on using the proper parameters in puzzle generation such as $h = g^{2^T}$ and strong RSA integer[8] N. Using class groups of imaginary quadratic fields [11] as a family of groups of unknown order instead of the well-known RSA group is an option to reduce the trust, but comes with

[7] Without loss of generality here we assume n is an even number.
[8] N is a strong RSA integer if it is the product of two safe primes.

higher (offline) computational investment for the puzzle generator to compute the parameter $h = g^{2^T}$ through sequential computation [16]. Observe that deploying the class groups solely does not eliminate the need for a trusted setup as it is still feasible that malicious sender (*i.e.*, dealer) fools the receiver (*i.e.*, party) into accepting shares that can never be opened if (g, h) is not a valid tuple. In order to further relax the role of trusted setup, we can protect against malicious sender (i.e. dealer) by enforcing honest behaviour via using proof of exponentiation (see Section 5.2) for $h = g^{2^T}$. Note that this protocol is secure under the adaptive root assumption [49] in hidden-order groups.

As the notion of class groups is less understood and its security and efficiency aspects have not yet been rigorously examined, conventional wisdom suggests choosing the RSA group. We observe that, given the setting of a secret sharing scheme where the privacy of the shares and secret is proven assuming that no collusion occurs between the dealer and each party, the trusted setup assumption can be circumvented while having the dealer as the one who knows the factorization of N. In another words, assume for the moment that the dealer would not collude with any of the parties, she can be forced to compute N correctly (*i.e.*, being a strong RSA integer) via zero-knowledge proof techniques by Camenisch and Michaels [13], ensuring that N is indeed the product of two safe primes. We stress this argument relies on the assumption that the dealer does not reveal the factorization of N to any of the parties, as this would give them the advantage to solve their puzzle much faster and therefore violates the privacy requirement of the PVTSS protocol.

Definition 1 provides our main definition of short-lived proofs. The public parameters pp potentially encapsulate both setup needed for an underlying proof system and setup needed for an underlying VDF. Either or both may represent a trusted setup if they require a secret parameter that can be used to break security assumptions if not destroyed

7 SECRET SHARING WITH ADDITIONAL SHARES

Any threshold secret sharing scheme guarantees *t-privacy* property, preventing any set of parties of size $\leq t$ to reconstruct the secret by pooling their shares. There is also an $n - t$*-robustness* assumption [9], ensuring the existence of sufficient number of valid shares at the reconstruction phase (*i.e.*, guaranteeing that the secret is always reconstructed). However, when it comes to making secret sharing time-based, it is natural to challenge such a liveness assumption and consider a scenario in which a *large* fraction of honest parties goes offline at the determined period for reconstruction, putting the system under the threat of failure.

Our goal is to mitigate this *liveness* assumption for a PVTSS scheme thanks to the capabilities of time-based cryptography. We observe this is feasible by having dealer provides parties with *additional time-locked* shares. By additional we mean some shares other than those that parties get during the Distribution phase of the scheme. Using Shamir secret sharing, the additional shares are computed by evaluating the sharing polynomial $f(\cdot)$ at some distinct publicly known points. So, even if there is less than a threshold of shares available at the reconstruction period (*i.e.*, $[T_1, T_2]$), the remaining parties will be able to open the additional time-locked shares after carrying out some sequential computation and retrieve the secret.

It turns out, however, the availability of additional time-locked shares for the sake of providing robustness may essentially lead to weakening the fault tolerance threshold t. This is clearly because the adversary corrupting at most t parties also gets the chance of acquiring the time-locked shares by putting certain amount of computational efforts. At first glance this seems to be a serious security issue. However, according to the security properties of PVTSS (Section 5.2), the fault tolerance of the system matters prior to the Reconstruction phase. This directly follows the publicity of the system, considering the ability of the adversary to observe the shares publicly posted (on a bulletin board) by (at least) a threshold of parties. So, with respect to this definition together with the existence of an upper time bound T_2 for secret reconstruction, revealing the additional locked shares at a proper time guarantees that the system holds its resiliency within the validity period.

We stress that, our security argument relies on the assumption that the adversary does not have an advantage over honest parties with respect to performing a certain number of steps of sequential computation.[10] An important shortcoming of the time-based cryptographic primitives is the lack of precise timing for the imposed sequential computation. Thus, the real time execution may vary depending on the *speed of available computing platforms/ speed of actual implementation*. Nevertheless, deploying repeated squaring based primitives with fairly simple evaluation function provides a plausible guarantee [40] that adversary cannot use her possible capability over to speedup the computation. *To manage this limitation, conventional wisdom suggests using a VDF with a relatively simple evaluation function for which highly optimized hardware is available to honest parties, limiting the speedup available to attackers. For this reason, repeated squaring based VDFs are considered the most practical candidates today.*

To focus on the core problem, we assume the (additional) time-locked shares are honestly generated. However, should a malicious dealer attempt to misbehave, the structure can be protected by using proper zero-knowledge proof mechanisms similar to the techniques used in the previous section.

Pessimistic condition. The setting we are considering can be referred to as *pessimistic condition*, where the number of *honest* parties drops lower than the required robustness threshold of $n/2$ in the synchronous model [17]. So, with the help of additional shares from the dealer one can do the reconstruction and finish the protocol in the absence of a large number of parties who abort. Note that this is rather in contrary to the *optimistic condition* in the literature, where for instance, it is assumed that the number of available honest parties is at least $\lfloor 3n/4 \rfloor + 1$.

Releasing Additional Shares In One Go. We provide a simple extension to the PVTSS scheme presented in Section 4.2 by having

[9] Note that this is the case in malicious setting where parties may not take part in the reconstruction phase

[10] it is, however, possible to grant the adversary a computational advantage allowing them to perform the computation α times faster. We elaborate on this later on.

dealer publish t additional time-locked shares, trying to bring liveness (robustness) even if all but one honest party [11] goes offline during the reconstruction period.

Construction. The dealer D performs the PVTSS.Sharing to share a secret s as specified in the description of the protocol Π_{PVTSS}. Moreover, she computes t additional shares $\{s'_1, \ldots, s'_t\}$ by evaluating the same Shamir polynomial $f(\cdot)$ containing the secret s at some known distinct points $\{a_1, \ldots, a_t\}$. The dealer then constructs and publishes an RSA-based TLP Z' with time parameter T_2 by running TLP.Gen embedding the concatenation of the shares $s'_1||\ldots||s'_t$ as its solution. The parties in $\mathcal{P}$ run PVTSS.Recover and PVTSS.Verify$_2$ as specified in the description of the protocol Π_{PVTSS}. In addition, anyone willing to obtain the additional shares starts solving the puzzle Z' upon receiving it. So, if less than a sufficient number of shares gets communicated by parties during the reconstruction period $[T_1, T_2]$, the shares locked in Z' would allow the solver/party to reconstruct the secret individually by the time T_2.

Theorem 7. *The above construction realizes the definition of PVTSS from Section* **??**.

Proof. Assuming the additional time-locked shares are contained in a secure TLP, the security properties follow directly from the underlying PVTSS scheme. For brevity we omit a formal treatment, but note the following nuance. In Π_{PVTSS}, the secret is encoded into the exponent and parties obtain $S = h^s$, and not s. Under the discrete logarithm (DL) assumption, the availability of t time-locked shares $s'_1, \ldots, s'_t$ reveals no information about s to anyone in possession of $v_1, \ldots, v_n$ and some shares of form h^{s_j}. Moreover, an adversary who corrupts up to t parties does not have any advantage in performing the sequential computation (containing additional shares) due to inherently non parallelizable nature of the puzzle. □

Remark 3. The reason why the time-locked shares should become accessible no sooner that T_2 is to ensure that, in pessimistic condition, the system achieves the security guarantees of PVTSS. That is (1) the adversary controlling at most t shares does not learn any information about secret s before honest parties do, (2) The fault tolerance of the system does not reduce, as the T_2 is the upper time bound for the system.

Application: Increasing the fault tolerance in packed secret sharing. In the packed Shamir secret sharing which is a beautiful generalization of Shamir secret sharing, the dealer shares l secrets $(s_1, \ldots, s_l)$ using one single (Shamir) polynomial $f(\cdot)$ of degree $t + l - 1$, where t is the fault tolerance threshold. The resulting throughput by packing more shares (increasing the degree of the polynomial) implies weakening the fault tolerance as $t + l$ shares are now needed for reconstruction, limiting the supported fault tolerance to $n - (t + l)$. To achieve the standard Shamir fault tolerance of t in a time-based setting, the dealer can send l additional time-locked shares to make up for the availability of more honest shares for reconstruction. In this way, the scheme can resist against up to $n - t$ faulty parties while the honest parties can reconstruct the secret after obtaining the additional shares by the time T_2.

Remark 4. The notion of *liveness* is close to *fairness* in a sense that both try to ensure the protocol execution terminates while the adversary gains no advantage over honest parties. But, the former targets honest parties in an honest/malicious setting, ensuring a successful termination and the latter aims at all parties in a rational setting, with not promise for a successful termination. Note that here we seek to address liveness, and not fairness. That is, due to the underlying fault tolerance of the scheme, adversary actually cannot get the secret even if a large number of honest parties goes offline, so satisfying fairness. However, due to the setting of our focus (*i.e.*, PVTSS) addressing liveness essentially leads to achieving fairness. In other words, what we accomplish here is *guaranteed output delivery* with respect to the secret.

Remark 5. We can think of using time-locked shares as a fallback strategy for recovering the secret. However, this should be used in a *time-sensitive* secret sharing scheme as the reconstruction time matters when it comes to analysing the security of the scheme. More precisely, since in the normal secret sharing it is not known in advance at what point reconstruction takes place, the availability of the time-locked shares may enable the adversary to recover the secret even before the reconstruction phase starts.

7.1 Decrementing-Threshold Secret Sharing

We observe that it is possible to derive an interesting *trade-off* between time and fault tolerance threshold by having the dealer releases each additional time-locked share periodically at different points in time (epochs). The consequence of this *gradual* release is twofold. Firstly, if necessary, it enables honest parties requiring some more shares (not necessarily t) to construct the secret without going through the sequential computation for the whole period of time *i.e.*, T_2. In fact, they can stop working up to a point where a *sufficient* number of additional shares is gained . Secondly, as time goes by and the reconstruction is not initiated, the adversary can get more additional shares, leading to a gradual lessening of the fault tolerance of the system. Now, we present a formal definition for our new scheme called detrementing-threshold secret sharing (DTSS).

Definition 10 (Decrementing-Threshold Secret Sharing)**.** *A (t, n) decrementing-threshold secret sharing (DTSS) scheme with a time parameter T is a tuple of algorithms* DTSS = (DTSS.Setup, DTSS.Sharing, DTSS.ShaRecover, DTSS.AddRecover, DTSS.Pool) *with the following behaviour:*

(1) **Initialization:**
   - Setup: DTSS.Setup$(1^\lambda, T_1) \rightarrow pp$, *on input security parameter λ, and time parameter T_1, generates system parameters pp. This involves parameters for the underlying secret sharing scheme and also for generating additional locked shares by the dealer D.*

(2) **Distribution:**
   - Sharing: DTSS.Sharing$(pp, s) \rightarrow \{Z_1, \ldots, Z_n, O_1, \ldots, O_t\}$, *on input a secret $s \in S_\lambda$, generates a locked share Z_i with time parameter T_1 for each party P_i in the set $\mathcal{P}$. Moreover, it publishes t additional locked shares $O_1, \ldots, O_t$, with O_i being locked using time parameter $(i + 1)T_1$.*

(3) **Reconstruction:**

[11] Here, we assume the verifier V is one of the participating parties.

– Share recovery: $\mathsf{DTSS.ShaRecover}(pp, Z_i) \to s_i$, *on input system parameters* pp *and locked share* Z_i, *recovers the share* s_i *no sooner than* T_1. *This algorithm is run by each party* P_i.
– Additional share recovery: $\mathsf{DTSS.AddRecover}(pp, O_1, \ldots, O_t) \to s_i'$ *on input system parameters* pp, *and additional locked shares* $O_1, \ldots, O_t$, *forcibly outputs the additional share* s_i' *no sooner than* $(i+1)T$.
– Pooling: $\mathsf{DTSS.Pool}(pp, \mathcal{S}, T_2) \to s$, *on input system parameters* pp *and a set* $\mathcal{S}$ *of shares of size* $> t$ *received before* T_2, *outputs the secret* s. *Note that the shares used can be obtained either through communicating parties' shares and/or obtaining additional time-locked shares over time.*

Formally, the properties that a DTSS scheme must provide the are as follows:

Correctness. DTSS satisfies correctness if for all secret $s \in S_\lambda$ it holds that

$$\Pr\left[\mathsf{TSS.Pool}(pp, \mathcal{S}, T_2) \to s \;:\; \begin{array}{l}\mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{DTSS.Sharing}(pp, s) \to \{\{Z_i\}_{i\in[n]}, \{O_i\}_{i\in[t]}\},\\ \mathsf{DTSS.ShaRecover}(pp, Z_i) \to s_i\\ \mathsf{DTSS.AddRecover}(pp, O_1, \ldots, O_t) \to \{s_i'\}_{i\in[j], 1\le j\le t}\end{array}\right] = 1$$

Privacy. A DTSS satisfies privacy if for all algorithms $\mathcal{A}$ running in time $T < jT_1$, where $1 \le j \le t$, with at most T_1 parallel processors, there exists a simulator Sim and a negligible function μ such that for all secret $s \in S_\lambda$ and $\forall\lambda \in \mathbb{N}$ it holds that

$$\Pr\left[\mathcal{A}(pp, Z_i, O_j) = 1 \;:\; \begin{array}{l}\mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{DTSS.Sharing}(pp, s) \to \{\{Z_i\}_{i\in[n]}, \{O_j\}_{j\in[t]}\}\end{array}\right] -$$

$$\Pr\left[\mathcal{A}(pp, Z_i, O_j) = 1 \;:\; \begin{array}{l}\mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{Sim}(pp) \to \{\{Z_i\}_{i\in[n]}, \{O_j\}_{j\in[t]}\}\end{array}\right] \le \mu(\lambda)$$

Security. Let $2T_1, \ldots, (t+1)T_1$ be times at which each additional time-locked share is forcibly obtained. A DTSS is secure if prior to $(j+1)T_1$, where $1 \le j \le t$, the adversary controlling at most $\le t-(j-1)$ parties learns no information about s in a computational sense.

$$\Pr\left[\mathsf{TSS.Pool}(pp, \mathcal{S}', T_2) \to s \;:\; \begin{array}{l}\mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{DTSS.Sharing}(pp, s) \to \{\{Z_i\}_{i\in[n]}, \{O_i\}_{i\in[t]}\},\\ \mathsf{DTSS.ShaRecover}(pp, Z_i) \to s_i,\\ \mathsf{DTSS.AddRecover}(pp, O_1, \ldots, O_t) \to \{s_i'\}_{i\in[j], 1\le j\le t}\end{array}\right] \le \mu(\lambda)$$

Robustness. A DTSS is robust if each party involved in the protocol can *eventually* reconstruct the secret s, either after receiving sufficient number of other parties' shares and/or obtaining the additional time-locked shares.

Definition 11 (Decrementing-Threshold Secret Sharing). *A* (t, n) *decrementing-threshold secret sharing (DTSS) scheme is a tuple of algorithms* DTSS = (DTSS.Setup, DTSS.Sharing, DTSS.ShaRecover, DTSS.Verify, DTSS.AddRecover, DTSS.Pool) *with the following behaviour:*

(1) **Initialization:**
– Setup: $\mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp$, *on input security parameter* λ *and time parameter* T_1, *generates system parameters* pp. *This involves parameters for the underlying secret sharing scheme and also for generating additional locked shares by the dealer* D. *Also, each party* P_i *announces a registered public key* pk_i *which its secret key* sk_i *is only known to them.*

(2) **Distribution:**
– Sharing: $\mathsf{DTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]}) \to \{C_1, \ldots, C_N, O_1, \ldots, O_t\}$, *on input system parameter* pp *and a randomly selected secret* $s \in S_\lambda$, *generates locked encrypted share* C_i *with time parameter* T_1 *for each party* P_i *with respect to its corresponding public key* pk_i *for* $i \in [n]$. *Moreover, it publicly outputs* t *additional locked shares* $O_1, \ldots, O_t$, *which* O_i *being locked with time parameter* $(i+1)T_1$.

(3) **Reconstruction:**
– Share recovery: $\mathsf{DTSS.ShaRecover}(pp, C_i, pk_i, sk_i) \to \{\tilde{s}_i, \pi_i\}$, *on input system parameters* pp, *locked encrypted share* C_i, *and the key pair* (pk_i, sk_i), *outputs a decrypted share* $\tilde{s}_i$ *together with proof* π_i *of valid decryption. This algorithm is run by each party* P_i.
– Recovery verification: $\mathsf{DTSS.Verify}(pp, C_i, \tilde{s}_i, \pi_i) \to \{0, 1\}$, *on input system parameters* pp, *locked encrypted share* C_i, *decrypted share* $\tilde{s}_i$ *and proof* π_i, *checks the validity of the decryption. This algorithm is run by any (external) verifier* V.
– Additional share recovery: $\mathsf{DTSS.AddRecover}(pp, O_1, \ldots, O_t) \to \{s_i'\}$, *on input system parameters* pp, *additional time-locked share* $O_1, \ldots, O_t$, *forcibly outputs the additional share* s_i' *at time* $(i+1)T_1$. *This algorithm is run by each party* P_i *wishing to obtain some additional share.*
– Pooling: $\mathsf{DTSS.Pool}(pp, \mathcal{S}, T_2) \to s$, *on input system parameters* pp *and a set* $\mathcal{S}$ *of shares of size* $> t$ *received before* T_2, *outputs the secret* s. *Note that shares can be obtained either through communicating parties' shares and/or obtaining additional time-locked shares over time.*

We could also include a verification algorithm $\mathsf{PVTSS.Verify}_2$ for any (external) verifier to check the validity of the presented additional share by a participating party. We refrain from formalizing this algorithm since we implicitly assume all the parties involved in reconstruction (including verifier) obtain and publish the additional timed-locks shares, negating the verification. However, such verification algorithm can introduce efficiency as it allows to reconstruct the secret while having *only one* party solves the puzzles containing additional time-locked shares and lets others know while proving the correctness of the published shares.

Formally, the properties that a DTSS scheme must provide the are as follows:

Correctness. A DTSS satisfies correctness if for all secret $s \in S_\lambda$ and all $i \in [n]$ it holds that

$$\Pr\left[\begin{array}{l}\mathsf{DTSS.Verify}(pp, C_i, \tilde{s}_i, \pi_i) = 1,\\ \mathsf{DTSS.Pool}(pp, \mathcal{S}, T_2) \to s\end{array} \;:\; \begin{array}{l}\mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp,\\ \mathsf{DTSS.Sharing}(pp, s) \to \{\{Z_i\}_{i\in[n]}, \{O_i\}_{i\in[t]}\},\\ \mathsf{DTSS.ShaRecover}(pp, C_i, pk_i, sk_i) \to \{\tilde{s}_i, \pi_i\}\\ \mathsf{DTSS.AddRecover}(pp, O_1, \ldots, O_t) \to \{s_i'\}\end{array}\right]$$

Soundness. A DTSS scheme is sound if there exists a negligible function μ such that for all PPT adversaries $\mathcal{A}$, all $\lambda \in \mathbb{N}$, and $i \in [n]$, it holds that

$$\Pr\left[b = 1 \ : \begin{array}{l} \mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathcal{A}(pp) \to (\{C_i\}_{i\in[n]}, \tilde{s}, \pi), \\ b := \mathsf{PVTSS.Verify}_2(pp, C_i, \tilde{s}_i, \pi_i), \wedge \nexists sk \text{ s.t.} \\ \mathsf{DTSS.ShaRecover}(pp, C_i, pk_i, sk_i) \to \{\tilde{s}_i, \pi_i\}, \end{array} \right] \leq \mu(\lambda)$$

Privacy. A DTSS satisfies privacy if for all algorithms $\mathcal{A}$ running in time $T < jT_1$, where $1 \leq j \leq t$, with at most T_1 parallel processors, there exists a simulator Sim and a negligible function μ such that for all secret $s \in S_\lambda$ and $\forall\lambda \in \mathbb{N}$ it holds that

$$\Pr\left[\mathcal{A}(pp, Z_i, O_j) = 1 \ : \begin{array}{l} \mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{DTSS.Sharing}(pp, s) \to \{\{Z_i\}_{i\in[n]}, \{O_j\}_{j\in[t]}\} \end{array} \right]$$

$$\Pr\left[\mathcal{A}(pp, Z_i, O_j) = 1 \ : \begin{array}{l} \mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{Sim}(pp) \to \{\{Z_i\}_{i\in[n]}, \{O_j\}_{j\in[t]}\} \end{array} \right] \leq \mu(\lambda)$$

Security. Let $2T_1, \ldots, (t+1)T_1$ be times at which each additional time-locked share is forcibly obtained. A DTSS is secure if prior to $(j+1)T_1$, where $1 \leq j \leq t$, the adversary controlling at most $\leq t-(j-1)$ parties learns no information about s in a computational sense.

$$\Pr\left[\mathsf{DTSS.Pool}(pp, \mathcal{S}', T_2) \to s \ : \begin{array}{l} \mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp, \\ \mathsf{DTSS.Sharing}(pp, s) \to \{\{Z_i\}_{i\in[n]}, \{O_i\}_{i\in[t]}\}, \\ \mathsf{DTSS.ShaRecover}(pp, Z_i) \to s_i, \\ \mathsf{DTSS.AddRecover}(pp, O_1, \ldots, O_t) \to \{s'_j\}_{j\in[t], 1\leq j\leq t} \end{array} \right] \leq \mu(\lambda)$$

Robustness. A DTSS is robust if each party involved in the protocol can *eventually* reconstruct the secret s, either after receiving sufficient number of other parties' shares and/or obtaining the additional time-locked shares.

We remark that the above definition for DTSS is generic and can be realized by combining the notion of gradual release of time-locked shares with an underlying threshold secret sharing scheme where the secret is made public after the reconstruction. However, we specifically consider the protocol Π_{PVTSS} for a concrete construction, making it decrementing-threshold.

Construction. We now present a (generalized) construction where the additional time-locked shares are revealed to the parties *gradually* and not in one go. We would like a protocol in which anybody is able to obtain each additional share s'_i at time $(i+1)T_1$ given that dealer's role must end with the distribution phase [12]. In a naive way, the dealer should create t puzzles each embedding one additional share to be opened at t different points in time. However, this inefficient solution comes with a high computation cost as each party wishing to access the shares needs to solve each puzzle separately in parallel, demanding up to $T_1 \sum_{j=1}^{t} j$ number of squaring. To get away with this issue, we use multi-instance time-lock puzzle (MTLP) [1], a primitive allowing sequential release of solutions where *the overall computation cost of solving t puzzles is equivalent to that of solving only the last one.*

(1) **Initialization:**
- Setup: $\mathsf{DTSS.Setup}(1^\lambda, T_1) \to pp$, the dealer D invokes two algorithms of $\mathsf{PVTSS.Setup}(1^\lambda, T_1)$ and $\mathsf{MTLP.Setup}(1^\lambda, T_1, t+1)$, and publishes the public parameters pp.

(2) **Distribution:**
- Sharing: $\mathsf{DTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]}) \to \{\{C_i\}_{i\in[n]}, \{Oj\}_{j\in[t]}\}$, the dealer D invokes $\mathsf{PVTSS.Sharing}(pp, s, \{pk_i\}_{i\in[n]})$ to generate N locked encrypted shares $\{C_i\}_{i\in[n]}$. Moreover, she computes t additional shares $f(a_i) = s'_j$ for $j \in [t]$, where $f(0) = s$ and $\{a_1, \ldots, a_t\}$ are some known distinct point. Finally, she invokes $\mathsf{MTLP.Gen}(\vec{\mathbf{s'}}, pk, sk, \vec{\mathbf{d}})$ to generate an MTLP containing $\{Oj\}_{j\in[t]}$.

(3) **Reconstruction:**
- Share recovery: $\mathsf{DTSS.ShaRecover}(pp, C_i, pk_i, sk_i) \to \{\tilde{s}_i, \pi_i\}$, each party P_i runs $\mathsf{PVTSS.Recover}(pp, C_i, pk_i, sk_i)$ to recover her share $\tilde{s}_i$, and generate a proof of correct decryption π_i.
- Recovery verification: $\mathsf{DTSS.Verify}(pp, C_i, \tilde{s}_i, \pi_i) \to \{0, 1\}$, any (external) verifier V runs $\mathsf{PVTSS.Verify}_2(pp, C_i, \tilde{s}_i, \pi_i)$ to check the validity of the published share $\tilde{s}_i$.
- Additional share recovery: $\mathsf{DTSS.AddRecover}(pp, \{Oj\}_{j\in[t]}) \to \{s'_j\}_{j\in[t]}$, anyone wishing to obtain some additional time-locked share $\{s'_j\}_{j\in[t]}$, runs $\mathsf{MTLP.Solve}(pp, \vec{\mathbf{O}})$.
- Pooling: $\mathsf{DTSS.Pool}(pp, \mathcal{S}, T_2) \to s$, Upon having sufficient number of shares (*i.e.*, $\geq t+1$) before time T_2, anybody can reconstruct the secret by invoking $\mathsf{PVTSS.Pool}(pp, \mathcal{S}, T_2)$.

Theorem 8. *The described construction realizes the security properties of a decrementing-threshold PVTSS with as described in Section 5.1.*

Proof. Apart from Soundness and Privacy which are implied by the underlying protocol Π_{PVTSS}, we show the construction satisfies the Robustness and Security.

Robustness. Since each participating party P_j has at least one share (*i.e.*, her own share), she can eventually reconstruct the secret at most by the time T_2 after solving the MTLP which contains t additional shares.

Security. Using a black box treatment and assuming that the underlying MTLP and PVTSS are secure, the security of the construction falls back to a $(t-i, n)$ PVTSS at time $T_1 + i\Delta$ for $0 \leq i < t$, where $t\Delta = T_2 - T_1$ □

Remark 6. By looking more closely, we see the trade-off between time and fault tolerance can be added as a property to any secret sharing scheme and not merely timed secret sharing. More precisely, the issue of fault tolerance reduction over time is implied by the gradual release of the shares and not the type of underlying secret sharing scheme. In addition, the use of time-locked additional shares implicitly provides an upper time bound. This is because by the time T_2 – which the last puzzle is supposed to open – the secret is revealed to all parties (it can be extended to an external verifier) So, this creates an upper time bound for the system by which anybody (including the adversary) learns the secret. However, we can still have the notion of delayed secret sharing.

[12] Without loss of generality we assume $T_2 = (t+1)T_1$, accommodating the periodic release of additional shares.

Remark 7. One may argue that due to the release of the additional shares anybody can just wait and do not take part in the reconstruction procedure until it gets enough number of shares to solely reconstruct the secret. But, notice that each honest party just has one share and the time they need to wait to solely reconstruct the secret at T_2 is longer than that of an adversary controlling at most $t-1$ parties. So, they never catch up without pooling their shares before availability of the first additional share at $T_{1+\Delta}$.

Protecting against invalid time-locked shares. The assumption about a dealer sending valid time-locked shares may seem to be strong at first glance. However, the crux of the design is to establish a trade-off between time and fault-tolerance by continual release of additional shares, causing the security diminish over time. So, the focus here is on the participation of the parties rather than the dealer's action. Moreover, the construction in [1] assumes an honest dealer and we do not know of any other construction for an MTLP which tolerates malicious dealer while providing the same level of efficiency in solving puzzle instances. We leave this for future work. Having said that, it is possible to protect against a malicious dealer using the same techniques for the PVTSS in Section 4, but at the cost of lowering the efficiently.

Some Notes. When we publish all the additional shares in T_2, we are not worried about the fault tolerance reduction as the system is supposed to stop at T_2, so the additional shares do not give the adversary any advantage as the secret will be reconstructed by all the parties (honest and dishonest) at the same time. However, in the DTSS, the gradual release is more of a scheme for encouraging the honest parties to participate than providing robustness while disclosing the shares in one go does not have the issue of threatening the fault tolerance.

In Dahlia's work [35], the required delay for secret reconstruction comes from the underlying fault-tolerance, meaning that there is always one honest party who does not take part into the reconstruction after some point in time (i.e. until after a transaction has been committed to the Consensus ordering). Note that given this, we need to reason about why we want to use puzzle to introduce a delay in the TSS (In fact we care about delaying shares and not the secret).

8 DISCUSSION

Timed secret sharing in absence of clock. An important observation regarding using "Short-lived proofs" to provide an upper bound is that to some extent we should rely on timing assumption for the proper operation of the system. *This issue becomes serious given that using these proofs doesn't make the system unverifiable after the upper-bound but inconvincible.(from the SLP paper)* So, this means if one has no way to identify the times (at least relatively) he may be fooled into accepting a wrong proof, as VDF only guarantees that it cannot be computed before T. One way to enable parties *to implicitly become aware of the notion of time* (probably relatively as we assume there is no shared clock in the system) is they start computing the TLP (to get to know the lower bound T1) and a VDF proof (to get to know the upper bound T2). So that they don't accept those proofs that arrived after that time. In fact, this is kind of providing the notion of time for parties by using their own computational power.

Why do we need SLP?. We actually focus on two aspects of shares to make the secret reconstruction time-based, namely *availability* and *validity*. More precisely, to enforce a lower bound T_1 for secret reconstruction, the shares should not be available to the parties up to T_1, preventing them to reconstruct the secret. One way we could achieve this goal is to ask dealer time-lock shares to ensure no party can access to its share only after performing some sequential computation for time T_1. For putting an upper bound T_2, however, we need to take a different approach as the shares are already available after time T_1. So, we base our time treatment according to the validity of the shares by considering an expiration time T_2 for shares and respectively for the secret. This is where the SLP comes into play. In fact, as in the reconstruction phase each party sends her (random) share together with a DLEQ proof showing the validity of the share (*i.e.*, the share corresponds to the one she initially received from the dealer), using a SLP on the DLEQ allows making the system time-sensitive in a way that the proofs (shares) are only convincing if sent before T_2.

I have a thought regarding the short-lived proofs (SLPs). One way that makes the use of such a scheme reasonable (even in the original paper that we argued about) is to consider an observer (i.e. verifier) who knows about time (or more accurately relative time) but is not online during the protocol execution. In other words, she's not constantly observing the protocol execution. (edited)

So, to ensure the validity of the received short-lived proofs (or signatures) she needs to be back at most by T_2; otherwise, she cannot make use of the received messages. I think this is the assumption that they make in their paper, having a verifier that knows about relative timing but is not observing the protocol (this also fits the example that I gave regarding email servers).

On having an upper bound. One may argue that the underlying fault-tolerance properties of secret sharing already offer this as the honest parties can do the reconstruction at any time that the protocol specifies. But, what if some of the honest parties go offline at that point and cannot provide their contribution on/right after (T_1)? So, we can think of $T_2 - T_1$ as a time frame in which the parties can make a contribution and reconstruct the secret. Another question question is how to encourage parties to take part in the reconstruction phase ignoring the fault tolerance property of the threshold secret sharing (*i.e.*, $n-t$-robustness). We observe that we can think of such an upper bound as the validity period of the shared secret. So, if parties pool their shares within this time period, it will be guaranteed that the resulting secret is indeed what it should be. Otherwise, the secret may be reconstructed wrongly while seemingly having enough valid shares. So, this possibility of obtaining an invalid secret implicitly forces parties to participate and release their shares within the determined time frame.

When not using SLP to provide the upper time bound, the correct secret can always be reconstructed. In other words, any verifier who is not aware of the notion of time (not tracking the time) can use the shares (even sent after the upper time bound) and obtain the valid secret. To make such a scenario time-sensitive security-wise, SLPs cause the reconstructed secret is only usable within the time

window, making it time sensitive (anybody wishing to reconstruct must be online in the reconstruction phase)

TSS as a generalization of the SS.. If we look a bit closely we realize that a time secret sharing scheme can be considered as a time-based generalization for the secret sharing scheme. In fact, if we set $T_1 = 0$ and $T_2 = \infty$ then what we have is actually a normal secret sharing scheme.

On use of DLEQ.. In fact, what happens in SCRAPE PVSS is that the parties use DLEQ on the knowledge of their secret key and not on the Shamir share s_i as they don't posse that (it is in the exponentiation). This is a crucial point. So each party generates a DLEQ proof for the relations $(h, pk_i, \hat{s}_i, \tilde{s}_i)$. As the encrypted shares are publicly available in "Scrape PVSS" this proof indeed implies that the submitted share is the correct one. But, if we want to put the encrypted share into TLPs then the parties cannot rely on such proof because a corrupt party can simple put two random values (plaintext, ciphertext), which passes the verification and publishes an incorrect share.

An important point regarding SLP. It is crucial to point out that deploying short-lived proof is necessary to force any reconstructor stay online (observe the received shares) during the determined time interval $[T_1, T_2]$. Otherwise, the shares are not guaranteed to be valid, so that the reconstructed secret.

Moreover, since we base our PVTSS protocol on SCRAPE PVSS featuring publicly publishable secret (i.e. anyone can obtain the secret after reconstruction), the protocol can only be executed once. This setting fits well with using Short-lived proofs using precomputed VDFs, where the pre-computed VDF should be used once and doesn't offer re-usable forgeability.

Combining different methods for bringing deniability. Recall that we offer constructions for proofs and signatures based on protocols, where deniability is added by combining the original statement with a VDF-related statement in a disjunction. As noted in Section 3, designated verifier proofs, proofs of work-or-knowledge, and KeyForge/TimeForge also add deniability via disjunction with a second statement. It is straightforward to combine these approaches. For example, a statement could be proven true in zero knowledge only if the proof is received by a specific recipient before a signed timekeeper statement is released or a VDF could have been computed. In this way, a proof can gain deniability in the absence of any trusted third party action in the future (as with short-lived proofs) while also gaining deniability without requiring anybody to solve a VDF if the third party acts faithfully. given the above note, we can also talk about how to get rid of VDF solving by considering a Timekeeping service who acts faithfully (the same definition in the Mattew Green Paper)

Achieving a stronger security guarantee based on time. - We can define the security based on the notion of time. That is, even if there exists an adversary who corrupts all the parties (excluding the dealer) up to some time (not permanently), he cannot obtain the secret. In fact, we get this stronger security guarantee in a timely manner. This holds after the distribution and before time T1, and then we fall back to the normal SS threshold assumption afterwards (after T1).
- Note that the issue with such definition is one may think that if an adversary corrupts a party, it will remain corrupted for ever. Then, this leads to a situation that the secret never gets reconstructed. However, we can reason about it by considering an adversary who may corrupt all the parties up to some time and then it will be restricted to corrupt only a threshold of parties.
- The other way to think about it is that, we can separate the notion of collusion and malicious. In fact, we can say such design protects against even n colluding parties (which only a threshold of them are malicious and may behave arbitrary). Does that make sense?

An application for multi-signature and fairness issue. In a multi-signature scheme, the signing algorithm requires the signing keys from the parties (any number of parties) and the verification algorithm verifies the resulting signature under the aggregated public key. So, if a large threshold of parties go offline, one can sign a transaction using the additional secrets (to be verified under the corresponding public keys that anybody can get to know it by having dealer publishes it online)

A Fairness Problem. In general, the source of difficulty in designing PCNs seems to be rooted in enforcing fairness (a weaker form of guaranteed output delivery) in the locking protocol: We want to ensure that either all parties P1, . . . , Pn learn a valid signature on the corresponding transaction, or none does. This makes it especially tricky to design general purpose solution for this problem, as fairness is notoriously difficult to achieve. Even using powerful cryptographic tools, such as general-purpose multi-party computation (MPC), does not seem to trivialize the problem since fairness for MPC is, in general, impossible to achieve [27, Limits on the security of coin flips when half the processors are faulty (extended abstract)]. The aim of this work is to evade this fairness barrier. We can make a similar argument in the TSS paper saying that if a large threshold of parties go offline, it essentially leads to a situation where there is no GOD while we have still fairness as no body including adversary corrupting up to t parties cannot reconstruct the secret. So, our goal is to provide GOD (for the remaining parties) // liveness (for the remaining parties) in a way that the adversary cannot use the outcome. The upper time bound and the one-time nature of a secret sharing makes such a thing possible.

On the use of non-interactive timed commitment. This paper presents efficient construction for time commitment which feature some useful properties, namely non-interactiveness, public verifiability of the correctness of the commitment, Non-Malleablity of the puzzle. The time commitment has an additional advantage compared to time-lock puzzle that is ensuring the solve that the puzzle is indeed solvable. A time commitment can be shown in an abstract way as com(xi, ri, T)
- Instead of relying on expensive ZK proofs for general NP languages as prior work, we show how to use Fiat-Shamir [FS87] NIZKs derived from Sigma protocols for simple algebraic languages.
- If we think of our design, we see that what we have in TSS is actually the same time-commitment which includes the time-lock puzzle c1 opening to blinding factor, an encryption of share, a DLEQ proof. So, it seems like we can make a black box use of a time commitment to commit to shares by the dealer. The question that should be asked here is if such timed commitment also convince the party that the committed share is indeed the valid share.
- One important question is that if this timed commitment proves any attributes with respected to the secret or it just provides some

proof of opening. Look at following: " The GaTC zero-knowledge proof does not prove any attribute of the secret that is committed to. Rather, it contains only a proof of opening. In contrast, our AVTC primitive, proves that a committed secret, possesses an attribute of the committer's choice."

How to reason about having an honest dealer in DTSS. - Furthermore this restriction can be lifted using additional zero-knowledge proofs
- For simplicity, we assume a semi-honest adversary for the first step (the commitment posting). This assumption can be removed using a generic NIZK proof that P0 generates to prove the well-formedness of the commitment [In their definition we cannot see any proof of correctness from the dealer]
- We can do the same and assume that the dealer is correct. Then say that we can lift such correctness assumption on dealer using the proofs shown in previous section.
- We can also present the complete definition but provide simplified construction.
- The problem with defining the protocol DTSS in the the semi-honest model is that we cannot elaborate on the robustness assumption. However, one may say that it doesn't matter if the parties are malicious or semi-honest. In the context of time, it's possible that some of the honest nodes go offline in the specific time-period so there is not a threshold of nodes to reconstruct the secret.
- However, if we consider the parties malicious, then the notion of robustness makes much more sense. This is because in this case, we can assume the worst case scenario where all if the corrupt parties don't show up and we only have $n - t$ honest parties that the unavailability of any of them would result in reconstruction failure. However, in the normal secret sharing this is not the case and we expect to hear from all the parties involved.

DTSS with publicly publishable secret? - I don't think we should define DTSS using a secret sharing scheme that the secret gets published publicly. In fact, what we're focusing on here is the fact that as time goes the adversary can get more chance to recover the secret as he obtains more share: but this doesn't' mean the secret should be reconstructed publicly.

- We can observe that the DTSS construction is quite interesting in the sense that : (1) first of all it doesn't add communication cost, (2) the dealer (who is honest) can efficiently generate the additional puzzles. (3) The construction nicely achieves our goal that is [breaking public goods] by putting the system under the treat of adversary (4) in the protocol, it's the adversary's job to put computational efforts to get some more shares and try to obtain the secret [the honest parties just need to take part to remain safe] (5) Our assumption regarding honest dealer and malicious parties does make sense with respect to the goal of DTSS (a direct application in MEV-prevention) (6) we can even extend the security of the scheme to protect against malicious adversary) - In the current PVSS definition. The dealer sends at most t shares. So, this means that only one of the participating parties can reconstruct the secret. We can make it public, but I think the current version makes more sense.

Do all the share holders need to obtain the additional shares or not? - We can think of this question in two flavors. First, consider using normal secret sharing with only those taking part in the reconstruction obtain the secret. Since we assume the dealer is honest, then not all the parties involved in the reconstruction should solve the puzzle and just one party is enough. Also, it doesn't matter when the parties get together, before obtaining the required additional shares or after it.
- If we use PVTSS, however, when a party presents some additional shares they should be checked by the verifier as they may be wrong. There are two options, (1) Any body including the verifier (who is / might be a participating party solves the puzzle itself, so it learns the correct additional shares), (2) any party who presents her share (with a DLEQ proof) also provide a proof of correct additional share recovery that is possible using MTLP. So, the verifier needs not to solve puzzles to learn some additional share and can just verify the correctness. I believe this is the better obtain that makes the system works even more efficient. [in this case, only one party is enough to obtain the addition share and post them along with their own share (and DLEQ) on the bulleting board, so reconstruction must start after obtaining the addition shares]

Generating SLPs by adversary without finding her valid encrypted shares at T_1? - It seems like the corrupted parties 9controlled by the adversary) can forge DLEQ proof and send am invalid share with forged proof when submitting at the reconstruction phase. So, it essentially means the adversary need not to even obtain their encrypted share at T_1 by solving their puzzle to generate a valid SLP. This makes the forging process even costly for the adversary as she can just compute a VDF upon the availability of the random beacon at T_1.

How to use time secret sharing for the problem of dead man switch?

9 APPLICATIONS

The use of blockchain to provide Fairness. We can think of how to present our work (TSS) as a way to provide fairness in secret-sharing-based MPC. The current solutions are mainly based on the use of blockchain as a platform to financially motivate participating parties by exchanging funds from those who failed in protocol execution to those who execute the protocol as specified to make up for not obtaining the output. It seems like we can elaborate on the replacement of this approach with TSS with the vital advantage that the notion of incentivization is implicit in our work, without requiring the existence of a blockchain for deriving time and penalizing the parties. In fact, the penalization in our works is done by wasting a large number of computational resources.

Using TSS to incentivize randomness generation. At first glance, it seems like our work can address the issue raised in the STROBE paper regarding incentivizing parties for taking part in the protocol execution for Randomness Generation (revealing the shares), given that the fault-tolerance aspect of the construction provides a public good game requiring only a threshold of parties contribute. The existing solution to break [such a public good game] is to give rewards only to t useful shares that are published first (i.e sooner than the others). However, [I think] our work is orthogonal to the proposed blockchain-based solution in the sense that it doesn't necessarily

encourage parties to publish their shares as they can never do it. But, if they want to do it sometime, they are implicitly forced to do it, as otherwise a computational resource they already put for obtaining the share (note that if this is the case if we want to have a lower bound. Otherwise they don't need to solve a TLP) and computing SLP (especially for a party who aims to behave dishonestly) would be wasted. Moreover, In the context of Randomness Beacon where the parties use PVSS to share their secret, we can think of our approach as an alternative solution for a rewards-based mechanism, but without resorting to the entire power of blockchain.

Regarding the use of our work as a way to incentivize parties to take part in the secret reconstruction (or MPC in some sense), as mentioned, the current approach based on using blockchain reward make use of *on-chain notion of time (or works based on deriving time from blockchain as in the Cross-chain Atomic Swap paper)*. So, we can highlight the fact that our protocol works without explicitly using the notion of time.

Using TSS for randomness with gradual (timed) release. Locking the coins of some parties indefinitely. This is about using the t out of n multisig scripts. *All of these works, however (implicitly) assume an expiration time T for the multi-sig scripts. This is used to ensure that, even if a large threshold of participants goes offline, the coins of the few remaining users are not locked indefinitely. Therefore the scope of the multisig-based protocol is limited to those cryptocurrencies that support the on-chain notion of time. Those [blockchains that do not offer the time-lock functionality] are therefore not compatible with these protocols.*

It seems like the above can be a target application for our third construction (PVTSS with additional shares)

Using TSS + additional shares for Multisig scripts. First, we can make the same argument regarding the possibility that some parties go offline and we address it [with additional shares in a TLPs]. But, there is one more interesting point. In our current scheme we assume that if some parties don't take part in the protocol, then others can make use of those additional shares to do the reconstruction. However, what if we want to have the shares of those parties who don't take part? It would be interesting to think of a setting in which the parties' shares are also put in a TLP (probably multi-instance TLP) so that parties can have access to their shares if needed (or others can get these shares if those parties go offline). If the purpose is to reconstruct the secret, just having some additional shares is enough and it is of no importance to reveal a party's share. But, if we want to use the share for a specific purpose (like the paper AVCS regarding using in the context of MPC for random share) then availability of the shares are important. However, one thing to point out is that the AVCS is in the context of VSS where the dealer had a private connection to each party and may not send a valid share (used as randomness in MPC). But, here as we use PVSS, the shares are guaranteed to be correct in the distribution phase. So, it seems the only way we can make use of those additional shares is by assuming a pessimistic scenario where a large threshold of honest parties may not take part and the dealer is also not available (dealer after the dealing phase goes offline). Please note that, as we discussed before, these additional shares should be given to the parties in a time-sensitive manner as we want to make sure the parties can only have access to it in the reconstruction phase and not before that (in the distribution phase) as it threats the t-privacy aspect of the secret sharing protocol.

Application: Breaking public goods game. The time-fault tolerance trade off property of DTSS encourages the honest parties in a threshold secret sharing scheme to participate in the reconstruction at the earliest time, otherwise, the adversary can make use of additional time-locked shares and obtain the secret even before the honest parties do. This can be considered as a way to break *public goods game* [6] in a secure system based on threshold secret sharing, where participation of only a threshold of parties is needed to obtain the outcome. In fact, it makes a race between parties to take part in the reconstruction stage and release their shares as soon as possible. Encouraging parties to speak out in the protocol execution by rewarding them for releasing their shares through the use of blockchain is a promising incentive mechanism that has been explored in some recent works [31]. Our solution can be thought complementary to the blockchain-based approach which does not require on-chain actions and does the job by *binding* the security of the secret to the notion of time.

Incentives for Randomness Generation. One open question is on managing the incentives of releasing shares for the randomness beacon. On a first sight the fault-tolerance of STROBE makes the problem a public goods game, since only a threshold needs to participate. One way to break this has been proposed in prior work [6,42] where the shares are published on chain and rewards are given only to "useful" shares (i.e. the first t that make it on-chain). This breaks the public goods game and makes it a race between the authorities who are eager to release their share and get rewarded.

Application: Robustenss for multisignature addresses. This special type of address enables control of funds in a community manner. This occurs by needing two or more people together with their cryptographic signatures in order to unlock the funds they contain. In a multi-signature address we need two or more digital signatures for an operation to be valid. In this way, if for example one of these signatures were stolen or hacked, it would be impossible to move the cryptocurrencies because the address is unlocked with more signatures.

Conceptually, a Multisig address is the hash of n public keys pk $1, \ldots, n$ *along with some number* $t \in 1, \ldots, n$ *called a threshold (see [2, 35] for details). To spend funds associated with this address, one creates a transaction containing all n public keys followed by t valid signatures from t of the n public keys, and writes this transaction to the blockchain. The message being signed is the same in all t signatures, namely the transaction data.*

Given above, assume a dealer D wants to establish *Multisignature* addresses. To mitigate the the single point of failure, he distributes the secret to a group of parties to be used to transfer some funds in the future (*i.e.*, delayed secret sharing). He then removes the secret to get away with the risk of compromising the key by an attacker. However, to make sure that the funds will be transferred by the due time he also broadcasts some time-locked shares as a large threshold of (honest) parties might go offline and don't provide their signature (which the signing key is the secret share). So, the remaining parties can make use of the additional shares and finish the protocol (Note that here the verification key for the multi signatures can be g^{s_i} when s_i is the secret share.

Application: Secret sharing in Order-then-Reveal methods. One good example for using DTSS as an incentivizing method is the use of order-then-reveal method for MEV mitigation [35]. In fact, here each user is a dealer (sharing a symmetric key) and BFT parties are shareholders. So, if we deploy the DTSS, then it can be thought of as an encouraging mechanism for (honest) BFT parties to take part and release their shares as soon as possible (here we can set the time for the release of the first additional share equal to the time it takes for an ordering to be committed. So, if we don't have enough number of shares before that time, then the adversary getting one additional share can recover the secret and learn the content of the secret before committing occurs, giving her an advantage for launching an MEV attack.

Application: Mitigating the ability of using secret in the future. It turns out that the secret sharing is not an ideal solution for setting up a threshold cryptographic operations such as threshold signing. This stems from two reasons. First, the existence of a trusted dealer creates a single point of failure where is in oppose to the distributed nature of a threshold setting. Second, when using a secret sharing scheme, at least one of parties would know the complete private key at some point which essentially means he can use it in the future to sign without the contribution (participation) of rest of the parties. Deploying the a timed secret sharing (using SLPs) scheme can mitigate the second issue. [13]

The use of secret Sharing in Order-then-Reveal methods. One good example for the DTSS is this work. In fact, here each user is a dealer (sharing a symmetric key) and BFT parties are shareholders. So, if we deploy the DTSS, then it can be thought of as an encouraging mechanism for BFT parties to take part and release their shares as soon as possible (here we can set the time for the release of the first additional share equal to the time it takes for an ordering to be committed. So, if we don't have enough number of shares before that time, then the adversary getting one additional share can recover the secret and learn the content of the secret before committing occurs, giving her an advantage for launching an MEV attack.

Justin Drake: *'Idea: Having an encryption mechanism that by default is threshold. But, if committee goes offline for some reason then after a period of time the delay kicks is and the it auto-magically decrypts'*

Regarding above, we can essentially address such situation via a DTSS (or the version with revealing at one go), enabling the remaining BFT parties to perform the decryption by having enough number of shares.

One open question is how to reason about encouraging *honest* parties to take part in the reconstruction as it is assumed that they do so as they are honest. This is exactly the same setting in which STROBE considers and raises awareness about the ways for encouragement. However, one way to think about this is that as in our protocol reconstruction is time-based, the honest parties can actually take part anytime within the validity period. Thus, it makes sense to assume some honest parties (for any reason) decide to take part lately. So, in this setting the reconstruction is time sensitive, meaning that time affects the honest parties' action is defined over time while in this paper the reconstruction is incident sensitive, meaning that an incident (*i.e.*, the fact that the ordering has been committed) affects the honest parties' action.

[13] These two issue could be addressed simultaneously via the use of a distributed key generation scheme, but the a DKG inherently comes with high overhead

10 CONCLUSION

REFERENCES

[1] Aydin Abadi and Aggelos Kiayias. 2021. Multi-instance publicly verifiable time-lock puzzle and its applications. In *International Conference on Financial Cryptography and Data Security*. Springer, 541–559.

[2] Ghada Almashaqbeh, Fabrice Benhamouda, Seungwook Han, Daniel Jaroslawicz, Tal Malkin, Alex Nicita, Tal Rabin, Abhishek Shah, and Eran Tromer. 2021. Gage mpc: Bypassing residual function leakage for non-interactive mpc. *Cryptology ePrint Archive* (2021).

[3] Arasu Arun, Joseph Bonneau, and Jeremy Clark. 2022. Short-lived zero-knowledge proofs and signatures. *Cryptology ePrint Archive* (2022).

[4] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2021. Dynamic ad hoc clock synchronization. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 399–428.

[5] Wacław Banasik, Stefan Dziembowski, and Daniel Malinowski. 2016. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *European symposium on research in computer security*. Springer, 261–280.

[6] Donald Beaver, Konstantinos Chalkias, Mahimna Kelkar, Lefteris Kokoris Kogias, Kevin Lewi, Ladi de Naurois, Valeria Nicolaenko, Arnab Roy, and Alberto Sonnino. 2021. STROBE: Stake-based Threshold Random Beacons. *Cryptology ePrint Archive* (2021).

[7] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Annual international cryptology conference*. Springer, 757–788.

[8] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In *Annual International Cryptology Conference*. Springer, 561–586.

[9] Dan Boneh and Moni Naor. 2000. Timed commitments. In *Annual international cryptology conference*. Springer, 236–254.

[10] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. 2015. On bitcoin as a public randomness source. *Cryptology ePrint Archive* (2015).

[11] Johannes Buchmann and Hugh C. Williams. 1988. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology* 1, 2 (1988), 107–118.

[12] Jeffrey Burdges and Luca De Feo. 2021. Delay encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 302–326.

[13] Jan Camenisch and Markus Michels. 1999. Proving in zero-knowledge that a number is the product of two safe primes. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 107–122.

[14] Ignacio Cascudo and Bernardo David. 2017. SCRAPE: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security*. Springer, 537–556.

[15] David Chaum and Torben Pryds Pedersen. 1992. Wallet databases with observers. In *Annual international cryptology conference*. Springer, 89–105.

[16] Alisa Cherniaeva, Ilia Shirobokov, and Omer Shlomovits. 2019. Homomorphic encryption random beacon. *Cryptology ePrint Archive* (2019).

[17] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. 1985. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE, 383–395.

[18] Peter Chvojka, Tibor Jager, Daniel Slamanig, and Christoph Striecks. 2021. Versatile and sustainable timed-release encryption and sequential time-lock puzzles. In *European Symposium on Research in Computer Security*. Springer, 64–85.

[19] Jeremy Clark and Urs Hengartner. 2010. On the use of financial data as a random beacon. In *2010 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 10)*.

[20] Alex Davidson, Peter Snyder, EB Quirk, Joseph Genereux, and Benjamin Livshits. 2021. STAR: Distributed Secret Sharing for Private Threshold Aggregation Reporting. *arXiv preprint arXiv:2109.10074* (2021).

[21] Yevgeniy Dodis and Dae Hyun Yum. 2005. Time capsule signature. In *International Conference on Financial Cryptography and Data Security*. Springer, 57–71.

[22] Yael Doweck and Ittay Eyal. 2020. Multi-party timed commitments. *arXiv preprint arXiv:2005.04883* (2020).

[23] Stephanie Dube Dwilson. 2019. What Happened to Julian Assange's Dead Man's Switch for the WikiLeaks Insurance Files? https://heavy.com/news/2019/04/julian-assange-dead-mans-switch-wikileaks-insurance-files/. Section: News.

[24] Paul Feldman. 1987. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. IEEE, 427–438.

[25] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application*

of cryptographic techniques. Springer, 186–194.

[26] Juan A Garay and Markus Jakobsson. 2002. Timed release of standard digital signatures. In *International Conference on Financial Cryptography*. Springer, 168–182.

[27] Somayeh Heidarvand and Jorge L Villar. 2008. Public verifiability from pairings in secret sharing schemes. In *International Workshop on Selected Areas in Cryptography*. Springer, 294–308.

[28] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 21–30.

[29] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. 2001. Designated verifier proofs and their applications. In *Advances in Cryptology—EUROCRYPT'96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings*. Springer, 143–154.

[30] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 177–194.

[31] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. 2018. Calypso: Private data management for decentralized ledgers. *Cryptology ePrint Archive* (2018).

[32] Yehuda Lindell. 2001. Parallel coin-tossing and constant-round secure two-party computation. In *Annual International Cryptology Conference*. Springer, 171–189.

[33] Yi Liu, Qi Wang, and Siu-Ming Yiu. 2022. Towards Practical Homomorphic Time-Lock Puzzles: Applicability and Verifiability. *Cryptology ePrint Archive* (2022).

[34] Angelique Faye Loe, Liam Medley, Christian O'Connell, and Elizabeth A Quaglia. 2021. TIDE: A novel approach to constructing timed-release encryption. *Cryptology ePrint Archive* (2021).

[35] Dahlia Malkhi and Pawel Szalachowski. 2022. Maximal Extractable Value (MEV) Protection on a DAG. *arXiv preprint arXiv:2208.00940* (2022).

[36] Yacov Manevich and Adi Akavia. 2022. Cross Chain Atomic Swaps in the Absence of Time via Attribute Verifiable Timed Commitments. *Cryptology ePrint Archive* (2022).

[37] Robert J. McEliece and Dilip V. Sarwate. 1981. On sharing secrets and Reed-Solomon codes. *Commun. ACM* 24, 9 (1981), 583–584.

[38] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*. Springer, 223–238.

[39] Krzysztof Pietrzak. 2018. Simple verifiable delay functions. In *10th innovations in theoretical computer science conference (itcs 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[40] Ronald L Rivest, Adi Shamir, and David A Wagner. 1996. Time-lock puzzles and timed-release crypto. (1996).

[41] Alexandre Ruiz and Jorge L Villar. 2005. Publicly verifiable secret sharing from Paillier's cryptosystem. In *WEWoRC 2005–Western European Workshop on Research in Cryptology*. Gesellschaft für Informatik eV.

[42] Berry Schoenmakers. 1999. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*. Springer, 148–164.

[43] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[44] Michael A Specter, Sunoo Park, and Matthew Green. 2021. {KeyForge}:{Non-Attributable} Email from {Forward-Forgeable} Signatures. In *30th USENIX Security Symposium (USENIX Security 21)*. 1755–1773.

[45] Shravan Srinivasan, Julian Loss, Giulio Malavolta, Kartik Nayak, Charalampos Papamanthou, and Sri AravindaKrishnan Thyagarajan. 2022. Transparent Batchable Time-lock Puzzles and Applications to Byzantine Consensus. *Cryptology ePrint Archive* (2022).

[46] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Giulio Malavolta, Nico Döttling, Aniket Kate, and Dominique Schröder. 2020. Verifiable timed signatures made practical. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1733–1750.

[47] Sri Aravinda Krishnan Thyagarajan, Guilhem Castagnos, Fabian Laguillaumie, and Giulio Malavolta. 2021. Efficient CCA Timed Commitments in Class Groups. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2663–2684.

[48] Sri Aravinda Krishnan Thyagarajan, Tiantian Gong, Adithya Bhat, Aniket Kate, and Dominique Schröder. 2021. OpenSquare: Decentralized Repeated Modular Squaring Service. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3447–3464.

[49] Benjamin Wesolowski. 2019. Efficient verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 379–407.