Scalable Time-Lock Puzzles

Aydin Abadi
Newcastle University
Newcastle, United Kingdom
aydin.abadi@ncl.ac.uk

Dan Ristea
University College London
London, United Kingdom
dan.ristea.19@ucl.ac.uk

Artem Grigor
University of Oxford
Oxford, United Kingdom
artem.grigor@cs.ox.ac.uk

Steven J. Murdoch
University College London
London, United Kingdom
s.murdoch@ucl.ac.uk

Abstract

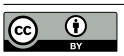
Time-Lock Puzzles (TLPs) enable a client to lock a message such that a server can unlock it only after a specified time. They have diverse applications, such as scheduled payments, secret sharing, and zero-knowledge proofs. In this work, we present a scalable TLP designed for real-world scenarios involving a large number of puzzles, where clients or servers may lack the computational resources to handle high workloads. Our contributions are both theoretical and practical. From a theoretical standpoint, we formally define the concept of a "Delegated Time-Lock Puzzle (D-TLP)", establish its fundamental properties, and introduce an *upper bound* for TLPs, addressing a previously overlooked aspect. From a practical standpoint, we introduce the "Efficient Delegated Time-Lock Puzzle" (ED-TLP) protocol, which implements the D-TLP concept. This protocol enables both the client and server to *securely outsource* their resource-intensive tasks to third-party helpers. It enables *real-time verification* of solutions and guarantees their delivery within predefined time limits by integrating an upper bound and a fair payment algorithm. ED-TLP allows combining puzzles from different clients, enabling a solver to process them sequentially, significantly reducing computational resources, especially for a large number of puzzles or clients. ED-TLP is the first protocol of its kind. We have implemented ED-TLP and conducted a comprehensive analysis of its performance for up to 10,000 puzzles. The results highlight its significant efficiency in TLP applications, demonstrating that ED-TLP securely delegates 99% of the client's workload and 100% of the server's workload with minimal overhead.

CCS Concepts

• **Security and privacy → Cryptography**; **Privacy-preserving protocols**.

Keywords

Time-lock puzzle, smart contracts, and fair payment.



ASIA CCS '25, Hanoi, Vietnam

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/3708821.3733907

ACM Reference Format:
Aydin Abadi, Dan Ristea, Artem Grigor, and Steven J. Murdoch. 2025. Scalable Time-Lock Puzzles. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3708821.3733907

1 Introduction

Time-Lock Puzzles (TLP) are elegant cryptographic primitives that enable the transmission of information to the future. A *client* can lock a message in a TLP such that no *server* can unlock it for a specified duration. May [30] was first to propose sending information into the future, i.e., time-lock puzzle/encryption. As May's scheme requires a trusted agent to release a secret on time, Rivest et al. [39] proposed a trustless TLP RSA-based using sequential modular squaring which became the basis of much of the work in the field. It is secure against a server with high computation resources that support parallelization. Applications of TLP include timely payments in cryptocurrencies [41], e-voting [14], sealed-bid auctions [39], timed secret sharing [28], timed commitments [27], zero-knowledge proofs [18], and verifiable delay functions [10].

To avoid requiring the client or another party trusted by the client to be online at the time of release, TLPs necessarily approximate time using computation, which impose high costs, especially in settings where multiple puzzles are required. In consequence, techniques to improve the scalability of TLPs have been proposed in the literature. However, there are generic settings these techniques do not address.

Consider two clients that need to send unrelated messages to a *resource-constrained* server. One client requires their message to be hidden for 24 hours; the other for 40 hours. Both clients encode their messages as RSA-based time-lock puzzles and send them independently to the server. To obtain both messages on time, the server would have to solve the puzzles in parallel. It would thus be performing total work equivalent to at least 64 hours of CPU time, an obvious inefficiency.

None of the techniques in the literature can reduce the workload in this setting. Homomorphic TLPs [2, 13, 29] enable computation on multiple puzzles to produce a single TLP containing the result and forgo solving individual puzzles separately. Unfortunately, these cannot help in this setting, as the two puzzles are unrelated. Batchable TLPs [40] cannot help either, as they only consider identical time intervals. Chained TLP (C-TLP) [3] enable a client to encode multiple puzzles separated by a fixed interval. The server can obtain each message on time only solving each chained puzzle

sequentially, rather than all puzzles in parallel. However, C-TLP requires that *a single client* generate all puzzles, so two clients cannot chain their puzzles, and the interval between puzzles is fixed, so arbitrary intervals cannot be chained (at least not without incurring a high communication and computational cost, as discussed in subsection 3.2). Therefore, none of the existing TLP variants enable the server to only do work equivalent to the longest puzzle while obtaining both messages at their intended times.

Moreover, for the fundamental feature of TLPs to hold – that they cannot be solved before the intended time – puzzles must be set assuming the capabilities of the most powerful solver in the world. Hence, a less capable server may still struggle to solve even a single, combined puzzle in time. This also introduces issues in settings with multiple servers with heterogeneous capabilities, as the servers will solve the puzzles at different times.

Setting puzzles for the most capable solver introduces its own challenge. Solving capabilities are difficult to estimate and, for longer-running puzzles, to predict how they will improve. The *LCS35 puzzle* released by Rivest [38] is a poignant example. Intended to take 35 years, it was solved in 3.5 years on a general purpose CPU [15]. Subsequently, it was opened in 2 months [16, 17] using field-programmable gate arrays (FPGA), showing that hardware specialized in repeated squaring provides a large speed-up.

Existing schemes consider the lower bound, representing the earliest possible time for the most powerful server to find a solution. However, an *upper bound* indicating the guaranteed time for a regular server to find a solution has not been studied and defined.

1.1 Our Contributions

These limitations inform the key features of the protocols in this paper. Critically, a solution requires secure delegation. On the server-side, delegation helps a server without capabilities offload their work to more powerful helpers. On the client-side, delegation helps clients combine unrelated puzzles into a single chain. To support this, a solution must also allow for arbitrary intervals in the chain.

We therefore introduce the concept of the "Delegated Time-Lock Puzzle" (D-TLP) and present the "Efficient Delegated Time-Lock Puzzle" (ED-TLP), a protocol that realizes this concept.

ED-TLP is a modular protocol that enables secure end-to-end delegation, efficiently handles the multi-puzzle setting, and accommodates varied time intervals. It lets clients and servers delegate tasks to potentially adversarial third-party helpers, who may have specialized hardware. This is especially beneficial when servers have varied capabilities or even lack the capabilities to directly engage in the protocol. Although ED-TLP includes both client and server delegation, each can be used separately. The protocol preserves the privacy of plaintext solutions and supports efficient real-time verification of solutions by relying solely on symmetric-key primitives. In a multi-client setting, when combined with client delegation, this lets multiple clients combine distinct messages into a single puzzle chain. The protocol supports resuming a previously generated puzzle chain, which allows a client to act as a helper for other clients by adding additional puzzles to an existing puzzle.

ED-TLP offers *accurate time* solution delivery, ensuring that a rational helper always delivers a solution before a predefined time. To capture this feature, we rely on two techniques. Firstly, we equip ED-TLP with an explicit upper bound for a helper to deliver a solution, determined by a "Customized Extra Delay Generating" function, which may hold independent interest and find applications in other delay-based primitives, such as Verifiable Delay Functions (VDFs) [10, 19, 45]. Secondly, we incorporate fair payment into ED-TLP, an algorithm that only compensates a solver if it provides a valid solution before a predefined time, made trustless through a smart contract.

We formally define and develop ED-TLP in a modular fashion. First, we define "Generic Multi-instance Time-Lock Puzzle" (GM-TLP) which supports the multi-puzzle setting with different time intervals and efficiently verifying the correctness of solutions. We then present an instantiation of GM-TLP: the "Generic Chained Time-Lock Puzzle" (GC-TLP) protocol. We enhance GM-TLP with client and server delegation to define D-TLP. Finally, we introduce ED-TLP which realizes GM-TLP. Our protocols use a standard time-lock puzzle scheme in a black-box manner.

We implemented and performed an in-depth evaluation of the protocols when generating and solving up to 10,000 puzzles. The implementation is open-source [37]. To the best of our knowledge, it is the first time parties' overhead for a large number of puzzles is studied. We found the computational overhead of ED-TLP to be negligible per puzzle and the cost of the smart contract is as low as 0.2 cents per puzzle. Applications of our (E)D-TLP include enhancing the scalability of (i) VDFs, by securely delegating VDFs computations to reduce the computational load on resource-constrained parties, and (ii) proof of storage, by offloading computationally intensive tasks—such as solving sequential challenges—to resource-capable helpers, as detailed in Section 6.

2 Preliminaries

2.1 Notations and Assumptions

We define function $\mathsf{parse}(\omega, y) \rightarrow (a, b)$, with inputs ω and y of at least ω bits, that parses y into a of length $|y| - \omega$ and b of length ω. To ensure generality in the definition of our verification algorithms, we adopt notations from zero-knowledge proof systems [9, 20]. Let R be an efficient binary relation which consists of pairs of the form (stm, wit), where stm is a statement and wit is a witness. Let $\mathcal{L}$ be the language (in $\mathcal{NP}$) associated with R, i.e., $\mathcal{L} = \{stm | \exists wit \text{ s.t. } R(stm, wit) = 1\}$. A (zero-knowledge) proof for $\mathcal{L}$ allows a prover to convince a verifier that $stm \in \mathcal{L}$ for a common input stm (without revealing wit).

In the formal definitions, we use $\Pr\left[\frac{\mathsf{Exp}}{\mathsf{Cond}}\right]$, where Exp is an experiment that involves an adversary $\mathcal{A}$, and Cond is the set of winning conditions for $\mathcal{A}$.

We denote network delay by Υ. As shown in [6, 22], after an honestly generated block is sent to the blockchain network, there is a maximum time period after which it will be observed by honest parties in the blockchain's unaltered part (a.k.a. state), ensuring a high probability of consistency. For further discussion on the network delay, please refer to Appendix A. We assume that parties use a secure channel when they interact with each other off-chain.

2.2 Symmetric-key Encryption Scheme

A symmetric-key encryption scheme consists of three algorithms: (1) $\mathsf{SKE.keyGen}(1^\lambda) \to k$ is a probabilistic algorithm that outputs a symmetric key k. (2) $\mathsf{SKE.Enc}(k, m) \to c$ takes as input k and a message m in some message space and outputs a ciphertext c. (3) $\mathsf{SKE.Dec}(k, c) \to m$ takes as input k and a ciphertext c and outputs a message m. We require that the scheme satisfies *indistinguishability under chosen-plaintext attacks* (IND-CPA). We refer readers to Appendix B for more details.

2.3 Time-Lock Puzzle

We restate the formal definition of a TLP [39]. Our focus will be on the RSA-based TLP, due to its simplicity and foundational role to the majority of TLPs.

Definition 1. A TLP is a scheme between a client C and a server $\mathcal{S}$, consisting of the following algorithms.

- $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, S) \to (pk, sk)$. A probabilistic algorithm run by C. It takes as input a security parameter, 1^λ, time parameter Δ that specifies how long a message must remain hidden in seconds, and time parameter S, the maximum number of squaring a server (with the highest level of computation resources) can perform per second. It outputs public and private keys (pk, sk).
- $\mathsf{GenPuzzle}_{\mathsf{TLP}}(s, pk, sk) \to p$. A probabilistic algorithm run by C. It takes as input solution s and (pk, sk) and outputs puzzle p.
- $\mathsf{Solve}_{\mathsf{TLP}}(pk, p) \to s$. A deterministic algorithm run by $\mathcal{S}$. It takes as input pk and p and outputs solution s.

TLP meets completeness and efficiency properties:

  * *Completeness*. For any honest C and $\mathcal{S}$, $\mathsf{Solve}_{\mathsf{TLP}}(pk, \mathsf{GenPuzzle}_{\mathsf{TLP}}(s, pk, sk)) = s$.
  * *Efficiency*. The run-time of $\mathsf{Solve}_{\mathsf{TLP}}(pk, p)$ is upper-bounded by a polynomial $poly(\Delta, \lambda)$.

The security of a TLP requires that the puzzle's solution stay confidential from all adversaries running in parallel within the time period, Δ. This is formally stated in Definition 2.

Definition 2. A TLP is secure if for all λ and Δ, all probabilistic polynomial time (PPT) adversaries $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$ where $\mathcal{A}_1$ runs in total time $O(poly(\Delta, \lambda))$ and $\mathcal{A}_2$ runs in time $\delta(\Delta) < \Delta$ using at most polynomial number $\xi(\Delta)$ of parallel processors, there is a negligible function $\mu(\cdot)$, such that:

$$\Pr\left[\begin{array}{l} pk, sk \leftarrow \mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, S) \\ s_0, s_1, \mathrm{state} \leftarrow \mathcal{A}_1(1^\lambda, pk, \Delta) \\ b \xleftarrow{\$} \{0,1\} \\ p \leftarrow \mathsf{GenPuzzle}_{\mathsf{TLP}}(s_b, pk, sk) \\ \hline b \leftarrow \mathcal{A}_2(pk, p, \mathrm{state}) \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

By definition, TLPs are sequential functions. Their construction requires that a sequential function, such as modular squaring, is invoked iteratively a fixed number of times. The sequential function and iterated sequential functions notions, in the presence of an adversary possessing a polynomial number of processors, are formally defined in [10]. We restate the definitions in Appendix C.

Although not required by the definition, a desirable property of a TLP is that generating a puzzle much faster than solving it. Rivest et al. [39] use a trapdoor RSA construction to meet the definition and achieve this property. The security of the RSA-based TLP relies on the hardness of the factoring problem, the security of the symmetric key encryption, and the sequential squaring assumption. This construction is detailed in Appendix D. We restate its formal definition below and refer readers to [3] for the proof.

Theorem 1. *Let N be a strong RSA modulus and Δ be the period for which the solution stays private. If the sequential squaring holds, factoring N is a hard problem and the symmetric-key encryption is semantically secure, then the RSA-based TLP scheme is a secure TLP.*

2.4 Multi-instance Time Lock Puzzle

Abadi and Kiayias [3] proposed the Chained Time-Lock Puzzle (C-TLP) to address the setting in which a server receives multiple puzzle instances separated by a fixed interval. It encodes information required to solve each puzzle in the preceding puzzle's solution. Only one puzzle in the chain needs to be solved at any one time, which, for z puzzles, reduces computations by a factor of $\frac{z+1}{2}$. The scheme also introduces hash-based proofs of the correct solutions. To date, it remains the most efficient time-lock puzzle for the multi-puzzle setting but it works only for fixed intervals.

2.5 Commitment Scheme

A commitment scheme involves two parties: *sender* and *receiver* and two phases: *commit* and *open*. In the commit phase, the sender commits to a message x as $\mathsf{Com}(x, r) = \mathsf{Com}_x$ using secret value $r \xleftarrow{\$} \{0,1\}^\lambda$. The commitment Com_x is sent to the receiver. In the open phase, the sender sends the opening (x, r) to the receiver who verifies its correctness: $\mathsf{Ver}(\mathsf{Com}_x, (x, r)) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy two properties: (1) *hiding*: it is infeasible for an adversary (the receiver) to learn any information about the committed message x until the commitment Com_x is opened, and (2) *binding*: it is infeasible for an adversary (the sender) to open a commitment to different values than those used in the commit phase, meaning it is infeasible to find (x', r') with $x \neq x'$ *s.t.* $\mathsf{Ver}(\mathsf{Com}_x, (x, r)) = \mathsf{Ver}(\mathsf{Com}_x, (x', r')) = 1$, . Efficient non-interactive commitment schemes exist in both the standard model, e.g., Pedersen scheme [35], and the random oracle model using the well-known hash-based scheme such that committing is: $\mathsf{G}(x||r) = \mathsf{Com}_x$ and $\mathsf{Ver}(\mathsf{Com}_x, x)$ requires checking: $\mathsf{G}(x||r) \stackrel{?}{=} \mathsf{Com}_x$, where $\mathsf{G} : \{0,1\}^* \to \{0,1\}^\lambda$ is a collision-resistant hash function; i.e., the probability to find x and x' such that $\mathsf{G}(x) = \mathsf{G}(x')$ is negligible in the security parameter, λ.

2.6 Smart Contract

A smart contract is a computer program that encodes the terms and conditions of an agreement between parties and often contains a set of variables and functions. Smart contract code is stored on a blockchain and is executed by the miners who maintain the blockchain. When (a function of) a smart contract is triggered by an external party and the associated fee (termed *gas*) is paid, every miner executes the smart contract. The correctness of the program execution is guaranteed by the security of the underlying blockchain.

In this work, we use a standard smart contract to retain a predefined deposit amount, check if delegated work was completed

correctly and on time, and distribute the deposit among parties accordingly. Its role can also be played by any semi-honest server (including a semi-honest "Trusted Execution Environment" [34]), even one with minimal computational resources.

3 Generic Multi-instance TLP

Although the Chained TLP (C-TLP) [3] can efficiently deal with multiple puzzles, it cannot handle time intervals of different sizes. We address this limitation by formally defining the Generic Multi-instance Time-Lock Puzzle (GM-TLP) and its concrete instantiation Generic Chained Time-Lock Puzzle (GC-TLP).

3.1 Definition and Security Properties

Definition 3 (Generic Multi-instance TLP)**.** A GM-TLP is a scheme between a client C, a server $\mathcal{S}$, and a public verifier $\mathcal{V}$, that consists of algorithms:

- $\mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \vec{\Delta}, S, z) \rightarrow (pk, sk)$. A probabilistic algorithm run by C. It takes as input security parameter 1^λ, a vector of time intervals $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$ (such that the j-th solution is not found before $\sum_{i=1}^{j} \bar{\Delta}_i$), S, and the total number of puzzles z. It outputs a set of public pk and private sk parameters.
- $\mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\vec{m}, pk, sk) \rightarrow \hat{p}$. A probabilistic algorithm run by C. It takes as input a message vector $\vec{m} = [m_1, \ldots, m_z]$ and (pk, sk) and outputs $\hat{p} \coloneqq (\vec{p}, \vec{g})$, where $\vec{p}$ is a puzzle vector and $\vec{g}$ is a public statement vector w.r.t. language $\mathcal{L}$ and $\vec{m}$. Each j-th element in vectors $\vec{p}$ and $\vec{g}$ corresponds to a solution s_j which itself contains m_j (and possibly witness parameters). C publishes $\vec{g}$ and sends $\vec{p}$ to $\mathcal{S}$.
- $\mathsf{Solve}_{\mathsf{GMTLP}}(pk, \vec{p}) \rightarrow (\vec{s}, \vec{m})$. A deterministic algorithm run by $\mathcal{S}$. It outputs solution vector $\vec{s}$ and plaintext message vector $\vec{m}$.
- $\mathsf{Prove}_{\mathsf{GMTLP}}(pk, s_j) \rightarrow \pi_j$. A deterministic algorithm run by $\mathcal{S}$. It takes as input pk and a solution: $s_j \in \vec{s}$ and outputs a proof π_j (asserting $m_j \in \mathcal{L}$). $\mathcal{S}$ sends $m_j \in s_j$ and π_j to $\mathcal{V}$.
- $\mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j, \pi_j, g_j) \rightarrow v_j \in \{0, 1\}$. A deterministic algorithm run by $\mathcal{V}$. It takes as input pk, j, m_j, π_j, and $g_j \in \vec{g}$ and outputs 1 if it accepts the proof or 0 otherwise.

GM-TLP satisfies completeness and efficiency:

- *Completeness*. For any honest prover and verifier, $\forall j \in [z]$, it holds that:
  - $\mathsf{Solve}_{\mathsf{GMTLP}}(pk, [p_1, \ldots, p_j]) \rightarrow ([s_1, \ldots, s_j], [m_1, \ldots, m_j])$.
  - $\mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j, \pi_j, g_j) \rightarrow 1$.
- *Efficiency*. The run-time of algorithm $\mathsf{Solve}_{\mathsf{GMTLP}}(pk, [p_1, \ldots, p_j]) = ([s_1, \ldots, s_j], [m_1, \ldots, m_j])$ is bounded by a polynomial $poly(\sum_{i=1}^{j} \bar{\Delta}_i, \lambda)$, $\forall j \in [z]$.

In the above, the prover is required to generate a witness/proof π_j for the language $\mathcal{L} \coloneqq \{stm \coloneqq (g_j, m_j) | \ R(stm, \pi_j) = 1\}$. The proposed definition is generalization of the definition provided in [3]: (1) different size intervals between puzzles are allowed and (2) a broad class of verification algorithms and schemes are supported, not just hash-based commitments. We allow $\mathsf{Solve}_{\mathsf{GMTLP}}$ to output both the solution vector and the plaintext message vector, as the solution to a puzzle may be an encoded version of the plaintext.

At a high level, GM-TLP satisfies a solution's *privacy* and *validity*. Solution's privacy requires the j-th solution to remain hidden from all adversaries that run in parallel in the period: $\sum_{i=1}^{j} \bar{\Delta}_i$. The solution's validity states that it is infeasible for a PPT adversary to compute an invalid solution and pass the verification. We formally define the two properties below.

Definition 4 (Privacy)**.** A GM-TLP is privacy-preserving if for all λ and $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, any number of puzzle: $z \geq 1$, any $j \in [z]$, any pair of randomized algorithms $\mathcal{A} \coloneqq (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in time $O(poly(\sum_{i=1}^{j} \bar{\Delta}_i, \lambda))$ and $\mathcal{A}_2$ runs in time $\delta(\sum_{i=1}^{j} \bar{\Delta}_i) < \sum_{i=1}^{j} \bar{\Delta}_i$ using at most polynomial $\xi(\max(\bar{\Delta}_1, \ldots, \bar{\Delta}_j))$ parallel processors, there exists a negligible function $\mu(\cdot)$, such that:

$$\Pr\left[\begin{array}{l} pk, sk \leftarrow \mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \vec{\Delta}, S, z) \\ \vec{m}_0, \vec{m}_1, \text{state} \leftarrow \mathcal{A}_1(1^\lambda, pk, z) \\ b_j \xleftarrow{\$} \{0,1\}, \forall j \in [z] \\ \hat{p} \leftarrow \mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\vec{m}_b, pk, sk) \\ \hline b', k \leftarrow \mathcal{A}_2(pk, \hat{p}, \text{state}) \\ \text{s.t. } b' = b_k \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

where $k \in [z]$, $\vec{m}_0 = [m_{0,1},\ldots,m_{0,z}]$, $\vec{m}_1 = [m_{1,1},\ldots,m_{1,z}]$ and $\vec{m}_b = [m_{b_{1,1}}, \ldots, m_{b_z,z}]$.

Definition 4 ensures solutions appearing after the j-th solution remain hidden from the adversary with a high probability. Moreover, similar to [3, 10, 23, 29], it ensures privacy even if $\mathcal{A}_1$ computes on the public parameters for a polynomial time, $\mathcal{A}_2$ cannot find the j-th solution in time $\delta(\sum_{i=1}^{j} \bar{\Delta}_i) < \sum_{i=1}^{j} \bar{\Delta}_i$ using at most $\xi(\max(\bar{\Delta}_1, \ldots, \bar{\Delta}_j))$ parallel processors, with a probability significantly greater than $\frac{1}{2}$. As shown in [10], we can set $\delta(\bar{\Delta}) = (1-\epsilon)\cdot\bar{\Delta}$ for a small ϵ, where $0 < \epsilon < 1$.

Definition 5 (Solution-Validity)**.** A GM-TLP preserves solution validity, if for all λ and $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, any number of puzzles: $z \geq 1$, all PPT adversaries $\mathcal{A} \coloneqq (\mathcal{A}_1, \mathcal{A}_2)$ that run in time $O(poly(\sum_{i=1}^{z} \bar{\Delta}_i, \lambda))$ there is a negligible function $\mu(\cdot)$, such that:

$$\Pr\left[\begin{array}{l} pk, sk \leftarrow \mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \vec{\Delta}, S, z) \\ \vec{m}, \text{state} \leftarrow \mathcal{A}_1(1^\lambda, pk, \vec{\Delta}) \\ \hat{p} \leftarrow \mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\vec{m}, pk, sk) \\ \vec{s}, \vec{m} \leftarrow \mathsf{Solve}_{\mathsf{GMTLP}}(pk, \vec{p}) \\ \hline j, \pi', m_j \leftarrow \mathcal{A}_2(pk, \vec{s}, \hat{p}, \text{state}) \\ \text{s.t. } \mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j, \pi', g_j) = 1 \\ m_j \notin \mathcal{L} \end{array}\right] \leq \mu(\lambda)$$

where $\vec{m} = [m_1, \ldots, m_z]$, and $g_j \in \vec{g} \in \hat{p}$.

Definition 6 (Security)**.** A GM-TLP is secure if it satisfies solution-privacy and solution-validity, w.r.t. Definitions 4 and 5, respectively.

3.2 Strawman Solutions

Before presenting our solution, we sketch potential schemes that provide variable intervals $\bar{\Delta}_j \neq \bar{\Delta}_{j+1}$ (or approximate them) and highlight their limitations to motivate our protocol.

One could use TLP as a black-box by symmetrically encrypting each message and including the key in the preceding puzzle to construct a chain. This would have varied intervals but would require a fresh RSA key for each message imposing a computational and communication costs for C. Another approach would be to

employ C-TLP [3] as a black-box by generating dummy puzzles to precede every real puzzle in the chain and approximate variable duration. Specifically, C identifies the largest value Δ' that exactly divides all time intervals $\bar{\Delta}_j$. To generate each puzzle p_i containing message m_i, C computes $c_i = \frac{\bar{\Delta}_i}{\Delta'}$, then uses C-TLP to produce c_i chained puzzles: $c_i - 1$ dummy puzzles followed by a puzzle containing m_i. Given the example in section 1, with two messages that are to be kept hidden for 24 hours and 40 hours, respectively, this would require a fixed interval of 8 hours and would produce 5 puzzles, in order: two dummy puzzles, a puzzle containing the message to be kept hidden for 24h, another dummy puzzle, and a puzzle containing the message to be kept hidden for 40h. Although only one RSA modulus is needed, C has to perform additional modular exponentiation and transmit dummy puzzles to $\mathcal{S}$. These costs are linear with the total number of dummy puzzles, which can explode when Δ' has to be very small to accurately capture the distinct intervals. For example, combining a 99 hour puzzle and a 100 hour puzzle would require 100 total puzzles, a 50x increase in communication costs.

The protocol described in the next section offers a high level of flexibility (w.r.t. the size of intervals) without suffering from the aforementioned downsides.

3.3 Generic Chained Time-lock Puzzle (GC-TLP)

We present GC-TLP that realizes GM-TLP. At a high level, GC-TLP lets a client C encode a vector of messages $\vec{m} = [m_1, \ldots, m_z]$ so a server $\mathcal{S}$ learns each message m_j at time $t_j \in \vec{t}, \forall j \in [z]$. In this setting, C is available only at an earlier time t_0, where $t_0 < t_1$.

In GC-TLP, we use the chaining technique, similar to C-TLP, and define the interval between puzzles as $\bar{\Delta}_j = t_j - t_{j-1}$, $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$. In the setup, for each $\bar{\Delta}_j$, C generates a set of parameters, including a time-dependent parameter a_j, which allows the message to be encoded with its corresponding time interval. In the puzzle generation phase, C uses the above parameters to encrypt each message m_j. The information required for decrypting m_j is contained in the ciphertext of message m_{j-1}. $\mathcal{S}$ must solve the puzzles sequentially in the specified order, starting with extracting message m_1, which uniquely has the values required to solve it publicly available. Below, we present GC-TLP in detail.

(1) $\mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \vec{\Delta}, S, z) \rightarrow (pk, sk)$. Involves C.
   (a) compute $N = q_1 \cdot q_2$, where q_i are large randomly chosen prime number. Next, compute Euler's totient function of N, as: $\phi(N) = (q_1 - 1) \cdot (q_2 - 1)$.
   (b) $\forall j \in [z]$, set $T_j = S \cdot \bar{\Delta}_j$ as the number of squarings needed to decrypt an encrypted message m_j after message m_{j-1} is revealed, where S is the maximum number of squarings modulo N per second the strongest server can perform, and set $\vec{T} = [T_1, \ldots, T_z]$.
   (c) compute values $a_j = 2^{T_j} \bmod \phi(N), \forall j \in [z]$, which yields vector $\vec{a} = [a_1, \ldots, a_z]$.
   (d) pick fixed size random generators: $r_j \xleftarrow{\$} \mathbb{Z}_N^*, \forall j \in [z+1]$ with $|r_j| = \omega_1$, and set $\vec{r} = [r_2, \ldots, r_{z+1}]$. Generate z symmetric-key encryption keys: $\vec{k} = [k_1, \ldots, k_z]$ and pick z fixed size sufficiently large random values: $\vec{d} = [d_1, \ldots, d_z]$, with $|d_j| = \omega_2$.
   (e) set public key as $pk \coloneqq (aux, N, \vec{T}, r_1, \omega_1, \omega_2)$ and secret key as $sk \coloneqq (q_1, q_2, \vec{a}, \vec{k}, \vec{r}, \vec{d})$, where aux contains a hash function's description and the size of the random values. Output pk and sk.

(2) $\mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\vec{m}, pk, sk) \rightarrow \hat{p}$. Involves C. Encrypt the messages $\forall j \in [z]$:
   (a) set $pk_j \coloneqq (N, T_j, r_j)$ and $sk_j \coloneqq (q_1, q_2, a_j, k_j)$. Recall r_1 is in pk; for $j > 1$, r_j is in $\vec{r}$.
   (b) generate a puzzle by calling the puzzle generator algorithm (in the RSA-based puzzle scheme): $\mathsf{GenPuzzle}_{\mathsf{TLP}}(m_j||d_j||r_{j+1}, pk_j, sk_j) \rightarrow p_j$
   (c) commit to each message: $\mathrm{G}(m_j||d_j) = g_j$ and output g_j.
   (d) output p_j as puzzle.

   This phase yields vectors of puzzles: $\vec{p} = [p_1, \ldots, p_z]$ and commitments: $\vec{g} = [g_1, \ldots, g_z]$. Let $\hat{p} \coloneqq (\vec{p}, \vec{g})$. All public parameters and puzzles are given to the server at time $t_0 < t_1$, where $\bar{\Delta}_1 = t_1 - t_0$, and $\vec{g}$ is sent to the public. This algorithm can be easily parallelized over the values of j.

(3) $\mathsf{Solve}_{\mathsf{GMTLP}}(pk, \vec{p}) \rightarrow (\vec{s}, \vec{m})$. Involves $\mathcal{S}$. Decrypt the messages in order from $j = 1, \forall j \in [z]$:
   (a) if $j = 1$, $r_1 \in pk$; otherwise, r_j is obtained by solving and parsing the previous puzzle.
   (b) set $pk_j \coloneqq (N, T_j, r_j)$.
   (c) call the puzzle solving algorithm in the TLP scheme: $\mathsf{Solve}_{\mathsf{TLP}}(pk_j, p_j) \rightarrow x_j$, where $p_j \in \vec{p}$.
   (d) parse $x_j = m_j||d_j||r_{j+1}$ as:
      (i) $\mathsf{parse}(\omega_1, m_j||d_j||r_{j+1}) \rightarrow (m_j||d_j, r_{j+1})$.
      (ii) $\mathsf{parse}(\omega_2, m_j||d_j) \rightarrow (m_j, d_j)$.
      (iii) output $s_j = m_j||d_j$ and m_j.

   By the end of this phase, vectors of solutions $\vec{s} = [s_1, \ldots, s_z]$ and plaintext messages $\vec{m} = [m_1, \ldots, m_z]$ are generated.

(4) $\mathsf{Prove}_{\mathsf{GMTLP}}(pk, s_j) \rightarrow \pi_j$. Involves $\mathcal{S}$.
   (a) parse s_j into (m_j, d_j).
   (b) send m_j and $\pi_j = d_j$ to the verifier.

(5) $\mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j, \pi_j, g_j) \rightarrow a_j \in \{0, 1\}$. Involves the verifier, e.g., the public or C.
   (a) verify the commitment: $\mathrm{G}(m_j, \pi_j) \stackrel{?}{=} g_j$.
   (b) if verification succeeds, accept the solution and output 1; otherwise, output 0.

Appendix E presents the security proof of GC-TLP.

Note that C can extend a puzzle chain later, only retaining pk and sk, without having to be online in the intervening time. A server must fully solve the existing chain before starting on new puzzles. Appendix G specifies the steps required.

4 Delegated Time-lock Puzzle

We introduce the notion of Delegated TLP (D-TLP), which is an enhancement of GM-TLP. We then proceed to present Efficient Delegated Time-Lock Puzzle (ED-TLP), a protocol embodying the properties of D-TLP. The key features that D-TLP offers include:

(1) *Client-side delegation*, that allows C to delegate setup and puzzle generation phases to a third-party helper $\mathcal{HC}$ (modeled as a semi-honest adversary) while preserving the privacy of its plaintext messages.

(2) *Server-side delegation*, which enables $\mathcal{S}$ to offload puzzle solving to another third-party helper $\mathcal{HS}$ (modeled as a rational adversary with a utility function equal to the sum of payments made to the helper minus the helper's computational costs) while ensuring the privacy and integrity of the plaintext solutions.
(3) *Accurate-time solution delivery*, that guarantees each solution is delivered before a pre-defined time.

To capture the latter property, we parameterize D-TLP with an *upper bound* and a *fair payment algorithm*.

An upper bound explicitly specifies the time by which a helper must find a solution. This parameter is not explicit in existing TLPs: the only time parameter is the lower bound. To establish the upper bound, we introduce the "Customized Extra Delay Generating" function, which produces a time parameter, denoted as Ψ. This specifies the duration after Δ when the helper is expected to discover the solution to the puzzle.

Definition 7 (Customized Extra Delay Generating Function)**.** Let ToC be the type of computational step a scheme relies on, S be the maximum number of steps a server with the best computational resources can compute per second, Δ be the interval in seconds a message must remain private, and aux_{ID} be auxiliary data about a specific server identified with ID, e.g., its computational resources. The Customized Extra Delay Generating function is defined as:

$$\mathsf{CEDG}(ToC, S, \Delta, aux_{ID}) \rightarrow \Psi_{ID}$$

It returns Ψ_{ID}: the extra time in seconds solver ID requires to solve the puzzle in addition to Δ.

A fair payment algorithm compensates a party only when it submits a valid solution before a pre-defined deadline. Implementing fair payments is crucial to guarantee the timely delivery of a solution by a third party helper $\mathcal{HS}$ in the context of delegated TLP. Note that simply relying on an upper bound would not guarantee that the solution will be delivered by a rational adversary that has already identified the solution. This algorithm will be incorporated into a smart contract.

4.1 Definition and Security Properties

We initially present the syntax of D-TLP in Definition 8 and then present its security properties, in Definitions 9, 10, and 11.

Definition 8 (Delegated Time-Lock Puzzle)**.** A D-TLP is a scheme between a client C, a server $\mathcal{S}$, a pair of third-party helpers ($\mathcal{HC}$, $\mathcal{HS}$), and a smart contract $\mathcal{SC}$ that consists of algorithms:

- $C.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}) \rightarrow (cpk, csk)$. A probabilistic algorithm run by C. It generates a set of public cpk and private csk parameters, given $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, where $\sum_{i=1}^{j} \bar{\Delta}_i$ is the period after which the j-th solution is discovered. It appends $\vec{\Delta}$ to cpk and outputs (cpk, csk). C sends cpk and csk to $\mathcal{S}$.
- $C.\mathsf{Delegate}_{\mathsf{DTLP}}(\vec{m}, cpk, csk) \rightarrow (\vec{m}^*, t_0)$. A probabilistic algorithm run by C. It encodes each element of $\vec{m} = [m_1, \ldots, m_z]$ to produce a vector of encoded messages $\vec{m}^* = [m_1^*, \ldots, m_z^*]$ and time point t_0 when puzzles are provided to $\mathcal{HS}$. C publishes t_0 and sends $\vec{m}^*$ to $\mathcal{HC}$.
- $\mathcal{S}.\mathsf{Delegate}_{\mathsf{DTLP}}(\mathsf{CEDG}, ToC, S, \vec{\Delta}, aux, adr_{\mathcal{HS}}, t_0, \Upsilon, \overrightarrow{coins}) \rightarrow adr_{\mathcal{SC}}$. A deterministic algorithm run by $\mathcal{S}$. It runs $\mathsf{CEDG}(ToC, S, \bar{\Delta}_i, aux) \rightarrow \Psi_i$, which uses the auxiliary information about the helper aux to determine the acceptable delay for every $\bar{\Delta}_i$ in $\vec{\Delta}$. It generates a smart contract $\mathcal{SC}$, deployed into the blockchain with address $adr_{\mathcal{SC}}$. $\mathcal{S}$ deposits $coins = \sum_{i=1}^{z} coins_i$ coins into $\mathcal{SC}$, where $coins_i \in \overrightarrow{coins}$ is the amount of coins paid to $\mathcal{HS}$ for solving the i-th puzzle. It registers $t_0, \vec{\Psi} = [\Psi_1, \ldots, \Psi_z]$, $adr_{\mathcal{HS}}$, and Υ in $\mathcal{SC}$. It returns $adr_{\mathcal{SC}}$, the address of the deployed $\mathcal{SC}$. $\mathcal{S}$ publishes $adr_{\mathcal{SC}}$.
- $\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}, S, z) \rightarrow (pk, sk)$. A probabilistic algorithm run by $\mathcal{HC}$. It outputs a set of public pk and private sk parameters. $\mathcal{HC}$ sends pk to $\mathcal{HS}$.
- $\mathsf{GenPuzzle}_{\mathsf{DTLP}}(\vec{m}^*, cpk, pk, sk, t_0) \rightarrow \hat{p}$. A probabilistic algorithm run by $\mathcal{HC}$. It outputs $\hat{p} := (\vec{p}, \vec{g})$, where $\vec{p}$ is a puzzle vector, $\vec{g}$ is a public statement vector w.r.t. language $\mathcal{L}$ and $\vec{m}^*$. Each j-th element in vectors $\vec{p}$ and $\vec{g}$ corresponds to a solution s_j that itself consists of m_j^* (and possibly witness parameters). $\mathcal{HC}$ sends $\vec{g}$ to $\mathcal{SC}$. It also sends $\vec{p}$ to $\mathcal{HS}$ at time t_0.
- $\mathsf{Solve}_{\mathsf{DTLP}}(\vec{\Psi}, aux, pk, \vec{p}, adr_{\mathcal{SC}}, \overrightarrow{coins}', \overrightarrow{coins}) \rightarrow (\vec{s}, q)$. A deterministic algorithm run by $\mathcal{HS}$, in which $\overrightarrow{coins}'$ is the vector of payments $\mathcal{HS}$ expects. It checks $\overrightarrow{coins}$ and $\vec{\Psi}$. If it does not agree on these parameters, it sets $q = 0$ and outputs $(, q)$ and the rest of the algorithms will not be run. Else, it sets $q = 1$, finds the solutions, and outputs a vector $\vec{s}$ of solutions and q.
- $\mathsf{Prove}_{\mathsf{DTLP}}(pk, s_j) \rightarrow \pi_j$. A deterministic algorithm run by $\mathcal{HS}$. It outputs a proof π_j asserting $m_j^* \in \mathcal{L}$.
- $\mathsf{Register}_{\mathsf{DTLP}}(s_j, \pi_j, adr_{\mathcal{SC}}) \rightarrow t_j$. A deterministic algorithm run by $\mathcal{HS}$. It registers m_j^* and π_j in $\mathcal{SC}$, where $m_j^* \in s_j$. It receives registration time t_j from $\mathcal{SC}$. It outputs t_j.
- $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j, t_0, \vec{\Delta}, \Upsilon) \rightarrow v_j$. A deterministic algorithm run by $\mathcal{SC}$. It checks whether (i) proof π_j is valid and (ii) the j-th solution was delivered on time. If both checks pass, it outputs $v_j = 1$; otherwise, it outputs $v_j = 0$.
- $\mathsf{Pay}_{\mathsf{DTLP}}(v_j, adr_{\mathcal{HS}}, \overrightarrow{coins}, j) \rightarrow u_j$. A deterministic algorithm run by $\mathcal{SC}$. It pays out based on the result of verification v_j. If $v_j = 1$, it sends $coins_j \in \overrightarrow{coins}$ coins to an account with address $adr_{\mathcal{HS}}$ and sets $u_j = 1$; otherwise, it sets $u_j = 0$. It outputs u_j.
- $\mathsf{Retrieve}_{\mathsf{DTLP}}(pk, csk, m_j^*) \rightarrow m_j$. A deterministic algorithm run by $\mathcal{S}$. It retrieves message m_j from m_j^* and outputs m_j.

D-TLP satisfies completeness and efficiency properties.

* *Completeness*. For honest C, $\mathcal{S}$, $\mathcal{SC}$, $\mathcal{HC}$, and $\mathcal{HS}$, $\forall j \in [z]$, it holds that:
- $\mathsf{Solve}_{\mathsf{DTLP}}(\vec{\Psi}, aux, pk, [p_1, \ldots, p_j]), adr_{\mathcal{SC}}, \overrightarrow{coins}', \overrightarrow{coins}) \rightarrow ([s_1, \ldots, s_j], 1)$.
- $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j, t_0, \vec{\Delta}, \Upsilon) \rightarrow v_j = 1$.
- $\mathsf{Pay}_{\mathsf{DTLP}}(ver_j, adr_{\mathcal{HS}}, \overrightarrow{coins}, j) \rightarrow u_j = 1$.
- $\mathsf{Retrieve}_{\mathsf{DTLP}}(pk, csk, m_j^*) \rightarrow m_j$, where $m_j \in \vec{m}$.

* *Efficiency*. The run-time of $\mathsf{Solve}_{\mathsf{DTLP}}(\vec{\Psi}, aux, pk, [p_1, \ldots, p_j], adr_{\mathcal{SC}}, \overrightarrow{coins}', \overrightarrow{coins}) \rightarrow ([s_1, \ldots, s_j], 1)$ is bounded by a polynomial $poly(\sum_{i=1}^{j} \bar{\Delta}_i, \lambda)$.

D-TLP's definition, similar to GM-TLP, supports a solution's privacy and validity with the addition that privacy requires plaintext messages $[m_1, \ldots, m_z]$ to remain hidden from $\mathcal{HC}$ and $\mathcal{HS}$.

In Case 1 in Definition 9, we state that given the algorithms' transcripts, an adversary that picks a pair of plaintext messages cannot tell which message is used for the puzzle with a probability

significantly greater than $\frac{1}{2}$. In Case 2 in Definition 9, we formally state that the privacy of an encoded solution m_j^* requires j-th encoded solution to remain hidden from all adversaries that run in parallel in period $\sum_{i=1}^{j} \bar{\Delta}_i$.

Definition 9 (Privacy)**.** A D-TLP is privacy-preserving if for all λ and $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, any number of puzzles: $z \geq 1$, any $j \in [z]$ the following hold:

(1) For any PPT adversary $\mathcal{A}_1$, there exists a negligible function $\mu(\cdot)$, such that:

$$\Pr\left[\begin{array}{l} cpk, csk \leftarrow C.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}) \\ \vec{m}_0, \vec{m}_1, \text{state} \leftarrow \mathcal{A}_1(1^\lambda, cpk, z) \\ b_i \xleftarrow{\$} \{0,1\}, \forall i \in [j] \\ \vec{m}^*, t_0 \leftarrow C.\mathsf{Delegate}_{\mathsf{DTLP}}(\vec{m}_b, cpk, csk) \\ adr_{\mathcal{SC}} \leftarrow \mathcal{S}.\mathsf{Delegate}_{\mathsf{DTLP}}(\mathsf{CEDG}, ToC, S, \vec{\Delta}, \\ \qquad aux, adr_{\mathcal{HS}}, t_0, \Upsilon, \overrightarrow{coins}) \\ pk, sk \leftarrow \mathcal{HC}.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}, S, z) \\ \hat{p} \leftarrow \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\vec{m}^*, cpk, pk, sk, t_0) \\ \vec{s}, q \leftarrow \mathsf{Solve}_{\mathsf{DTLP}}(\vec{\Psi}, aux, pk, \vec{p}, adr_{\mathcal{SC}}, \overrightarrow{coins}', \\ \qquad \overrightarrow{coins}) \\ \hline b', k \leftarrow \mathcal{A}_1(pk, cpk, \vec{m}_0, \vec{m}_1, \vec{m}^*, \text{state}, t_0, \mathsf{CEDG}, \\ \qquad ToC, aux, \vec{\Psi}, \vec{\Delta}, \vec{s}, \hat{p}, \overrightarrow{coins}, adr_{\mathcal{HS}}, adr_{\mathcal{SC}}) \\ \text{s.t. } b' = b_k \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

where $\vec{m}_0 = [m_{0,1}, \ldots, m_{0,z}]$, $\vec{m}_1 = [m_{1,1}, \ldots, m_{1,z}]$, $\vec{m}_b = [m_{b_1,1}, \ldots, m_{b_z,z}]$.

(2) For any two randomized algorithms $\mathcal{A} \coloneqq (\mathcal{A}_2, \mathcal{A}_3)$, where $\mathcal{A}_2$ runs in time $O(poly(\sum_{i=1}^{j} \bar{\Delta}_i, \lambda))$ and $\mathcal{A}_3$ runs in time $\delta(\sum_{i=1}^{j} \bar{\Delta}_i) < \sum_{i=1}^{j} \bar{\Delta}_i$ using at most $\xi(\max(\bar{\Delta}_1, \ldots, \bar{\Delta}_j))$ parallel processors, there is a negligible function $\mu(\cdot)$, such that:

$$\Pr\left[\begin{array}{l} cpk, csk \leftarrow C.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}) \\ \vec{m}_0^*, \vec{m}_1^*, \text{state} \leftarrow \mathcal{A}_2(1^\lambda, cpk, csk, z) \\ adr_{\mathcal{SC}} \leftarrow \mathcal{S}.\mathsf{Delegate}_{\mathsf{DTLP}}(\mathsf{CEDG}, ToC, \\ \qquad S, \vec{\Delta}, aux, adr_{\mathcal{HS}}, t_0, \Upsilon, \overrightarrow{coins}) \\ pk, sk \leftarrow \mathcal{HC}.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}, S, z) \\ b_i \xleftarrow{\$} \{0,1\}, \forall i \in [j] \\ \hat{p} \leftarrow \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\vec{m}_b^*, cpk, pk, sk, t_0) \\ \hline b', k \leftarrow \mathcal{A}_3(pk, cpk, \hat{p}, \text{state}, \vec{\Psi}, \vec{\Delta}, adr_{\mathcal{HS}}, \\ \qquad adr_{\mathcal{SC}}, aux, \mathsf{CEDG}, ToC, t_0) \\ \text{s.t. } b' = b_k \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

where $\vec{m}_0^* = [m_{0,1}^*, \ldots, m_{0,j}^*]$, $\vec{m}_0^* = [m_{1,1}^*, \ldots, m_{1,j}^*]$, $\vec{m}_b^* = [m_{b_1,1}^*, \ldots, m_{b_j,j}^*]$.

Intuitively, solution validity requires that a prover cannot persuade a verifier (1) to accept a solution that is not equal to the encoded solution m_j^* or (2) to accept a proof that has been registered after the deadline, except for a probability negligible in the security parameter.

Definition 10 (Solution-Validity)**.** A D-TLP preserves a solution validity, if for all λ and $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, any number of puzzles: $z \geq 1$, all PPT adversaries $\mathcal{A} \coloneqq (\mathcal{A}_1, \mathcal{A}_2)$ that run in time $O(poly(\sum_{i=1}^{z} \bar{\Delta}_i, \lambda))$ there is a negligible function $\mu(\cdot)$, such that:

$$\Pr\left[\begin{array}{l} cpk, csk \leftarrow C.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}) \\ \vec{m}, \vec{m}^*, \text{state} \leftarrow \mathcal{A}_1(1^\lambda, pk, \vec{\Delta}, z) \\ adr_{\mathcal{SC}} \leftarrow \mathcal{S}.\mathsf{Delegate}_{\mathsf{DTLP}}(\mathsf{CEDG}, ToC, S, \vec{\Delta}, aux, \\ \qquad adr_{\mathcal{HS}}, t_0, \Upsilon, \overrightarrow{coins}) \\ pk, sk \leftarrow \mathcal{HC}.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}, S, z) \\ \hat{p} \leftarrow \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\vec{m}^*, cpk, pk, sk, t_0) \\ \vec{s}, q \leftarrow \mathsf{Solve}_{\mathsf{DTLP}}(\vec{\Psi}, aux, pk, \vec{p}, adr_{\mathcal{SC}}, \overrightarrow{coins}', \overrightarrow{coins}) \\ \hline j, \pi', m_j^*, t_j, t_j' \leftarrow \mathcal{A}_2(pk, cpk, \vec{s}, \hat{p}, \text{state}, csk, \vec{\Psi}, \vec{\Delta}, \\ \qquad aux, \overrightarrow{coins}, adr_{\mathcal{SC}}, adr_{\mathcal{HS}}, \mathsf{CEDG}, ToC, t_0) \\ \text{s.t. } (con_1 \wedge con_2 \wedge con_3) \vee (\neg con_1 \wedge con_4 \wedge con_5) \end{array}\right] \leq \mu(\lambda)$$

where $\vec{m} = [m_1, \ldots, m_z]$, and $g_j \in \vec{g} \in \hat{p}$. Each condition con_i is defined as follows:

- con_1: j-th solution delivered on time: $t_j - t_0 \leq \sum_{i=1}^{j} (\bar{\Delta}_i + \Psi_i + \Upsilon)$.
- con_2: generates invalid proof π': $m_j^* \notin \mathcal{L}$.
- con_3: the invalid proof π' is accepted: $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi', g_j, \Psi_j, t_j, t_0, \vec{\Delta}, \Upsilon) = 1$.
- con_4: generates valid proof π_j: $m_j^* \in \mathcal{L}$.
- con_5: a valid proof is accepted, despite late delivery: $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j', t_0, \vec{\Delta}, \Upsilon) = 1$.

Informally, fair payment states that the verifier pays the prover if and only if the verifier accepts the proof.

Definition 11 (Fair Payment)**.** A D-TLP supports fair payment, if $\forall j \in [z]$ and any PPT adversary $\mathcal{A}$, there is a negligible function $\mu(\cdot)$, such that:

$$\Pr\left[\begin{array}{l} cpk, csk \leftarrow C.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}) \\ \vec{m}, \text{state} \leftarrow \mathcal{A}(1^\lambda, cpk, z) \\ \vec{m}^*, t_0 \leftarrow \mathsf{Delegate}_{\mathsf{DTLP}}(\vec{m}, cpk, csk) \\ adr_{\mathcal{SC}} \leftarrow \mathcal{S}.\mathsf{Delegate}_{\mathsf{DTLP}}(\mathsf{CEDG}, ToC, S, \vec{\Delta}, aux, \\ \qquad adr_{\mathcal{HS}}, t_0, \Upsilon, \overrightarrow{coins}) \\ pk, sk \leftarrow \mathcal{HC}.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}, S, z) \\ \hat{p} \leftarrow \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\vec{m}^*, cpk, pk, sk, t_0) \\ s_j, t_j, \pi_j \leftarrow \mathcal{A}(cpk, \vec{m}, \text{state}, t_0, \mathsf{CEDG}, ToC, aux, \\ \qquad \vec{\Psi}, \vec{\Delta}, \overrightarrow{coins}, adr_{\mathcal{SC}}, adr_{\mathcal{HS}}, pk, \vec{p}) \\ v_j \leftarrow \mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j, t_0, \vec{\Delta}, \Upsilon) \\ u_j \leftarrow \mathsf{Pay}_{\mathsf{DTLP}}(v_j, adr_{\mathcal{HS}}, \overrightarrow{coins}, j) \\ \hline v_j \neq u_j \end{array}\right] \leq \mu(\lambda)$$

Definition 12 (Security)**.** A D-TLP is secure if it satisfies solution privacy, solution-validity, and fair payment, w.r.t. definitions 9, 10, and 11 respectively.

4.2 Efficient Delegated Time-locked Puzzle

At a high level, ED-TLP operates as follows. During the setup, C encrypts all the plaintext solutions using symmetric-key encryption with key csk, which is sent to $\mathcal{S}$. In turn, $\mathcal{S}$ picks a $\mathcal{HS}$ and determines the extra time it needs to find each solution using $\mathsf{CEDG}(\cdot)$. Next, $\mathcal{S}$ constructs a smart contract $\mathcal{SC}$ in which it specifies the expected delivery time for each solution. It deploys $\mathcal{SC}$ to the blockchain and deposits enough coins for z valid solutions.

C sends the ciphertexts to $\mathcal{HC}$ who (1) generates all required secret and public keys on C's behalf and (2) builds puzzles on the

ciphertexts (instead of the plaintext solutions in GC-TLP). It sends all puzzles to $\mathcal{HS}$ who checks the deposit and the $\mathcal{SC}$ parameters.

If $\mathcal{HS}$ agrees to proceed, it solves each puzzle and generates a proof of the solutions' correctness. It sends the solution and proof to $\mathcal{SC}$ who checks if the solution-proof pair has been delivered on time and the solution is valid with the help of the proof. If the two checks pass, $\mathcal{SC}$ sends a portion of the deposit to $\mathcal{HS}$. Given a valid encrypted solution stored in $\mathcal{SC}$ and a secret key csk, $\mathcal{S}$ decrypts the ciphertext to retrieve a plaintext solution. Figure 4, in Appendix F, outlines the interaction between the parties in ED-TLP. Below, we present ED-TLP in detail.

(1) $C.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}) \rightarrow (cpk, csk)$. Involves C.
   (a) generate a secret key for symmetric-key encryption: $\mathsf{SKE.keyGen}(1^\lambda) \rightarrow csk$.
   (b) set $cpk = \vec{\Delta}$ and send (cpk, csk) to $\mathcal{S}$.
(2) $C.\mathsf{Delegate}_{\mathsf{DTLP}}(\vec{m}, csk) \rightarrow (\vec{m}^*, t_0)$. Involves C.
   (a) encrypt each plaintext solution in vector $\vec{m}$: $\mathsf{SKE.Enc}(csk, m_i) \rightarrow m_i^*, \forall i \in [z]$.
   (b) send t_0 and $\vec{m}^* = [m_1^*, .., m_z^*]$ to $\mathcal{HC}$ and t_0 to $\mathcal{S}$.
(3) $\mathcal{S}.\mathsf{Delegate}_{\mathsf{DTLP}}(\mathsf{CEDG}, ToC, S, \vec{\Delta}, aux, adr_{\mathcal{HS}}, t_0, \Upsilon, \overrightarrow{coins}) \rightarrow adr_{\mathcal{SC}}$. Involves $\mathcal{S}$.
   (a) determine the extra delay Ψ_j that $\mathcal{HS}$ needs to find the j-th solution, $\forall j \in [z]$, by calling $\mathsf{CEDG}(ToC, S, \bar{\Delta}_j, aux) \rightarrow \Psi_j$.
   (b) construct a smart contract $\mathcal{SC}$, deploy $\mathcal{SC}$ into the blockchain, and deposit $coins = \sum_{j=1}^{z} coins_j$ amount of coins into $\mathcal{SC}$, where $coins_j \in \overrightarrow{coins}$. Let $adr_{\mathcal{SC}}$ be the address of the deployed $\mathcal{SC}$.
   (c) set the delivery time for the j-th solution as $T_j = t_0 + \sum_{i=1}^{j}(\bar{\Delta}_i + \Psi_i + \Upsilon), \forall j \in [z]$.
   (d) register $\vec{T} = [T_1, \ldots, T_z]$ and $adr_{\mathcal{HS}}$ in $\mathcal{SC}$.
   (e) send $adr_{\mathcal{SC}}$ to $\mathcal{HC}$ and $\mathcal{HS}$.
(4) $\mathcal{HC}.\mathsf{Setup}_{\mathsf{DTLP}}(1^\lambda, \vec{\Delta}, S, z) \rightarrow (pk, sk)$. Involves $\mathcal{HC}$.
   - call $\mathsf{Setup}_{\mathsf{GM-TLP}}(1^\lambda, \vec{\Delta}, S, z) \rightarrow (pk, sk)$, to generate public and private parameters for z puzzles.
(5) $\mathsf{GenPuzzle}_{\mathsf{DTLP}}(\vec{m}^*, pk, sk, t_0) \rightarrow \hat{p}$. Involves $\mathcal{HC}$.
   (a) call $\mathsf{GenPuzzle}_{\mathsf{DTLP}}(\vec{m}^*, pk, sk) \rightarrow \hat{p}$, to generate z puzzles and their commitments. Recall, $\hat{p} := (\vec{p}, \vec{g})$, where $\vec{p}$ is a vector of puzzles and $\vec{g}$ is a vector of commitments.
   (b) at time t_0, send $\vec{p}$ to $\mathcal{HS}$ and $\vec{g}$ to $\mathcal{SC}$.
(6) $\mathsf{Solve}_{\mathsf{DTLP}}(\vec{\Psi}, aux, pk, \vec{p}, adr_{\mathcal{SC}}, \overrightarrow{coins}', \overrightarrow{coins}) \rightarrow (\vec{s}, q)$. Involves $\mathcal{HS}$.
   (a) check if the deposit is sufficient ($coins_j \geq coins'_j$) and elements of $\vec{\Psi}$ are large enough. If not, set $q = 0$ and halt.
   (b) call $\mathsf{Solve}_{\mathsf{GM-TLP}}(pk, \vec{p}) \rightarrow (\vec{s}, \vec{m}^*)$, to solve z puzzles.
(7) $\mathsf{Prove}_{\mathsf{DTLP}}(pk, s_j) \rightarrow \pi_j$. Involves $\mathcal{HS}$.
   - call $\mathsf{Prove}_{\mathsf{GM-TLP}}(pk, s_j) \rightarrow \pi_j$, to generate a proof, upon discovering a solution $s_j \in \vec{s}$.
(8) $\mathsf{Register}_{\mathsf{DTLP}}(s_j, \pi_j, adr_{\mathcal{SC}}) \rightarrow t_j$. Involves $\mathcal{HS}$.
   - send the j-th encoded message m_j^* (where $m_j^* \in s_j$) and its proof π_j to $\mathcal{SC}$, to be registered in $\mathcal{SC}$ by time t_j.
(9) $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j, t_0, \vec{\Delta}, \Upsilon) \rightarrow v_j$. Involves $\mathcal{SC}$.
   (a) if this is the first time it is invoked (when $j = 1$), set two vectors $\vec{v} = [v_1, \ldots, v_z]$ and $\vec{u} = [u_1, \ldots, u_z]$ whose elements are initially set to empty ϵ.
   (b) read the content of $\mathcal{SC}$ and check whether (s_j, π_j) was delivered to $\mathcal{SC}$ on time: $t_j \leq T_j = t_0 + \sum_{i=1}^{j}(\bar{\Delta}_i + \Psi_i + \Upsilon)$
   (c) call $\mathsf{Verify}_{\mathsf{GM-TLP}}(pk, j, m_j^*, \pi_j, g_j) \rightarrow v'_j \in \{0, 1\}$, to check a proof's validity, where $g_j \in \vec{g}$.
   (d) set $v_j = 1$, if both checks in steps 9b and 9c pass; set $v_j = 0$, otherwise.
(10) $\mathsf{Pay}_{\mathsf{DTLP}}(v_j, adr_{\mathcal{HS}}, \overrightarrow{coins}, j) \rightarrow u_j$. Involves $\mathcal{SC}$. Invoked either by C or $\mathcal{S}$.
   - if $v_j = 1$, then:
     (a) if $u_j \neq 1$, for j-th puzzle, send $coins_j$ coins to $\mathcal{HS}$, where $coins_j \in \overrightarrow{coins}$.
     (b) set $u_j = 1$ to ensure $\mathcal{HS}$ will not be paid multiple times for the same solution.
   - if $v_j = 0$, send $coins_j$ coins back to $\mathcal{S}$ and set $u_j = 0$.
(11) $\mathsf{Retrieve}_{\mathsf{DTLP}}(pk, csk, m_j^*) \rightarrow m_j$. Involves $\mathcal{S}$.
   (a) read m_j^* from $\mathcal{SC}$.
   (b) locally decrypt m_j^*: $\mathsf{SKE.Dec}(csk, m_j^*) \rightarrow m_j$.

In the ED-TLP (and the definition of D-TLP) we assumed $\mathcal{HS}$ is a rational adversary, to ensure only the timely delivery of the solution through our incentive mechanism. Nevertheless, it is crucial to note that the privacy of the scheme remains intact even in the presence of a fully malicious $\mathcal{HS}$.

Theorem 2. *If the symmetric-key encryption meets IND-CPA, GC-TLP is secure (w.r.t. Definition 6), the blockchain is secure (i.e., it meets persistence and liveness properties [22], and the underlying signature satisfies "existential unforgeability under chosen message attacks") and $\mathcal{SC}$'s correctness holds, then ED-TLP is secure, w.r.t. Definition 12.*

Appendix H presents the proof of Theorem 2.

4.2.1 Satisfying the Primary Features. Let us explain how ED-TLP satisfies the primary properties.

- *Varied Size Time Intervals and Efficiently Handling Multiple Puzzles.* ED-TLP uses GC-TLP in a block-box manner, therefore, it inherits these features.
- *Privacy.* To ensure plaintext solutions' privacy, C encrypts all plaintext solutions using symmetric-key encryption and asks $\mathcal{HC}$ to treat the ciphertexts as puzzles' solutions. C sends the secret key of the encryption only to $\mathcal{S}$. This approach protects the privacy of plaintext messages from both $\mathcal{HC}$ and $\mathcal{HS}$.
- *Exact-time Solutions Recovery.* In ED-TLP, given the exact computational power of $\mathcal{HS}$, $\mathcal{S}$ uses $\mathsf{CEDG}(\cdot)$ to determine the extra time $\mathcal{HS}$ needs for each solution. Thus, instead of assuming that $\mathcal{HS}$ possesses computing resources to perform the maximum number of squarings S per second and find a solution on time, $\mathcal{S}$ considers the available resources of $\mathcal{HS}$.
- *Timely Delivery of Solutions and Fair Payment.* To ensure the timely *delivery* of a solution, $\mathcal{S}$ constructs a smart contract $\mathcal{SC}$ and specifies by when each solution must be delivered, based on $\mathsf{CEDG}(\cdot)$ output, and deposits in $\mathcal{SC}$ a certain amount of coins. $\mathcal{S}$ requires $\mathcal{SC}$ to check if each (encrypted) solution is valid and delivered on time and, if the two checks pass, pay $\mathcal{HS}$. This gives $\mathcal{HS}$ the assurance it will get paid if it provides a valid solution on time. The hash-based $\mathcal{SC}$-side verification imposes a low computation cost.

5 Evaluation

We conducted an in-depth evaluation of GC-TLP and ED-TLP against (1) the original RSA-based TLP of Rivest et al. [39] because it serves as the foundation for numerous TLPs and VDFs, and (2) the C-TLP of Abadi and Kiayias [3] because it is the most efficient multi-instance puzzle that supports efficient verification. We analyze the features of all protocols and compare their overheads asymptotically and concretely.

5.1 Features

Table 1 compares the four schemes' features. Figure 3 in Appendix F further illustrates differences among these four schemes concerning the support for multi-puzzle and varied-size time intervals.

The features of our protocols were motivated by limitations in the existing literature, illustrated by the multi-client example in Section 1. Our scheme enables clients with different puzzles to combine them in a single chain the server can solve sequentially. We briefly explain how it can be done in a simple two-client case. Consider clients C_1 and C_2 who encode messages m_1 and m_2 in puzzles $\hat{p}_1$ and $\hat{p}_2$ with intervals Δ_1 and Δ_2, respectively, then send them to a server $\mathcal{S}$ at the same time. Assume $\Delta_1 < \Delta_2$. After $\mathcal{S}$ receives $\hat{p}_1$ and $\hat{p}_2$, it can start working on $\hat{p}_1$ immediately, but also asks (or incentivizes) the clients to chain their messages. To do this, C_2, with a longer interval, delegates its messages to C_1. Acting as $\mathcal{HC}$, C_1 uses its existing keys to extend $\hat{p}_1$ with a puzzle containing m_2 set for interval of $\bar{\Delta}_2 = \Delta_2 - \Delta_1$, which is sent to $\mathcal{S}$. After solving the initial puzzle in $\hat{p}_1$, $\mathcal{S}$ can start solving the new puzzle. It is thus able to recover both m_1 and m_2 in time Δ_2 using only a single CPU, doing equivalent work to the work $\hat{p}_2$ initially required. This *scales to any number of clients*, as puzzles chains can be arbitrarily long. Alternatively, clients can delegate puzzle generation to a common helper $\mathcal{HC}$ from the start. In addition, if $\mathcal{S}$ is unable to solve the puzzle chain in useful time, it can use server delegation to securely offload its work, with fair payment and timely delivery of solutions

The protocols are modular, so the main features of ED-TLP: variable intervals, client delegation, and server delegation can be used separately. For example, if C_1 and C_2 are offline after sending their puzzles and cannot chain them, $\mathcal{S}$ can still delegate the work and obtain the solutions on time (at a higher cost).

5.2 Asymptotic Cost Analysis

We provide a brief asymptotic cost analysis of the protocol. Table 2 summarizes the cost comparison between the four schemes. In our cost evaluation, for the sake of simplicity, we do not include the output of $\mathsf{CEDG}(\cdot)$, as the output is a fixed value and remains the same for all the schemes we analyze in this section.

5.2.1 Computation Cost. As Table 2 demonstrates, ED-TLP has the lowest client-side setup and puzzle generation cost and negligible computation cost for the server. Schemes that support verification: C-TLP, GC-TLP, and ED-TLP, all have $O(z)$ verification cost.

Thus, in ED-TLP, the client does not need to perform any modular exponentiation. However, in the rest of the schemes, the client has to perform $O(z)$ modular exponentiation in the setup and/or puzzle generation phases. Furthermore, in ED-TLP, the server does not engage in any modular exponentiation, in contrast to other schemes

Table 1: Differences in features between: (1) the original TLP of Rivest et al. [39], (2) the C-TLP of Abadi and Kiayias [3] (3) our GC-TLP, and (4) our ED-TLP.

	Scheme			
Features	TLP	C-TLP	GC-TLP	ED-TLP
Multi-puzzle	✗	✓	✓	✓
Varied-size	✓	✗	✓	✓
Verification	✗	✓	✓	✓
Delegation	✗	✗	✗	✓
Exact-time Solution Recovery	✗	✗	✗	✓
Fair Payment	✗	✗	✗	✓

that require the server's involvement in performing $O(S \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$ exponentiation in GC-TLP, $O(z \cdot S \cdot \Delta)$ exponentiation within C-TLP, and $O(S \cdot \sum_{i=1}^{z} \Delta_i)$ exponentiation in the original TLP.

5.2.2 ED-TLP's Added Cost. ED-TLP imposes an additional cost, $O(z)$, during the retrieve phase, distinguishing it from the other schemes that do not incur such a cost. Note that the retrieve phase exclusively involves invocations of the symmetric key encryption scheme, which imposes low computational costs.

5.2.3 Communication Cost. As depicted in Table 2, for the z-puzzle setting, the communication complexity for all four schemes is $O(z)$. However, in ED-TLP, both C and $\mathcal{S}$ only transmit random values and outputs of symmetric-key encryption, which are shorter than the messages in the other three schemes. Thus, ED-TLP exhibits the lowest client-side and server-side concrete communication costs.

5.3 Concrete Run-time Analysis

To complement the asymptotic analysis, we implemented ED-TLP and GC-TLP, as well as C-TLP [3] and the Rivest et al. TLP [39] in Python 3 using the high-performance multiple precision integer library GMP [24] via the gmpy2 [25] library. The implementation strictly adhere to the protocol descriptions for TLP, C-TLP, and GC-TLP. Smart contract functionality was implemented using Solidity. For protocols requiring a commitment scheme, we used a SHA512 hash-based commitment with 128-bit witness values.

5.3.1 Setting the Parameters. The evaluation required a concrete value of S, the number of repeated squaring a CPU can perform. We benchmarked 10 different commodity computing machines and consumer devices. The results are presented in Figure 1.

5.3.2 Run-time Comparison. For the concrete analysis, we ran the four protocols on a 2021 MacBook Pro equipped with an Apple M1 Pro CPU. As we use execution time as a proxy for computation costs, for benchmarking we used a mock smart contract running locally, without an underlying blockchain or network delay. Table 3 lists the running time of each protocol step and totals per actor.

The most salient observation is $\mathcal{S}$ can offload 100% of its workload to $\mathcal{HS}$ with negligible computational overhead. Although ED-TLP adds the Delegate and Retrieve steps, their CPU costs for $\mathcal{S}$ are negligible per puzzle. In our benchmarks, both $\mathcal{S}$ and $\mathcal{HS}$ were run on the same hardware, but as Figure 1 shows, delegating to specialized hardware can result in large gains, especially

Table 2: Asymptotic costs comparison. In the table, z is the total number of puzzles. Δ and Δ_i are time intervals in C-TLP and TLP respectively. $\bar{\Delta}_i$ is a time interval in GC-TLP and ED-TLP with $\Delta_i = \sum_{j=1}^{i} \bar{\Delta}_j$.

Scheme	Operation	Algorithms Complexity										Total Complexity	
		Setup		Delegate		GenPuzzle		Solve		Verify	Retrieve	Comp.	Comm.
		$\mathcal{C}$	$\mathcal{HC}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{C}$	$\mathcal{HC}$	$\mathcal{S}$	$\mathcal{HS}$				
ED-TLP	Exp.	–	$O(z)$	–	–	–	$O(z)$	–	$O(S \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$	–	–	$O(S \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$	$O(z)$
	Add./Mul.	–	–	–	$O(z)$	–	$O(z)$	–	$O(z)$	–	–		
	Hash	–	–	–	–	–	$O(z)$	–	–	$O(z)$	–		
	Sym. Enc	–	–	$O(z)$	–	$O(z)$	$O(z)$	–	$O(z)$	–	$O(z)$		
GC-TLP	Exp.	$O(z)$				$O(z)$		$O(S \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$		–		$O(S \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$	$O(z)$
	Add./Mul.	–				$O(z)$		$O(z)$		–			
	Hash	–				$O(z)$		–		$O(z)$			
	Sym. Enc	–				$O(z)$		$O(z)$		–			
C-TLP	Exp.	1				$O(z)$		$O(z \cdot S \cdot \Delta)$		–		$O(z \cdot S \cdot \Delta)$	$O(z)$
	Add./Mul.	–				$O(z)$		$O(z)$		–			
	Hash	–				$O(z)$		–		$O(z)$			
	Sym. Enc	–				$O(z)$		$O(z)$		–			
TLP	Exp.	$O(z)$				$O(z)$		$O(S \cdot \sum_{i=1}^{z} \Delta_i)$				$O(S \cdot \sum_{i=1}^{z} \Delta_i)$	$O(z)$
	Add./Mul.	–				$O(z)$		$O(z)$					
	Hash	–				–		–					
	Sym. Enc	–				$O(z)$		$O(z)$					

Table 3: Run time using 2048-bit moduli and 1 second intervals (or equivalent cumulative durations in TLP).

Scheme	Number of puzzles	Algorithms Run Time (in seconds)														
		Setup		Delegate		GenPuzzle		Solve		Verify	Retrieve	Total time				
		$\mathcal{C}$	$\mathcal{HC}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{C}$	$\mathcal{HC}$	$\mathcal{S}$	$\mathcal{HS}$			$\mathcal{C}$	$\mathcal{HC}$	$\mathcal{S}$	$\mathcal{HS}$	Overall
ED-TLP	10	<0.01	0.03	<0.01	<0.01		0.03		10.67	<0.01	<0.01	<0.01	0.06	<0.01	10.67	10.72
	100	<0.01	0.03	<0.01	<0.01		0.19		107.50	<0.01	<0.01	<0.01	0.22	<0.01	107.50	107.72
	1000	<0.01	0.05	0.02	<0.01		1.80		1071.01	<0.01	0.02	0.02	1.85	0.02	1071.01	1072.88
	10000	<0.01	0.23	0.13	<0.01		18.12		10669.92	0.03	0.13	0.13	18.12	0.13	10669.92	10688.54
GC-TLP	10	0.03				0.03		10.71		<0.01		0.05		10.71		10.76
	100	0.03				0.19		107.28		<0.01		0.22		107.28		107.50
	1000	0.05				1.81		1072.96		0.02		1.85		1072.96		1074.81
	10000	0.22				18.07		10689.07		0.02		18.29		10689.07		10707.37
C-TLP	10	0.03				0.02		10.70		<0.01		0.05		10.70		10.75
	100	0.03				0.19		107.09		<0.01		0.21		107.01		107.22
	1000	0.04				1.83		1070.36		0.03		1.86		1070.36		1072.23
	10000	0.14				18.10		10680.29		0.02		18.24		10680.29		10698.54
TLP	10	0.14				0.02		58.50				0.16		58.50		58.65
	100	1.86				0.19		5075.49				2.05		5075.49		5077.53
	1000	19.78				1.80		N/A[a]				21.58		N/A[a]		N/A[a]
	10000	181.77				17.15		N/A[b]				198.92		N/A[b]		N/A[b]

[a] Estimated total runtime: 5.8 days [b] Estimated total runtime: 1.5 years

for an underpowered server. Client-side delegation also has low overheads. Despite its much lower computational complexity, delegating puzzle generation results in computational savings of 99% for the client. As puzzle generation can be parallelized, generating large numbers of puzzles can benefit from delegation to a $\mathcal{HC}$ with a high number of powerful CPU cores. For example, 10000 puzzles

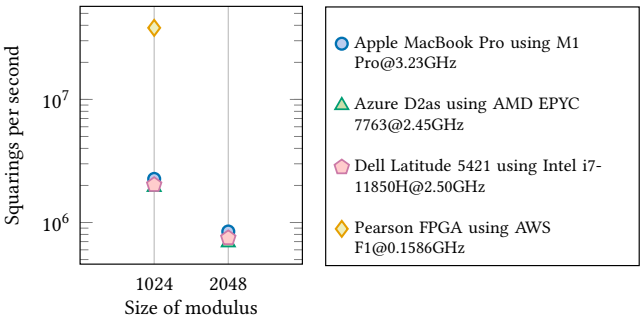


Figure 1: Log plot of the squaring capabilities of consumer and commodity cloud hardware for 1024-bit and 2048-bit RSA groups, compared to an FPGA-based solution for 1024-bit moduli as reported in [44]. General purpose CPUs have similar capabilities, but the FGPA solution is an order of magnitude faster.

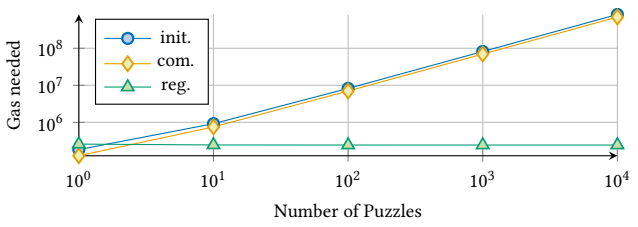


Figure 2: Gas amount needed to execute the smart contract functions (initialization, committing to the puzzles, and registering solutions) for different puzzle instances

only require 2.48s on 8 cores, compared to 18.12s on a single core. To support variable durations, GC-TLP has a more complex setup compared to C-TLP (Table 3). In practice, the effect is minor per puzzle, with a total increase of 0.08s for 10000 puzzles.

5.4 ED-TLP Gas Cost Estimates

We implemented the ED-TLP smart contract in Solidity and estimated its costs on the Ethereum [46] Mainnet and two Ethereum-based level 2 networks: Arbitrum One [26] and Polygon PoS [8]. Although the protocol is agnostic of the underlying blockchain, we chose Solidity and Ethereum-based chains due to their prevalence. Figure 2 shows the gas cost scaling with corresponding USD costs list in Table 4. Initializing the contract, executed when $\mathcal{S}$ deploys the contract in $\mathcal{S}.\text{Delegate}_{\text{DTLP}}$, and committing to the puzzles, executed by $\mathcal{HC}$ during $\text{GenPuzzle}_{\text{DTLP}}$, are $O(n)$, which the costs reflect. There is, however, a large constant gas cost inherent to the deployment of the contract. Registering solutions is asymptotically constant. The minor variations seen in concrete gas cost are due to different parameters.

5.5 Further Discussion

Delegation with fair payment incentives developing and making available specialized hardware for squaring-based puzzles and VDFs.

Table 4: Cost comparison of initializing the contract, committing to the puzzle chain, and registering a solution across different networks.

Number of puzzles	Algorithms' Gas Cost (in USD*)								
	Etherium			Arbitrum			Polygon		
	init.	com.	reg.	init.	com.	reg.	init.	com.	reg.
1	0.98	0.67	1.38	0.05	0.04	0.07	0.01	0.01	0.01
10	4.89	3.99	1.31	0.25	0.20	0.07	0.02	0.01	0.01
100	43.72	37.17	1.30	2.19	1.86	0.07	0.11	0.10	0.01
1000	433.44	369.10	1.30	21.67	18.46	0.07	1.08	0.92	0.01
10000	4331.40	3690.14	1.30	216.54	184.48	0.07	10.70	9.12	0.01

* Transaction USD price on 20 Aug 2024, 11:28 UTC using www.cryptoneur.xyz/en/gas-fees-calculator.

This levels the playing field between parties with different computational capabilities and addresses one of the fundamental issues of time-based cryptography: the heterogeneity of solvers which otherwise causes the solve time to vary substantially. Figure 1 shows that commodity cloud and high-end consumer hardware are comparable in solving speed[1]. In contrast, FPGA-based squaring requires only 4 cycles per operation [44]. Therefore, speed-up by a factor of 15 can be achieved by delegating solving to a helper using the FPGA described in [44] compared to the fastest general-purpose CPU benchmarked (Apple M1).

As the overhead of ED-TLP is low, delegation and the choice of helper become economics decisions. Even if $\mathcal{C}$ and $\mathcal{S}$ have the capabilities to run GC-TLP on their own, they can still consider ED-TLP if the speed-up offered by delegation provides enough value compared to the monetary cost of delegation. Helpers with different capabilities and costs can be considered. A slower helper with adequate performance for the needs of $\mathcal{S}$ at a lower cost may be preferred over a costlier, faster helper. In addition, if faster helpers are busy, slower helpers may provide the solution sooner overall, which an appropriate instantiation of CEDG can capture.

Setting the value of S, the number of squarings per second the most capable server can perform, is currently a challenge as numbers are not readily available. If a market for delegation of repeated squaring develops, puzzle generation can be calibrate to the strongest offerings on the market to ensure that puzzles cannot be solved before the intended time. Additionally, by allowing $\mathcal{C}$ to extend the chain, this means new puzzles in a chain can be generated using updated information about the strongest servers.

6 Applications of (E)D-TLP

We briefly discuss applications of our (E)D-TLP within the contexts of VDF and proofs of data storage

6.1 Delegated VDF

Verifiable delay functions (VDFs) let a prover provide a publicly verifiable proof that it has performed a predetermined number of sequential computations [10, 11]. As VDF schemes have been

[1]This may be explained by the focus on highly parallelizable tasks, especially for cloud computing; puzzle solving is inherently sequential.

built on TLPs [36, 45], the ideas behind D-TLP can also directly contribute to VDF scalability. A typical VDF involves algorithms: (1) $\mathsf{Setup}(\cdot)$ that returns system parameters, (2) $\mathsf{Eval}(\cdot)$ that returns a value y and proof that the value has been constructed correctly, and (3) $\mathsf{Verify}(\cdot)$ that decides if a proof is valid. By definition, $\mathsf{Eval}(\cdot)$ demands considerable computational resources to compute a predefined number of sequential steps until it obtains the final result, similarly to TLP.

However, to date, the scalability of VDF schemes has been overlooked. The literature has primarily concentrated on the specific case where the solver deals with only a single instance of VDF, ignoring the generic multi-instance VDF cases. Using a current VDF scheme, if a resource-constrained party (lacking a sufficient level of computation power) has to run multiple VDFs within a period, it must defer some instances of $\mathsf{Eval}(\cdot)$ until others have completed. However, this leads to a significant delay in computing all VDFs instances. This becomes particularly problematic in a competitive environment, where different parties need to submit the output of VDF ($\mathsf{Eval}(\cdot)$) instances to a smart contract on time to receive a reward, e.g., in [42]. By adapting D-TLP, VDFs can be delegated with fair payment for timely delivery, letting systems scale as the number of VDF instances increases beyond their immediate capabilities.

6.2 Scalable Proof of Storage

There are scenarios in which a client desires a server to acquire distinct "random challenges" at various points in time within a specified period, without the client's involvement during that timeframe. Such challenges would enable the server to generate specific proofs, including, but not limited to, demonstrating the *continuous availability of services*, such as proofs of data storage [3, 5, 47]. In scenarios where a client outsources its file to a storage server and aims to ensure the file's integrity and accessibility at different intervals, these challenges become crucial. The idea involves the client computing random challenges, encoding them into puzzles, and transmitting them to the server. The server can then solve each puzzle, extract a subset of challenges, and employ them for the relevant proof scheme.

In cases where a (set of) client needs to simultaneously activate multiple instances of the scheme (e.g., invoking multiple instances of proofs of data storage scheme when the client has multiple independent files), thin clients with limited resources can not generate all the puzzles and the storage server with insufficient computational resources may struggle to solve the puzzles on time.

To surmount this challenge in a privacy-preserving manner, our ED-TLP can be employed to (i) enable the client to delegate the generation of puzzles to a more powerful server and (ii) allow the storage server to verifiably delegate the task of solving the puzzles to a resourceful helper. In the multi-client case, the server can ask clients to combine their puzzles into a single chain to further reduce the overall workload.

7 Related Work

Since the introduction of the RSA-based TLP, various variants have been proposed. For a comprehensive survey, we refer readers to a recent survey [31]. Boneh [12] and Garay and Jakobsson [21] have proposed TLPs for the setting where a client can be malicious and needs to prove (in zero-knowledge) to a solver that the correct solution will be recovered after a certain time. Baum et al. [7] have developed a composable TLP that can be defined and proven in the universal composability framework. Malavolta and Thyagarajan [29] and Brakerski et al. [13] proposed the notion of homomorphic TLPs, which let an arbitrary function run over puzzles before they are solved. They use the RSA-based TLP and fully homomorphic encryption, which imposes high overheads. To improve the above homomorphic TLPs, Srinivasan et al. [40] proposed a scheme that supports the unbounded batching of puzzles.

Thyagarajan et al. [42] proposed a blockchain-based scheme that allows a party to delegate the computation of sequential squaring to a set of competing solvers. To date, this is the only scheme that considers verifiably outsourcing sequential squaring for rewards. In this scheme, a client posts the number of squarings required and its related public parameters to a smart contract, alongside a deposit. The solvers are required to propose a solution before a certain time point. This scheme suffers from a security issue that allows colluding solvers to send incorrect results and invalid proofs, but still they are paid without being detected [1]. Also, this scheme does not offer any solution for multi-puzzle settings. It offers no formal definition for delegated time-lock puzzles (they only present a formal definition for delegated squaring) and supports only solver-side delegation. Our (E)D-TLP addresses these limitations.

8 Conclusion

In this work, we introduced a scalable TLP tailored for real-world scenarios involving numerous puzzles, effectively addressing the computational constraints of clients and servers under heavy workloads. Specifically, we proposed the concept of Delegated Time-Lock Puzzle (D-TLP) and developed a novel protocol, ED-TLP, to implement D-TLP, significantly enhancing the scalability of TLPs. This protocol allows a server to offload its workload to a more capable helper with minimal overhead, ensuring timely delivery and fair payment. Delegating tasks to a helper with specialized hardware is particularly advantageous, as it enables significant speed-ups. In a multi-client scenario, ED-TLP optimizes efficiency by allowing clients to merge their puzzles into a single puzzle chain, equivalent in complexity to the longest puzzle.

We implemented ED-TLP, configured its concrete parameters, and evaluated its on-chain and off-chain costs. For the first time, we assessed the performance of state-of-the-art TLPs for handling a large number of puzzles. The results demonstrate that our scheme is both efficient and highly scalable. ED-TLP enables the delegation of 99% of the client workload and 100% of the server workload with minimal overhead, achieving smart contract gas costs as low as 0.2 cents per puzzle. ED-TLP can enhance the scalability of systems reliant on delay-based cryptography.

Acknowledgments

Aydin Abadi was supported in part by REPHRAIN: The National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online, under UKRI grant: EP/V011189/1. Dan Ristea is funded by UK EPSRC grant EP/S022503/1 supporting the CDT in Cybersecurity at UCL. Steven J. Murdoch was supported by REPHRAIN.

References

[1] Aydin Abadi. 2023. Decentralised Repeated Modular Squaring Service Revisited: Attack and Mitigation. Cryptology ePrint Archive, Paper 2023/1347. https://eprint.iacr.org/2023/1347

[2] Aydin Abadi. 2024. Tempora-Fusion: Time-Lock Puzzle with Efficient Verifiable Homomorphic Linear Combination. Cryptology ePrint Archive, Paper 2024/1013. https://eprint.iacr.org/2024/1013

[3] Aydin Abadi and Aggelos Kiayias. 2021. Multi-instance publicly verifiable time-lock puzzle and its applications. In *Financial Cryptography and Data Security – FC*, Vol. 12675. Springer, Virtual Event, 541–559. doi:10.1007/978-3-662-64331-0_28

[4] Amazon. 2023. EC2 F1 instance. https://aws.amazon.com/ec2/instance-types/f1/

[5] Giuseppe Ateniese, Long Chen, Mohammad Etemad, and Qiang Tang. 2020. Proof of Storage-Time: Efficiently Checking Continuous Data Availability. In *Network and Distributed System Security Symposium – NDSS*, Vol. 2. The Internet Society, San Diego, California, USA, 1345–1359. doi:10.14722/ndss.2020.24427

[6] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a Transaction Ledger: A Composable Treatment. In *Advances in Cryptology – CRYPTO*, Vol. 10401. Springer, Santa Barbara, CA, USA, 324–356. doi:10.1007/978-3-319-63688-7_11

[7] Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. 2021. TARDIS: A Foundation of Time-Lock Puzzles in UC. In *Advances in Cryptology – EUROCRYPT*, Vol. 12698. Springer, Zagreb, Croatia, 429–459. doi:10.1007/978-3-030-77883-5_15

[8] Mihailo Bjelic, Sandeep Nailwal, Amit Chaudhary, and Wenxuan Deng. 2017. POL: one token for all polygon chains. https://polygon.technology/papers/pol-whitepaper

[9] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. 1991. Noninteractive Zero-Knowledge. *SIAM J. Comput.* 20, 6 (1991), 1084–1118. doi:10.1137/0220068

[10] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable Delay Functions. In *Advances in Cryptology – CRYPTO*, Vol. 10991. Springer, Santa Barbara, CA, USA, 757–788. doi:10.1007/978-3-319-96884-1_25

[11] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2018. A Survey of Two Verifiable Delay Functions. Cryptology ePrint Archive, Paper 2018/712. https://eprint.iacr.org/2018/712

[12] Moni Boneh, Danand Naor. 2000. Timed Commitments. In *Advances in Cryptology – CRYPTO*. Springer, Berlin, Heidelberg, 236–254. doi:10.1007/3-540-44598-6_15

[13] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. 2019. Leveraging Linear Decryption: Rate-1 Fully-Homomorphic Encryption and Time-Lock Puzzles. In *Theory of Cryptography – TCC*, Vol. 11892. Springer, Nuremberg, Germany, 407–437. doi:10.1007/978-3-030-36033-7_16

[14] Hsing-Chung Chen and Rini Deviani. 2012. A Secure E-Voting System Based on RSA Time-Lock Puzzle Mechanism. In *International Conference on Broadband, Wireless Computing, Communication and Applications*. IEEE, Victoria, BC, Canada, 596–601. doi:10.1109/BWCCA.2012.104

[15] Adam Conner-Simons. 2019. Programmers solve MIT's 20-year-old cryptographic puzzle. https://www.csail.mit.edu/news/programmers-solve-mits-20-year-old-cryptographic-puzzle

[16] Cryptophage. 2019. Cryptophage LCS35 Solver. https://github.com/supranational/lcs35

[17] Cryptophage. 2019. Cryptophage Project. https://web.archive.org/web/20190506050059/http://www.cryptophage.com/ Archived on 2019/05/06.

[18] Cynthia Dwork and Moni Naor. 2000. Zaps and their applications. In *Symposium on Foundations of Computer Science – FOCS*. IEEE Computer Society, Redondo Beach, California, USA, 283–293. doi:10.1109/SFCS.2000.892117

[19] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. 2020. Continuous Verifiable Delay Functions. In *Advances in Cryptology – EUROCRYPT*, Vol. 12107. Springer, Zagreb, Croatia, 125–154. doi:10.1007/978-3-030-45727-3_5

[20] Uriel Feige, Dror Lapidot, and Adi Shamir. 1990. Multiple Non-Interactive Zero Knowledge Proofs Based on a Single Random String (Extended Abstract). In *Symposium on Foundations of Computer Science, Volume I*. IEEE Computer Society, St. Louis, Missouri, USA, 308–317. doi:10.1109/FSCS.1990.89549

[21] Juan A. Garay and Markus Jakobsson. 2002. Timed Release of Standard Digital Signatures. In *Financial Cryptography – FC Revised Papers*, Vol. 2357. Springer, Southampton, Bermuda, 168–182. doi:10.1007/3-540-36504-4_13

[22] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology – EUROCRYPT*, Vol. 9057. Springer, Sofia, Bulgaria, 281–310. doi:10.1007/978-3-662-46803-6_10

[23] Juan A. Garay, Aggelos Kiayias, and Giorgos Panagiotakos. 2019. Blockchains from Non-Idealized Hash Functions. Cryptology ePrint Archive, Paper 2019/315. https://eprint.iacr.org/2019/315

[24] Torbjörn Granlund and the GMP development team. 2023. GNU MP: The GNU Multiple Precision Arithmetic Library. http://gmplib.org/

[25] Case Van Horsen and the GMPY 2 development team. 2023. gmpy2. https://github.com/aleaxit/gmpy

[26] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *USENIX Security Symposium*. USENIX Association, Baltimore, MD, USA, 1353–1370. doi:10.5555/3277203.3277305

[27] Jonathan Katz, Julian Loss, and Jiayu Xu. 2020. On the Security of Time-Lock Puzzles and Timed Commitments. In *Theory of Cryptography – TCC*, Vol. 12552. Springer, Durham, NC, USA, 390–413. doi:10.1007/978-3-030-64381-2_14

[28] Alireza Kavousi, Aydin Abadi, and Philipp Jovanovic. 2024. Timed Secret Sharing. In *Advances in Cryptology – ASIACRYPT*. Springer Nature, Singapore, 129–164. doi:10.1007/978-981-96-0941-3_5

[29] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. 2019. Homomorphic Time-Lock Puzzles and Applications. In *Advances in Cryptology – CRYPTO*, Vol. 11692. Springer, Santa Barbara, CA, USA, 620–649. doi:10.1007/978-3-030-26948-7_22

[30] Timothy C May. 1993. Timed-Release Crypto. https://cypherpunks.venona.com/date/1993/02/msg00129.html

[31] Liam Medley, Angelique Faye Loe, and Elizabeth A Quaglia. 2023. Sok: Delay-based cryptography. In *IEEE Computer Security Foundations Symposium – CSF*. IEEE, Dubrovnik, Croatia, 169–183. doi:10.1109/CSF57540.2023.00028

[32] Microsoft Azure. 2023. Azure naming conventions. https://learn.microsoft.com/en-us/azure/virtual-machines/vm-naming-conventions

[33] Microsoft Azure. 2023. Azure VM sizes. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes

[34] Rafael Pass, Elaine Shi, and Florian Tramèr. 2017. Formal Abstractions for Attested Execution Secure Processors. In *Advances in Cryptology – EUROCRYPT*, Vol. 10210. Springer, Paris, France, 260–289. doi:10.1007/978-3-319-56620-7_10

[35] Torben P. Pedersen. 1991. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology – CRYPTO*, Vol. 576. Springer, Santa Barbara, California, USA, 129–140. doi:10.1007/3-540-46766-1_9

[36] Krzysztof Pietrzak. 2019. Simple Verifiable Delay Functions. In *Innovations in Theoretical Computer Science Conference – ITCS*, Vol. 124. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, San Diego, California, USA, 60:1–60:15. doi:10.4230/LIPICS.ITCS.2019.60

[37] Dan Ristea and Artem Grigor. 2024. TLP_lib. https://github.com/danrr/TLP_lib

[38] Ronald L. Rivest. 1999. Description of the LCS35 Time Capsule Crypto-Puzzle. https://people.csail.mit.edu/rivest/pubs/Riv99b.lcs35-puzzle-description.txt

[39] R. L. Rivest, A. Shamir, and D. A. Wagner. 1996. *Time-lock Puzzles and Timed-release Crypto*. Technical Report. Massachusetts Institute of Technology.

[40] Shravan Srinivasan, Julian Loss, Giulio Malavolta, Kartik Nayak, Charalampos Papamanthou, and Sri Aravinda Krishnan Thyagarajan. 2023. Transparent Batchable Time-lock Puzzles and Applications to Byzantine Consensus. In *Public-Key Cryptography – PKC*, Vol. 13940. Springer, Atlanta, GA, USA, 554–584. doi:10.1007/978-3-031-31368-4_20

[41] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Giulio Malavolta, Nico Döttling, Aniket Kate, and Dominique Schröder. 2020. Verifiable Timed Signatures Made Practical. In *ACM SIGSAC Conference on Computer and Communications Security – CCS*. ACM, Virtual Event, 1733–1750. doi:10.1145/3372297.3417263

[42] Sri Aravinda Krishnan Thyagarajan, Tiantian Gong, Adithya Bhat, Aniket Kate, and Dominique Schröder. 2021. OpenSquare: Decentralized Repeated Modular Squaring Service. In *ACM SIGSAC Conference on Computer and Communications Security – CCS*. ACM, Virtual Event, 3447–3464. doi:10.1145/3460120.3484809

[43] VDF Alliance. 2019. FPGA Competition. https://supranational.atlassian.net/wiki/spaces/VA/pages/36569208/FPGA+Competition

[44] VDF Alliance. 2019. FPGA Competition Round 2 Results. https://github.com/supranational/vdf-fpga-round2-results

[45] Benjamin Wesolowski. 2019. Efficient Verifiable Delay Functions. In *Advances in Cryptology – EUROCRYPT*, Vol. 11478. Springer, Darmstadt, Germany, 379–407. doi:10.1007/978-3-030-17659-4_13

[46] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf

[47] Haiyang Yu, Qi Hu, Zhen Yang, and Huan Liu. 2021. Efficient Continuous Big Data Integrity Checking for Decentralized Storage. *IEEE Transactions on Network Science and Engineering* 8, 2 (2021), 1658–1673. doi:10.1109/TNSE.2021.3068261

A Discussion on Network Delay Parameters

As discussed in [22], liveness states that an honestly generated transaction will eventually be included more than κ blocks deep in an honest party's blockchain. It is parameterized by wait time: u and depth: κ. They can be fixed by setting κ as the minimum depth of a block considered as the blockchain's state (i.e., a part of the blockchain that remains unchanged with a high probability, e.g., $\kappa \geq 6$) and u the waiting time that the transaction gets κ blocks deep.

As shown in [6], there is a slackness in honest parties' view of the blockchain. In particular, there is no guarantee that at any

given time, all honest miners have the same view of the blockchain or even the state. But, there is an upper bound on the slackness, denoted by `WindowSize`, after which all honest parties would have the same view on a certain part of the blockchain state.

This means when an honest party (e.g., the server) propagates its transaction (containing the proof) all honest parties will see it on their chain after at most: $\Upsilon = \texttt{WindowSize} + u$ time period.

B Symmetric-key Encryption Scheme

A symmetric-key encryption scheme consists of three algorithms: (1) $\mathsf{SKE.keyGen}(1^\lambda) \rightarrow k$ is a probabilistic algorithm that outputs a symmetric key k. (2) $\mathsf{SKE.Enc}(k, m) \rightarrow c$ takes as input k and a message m in some message space and outputs a ciphertext c. (3) $\mathsf{SKE.Dec}(k, c) \rightarrow m$ takes as input k and a ciphertext c and outputs a message m.. The correctness requirement is that for all messages m in the message space: $\Pr[\mathsf{SKE.Dec}(k, \mathsf{SKE.Enc}(k, m)) = m : \mathsf{SKE.keyGen}(1^\lambda) \rightarrow k] = 1$.

The symmetric-key encryption scheme satisfies *indistinguishability under chosen-plaintext attacks (IND-CPA)*, if for any PPT adversary $\mathcal{A}$ there is a negligible function $\mu(\cdot)$ for which $\mathcal{A}$ has no more than $\frac{1}{2} + \mu(\lambda)$ probability in winning the following game.

The challenger generates a symmetric key $\mathsf{SKE.keyGen}(1^\lambda) \rightarrow k$. The adversary $\mathcal{A}$ is given access to an encryption oracle $\mathsf{SKE.Enc}(k, \cdot)$ and eventually sends to the challenger a pair of messages m_0, m_1 of equal length. In turn, the challenger chooses a random bit b and provides $\mathcal{A}$ with a ciphertext $\mathsf{SKE.Enc}(k, m_b) \rightarrow c_b$. Upon receiving c_b, $\mathcal{A}$ continues to have access to $\mathsf{SKE.Enc}(k, \cdot)$ and wins if its guess b' is equal to b.

C Sequential and Iterated Functions

Definition 13 ($\Delta, \delta(\Delta)$)-Sequential function)**.** For a function: $\delta(\Delta)$, time parameter: Δ and security parameter: $\lambda = O(\log(|X|))$, $f : X \rightarrow Y$ is a $(\Delta, \delta(\Delta))$-sequential function if the following conditions hold:

- There is an algorithm that for all $x \in X$ evaluates f in parallel time Δ, by using $poly(\log(\Delta), \lambda)$ processors.
- For all adversaries $\mathcal{A}$ which execute in parallel time strictly less than $\delta(\Delta)$ with $poly(\Delta, \lambda)$ processors there is a negligible function $\mu(\cdot)$:

$$\Pr\left[y_{\mathcal{A}} \xleftarrow{\$} \mathcal{A}(\lambda, x), x \xleftarrow{\$} X : y_{\mathcal{A}} = f(x)\right] \leq \mu(\lambda)$$

where $\delta(\Delta) = (1 - \epsilon)\Delta$ and $\epsilon < 1$.

Definition 14 (Iterated Sequential function)**.** Let $\beta : X \rightarrow X$ be a $(\Delta, \delta(\Delta))$-sequential function. A function $f : \mathbb{N} \times X \rightarrow X$ defined as $f(k, x) = \beta^{(k)}(x) = \overbrace{\beta \circ \beta \circ \ldots \circ \beta}^{k \text{ Times}}$ is an iterated sequential function, with round function β, if for all $k = 2^{o(\lambda)}$ function $h : X \rightarrow X$ defined by $h(x) = f(k, x)$ is $(k\Delta, \delta(\Delta))$-sequential.

The primary property of an iterated sequential function is that the iteration of the round function β is the quickest way to evaluate the function. Iterated squaring in a finite group of unknown order is widely believed to be a suitable candidate for an iterated sequential function. Below, we restate its definition.

Assumption 1 (Iterated Squaring). *Let N be a strong RSA modulus, r be a generator of $\mathbb{Z}_N$, Δ be a time parameter, and $T = poly(\Delta, \lambda)$. For any $\mathcal{A}$, defined above, there is a negligible function $\mu(\cdot)$ such that:*

$$\Pr\left[\begin{array}{l} r \xleftarrow{\$} \mathbb{Z}_N, b \xleftarrow{\$} \{0,1\} \\ \text{if } b = 0,\ y \xleftarrow{\$} \mathbb{Z}_N \\ \text{else } y = r^{2^T} \end{array} : \mathcal{A}(N, r, y) \rightarrow b\right] \leq \frac{1}{2} + \mu(\lambda)$$

D The original RSA-based TLP

Below, we restate the original RSA-based time-lock puzzle proposed by Rivest et al. [39].

(1) $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, S)$.
  (a) pick two large random prime numbers, q_1 and q_2. Set $N = q_1 \cdot q_2$ and compute Euler's totient function of N as follows, $\phi(N) = (q_1 - 1) \cdot (q_2 - 1)$.
  (b) set $T = S \cdot \Delta$ the total number of squarings needed to decrypt an encrypted message m, where S is the maximum number of squaring modulo N per second that the (strongest) server can perform, and Δ is the period, in seconds, for which the message must remain private.
  (c) generate a key for the symmetric-key encryption: $\mathsf{SKE.keyGen}(1^\lambda) \rightarrow k$.
  (d) choose a uniformly random value r: $r \xleftarrow{\$} \mathbb{Z}_N^*$.
  (e) set $a = 2^T \bmod \phi(N)$.
  (f) set $pk \coloneqq (N, T, r)$ as the public key and $sk \coloneqq (q_1, q_2, a, k)$ as the secret key.
(2) $\mathsf{GenPuzzle}_{\mathsf{TLP}}(m, pk, sk)$.
  (a) encrypt the message under key k using the symmetric-key encryption, as follows: $p_1 = \mathsf{SKE.Enc}(k, m)$.
  (b) encrypt the symmetric-key encryption key k, as follows: $p_2 = k + r^a \bmod N$.
  (c) set $p \coloneqq (p_1, p_2)$ as puzzle and output the puzzle.
(3) $\mathsf{Solve}_{\mathsf{TLP}}(pk, p)$.
  (a) find b, where $b = r^{2^T} \bmod N$, through repeated squaring of r modulo N.
  (b) decrypt the key's ciphertext, i.e., $k = p_2 - b \bmod N$.
  (c) decrypt the message's ciphertext, i.e., $m = \mathsf{SKE.Dec}(k, p_1)$. Output the solution, m.

E Security Analysis of GC-TLP

Theorem 3. *GC-TLP is a secure multi-instance time-lock puzzle, w.r.t. Definition 6.*

Proof. The proof has similarities with that of C-TLP [3]. Nevertheless, it has significant differences. First, we prove a solver cannot find the parameters needed to solve j-th puzzle, without solving the previous puzzle, $(j-1)$-th.

Lemma 1. *Let N be a large RSA modulus, k be a random key for symmetric-key encryption, and $\lambda = \log_2(N) = \log_2(k)$ be the security parameter. In GC-TLP, given puzzle vector $\vec{p}$ and public key pk, an adversary $\mathcal{A} \coloneqq (\mathcal{A}_1, \mathcal{A}_2)$, defined in Section 3.1, cannot find the next group generator r_j, where $r_j \xleftarrow{\$} \mathbb{Z}_N^*$ and $j \geq 1$, significantly in time smaller than $T_j = \delta(\sum_{i=1}^{j-1} \bar{\Delta}_i)$, except with a negligible probability in the security parameter, $\mu(\lambda)$.*

Proof. The next generator, r_j, is picked uniformly at random from $\mathbb{Z}_N^*$ and is encrypted along with the $(j-1)$-th puzzle solution, s_{j-1}. For the adversary to find r_j without performing enough squaring, it has to either break the security of the symmetric-key scheme, decrypt the related ciphertext s_{i-1} and extract the random value from it, or guess r_j correctly. In both cases, the adversary's probability of success is negligible $\mu(\lambda)$. □

Next, we prove GC-TLP preserves a solution's privacy with regard to Definition 4.

Theorem 4. *Let $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$ be a vector of time parameters and N be a strong RSA modulus. If the sequential squaring assumption holds, factoring N is a hard problem, $\mathrm{G}(\cdot)$ is a random oracle and the symmetric-key encryption is IND-CPA secure, then GC-TLP (which encodes z solutions) is a privacy-preserving GM-TLP according to Definition 4.*

Proof. For adversary $\mathcal{A} \coloneqq (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in total time $O(poly(\sum_{i=1}^{z} \bar{\Delta}_i, \lambda))$, $\mathcal{A}_2$ runs in time $\delta(\sum_{i=1}^{j} \bar{\Delta}_i) < \sum_{i=1}^{j} \bar{\Delta}_i$ using at most $\xi(\max(\bar{\Delta}_1, \ldots, \bar{\Delta}_j))$ parallel processors, and $j \in [z]$, we have two cases. In case $z = 1$: to find s_1 earlier than $\delta(\bar{\Delta}_1)$, it has to break the original TLP scheme [39], as the two schemes are identical, yet TLP is known to be secure from Theorem 1. In case $z > 1$: to find s_j earlier than $T_j = \delta(\sum_{i=1}^{j} \bar{\Delta}_i)$, it has to either find one of the prior solutions earlier than its predefined time, which would require it to break the TLP scheme again, or to find the related generator, r_j, earlier than it is supposed to yet its success probability is negligible due to Lemma 1.

The adversary may want to find partial information of the commitment, g_j, pre-image (which contains the solution) before solving the puzzle. But, this is infeasible for a PPT adversary, given that $\mathrm{G}(\cdot)$ is a random oracle. We conclude that GC-TLP is a privacy-preserving generic multi-instance time-lock puzzle scheme. □

Next, we present the theorem and proof for GC-TLP solution's validity, w.r.t. Definition 5.

Theorem 5. *Let $\mathrm{G}(\cdot)$ be a hash function modeled as a random oracle. Then, GC-TLP preserves a solution's validity, according to Definition 5.*

Proof. The proof is reduced to the security of the hash-based commitment. Given the commitment $g_j = \mathrm{G}(m_j, d_j)$ and an opening $\pi \coloneqq (m_j, d_j)$, for an adversary to break solution validity, it must find (m'_j, d'_j), such that $\mathrm{G}(m'_j, d'_j) = g_j$, where $m_j \neq m'_j$, i.e., a collision. However, this is infeasible for a PPT adversary, as $\mathrm{G}(\cdot)$ is collision-resistant in the random oracle model. □

As indicated in the proofs of Theorems 4 and 5, GC-TLP preserves the privacy and validity of a solution, respectively. Hence, per Definition 6, GC-TLP is a *secure* generic multi-instance time-lock puzzle scheme. □

F Figures and tables

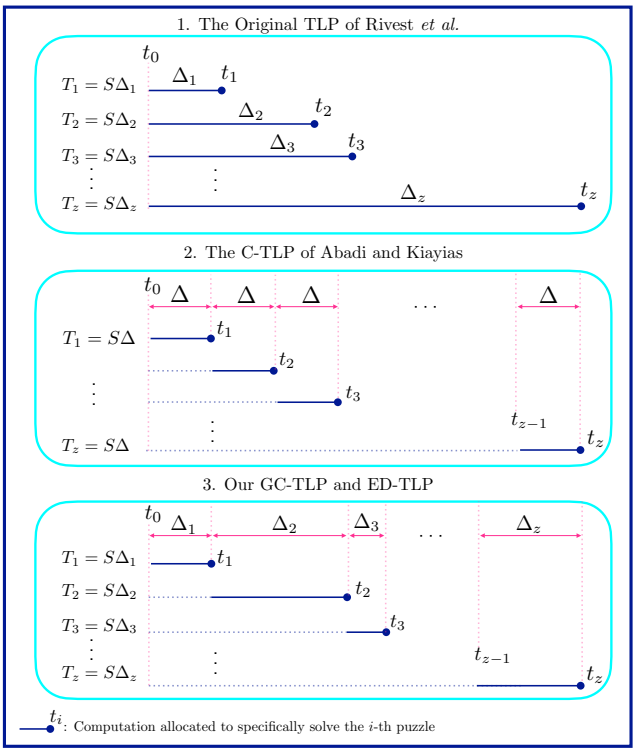


Figure 3: Differences between the following four schemes in terms of supporting multi-puzzle and varied-size time intervals: the original TLP of Rivest et al. [39], the C-TLP of Abadi and Kiayias [3], our GC-TLP, and our ED-TLP.

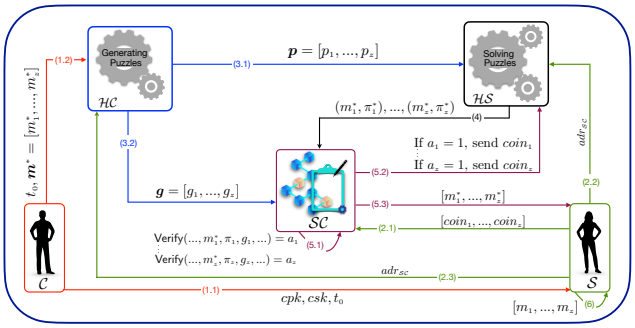


Figure 4: Outline of the interactions between parties in ED-TLP. $\mathcal{C}$ is the client, $\mathcal{S}$ is the server, $\mathcal{HC}$ is $\mathcal{C}$'s helper, $\mathcal{HS}$ is $\mathcal{S}$'s helper, and $\mathcal{SC}$ is the smart contract.

Table 5: Squaring capabilities of consumer and commodity cloud hardware for 1024-bit and 2048-bit RSA groups. Figure 1 displays a selection of the results.

Machine	CPU	Freq (GHz)	Squarings per second	
			1024-bit	2048-bit
Apple MacBook Pro	Apple M1 Pro	3.23	$2.260 \cdot 10^6$	$0.845 \cdot 10^6$
Dell Latitude 5421	Intel i7-11850H	2.50	$2.025 \cdot 10^6$	$0.749 \cdot 10^6$
University Cluster	Intel Haswell	2.4	$1.683 \cdot 10^6$	$0.671 \cdot 10^6$
Azure F2s_v2 [32, 33]	Intel Xeon 8272CL	2.60	$1.612 \cdot 10^6$	$0.628 \cdot 10^6$
Azure DS2_v2 [32, 33]	Intel Xeon E5-2673 v3	2.40	$1.201 \cdot 10^6$	$0.480 \cdot 10^6$
Azure D2as_v5 [32, 33]	AMD EPYC 7763	2.45	$1.935 \cdot 10^6$	$0.686 \cdot 10^6$
Azure D2ls_v5 [32, 33]	Intel Xeon 8370C	2.80	$1.509 \cdot 10^6$	$0.573 \cdot 10^6$
Azure D2pls_v5 [32, 33]	Ampere Altra	3.0	$0.824 \cdot 10^6$	$0.262 \cdot 10^6$
Azure FX4mds [32, 33]	Intel Xeon 6246R CPU	3.40	$1.911 \cdot 10^6$	$0.736 \cdot 10^6$
*Pearson FPGA**	*AWS F1 [4]*	0.1586	$38.168 \cdot 10^6$	N/A

* For comparison, we include the squaring capabilities of an FPGA-based solution developed by Eric Pearson for the VDF Alliance FPGA Competition [43], as reported in [44].

G Extending a puzzle chain

(1) $\mathsf{GenPuzzleExt}_{\mathsf{GM-TLP}}(1^\lambda, \vec{\Delta}', S', z, z', pk, sk) \rightarrow (pk', sk')$. Involves C. Requires an existing set of keys pk, sk, the previous number of puzzles z and the number of new puzzles z'.
   (a) $\forall j, z < j \leq z + z'$, set $T_j = S' \cdot \bar{\Delta}'_j$.
   (b) Set $\vec{T}\,' = [T_1, \ldots, T_z, T_{z+1}, \ldots, T_{z+z'}]$ containing values $\vec{T} = [T_1, \ldots, T_z] \in pk$.
   (c) compute values a_j: $\forall j, z < j \leq z + z' : a_j = 2^{T_j} \bmod \phi(N)$. This yields vector $\vec{a}' = [a_1, \ldots, a_z, a_{z+1}, \ldots, a_{z+z'}]$ containing values $\vec{a} = [a_1, \ldots, a_z] \in sk$.
   (d) choose z' fixed size random generators: $r_j \xleftarrow{\$} \mathbb{Z}^*_N$, where $|r_j| = \omega_1$, $\forall j,\ z < j \leq z + z'$ and set $\vec{r}' = [r_2, \ldots, r_{z+1}, r_{z+2}, \ldots, r_{z+z'+1}]$ containing values $\vec{r} \in sk$. Also, pick z' random keys for the symmetric-key encryption and set $\vec{k}' = [k_1, \ldots, k_z, k_{z+1}, \ldots, k_{z+z'}]$ containing $\vec{k} \in sk$. Choose z' fixed size sufficiently large random values, where $|d_j| = \omega_2\ \forall j,\ z < j \leq z + z'$ and set $\vec{d}' = [d_1, \ldots, d_z, d_{z+1}, \ldots, d_{z+z'}]$ containing values $\vec{d} \in sk$.
   (e) set public key $pk' \coloneqq (aux, N, \vec{T}\,', r_1, \omega_1, \omega_2)$ and secret key $sk' \coloneqq (q_1, q_2, \vec{a}', \vec{k}', \vec{r}', \vec{d}')$ where $aux, r_1, \omega_1, \omega_2 \in pk$ and $q_1, q_2 \in sk$. Output pk' and sk'.

(2) $\mathsf{GenPuzzleExt}_{\mathsf{GM-TLP}}(\overrightarrow{m}', pk', sk') \rightarrow \hat{p}'$. Involves C. Encrypt the messages $\forall j, z < j \leq z+z'$ as in $\mathsf{GenPuzzle}_{\mathsf{GM-TLP}}$ and output the new puzzle vector $\hat{p}'$. Send $\hat{p}'$ and pk' to $\mathcal{S}$. The new $\vec{g}'$ is made public.

$\mathcal{S}$ proceeds to solve the newly generated puzzles as normal but only once all puzzles in the original chain have been solved. As extending a message chain is equivalent to creating a longer chain initially, the security properties of the protocol are not affected.

H Security Analysis of ED-TLP

We prove the security of ED-TLP, i.e., Theorem 2.

Proof. We first focus on the solutions' privacy, w.r.t. Definition 9.

Claim 1. If the symmetric-key encryption satisfies IND-CPA, then ED-TLP preserves solutions' privacy from $\mathcal{HC}$ and $\mathcal{HS}$, w.r.t. Case 1 in Definition 9.

Proof. $\mathcal{HC}$ receives a vector $\overrightarrow{m}^*$ of ciphertexts (i.e., encrypted plaintext solutions), and public parameters in set $A = \{cpk, \mathsf{CEDG}, ToC, S, \vec{\Delta}, aux, coins, t_0, adr_{\mathcal{HS}}, adr_{SC}\}$. Other parameters given to $\mathcal{A}_1$ in the experiment (i.e., parameters in set $B = \{pk, \vec{s}, \hat{p}\}$) are generated by $\mathcal{HC}$ itself and may seem redundant. However, we have given these parameters to $\mathcal{A}_1$ for the case where $\mathcal{HS}$ is corrupt as well, which we will discuss shortly.

Since the public parameters were generated independently of the plaintext solutions $m_1, \ldots, m_z$, they do not reveal anything about each m_i. As the symmetric-key encryption scheme is IND-CPA, the vector $\overrightarrow{m}^*$ of ciphertext reveals no information about each m_i. Specifically, in Case 1 in Definition 9, the probability that $\mathcal{A}_1$ can tell whether a ciphertext $m^*_{b_i,i} \in \overrightarrow{m}^*$ is an encryption of message $m_{0,i}$ or $m_{1,i}$, chosen by $\mathcal{A}_1$, is at most $\frac{1}{2} + \mu(\lambda)$.

Now, we focus on the case where $\mathcal{HS}$ is corrupt. The messages that $\mathcal{HS}$ receives include a vector $\overrightarrow{m}^*$ of ciphertexts and parameters in set $A + B$. As long as the symmetric-key encryption meets IND-CPA, the vector $\overrightarrow{m}^*$ of ciphertext reveals no information about each m_i. Hence, when $\mathcal{HS}$ is corrupt, the probability that $\mathcal{A}_1$, in Case 1 in Definition 9, can tell if a ciphertext $m^*_{b_i,i} \in \overrightarrow{m}^*$ is the encryption of message $m_{0,i}$ or $m_{1,i}$, both of which were initially chosen by $\mathcal{A}_1$, is at most $\frac{1}{2} + \mu(\lambda)$.

Recall the public parameters in A do not reveal anything about each m_i. The same holds for the public parameter $pk \in B$. Also, parameters $\vec{s}$ and $\hat{p}$ in B were generated by running $\mathsf{GenPuzzle}_{\mathsf{DTLP}}$ and $\mathsf{Solve}_{\mathsf{DTLP}}$ on the ciphertexts in vector $\overrightarrow{m}^*$. Thus, $\vec{s}$ and $\hat{p}$ will not reveal anything about the plaintext messages, as long as the symmetric-key encryption satisfies IND-CPA. □

Claim 2. If GC-TLP protocol is privacy-preserving (w.r.t. Definition 4), then ED-TLP preserves solutions' privacy from $\mathcal{S}$ and $\mathcal{HS}$, w.r.t. Case 2 in Definition 9.

Proof. Before solving the puzzles, the messages that $\mathcal{HS}$ or $\mathcal{S}$ receives include the elements of sets $A = \{pk, \hat{p}\}$ and $B = \{\vec{\Psi}$, CEDG, t_0, *coins*, $adr_{\mathcal{HS}}$, adr_{SC}, ToC, $aux\}$. The elements of set A are identical to what $\mathcal{S}$ receives in GC-TLP. The elements of set B are independent of the plaintext solutions and the puzzles' secret and public parameters. Therefore, knowledge of B does not help $\mathcal{S}$ or $\mathcal{HS}$ learn the plaintext solutions before solving the puzzles. More formally, given $A+B$, in Case 2 in Definition 9, the probability that $\mathcal{A}_3$ can tell whether a puzzle $p_{b_i,i} \in \vec{p} \in \hat{p}$ has been created for plaintext message $m_{0,i}$ or $m_{1,i}$ is at most $\frac{1}{2} + \mu(\lambda)$, due to the privacy property of GC-TLP, i.e., Theorem 4. □

Claim 3. If GC-TLP meets solution-validity (w.r.t. Definition 5), the blockchain is secure (i.e., it meets persistence and liveness properties [22], and the signature satisfies existential unforgeability under chosen message attacks) and the smart contract's correctness holds, ED-TLP preserves a solution validity, w.r.t. Definition 10.

Proof. First, we focus on event I = $(con_1 \wedge con_2 \wedge con_3)$: a prover submits proof on time and passes the verification despite the proof containing an opening for a different message than the one already committed to. Compared to GC-TLP, the extra information that a prover (in this case $\mathcal{HS}$) learns in ED-TLP includes parameters in set $A + B$. Nevertheless, these parameters are independent of the plaintext messages and the parameters used for the commitment. Therefore, the solutions' validity is reduced to the solution validity of GC-TLP and, in turn, to the security of the commitment scheme. Specifically, given commitment $g_j = \mathrm{G}(m_j, d_j)$ and an opening $\pi \coloneqq (m_j, d_j)$, for an adversary to break the solution validity, it must generate (m'_j, d'_j), such that $\mathrm{G}(m'_j, d'_j) = g_j$, where $m_j \neq m'_j$. However, this is infeasible for a PPT adversary, as $\mathrm{G}(\cdot)$ is collision-resistant, in the random oracle model. Thus, in the experiment in Definition 10, event I occurs with a probability at most $\mu(\lambda)$.

Now, we focus on event II = $(\neg con_1 \wedge con_4 \wedge con_5)$: the adversary has generated a valid proof and passed the verification despite registering the proof late. Due to the persistency property of the blockchain, once a transaction goes more than v blocks deep into the blockchain of one honest player (where v is a security parameter), it will be included in every honest player's blockchain with overwhelming probability, and it will be assigned a permanent position in the blockchain (so it will not be modified with an overwhelming probability).

Due to the liveness property, all transactions originating from honest parties will eventually end up at a depth of more than v blocks in an honest player's blockchain; so, the adversary cannot perform a selective denial of service attack against honest account holders. Thus, with a high probability when a (well-formed transaction containing) proof is sent late to the smart contract, the smart contract declares late; accordingly, $\mathsf{Verify}_{\mathsf{DTLP}}(\cdot)$ outputs 0 except for a negligible probability, $\mu(\lambda)$. Hence, in the experiment in Definition 10, event II occurs with a probability at most $\mu(\lambda)$. □

Now, we focus on fair payment, w.r.t. Definition 11.

Claim 4. If the blockchain is secure (it meets persistence and liveness properties [22]), then ED-TLP offers fair payment, w.r.t. Definition 11.

Proof. The proof reduces to the security of the blockchain (and smart contracts). Specifically, due to the persistence and liveness properties, (1) the state of a smart contract cannot be tampered with and (2) each function implemented in a smart contract correctly computes a result, except for a negligible probability $\mu(\lambda)$. Thus, in the experiment in Definition 11, when $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, \pi_j, g_j, \Psi_j, t_j, t_0, \vec{\Delta}, \Upsilon) \rightarrow v_j \in \{0, 1\}$, then (1) the intact v_j is passed on to $\mathsf{Pay}_{\mathsf{DTLP}}(v_j, adr_{\mathcal{HS}}, \overrightarrow{coins}, j)$ as input (because the smart contract generates and maintains v_j and passes it to $\mathsf{Verify}_{\mathsf{DTLP}}$) and (2) $u_j = v_j$, except for the probability of $\mu(\lambda)$. □

Hence, ED-TLP is secure, w.r.t. Definition 12, given that ED-TLP satisfies the solutions' privacy (w.r.t. Definition 9), the solutions' validity (w.r.t. Definition 10), and fair payment (w.r.t. Definition 11). ■