Delegated Time-Lock Puzzle

Aydin Abadi* Dan Ristea** Steven J. Murdoch***

University College London

Abstract. Time-Lock Puzzles (TLPs) are cryptographic protocols that enable a client to lock a message in such a way that a server can only unlock it after a specific time period. However, existing TLPs have certain limitations: (i) they assume that both the client and server always possess *sufficient computational resources* and (ii) they solely focus on the lower time bound for finding a solution, disregarding the upper bound that guarantees a *regular server* can find a solution within a certain time frame. Additionally, existing TLPs designed to handle multiple puzzles either (a) entail high verification costs or (b) *lack generality*, requiring identical time intervals between consecutive solutions. To address these limitations, this paper introduces, for the first time, the concept of a "Delegated Time-Lock Puzzle" and presents a protocol called "Efficient Delegated Time-Lock Puzzle" (ED-TLP) that realises this concept. ED-TLP allows the client and server to *delegate* their resource-demanding tasks to third-party helpers. It facilitates *real-time verification* of solution correctness and efficiently handles multiple puzzles with varying time intervals. ED-TLP ensures the delivery of solutions within predefined time limits by incorporating both an upper bound and a fair payment algorithm. We have implemented ED-TLP and conducted a comprehensive analysis of its overheads, demonstrating the efficiency of the construction.

1 Introduction

Time-Lock Puzzles (TLPs) are elegant cryptographic primitives that enable the transmission of information to the future. They allow a party to lock a message in such a way that no one else can unlock it until a certain amount of time has elapsed. [1] TLPs have various applications, including timely payments in cryptocurrencies [29], e-voting [9], fair contract signing [7], timed secret sharing [20], and sealed-bid auctions [27]. Researchers have proposed a variety of time-lock puzzles over the last two decades.

1.1 Critical Limitations of Existing Solutions

Existing TLPs have certain assumptions and limitations. Firstly, these schemes assume that both the client and server possess sufficient computational resources to handle the

* aydin.abadi@ucl.ac.uk
** dan.ristea.19@ucl.ac.uk
*** s.murdoch@ucl.ac.uk

[1] There are time-lock puzzles that use the help of a third party to support the time-release of a secret. This protocols category is not the focus of this paper.

resource-demanding tasks themselves during different phases.[2] Nevertheless, this assumption needs to be relaxed when the client and/or server are thin clients (e.g., the Internet of Things devices) with limited computational power.

Secondly, existing schemes primarily concentrate on the lower bound, representing the earliest possible time for the most powerful server to find a solution. However, they have not considered or defined the *upper bound*, which indicates the guaranteed time for a regular server to find a solution.

Thirdly, the existing time-lock puzzle schemes that efficiently handle the multi-puzzle setting[3] face specific limitations. Either (a) they impose a high verification cost on the party responsible for checking the correctness of solutions, or (b) they only work efficiently when the time interval between the discovery of every two consecutive solutions is identical.

1.2 Our Contributions

In this paper, we address the aforementioned limitations by introducing the concept of the "Delegated Time-Lock Puzzle" (D-TLP) and presenting the "Efficient Delegated Time-Lock Puzzle" (ED-TLP), a protocol that realizes this concept. ED-TLP enables end-to-end delegation, allowing both the client and server to independently delegate their resource-demanding tasks to third-party helpers while ensuring the privacy of the plaintext solutions from these helpers. It efficiently supports real-time verification of solution correctness by relying solely on symmetric-key primitives. Furthermore, ED-TLP accommodates varied time intervals and efficiently handles the multi-puzzle setting, wherein a server receives a large set of puzzles and must find different solutions at different points in time.

Additionally, ED-TLP offers accurate-time solution delivery, ensuring that a rational solver always delivers a solution before a predefined time. To capture this feature, we rely on two techniques. First, we equip ED-TLP with an upper bound that explicitly specifies the time when a solver will find a solution. To determine this upper bound, we introduce a new predicate called "Customised Extra Delay Generating" which may be of independent interest and can find applications in other delay-based primitives, such as Verifiable Delay Functions (VDFs) [5,31,12]. Second, we incorporate fair payment into ED-TLP; fair payment is an algorithm that pays a solver only if it provides a valid solution before a predefined time.

We define and develop ED-TLP in a modular fashion. Firstly, we propose the concept of a "Generic Multi-instance Time-Lock Puzzle" (GM-TLP) and devise an efficient protocol called "Generic Chained Time-Lock Puzzle" (GC-TLP) that realizes GM-TLP. Subsequently, we enhance GM-TLP to the notion of D-TLP and build ED-TLP based on GC-TLP. Our protocols utilise a standard time-lock puzzle scheme in a black-box manner. ED-TLP is the first time-lock puzzle that simultaneously offers all of the aforementioned features. We will also show that the ideas behind D-TLP can be utilised to transform VDF to delegated VDF.

[2] There is a scheme that aims to facilitate delegation, but (as we will briefly discuss in Section 2) it suffers from a security issue.

[3] In the multi-puzzle setting, a server is given a (large) set of puzzles at once and needs to find different solutions at different points in time

We have also implemented ED-TLP and GC-TLP and conducted a thorough evaluation of their concrete overheads. The results of our evaluation demonstrate the remarkable efficiency of these protocols. For instance, when considering a total number of 100 puzzles, the outcomes are as follows:

- ED-TLP's client-side run-time is:
  - 74 times faster than client-side run-time in the state-of-the-art multi-puzzle TLP in [1].
  - 1165 times faster than client-side run-time of the original TLP of Rivest *et al.* [27] which serves as the foundation for numerous TLPs and VDFs.
- ED-TLP's server-side run-time is:
  - 102 times faster than the server-side run-time in [1].
  - 5055 times faster than the server-side run-time in [27].

The implementation source code is publicly available.

2 Related Work

In this section, we present a summary of related work. For a comprehensive survey, we refer readers to a recent survey in [23]. Timothy C. May [22] was the first who proposed the idea of sending information into the future, i.e., time-lock puzzle/encryption. A time-lock puzzle lets a party/client encrypt a message such that it cannot be decrypted until a certain time elapses. A basic property of a time-lock scheme is that generating a puzzle takes less time than solving it. Since the scheme that May proposed uses a trusted agent which releases a secret on time for a puzzle to be solved and relying on a trusted agent can be a strong assumption, Rivest *et al.* [27] proposed an RSA-based puzzle scheme. This scheme does not require a trusted agent, relies on sequential (modular) squaring, and is secure against a receiver who may have access to many computation resources that run in parallel.

Since the introduction of the RSA-based time-lock puzzle, various variants of it have been proposed. For instance, researchers such as Boneh *et al.* [7] and Garay *et al.* [15] have proposed time-lock puzzles schemes which consider 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. Also, Baum *et al.* [3] have developed a composable time-lock puzzle scheme that can be defined and proven in the universal composability framework.

2.1 Homomorphic Time-lock Puzzles.

Malavolta *et al.* [21] and Brakerski *et al.* [8] proposed the notion of homomorphic time-lock puzzles, which allow an arbitrary function to run over puzzles before they are solved. They are based on the RSA-based puzzle and fully homomorphic encryption which imposes high overheads. To improve the above homomorphic time-lock puzzle schemes, Srinivasan *et al.* [28] proposed a scheme that supports the unbounded batching of puzzles. The construction is based on indistinguishability obfuscation and puncturable pseudorandom function.

2.2 Multi-instance Time Lock Puzzle.

Researchers also have considered the setting where a server may receive multiple puzzles/instances at once from a client and needs to solve all of them; the time-lock puzzles in [10,1,11] deal with such a setting. The schemes in [10,11] use the original RSA time-lock puzzle in a non-black-box way and are based on an asymmetric-key encryption scheme, as opposed to a symmetric-key one, used in the original time-lock puzzle. Also, the one in [10] offers no verification. To address these limitations Abadi *et al.* [1] proposed the chained time-lock-puzzle protocol (C-TLP). The scheme uses the RSA-based scheme in a black-box manner. It also uses efficient (hash-based) verification to let the solver prove that it found the correct solutions. To date, it remains the most efficient time-lock puzzle for the multi-puzzle setting. Nevertheless, the scheme efficiently works only for identical-size time intervals, i.e., the size of the time window between the discovery of every two consecutive solutions must be identical. This shortcoming limits the scheme's application and adoption.

2.3 Outsourced Sequential Squaring.

Thyagarajan *et al.* [30] at CCS 2021 have 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. In this scheme, a client posts the number of squaring required and its related public parameters to a smart contract, say D, and places a certain amount of deposit in D. The solvers (whose identities are not fixed before the protocol's execution) are required to propose a solution before a certain time point, called "t_{sol}" in [30].

Then, each solver locally computes the solution and related proof of correctness. It sends to D, the solution, proof, and an asking price. The contract allows any users or solvers to locally check the proof and send their complaints if the proof is invalid. If no complaints are received, then D pays the solver who asked for the lowest price.

The scheme uses a proof system which requires at least 4 exponentiations to verify each proof. Since performing a fixed computation on a smart contract, say D, costs more than performing the same computation locally, the scheme requires users (that can include the competing solvers) to locally verify each proof and report it to D if the verification fails. In this case, D verifies the proof itself (that still imposes a high cost).

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. Also, this scheme does not offer any solution for multi-puzzle settings and it does not explain how the value of t_{sol} is set. Furthermore, it treats the original RSA-based time-lock puzzle in a non-black-box manner, offers no formal definition for delegated time-lock puzzles (they only present a formal definition for delegated squaring), and supports only one-side (i.e., solver-side) delegation. We address all of these limitations.

2.4 Verifiable Delay Function (VDF)

A VDF enables a prover to provide a publicly verifiable proof stating that it has performed a pre-determined number of sequential computations [5,31,6,25]. VDFs have

various applications, such as in decentralised systems to extract reliable public randomness from a blockchain. VDF was first formalised by Boneh *et al* in [5]. They proposed several VDF constructions based on SNARKs along with either incrementally verifiable computation or injective polynomials, or based on time-lock puzzles, where the SNARKs based approaches require a trusted setup.

Later, Wesolowski [31] and Pietrzak [25] concurrently improved the previous VDFs from different perspectives and proposed schemes based on sequential squaring. They also support efficient verification. Most VDFs have been built upon time-lock puzzles. Nevertheless, the converse is not necessarily the case. Because, VDFs are not suitable to encapsulate an arbitrary private message and they take a public message as input, whereas time-lock puzzles have been designed to conceal a private input message.

3 Preliminaries

3.1 Notations and Assumptions

Table 1. Notation Table.

Symbol	Description
λ	Security parameter
z	Number of puzzles
$\mathcal{S}$	Server
$\mathcal{C}$	Client
$\mathcal{TPH}$	Helper of client
$\mathcal{TPH}'$	Helper of server
$\mathcal{SC}$	Smart contract
$\mathcal{NP}$	Nondeterministic polynomial time
$\mathcal{L}$	Language in $\mathcal{NP}$
sk, pk	Secret and public keys
N	RSA modulus
TLP	Time-lock puzzle
GMTLP	Generic multi-instance time-lock puzzle
DTLP	Delegated time-lock puzzle
Δ	Period that a message must remain hidden
S	Max. squaring that the strongest solver can do
$T = S\Delta$	Number of squaring needed to solve a puzzle
j	Puzzle's index
s or s_j	Puzzle's solution
p or p_j	Puzzle
$\boldsymbol{p}$	$[p_1, ..., p_z]$
g_j	Public statement
$\boldsymbol{g}$	$[g_1, ..., g_z]$
$\hat{p}$	$\hat{p} := (\boldsymbol{p}, \boldsymbol{g})$
m, m_j	Plaintext message
r, r_j, d_j	Random value
$\mathtt{G}$	Hash function

Symbol	Description
$\bar{\Delta}_j$	Time interval
$\sum_{i=1}^{j} \bar{\Delta}_i$	Period after which j-th solution is found in D-TLP/GM-TLP
$T = S\bar{\Delta}_j$	Number of squaring needed to solve a puzzle
π_j	Proof of solution's correctness
$\mathtt{CEDG}$	Customised extra delay generating predicate
ToC	Type of computational step
aux, aux_{ID}	Specification of a certain solver
Υ	Network delay
m_j^*	Encoded/encrypted message
$\boldsymbol{m}^*$	$[m_1^*, ..., m_z^*]$
t_0	Time when puzzles are given to solver
$coins_j$	Coins paid to solver for finding j-th solution
$coins$	$coins_1 + ... + coins_z$
$\boldsymbol{coins}$	$[coins_1, ..., coins_z]$
$coins'$	Coins a solver wants to receive for finding z puzzles
t_j	Time when j-th solution is registered in $\mathcal{SC}$
Ψ_j	Extra time needed to find j-th solution
$\boldsymbol{\Psi}$	$[\Psi_1, ..., \Psi_z]$
k_j	Symm-key encryption random key
adr_I	Party I's account addresses
μ	Negligible function
$\mathtt{SKE.keyGen}$	Symm-key encryption's key gen. algo.
$\mathtt{Enc}$	Symm-key encryption's encryption algo.
$\mathtt{Dec}$	Symm-key encryption's decryption algo.
$\mathtt{Com}$	Commitment scheme's commit algo.
$\mathtt{Ver}$	Commitment scheme's verify algo.

We define a parse function as $\mathtt{parse}(\omega, y) \rightarrow (a, b)$, which takes as input a value ω and a value y of length at least ω-bit. It parses y into two values a and b and returns (a, b) where the bit length of a is $|y| - \omega$ and the bit length of b is ω.

To ensure generality in the definition of our verification algorithms, we adopt notations from zero-knowledge proof systems [4,13]. 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., $\mathcal{L} = \{stm| \ \exists wit$ s.t. $R(stm, wit) = 1\}$. A (zero-knowledge) proof for $\mathcal{L}$ lets a prover convince a verifier that $stm \in \mathcal{L}$ for a common input stm (without revealing wit).

We denote network delay by Υ. As shown in [16,2], after an honestly generated block is sent to the 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 network delay, please refer to Appendix C. We summarise our notations in Table 1. We assume that parties use a secure channel when they interact with each other off-chain.

3.2 Symmetric-key Encryption Scheme

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

3.3 Time-lock Puzzle Scheme

In this section, we restate the time-lock puzzle's formal definitions, and the RSA-based time-lock puzzle protocol [27]. Our focus will be on the RSA-based puzzle, because of its simplicity and because it is the core of the majority of time-lock puzzle schemes.

Definition 1. *A time-lock puzzle scheme consists of the following three algorithms* ($\mathsf{Setup}_{\mathsf{TLP}}$, $\mathsf{GenPuzzle}_{\mathsf{TLP}}$, $\mathsf{Solve}_{\mathsf{TLP}}$) *that are defined below and meets completeness and efficiency properties. It involves two parties, a client $\mathcal{C}$ and a server $\mathcal{S}$.*

* *Algorithms:*
- $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, S) \rightarrow (pk, sk)$. *A probabilistic algorithm run by $\mathcal{C}$. 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 which is the maximum number of squaring that a solver with the highest level of computation resources can perform per second. It outputs pair (pk, sk) that contains public and private keys.*
- $\mathsf{GenPuzzle}_{\mathsf{TLP}}(s, pk, sk) \rightarrow p$. *A probabilistic algorithm run by $\mathcal{C}$. It takes as input a solution s and (pk, sk). It outputs a puzzle p.*
- $\mathsf{Solve}_{\mathsf{TLP}}(pk, p) \rightarrow s$. *A deterministic algorithm run by $\mathcal{S}$. It takes as input pk and p. It outputs a solution s.*

* *Completeness. For any honest $\mathcal{C}$ and $\mathcal{S}$, it always holds that* $\mathsf{Solve}_{\mathsf{TLP}}(pk, p) = s$.

* *Efficiency. the run-time of algorithm* $\mathsf{Solve}_{\mathsf{TLP}}(pk, p)$ *is upper-bounded by* $poly(\Delta, \lambda)$*, where* $poly(.)$ *is a polynomial.*

The security of a time-lock puzzle requires that the puzzle's solution stay confidential from all adversaries running in parallel within the time period, Δ. It also requires that an adversary cannot extract a solution in time $\delta(\Delta) < \Delta$, using $\xi(\Delta)$ processors that run in parallel and after a large amount of pre-computation.

Definition 2. *A time-lock puzzle 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* $\xi(\Delta)$ *parallel processors, there exists a negligible function* $\mu(.)$*, such that:*

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

Below, we restate the original RSA-based time-lock puzzle proposed in [27].

1. Setup: $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, S)$.
   (a) pick at random two large prime numbers, q_1 and q_2. Then, compute $N = q_1 q_2$. Next, compute Euler's totient function of N as follows, $\phi(N) = (q_1 - 1)(q_2 - 1)$.
   (b) set $T = S\Delta$ the total number of squaring needed to decrypt an encrypted message m, where S is the maximum number of squaring modulo N per second that the (strongest) solver 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, i.e., $\mathtt{SKE.keyGen}(1^\lambda) \rightarrow k$.
   (d) choose a uniformly random value r, i.e., $r \xleftarrow{\$} \mathbb{Z}_N^*$.
   (e) set $a = 2^T \bmod \phi(N)$.
   (f) set $pk := (N, T, r)$ as the public key and $sk := (q_1, q_2, a, k)$ as the secret key.
2. Generate Puzzle: $\mathsf{GenPuzzle}_{\mathsf{TLP}}(m, pk, sk)$.
   (a) encrypt the message under key k using the symmetric-key encryption, as follows: $p_1 = \mathtt{Enc}(k, m)$.
   (b) encrypt the symmetric-key encryption key k, as follows: $p_2 = k + r^a \bmod N$.
   (c) set $p := (p_1, p_2)$ as puzzle and output the puzzle.
3. Solve Puzzle: $\mathsf{Solve}_{\mathsf{TLP}}(pk, p)$.
   (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., $m = \mathtt{Dec}(k, p_1)$. Output the solution, m.

By definition, time-lock puzzles are sequential functions. Their construction (as shown above) requires that a sequential function, such as modular squaring, is called 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 [5]. We restate the definitions in Appendix B. The security of the RSA time-lock puzzle is based on the hardness of the factoring problem, the security of the symmetric key encryption, and sequential squaring assumption. We restate its formal definition below and refer readers to [1] for the proof.

Theorem 1. *Let N be a strong RSA modulus and Δ be the period within 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 time-lock puzzle.*

3.4 Commitment Scheme

A commitment scheme involves two parties, *sender* and *receiver*. It also includes two phases: *commit* and *open*. In the commit phase, the sender commits to a message: x as $\mathsf{Com}(x, r) = \mathsf{Com}_x$, that involves a secret value: $r \xleftarrow{\$} \{0,1\}^\lambda$. In the end of the commit phase, the commitment Com_x is sent to the receiver. In the open phase, the sender sends the opening $x := (x, r)$ to the receiver who verifies its correctness: $\mathsf{Ver}(\mathsf{Com}_x, x) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy two properties: (a) *hiding*: it is infeasible for an adversary (i.e., the receiver) to learn any information about the committed message x, until the commitment Com_x is opened, and (b) *binding*: it is infeasible for an adversary (i.e., the sender) to open a commitment Com_x to different values $x' := (x', r')$ than that was used in the commit phase, i.e., infeasible to find x', *s.t.* $\mathsf{Ver}(\mathsf{Com}_x, x) = \mathsf{Ver}(\mathsf{Com}_x, x') = 1$, where $x \neq x'$. There exist efficient non-interactive commitment schemes both in (a) the standard model, e.g., Pedersen scheme [24], and (b) the random oracle model using the well-known hash-based scheme such that committing is: $\mathtt{G}(x||r) = \mathsf{Com}_x$ and $\mathsf{Ver}(\mathsf{Com}_x, x)$ requires checking: $\mathtt{G}(x||r) \stackrel{?}{=} \mathsf{Com}_x$, where $\mathtt{G} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a collision-resistant hash function; i.e., the probability to find x and x' such that $\mathtt{G}(x) = \mathtt{G}(x')$ is negligible in the security parameter, λ.

3.5 Smart Contract

A smart contract is a computer program. It encodes the terms and conditions of an agreement between parties and often contains a set of variables and functions. A smart contract code is stored on a blockchain and is maintained by the miners who maintain the blockchain. When (a function of) a smart contract is triggered by an external party, every miner executes the smart contract's code. The program execution's correctness is guaranteed by the security of the underlying blockchain. Ethereum [32] has been the most predominant cryptocurrency framework that lets users define *arbitrary* smart contracts.

4 Generic Multi-instance Time-lock Puzzle

As we previously stated, even though C-TLP in [1] can efficiently deal with multiple puzzles, it has a serious limitation; namely, it cannot deal with the situation where the time intervals have different sizes. In this section, we address this limitation, by formally defining the notion of the Generic Multi-instance Time-Lock Puzzle (GM-TLP) and its concrete instantiation Generic Chained Time-Lock Puzzle (GC-TLP) which inherits C-TLP's interesting features and supports varied size time intervals.

4.1 GM-TLP Definition

In this section, we provide a formal definition of GM-TLP.

Definition 3 (Generic Multi-instance Time-lock Puzzle). *A GM-TLP consists of five algorithms* $(\mathsf{Setup}_{\mathsf{GMTLP}}, \mathsf{GenPuzzle}_{\mathsf{GMTLP}}, \mathsf{Solve}_{\mathsf{GMTLP}}, \mathsf{Prove}_{\mathsf{GMTLP}}, \mathsf{Verify}_{\mathsf{GMTLP}})$ *which are defined below. It satisfies completeness and efficiency properties. It also involves three entities: a client* $\mathcal{C}$*, a server* $\mathcal{S}$*, and a public verifier* $\mathcal{V}$*.*

* *Algorithms:*

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

* *Completeness. for any honest prover and verifier, it always holds that:*

- $\mathsf{Solve}_{\mathsf{GMTLP}}(pk, [p_1, \ldots, p_j]) = ([s_1, \ldots, s_j], [m_1, \ldots, m_j])$*,* $\forall j,\ 1 \leq j \leq z$*.*
- $\mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j, \pi_j, g_j) \rightarrow 1$*.*

* *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:* $poly(\sum_{i=1}^{j} \bar{\Delta}_i, \lambda)$*, where* $poly(.)$ *is a fixed polynomial and* $1 \leq j \leq z$*.*

To be more precise, in the above, the prover is required to generate a witness/proof π_j for the language $\mathcal{L} := \{stm := (g_j, m_j) |\ R(stm, \pi_j) = 1\}$.

Note that our proposed definition is a generalised version of the definition proposed in [1] from a couple of perspectives. Specifically, the later definition (1) only considers a specific case where all time intervals, i.e. $\bar{\Delta}_i$, are identical and (2) defines a very specific verification algorithm (similar to the one in [3]), compatible for the hash-based commitment. Whereas our proposed definition allows the time intervals to be of different sizes and captures a broad class of verification algorithms/schemes.

At a high level, our definition (similar to [1]) satisfies two properties: a solution's *privacy* and *validity*. Solution's privacy requires j-th solution to remain hidden from all adversaries that run in parallel in period: $\sum_{i=1}^{j} \bar{\Delta}_i$. On the other hand, 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 $\boldsymbol{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, any number of puzzle: $z \geq 1$, any j (where $1 \leq j \leq z$), any pair of randomised algorithm $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in time $O(poly(\sum_{i=1}^{j} \bar{\Delta}_i, \lambda))$ 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 $\xi(Max(\bar{\Delta}_1, \ldots, \bar{\Delta}_j))$ parallel processors, there exists a negligible function $\mu(.)$, such that:*

$$Pr\begin{bmatrix} \mathcal{A}_2(pk, \hat{p}, state) \rightarrow a \\ s.t. \\ a := (b_i, i) \\ m_{b_i,i} = m_{b_j,j} \end{bmatrix} : \begin{bmatrix} \mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk) \\ \mathcal{A}_1(1^\lambda, pk, z) \rightarrow (\boldsymbol{m}, state) \\ \forall j, 1 \leq j \leq z : b_j \xleftarrow{\$} \{0,1\} \\ \mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\boldsymbol{m}', pk, sk) \rightarrow \hat{p} \end{bmatrix} \leq \frac{1}{2} + \mu(\lambda)$$

where $1 \leq i \leq z$, $\boldsymbol{m} = [(m_{0,1}, m_{1,1}), \ldots, (m_{0,z}, m_{1,z})]$, and $\boldsymbol{m}' = [m_{b_1,1}, \ldots, m_{b_z,z}]$.

The above definition ensures the solutions appear after j-th one, stays hidden from the adversary with a high probability, as well. Moreover, similar to [1,5,21,17], it captures that even if $\mathcal{A}_1$ computes on the public parameters for a polynomial time, $\mathcal{A}_2$ cannot find 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 $\frac{1}{2}$. In general, we can set $\delta(\bar{\Delta}) = (1 - \epsilon)\bar{\Delta}$ for a small ϵ, where $0 < \epsilon < 1$, as stated in [5].

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

$$Pr\begin{bmatrix} \mathcal{A}_2(pk, \boldsymbol{s}, \hat{p}, state) \rightarrow a \\ s.t. \\ a := (j, \pi', m_j) \\ \mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j, \pi', m_j, g_j) = 1 \\ m_j \notin \mathcal{L} \end{bmatrix} : \begin{bmatrix} \mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk) \\ \mathcal{A}_1(1^\lambda, pk, \boldsymbol{\Delta}) \rightarrow (\boldsymbol{m}, state) \\ \mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\boldsymbol{m}, pk, sk) \rightarrow \hat{p} \\ \mathsf{Solve}_{\mathsf{GMTLP}}(pk, \boldsymbol{p}) \rightarrow (\boldsymbol{s}, \boldsymbol{m}) \end{bmatrix} \leq \mu(\lambda)$$

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

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

4.2 Protocol: Generic Chained Time-lock Puzzle (GC-TLP)

In this section, we present GC-TLP that realises GM-TLP. At a high level, GC-TLP works as follows.

Let $\mathcal{C}$ be the client who has a vector of messages: $\boldsymbol{m} = [m_1, \ldots, m_z]$ and wants a server, $\mathcal{S}$, to learn each message m_i at time $t_i \in \boldsymbol{t}$, where $\boldsymbol{t} = [t_1, \ldots, t_z]$, $\bar{\Delta}_j = t_j - t_{j-1}$, $\boldsymbol{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, $T_j = S\bar{\Delta}_j$ and $1 \leq j \leq z$. In this setting, $\mathcal{C}$ is available only at an earlier time t_0, where $t_0 < t_1$. In the setup phase, similar to C-TLP, $\mathcal{C}$ for each time interval, $\bar{\Delta}_j$, generates a set of parameters. Nevertheless, unlike C-TLP, in the same phase for each time interval, it also generates a time-dependent parameter a_j. Loosely speaking, the time-dependent parameters, $\boldsymbol{a} = [a_1, \ldots, a_z]$, allow the messages to be encoded such that the above time intervals have different sizes. In GC-TLP (similar to C-TLP) we use the chaining technique.

In the puzzle generation phase, $\mathcal{C}$ first uses the above parameters to encrypt the message that will be decrypted after the rest, m_z. However, it also uses a_j when it encrypts the related message. It integrates the information required for decrypting it into the ciphertext of message m_{z-1} that will be decrypted prior to m_z. It repeats the above process (using different values of a_j) until it encodes all messages. For $\mathcal{S}$ to solve the puzzles, it starts in the opposite direction where the puzzles were created. In particular, it first extracts message m_1 at time t_1. By finding m_1, $\mathcal{S}$ learns enough information that is needed to perform the sequential squaring to decrypt the next message, m_2. So, $\mathcal{S}$ after fully decrypting m_1, begins sequentially squaring to decrypt m_2. The server, $\mathcal{S}$, repeats the above process to find all messages. Below, we present the GC-TLP in detail.

1. Setup. $\mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk)$
   This phase involves the client, $\mathcal{C}$.
   (a) compute $N = q_1 q_2$, where q_i is a large randomly chosen prime number. Next, compute Euler's totient function of N, as: $\phi(N) = (q_1 - 1)(q_2 - 1)$.
   (b) set $T_j = S\bar{\Delta}_j$ as the total number of squaring needed to decrypt an encrypted message m_j after previous m_{j-1} message is revealed, where S is the maximum number of squaring modulo N per second that the strongest solver can perform and $1 \leq j \leq z$.
   (c) compute values a_j as follows.

$$\forall j, 1 \leq j \leq z : a_j = 2^{T_j} \bmod \phi(N)$$

   This yield vector $\boldsymbol{a} = [a_1, \ldots, a_z]$.
   (d) choose z fixed size random generators, i.e., $r_j \xleftarrow{\$} \mathbb{Z}_N^*$, and set $\boldsymbol{r} = [r_2, \ldots, r_z]$, where $1 \leq j \leq z$ and $|r_j| = \omega_1$. Also, pick z random keys: $\boldsymbol{k} = [k_1, \ldots, k_z]$ for the symmetric-key encryption. Choose z fixed size sufficiently large random values: $\boldsymbol{d} = [d_1, \ldots, d_z]$, where $|d_j| = \omega_2$.
   (e) set public key as $pk := (aux, N, \boldsymbol{T}, r_1, \omega_1, \omega_2)$ and secret key as $sk := (q_1, q_2, \boldsymbol{a}, \boldsymbol{k}, \boldsymbol{r}, \boldsymbol{d})$, where $\boldsymbol{T} = [T_1, \ldots, T_z]$ and *aux* contains a cryptographic hash function's description and the size of the random values. Output pk and sk.
2. Generate Puzzle. $\mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\boldsymbol{m}, pk, sk) \rightarrow \hat{p}$
   This phase involves $\mathcal{C}$.
   Encrypt the messages, starting with $j = z$, in descending order. $\forall j, z \geq j \geq 1$:

(a) set $pk_j := (N, T_j, r_j)$ and $sk_j := (q_1, q_2, a_j, k_j)$. Recall that if $j = 1$, then r_j is in pk; otherwise (if $j > 1$), r_j is in $\boldsymbol{r}$.
(b) generate a puzzle:
- if $j = z$, then call the puzzle generator algorithm (in the RSA-based puzzle scheme), i.e., call: $\mathsf{GenPuzzle}_{\mathsf{TLP}}(m_j||d_j, pk_j, sk_j) \rightarrow p_j := (p_{j,1}, p_{j,2})$.
- otherwise, call: $\mathsf{GenPuzzle}_{\mathsf{TLP}}(m_j||d_j||r_{j+1}, pk_j, sk_j) \rightarrow p_j := (p_{j,1}, p_{j,2})$.

Note, the difference between the above two cases is that, in the second case r_{j+1} is a part of the message, while in the first case it is not.
(c) commit to each message, i.e., $\mathtt{G}(m_j||d_j) = g_j$ and output g_j.
(d) output $p_j := (p_{j,1}, p_{j,2})$ as puzzle.

This phase results in vectors of puzzles: $\boldsymbol{p} = [p_1, \ldots, p_z]$ and commitments: $\boldsymbol{g} = [g_1, \ldots, g_z]$. Let $\hat{p} := (\boldsymbol{p}, \boldsymbol{g})$. All public parameters and puzzles are given to a server at time $t_0 < t_1$, where $\bar{\Delta}_1 = t_1 - t_0$. Also, $\boldsymbol{g}$ is sent to the public.

3. Solve Puzzle. $\mathsf{Solve}_{\mathsf{GMTLP}}(pk, \boldsymbol{p}) \rightarrow (\boldsymbol{s}, \boldsymbol{m})$
This phase involves the server, $\mathcal{S}$.
Decrypt the messages, starting with $j = 1$, in ascending order. $\forall j, 1 \leq j \leq z$:
(a) if $j = 1$, then set $r_j = r_1$, where $r_1 \in pk$; Otherwise, set $r_j = u$.
(b) set $pk_j := (N, T_j, r_j)$.
(c) call the puzzle solving algorithm in TLP scheme, i.e., call: $\mathsf{Solve}_{\mathsf{TLP}}(pk_j, p_j) \rightarrow x_j$, where $p_j \in \boldsymbol{p}$.
(d) parse x_j. In case $j < z$, we have $x_j = m_j||d_j||r_{j+1}$; otherwise, we have $x_j = m_j||d_j$. So, x_j is parsed as follows.
- if $j < z$:
  i. $\mathtt{parse}(\omega_1, m_j||d_j||r_{j+1}) \rightarrow (m_j||d_j, r_{j+1})$.
  ii. set $u = r_{j+1}$.
  iii. $\mathtt{parse}(\omega_2, m_j||d_j) \rightarrow (m_j, d_j)$.
  iv. output $s_j = m_j||d_j$ and m_j.
- if $j = z$:
  i. $\mathtt{parse}(\omega_2, m_j||d_j) \rightarrow (m_j, d_j)$.
  ii. output $s_j = x_j = m_j||d_j$ and m_j.

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

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

5. Verify. $\mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j, \pi_j, g_j) \rightarrow \{0, 1\}$
This phase involves the verifier, e.g., the public or $\mathcal{C}$.
- verify the commitment, i.e., $\mathtt{G}(m_j, \pi_j) \stackrel{?}{=} g_j$.
- if the verification is passed, then accept the solution and output 1; otherwise (if it is rejected), output 0.

The C-TLP scheme in [1] does not directly support different size time intervals, i.e., $\bar{\Delta}_j \neq \bar{\Delta}_{j+1}$, because, in the setup phase, the protocol makes a black-box call to algorithm $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta)$ which takes only a *single time parameter*, Δ. The same time parameter is used throughout the C-TLP protocol. If the C-TLP protocol in [1]

is directly used for the setting where the time intervals are different, then it imposes a high cost and introduces an additional layer of complexity, for the following reasons. In the setup, the protocol makes a black-box call to algorithm $\mathsf{Setup}_{\mathsf{TLP}}(1^{\lambda}, \Delta)$, where the algorithm takes only a *single time parameter*, Δ, and in each call it outputs a distinct public-private keys. This is sufficient in the C-TLP setting, as all the time intervals have the same size.

Nevertheless, if there are multiple distinct time parameters (i.e., $\bar{\Delta}_1, \ldots, \bar{\Delta}_z$, s.t. $\bar{\Delta}_j \neq \bar{\Delta}_{j+1}$), then $\mathsf{Setup}_{\mathsf{TLP}}(.)$ has to be called as many times as the total number of time parameters, i.e., z times. This results in z public keys and z secret keys (unlike our GC-TLP that requires a single pair of keys), which ultimately leads to a high communication cost, as all the public keys have to be published. Having different public keys introduces an additional layer of complexity as well. Because each puzzle's parameters are now defined over a ring of different size (as N's are different now).

5 Security Analysis of GC-TLP

In this section, we present the security proof of GC-TLP. The proof has similarities with the C-TLP's security proof in [1]. Nevertheless, it has significant differences. Therefore, for the sake of completeness, we provide the GC-TLP's security proof. 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 symmetric-key encryption's random key, and $\lambda = \log_2(N) = \log_2(k)$ be the security parameter. In GC-TLP, given puzzle vector $\boldsymbol{p}$ and public key pk, an adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$, defined in Section 4.1, cannot find the next group generator: r_j, where $r_j \xleftarrow{\$} \mathbb{Z}_N^*$ and $j \geq 1$, significantly 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. 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. However, in both cases, the adversary's probability of success is negligible $\sigma(\lambda)$. Because the next generator, r_j, is encrypted along with the $(j-1)$-th puzzle solution, s_{j-1}, and is picked uniformly at random from $\mathbb{Z}_N^*$. □

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

Theorem 2. *Let $\boldsymbol{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$ 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, $\mathtt{G}(.)$ 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. We will show below, for adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$, 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 $1 \leq j \leq z$,

- in the case $z = 1$: to find s_1 earlier than $\delta(\bar{\Delta}_1)$, it has to break the original TLP scheme [27].
- in the case $z > 1$: to find s_j earlier than $T_j = \delta(\sum_{i=1}^{j} \bar{\Delta}_i)$, it has to either find (at least) 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; but its success probability is negligible due to Lemma 1.

In particular, in the case $z = 1$, the security of GC-TLP boils down to the TLP's security. Therefore, in this case, GC-TLP scheme is secure as long as TLP is; because, the two schemes are identical, in this case. Furthermore, when $z > 1$, the adversary has to find solution s_j earlier than time t_j, when the previous solution, s_{j-1}, is extracted. However, this needs either breaking the original TLP scheme, or finding generator r_j prior to extracting s_{j-1}, where $2 < j \leq z$. We know that the TLP scheme is secure according to Theorem 1 and the probability of finding the next generator, r_j, prior to time t_{j-1} is negligible, according to Lemma 1. Also, for the adversary to find a solution earlier, it may also try to find partial information of the commitment, g_j, pre-image (which contains the solution) before fully solving the puzzle. Nevertheless, this is infeasible for a PPT adversary, given a random oracle's, $\mathtt{G}(.)$, output. 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 3. *Let $\mathtt{G}(.)$ 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 scheme. Specifically, given the commitment $g_j = \mathtt{G}(m_j, d_j)$ and an opening $\pi := (m_j, d_j)$, for an adversary to break the solution validity, it must come up (m'_j, d'_j), such that $\mathtt{G}(m'_j, d'_j) = g_j$, where $m_j \neq m'_j$, i.e., finds a collision of $\mathtt{G}(.)$. However, this is infeasible for a PPT adversary, as $\mathtt{G}(.)$ is collision-resistant in the random oracle model. □

Finally, we state and prove the main security theorem of GC-TLP.

Theorem 4. *GC-TLP is a secure multi-instance time-lock puzzle, w.r.t. Definition 6.*

Proof. As indicated in the proofs of Theorems 2 and 3, GC-TLP preserves the privacy and validity of a solution, respectively. Hence, according to Definition 6, GC-TLP is a *secure* generic multi-instance time-lock puzzle scheme. □

6 Delegated Time-lock Puzzle

In this section, we present Delegated TLP (D-TLP), which is an enhanced version of the GM-TLP notion, that also offers:

1. *client-side delegation*: allowing $\mathcal{C}$ to delegate setup and puzzle generation phases to a powerful Third-Party Helper ($\mathcal{TPH}$) (potentially a semi-honest adversary) while ensuring the privacy of its plaintext messages (i.e., puzzle solutions) is preserved. Supporting the delegation of setup and puzzle generation phases is an important feature, especially when these two phases are resource-demanding and a client with limited resources is not able to meet these phases' resource requirements.
2. *server-side delegation*: allowing $\mathcal{S}$ to delegate the role of finding solutions to another Third-Party Helper ($\mathcal{TPH}'$) (potentially rational adversary) while ensuring the privacy (and integrity) of the plaintext solutions.
3. *accurate-time solution delivery*: ensuring that each puzzle solution will be delivered before a pre-defined point in time.

To capture the latter property, we equip D-TLP with:

- *upper bound*: a parameter that explicitly specifies the time when a solver will find a solution. Such a parameter is not explicit in the definition of existing time-lock puzzles[4]. In these schemes, the only time parameter is the lower bound S (in $T = S\Delta$) which refers to the *maximum* number of squarings that a solver can perform per second. Given S, a puzzle creator can set T and the puzzle such that an adversary cannot find a solution earlier than it is supposed to. To determine the upper bound, we define a new predicate called "Customised Extra Delay Generating" that outputs a time parameter Ψ which determines how long after Δ the solver will find the puzzle's solution. Definition 7 presents further detail.
- *fair payment*: an algorithm that pays a party (i.e., $\mathcal{TPH}'$) if and only if the party provides a valid solution before a pre-defined time. Such an algorithm is necessary to ensure the timely *delivery* of a solution. Because having only an upper bound would not guarantee that the solution will be delivered by a rational adversary (e.g., $\mathcal{TPH}'$) who has already found the solution.

Definition 7 (Customised Extra Delay Generating Predicate). *Let ToC be a specific type of computational step (or hardness assumption) that a specific scheme relies on, S be the maximum number of computational steps (of type ToC) that a solver equipped with the highest level of computer resources can take per second, Δ be the time period (in seconds) that a message/solution must remain private, and aux_{ID} be auxiliary data about a specific solver identified with ID, e.g., the computational resources available to the solver. Then, predicate Customised Extra Delay Generating is defined as*

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

which takes ToC, S, Δ, and aux_{ID} as input and returns Ψ_{ID}: the extra time (in seconds) that the solver (in addition to Δ) must have, to discover the message/solution.

[4] The first time-lock proposal, introduced by May, which assumed the existence trusted server which sends a secret key to the recipient at a certain time, considered the upper bound. However, the later RSA-based time-lock puzzle of Rivest *et al.* considered only the lower bound (which would suffice in the non-delegated setting in which the solver is the same party who is interested in the solution). Since then, other proposals that relied on the computational power of the solver (e.g., RSA-based or random oracle) also did not (need to) define the upper bound.

Predicate $\texttt{CEDG}(.)$ can have applications in other delayed-based primitives beyond time-lock puzzles. For instance, it can be integrated into a VDF scheme to determine the exact time when a regular solver can find a valid output of the VDF.

6.1 D-TLP Definition

In this section, we first 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 consists of the following algorithms:* $(\mathsf{C.Setup}_{\mathsf{DTLP}}, \mathsf{C.Delegate}_{\mathsf{DTLP}}, \mathsf{S.Delegate}_{\mathsf{DTLP}}, \mathsf{TPH.Setup}_{\mathsf{DTLP}}, \mathsf{GenPuzzle}_{\mathsf{DTLP}}, \mathsf{Solve}_{\mathsf{DTLP}}, \mathsf{Prove}_{\mathsf{DTLP}}, \mathsf{Register}_{\mathsf{DTLP}}, \mathsf{Verify}_{\mathsf{DTLP}}, \mathsf{Pay}_{\mathsf{DTLP}}, \mathsf{Retrieve}_{\mathsf{DTLP}})$ *which are defined below. It satisfies completeness and efficiency properties. It involves a client $\mathcal{C}$, a pair of third-party helpers $(\mathcal{TPH}, \mathcal{TPH}')$, a server $\mathcal{S}$, and a smart contract $\mathcal{SC}$.*

* *Algorithms:*

- $\underline{\mathsf{C.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}) \rightarrow (cpk, csk)}$. *A probabilistic algorithm run by $\mathcal{C}$. It takes as input 1^λ and $\boldsymbol{\Delta}$. It generates a set of public cpk and private csk parameters. It appends $\boldsymbol{\Delta}$ to cpk and outputs (cpk, csk). $\mathcal{C}$ sends $\boldsymbol{\Delta}$ to $\mathcal{S}$ and csk to $\mathcal{S}$.*
- $\underline{\mathsf{C.Delegate}_{\mathsf{DTLP}}(\boldsymbol{m}, cpk, csk) \rightarrow (\boldsymbol{m}^*, t_0)}$. *A probabilistic algorithm run by $\mathcal{C}$. It takes as input a message vector $\boldsymbol{m} = [m_1, ..., m_z]$ and (cpk, csk). It encodes each element of vector $\boldsymbol{m}$. It outputs a vector of the encoded messages $\boldsymbol{m}^* = [m_1^*, ..., m_z^*]$ and time point t_0 when puzzles are provided to solver $\mathcal{TPH}'$. $\mathcal{C}$ publishes t_0 and sends $\boldsymbol{m}^*$ to $\mathcal{TPH}$.*
- $\underline{\mathsf{S.Delegate}_{\mathsf{DTLP}}(\texttt{CEDG}, ToC, S, \boldsymbol{\Delta}, aux, adr_{\mathcal{TPH}'}, t_0, \Upsilon, \boldsymbol{coins}) \rightarrow adr_{\mathcal{SC}}}$. *A deterministic algorithm run by $\mathcal{S}$. It takes as input $\texttt{CEDG}$, ToC, S , $\boldsymbol{\Delta}$, aux, $adr_{\mathcal{TPH}'}$, t_0, Υ, and $\boldsymbol{coins} = [coins_1, ..., coins_z]$, where "$coins_i$" amount of coins will be paid to $\mathcal{TPH}'$ for solving i-th puzzle. It executes $\texttt{CEDG}(ToC, S, \bar{\Delta}_i, aux) \rightarrow \Psi_i$, for every element $\bar{\Delta}_i$ of $\boldsymbol{\Delta}$. It generates a smart contract $\mathcal{SC}$, deploys $\mathcal{SC}$ into the blockchain, and deposits $coins = \sum_{j=1}^{z} coins_j$ amount of coins into $\mathcal{SC}$. It registers $t_0, \boldsymbol{\Psi} = [\Psi_1, ..., \Psi_z]$, $adr_{\mathcal{TPH}'}$, and Υ in $\mathcal{SC}$. Let $adr_{\mathcal{SC}}$ be the address of the deployed $\mathcal{SC}$. It returns $adr_{\mathcal{SC}}$. $\mathcal{S}$ publishes $adr_{\mathcal{SC}}$.*
- $\underline{\mathsf{TPH.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk)}$. *A probabilistic algorithm run by $\mathcal{TPH}$. It takes as input 1^λ, $\boldsymbol{\Delta}$, S, and z. It generates a set of public pk and private sk parameters. It outputs (pk, sk). $\mathcal{TPH}$ send pk to $\mathcal{TPH}'$.*
- $\underline{\mathsf{GenPuzzle}_{\mathsf{DTLP}}(\boldsymbol{m}^*, cpk, pk, sk, t_0) \rightarrow \hat{p}}$. *A probabilistic algorithm run by $\mathcal{TPH}$. It takes as input $\boldsymbol{m}^* = [m_1^*, ..., m_z^*]$, cpk, (pk, sk), and t_0. It outputs $\hat{p} := (\boldsymbol{p}, \boldsymbol{g})$, where $\boldsymbol{p}$ is a puzzle vector, $\boldsymbol{g}$ is a public statement vector w.r.t. language $\mathcal{L}$ and $\boldsymbol{m}^*$. Each j-th element in vectors $\boldsymbol{p}$ and $\boldsymbol{g}$ corresponds to a solution s_j that itself consists of m_j^* (and possibly witness parameters). $\mathcal{TPH}$ sends $\boldsymbol{g}$ to $\mathcal{SC}$. It also sends $\boldsymbol{p}$ to $\mathcal{TPH}'$ at time t_0.*
- $\underline{\mathsf{Solve}_{\mathsf{DTLP}}(\boldsymbol{\Psi}, aux, pk, \boldsymbol{p}, adr_{\mathcal{SC}}, coins', coins) \rightarrow (\boldsymbol{s}, q)}$. *A deterministic algorithm run by $\mathcal{TPH}'$. It takes as input $\boldsymbol{\Psi}$, aux, pk, $\boldsymbol{p}$, $adr_{\mathcal{SC}}$, the total amount of coins: $coins'$ that $\mathcal{TPH}'$ expects to receive, and $coins$. It checks $coins$ and $\boldsymbol{\Psi}$, if it does not agree on these parameters, it sets $q = False$ and outputs $(, q)$.*

Otherwise, it sets $q = True$, finds the solutions, and it outputs a vector $\boldsymbol{s}$ of solutions and q. When $q = False$, the rest of the algorithms will not be executed.

- $\underline{\mathsf{Prove}_{\mathsf{DTLP}}(pk, s_j) \rightarrow \pi_j}$. *A deterministic algorithm run by $\mathcal{TPH}'$. It takes as input pk and s_j. It outputs a proof π_j (asserting $m_j^* \in \mathcal{L}$).*
- $\underline{\mathsf{Register}_{\mathsf{DTLP}}(s_j, \pi_j, adr_{\mathcal{SC}}) \rightarrow t_j}$. *A deterministic algorithm run by $\mathcal{TPH}'$. It takes as input s_j, π_j, and $adr_{\mathcal{SC}}$. 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.*
- $\underline{\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j, t_0, \boldsymbol{\Delta}, \Upsilon) \rightarrow a_j}$. *A deterministic algorithm run by $\mathcal{SC}$. It takes as input pk, j, m_j^*, π_j, $g_j \in \boldsymbol{g}$, Ψ_j, t_j, t_0, $\boldsymbol{\Delta}$, and Υ. It checks whether (i) proof π_j is valid and (ii) j-th solution was delivered on time. If both checks pass, it outputs $a_j = 1$; otherwise, it outputs $a_j = 0$.*
- $\underline{\mathsf{Pay}_{\mathsf{DTLP}}(a_j, adr_{\mathcal{TPH}'}, \boldsymbol{coins}, j) \rightarrow b_j}$. *It is a deterministic algorithm run by $\mathcal{SC}$. It takes as input the result of verification a_j, $adr_{\mathcal{TPH}'}$, $\boldsymbol{coins}$, and j. If $a_j = 1$, then it sends $coins_j$ coins (where $coins_j \in \boldsymbol{coins}$) to an account with address $adr_{\mathcal{TPH}'}$ and sets $b_j = 1$; otherwise (i.e., $a_j = 0$), it sets $b_j = 0$. It outputs b_j.*
- $\underline{\mathsf{Retrieve}_{\mathsf{DTLP}}(pk, csk, m_j^*) \rightarrow m_j}$. *A deterministic algorithm run by $\mathcal{S}$. It takes as input pk, csk, and m_j^*. It retrieves message m_j from m_j^* and outputs m_j.*

* *Completeness. for honest $\mathcal{C}$, $\mathcal{S}$, $\mathcal{SC}$, $\mathcal{TPH}$, and $\mathcal{TPH}'$, it always holds that:*

- $\mathsf{Solve}_{\mathsf{DTLP}}(\boldsymbol{\Psi}, aux, pk, [p_1, ..., p_j], adr_{\mathcal{SC}}, coins', coins) = ([s_1, ..., s_j], True)$, *for every j, $1 \leq j \leq z$.*
- $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j, t_0, \boldsymbol{\Delta}, \Upsilon) \rightarrow a_j = 1$.
- $\mathsf{Pay}_{\mathsf{DTLP}}(a_j, adr_{\mathcal{TPH}'}, \boldsymbol{coins}, j) \rightarrow b_j = 1$.
- $\mathsf{Retrieve}_{\mathsf{DTLP}}(pk, csk, m_j^*) \rightarrow m_j$, *where $m_j \in \boldsymbol{m}$.*

* *Efficiency. the run-time of algorithm* $\mathsf{Solve}_{\mathsf{DTLP}}(\boldsymbol{\Psi}, aux, pk, [p_1, ..., p_j], adr_{\mathcal{SC}}, coins', coins) = ([s_1, ..., s_j], True)$ *is bounded by: $poly(\sum_{i=1}^{j} \bar{\Delta}_i, \lambda)$, where $poly(.)$ is a fixed polynomial and $1 \leq j \leq z$.*

D-TLP's definition, similar to GM-TLP, supports a solution's privacy and validity, with a main difference. Now, privacy also requires plaintext solutions $[m_1, ..., m_z]$ to remain hidden from $\mathcal{TPH}$ and $\mathcal{TPH}'$ (in addition to requiring the solution to remain hidden from the adversaries that have access to puzzles and parallel processors).

Therefore, in case 1 in Definition 9, we state that given the algorithms' transcripts, an adversary (which initially picks a pair of plaintext messages) cannot tell which message is used for the puzzle with a probability significantly greater than $\frac{1}{2}$. Moreover, 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 $\boldsymbol{\Delta} = [\bar{\Delta}_1, ..., \bar{\Delta}_z]$, any number of puzzles: $z \geq 1$, any j (where $1 \leq j \leq z$) the following hold:*

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

$$Pr\left[\begin{array}{l}\mathcal{A}_1(pk, cpk, \boldsymbol{m}, \\ \hat{\boldsymbol{m}}^*, \textit{state}, t_0, \\ \texttt{CEDG}, ToC, aux, \\ \boldsymbol{\Psi}, \boldsymbol{\Delta}, \boldsymbol{s}, \hat{p}, \\ \boldsymbol{coins}, adr_{\mathcal{TPH}'}, \\ adr_{\mathcal{SC}}) \rightarrow a \\ \textit{s.t.} \\ a := (b_i, i) \\ m_{b_i,i} = m_{b_j,j}\end{array} : \begin{array}{l}\mathsf{C.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}) \rightarrow (cpk, csk) \\ \mathcal{A}_1(1^\lambda, cpk, z) \rightarrow (\boldsymbol{m}, \textit{state}) \\ \forall j', 1 \leq j' \leq z : b_{j'} \xleftarrow{\$} \{0,1\} \\ \mathsf{C.Delegate}_{\mathsf{DTLP}}(\hat{\boldsymbol{m}}, cpk, csk) \rightarrow (\hat{\boldsymbol{m}}^*, t_0) \\ \mathsf{S.Delegate}_{\mathsf{DTLP}}(\texttt{CEDG}, ToC, S, \boldsymbol{\Delta}, aux, \\ adr_{\mathcal{TPH}'}, t_0, \Upsilon, \boldsymbol{coins}) \rightarrow adr_{\mathcal{SC}} \\ \mathsf{TPH.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk) \\ \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\hat{\boldsymbol{m}}^*, cpk, pk, sk, t_0) \rightarrow \hat{p} \\ \mathsf{Solve}_{\mathsf{DTLP}}(\boldsymbol{\Psi}, aux, pk, \boldsymbol{p}, \\ adr_{\mathcal{SC}}, coins', coins) \rightarrow (\boldsymbol{s}, q)\end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

2. *For any pair of randomised algorithm $\mathcal{A} := (\mathcal{A}_2, \mathcal{A}_3)$, 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 $\xi(Max(\bar{\Delta}_1, ..., \bar{\Delta}_j))$ parallel processors, there exists a negligible function $\mu(.)$, such that:*

$$Pr\left[\begin{array}{l}\mathcal{A}_3(pk, cpk, \hat{p}, \\ \textit{state}, \boldsymbol{\Psi}, \boldsymbol{\Delta}, \\ adr_{\mathcal{TPH}'}, adr_{\mathcal{SC}}, \\ aux, \texttt{CEDG}, \\ ToC, t_0) \rightarrow a \\ \textit{s.t.} \\ a := (b_i, i) \\ m^*_{b_i,i} = m^*_{b_j,j}\end{array} : \begin{array}{l}\mathsf{C.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}) \rightarrow (cpk, csk) \\ \mathcal{A}_2(1^\lambda, cpk, csk, z) \rightarrow (\boldsymbol{m}, \boldsymbol{m}^*, \textit{state}) \\ \mathsf{S.Delegate}_{\mathsf{DTLP}}(\texttt{CEDG}, ToC, S, \\ \boldsymbol{\Delta}, aux, adr_{\mathcal{TPH}'}, t_0, \Upsilon, \boldsymbol{coins}) \rightarrow adr_{\mathcal{SC}} \\ \mathsf{TPH.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk) \\ \forall j', 1 \leq j' \leq z : b_{j'} \xleftarrow{\$} \{0,1\} \\ \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\hat{\boldsymbol{m}}^*, cpk, pk, sk, t_0) \rightarrow \hat{p}\end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

where $1 \leq i \leq z$, $\boldsymbol{m} = [(m_{0,1}, m_{1,1}), ..., (m_{0,z}, m_{1,z})]$, $\hat{\boldsymbol{m}} = [m_{b_1,1}, ..., m_{b_z,z}]$, $\boldsymbol{m}^* = [(m^*_{0,1}, m^*_{1,1}), ..., (m^*_{0,z}, m^*_{1,z})]$, and $\hat{\boldsymbol{m}}^* = [m^*_{b_1,1}, ..., m^*_{b_z,z}]$.

Intuitively, solution validity requires that a prover cannot persuade a verifier (i) to accept a solution that is not equal to the encoded solution m^*_j or (ii) 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 $\boldsymbol{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, any number of puzzles: $z \geq 1$, all PPT adversaries $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$ that run in time $O(poly(\sum_{i=1}^{z} \bar{\Delta}_i, \lambda))$ there is a negligible function $\mu(.)$, such that:*

$$Pr\left[\begin{array}{l|l} \mathcal{A}_2(pk, cpk, \boldsymbol{s}, & \\ \hat{p}, \textit{state}, csk, & \mathsf{C.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}) \rightarrow (cpk, csk) \\ \boldsymbol{\Psi}, \boldsymbol{\Delta}, aux, & \mathcal{A}_1(1^\lambda, pk, \boldsymbol{\Delta}, z) \rightarrow (\boldsymbol{m}, \boldsymbol{m}^*, \textit{state}) \\ \boldsymbol{coins}, adr_{\mathcal{SC}}, & \mathsf{S.Delegate}_{\mathsf{DTLP}}(\texttt{CEDG}, ToC, S, \boldsymbol{\Delta}, \\ adr_{\mathcal{TPH}'}, \texttt{CEDG}, & aux, adr_{\mathcal{TPH}'}, t_0, \Upsilon, \boldsymbol{coins}) \rightarrow adr_{\mathcal{SC}} \\ ToC, t_0) \rightarrow (a, t_j, t'_j) : & \mathsf{TPH.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk) \\ \textit{s.t.} & \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\boldsymbol{m}^*, cpk, pk, sk, t_0) \rightarrow \hat{p} \\ (con_1 \wedge con_2 \wedge con_3) & \mathsf{Solve}_{\mathsf{DTLP}}(\boldsymbol{\Psi}, aux, pk, \\ \vee & \boldsymbol{p}, adr_{\mathcal{SC}}, coins', coins) \rightarrow (\boldsymbol{s}, q) \\ (con_4 \wedge con_5 \wedge con_6) & \end{array}\right] \leq \mu(\lambda)$$

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

- con_1 : *generates an invalid proof, i.e., sets $a := (j, \pi', m^*_j)$, s.t., $m^*_j \notin \mathcal{L}$ w.r.t. proof π'.*
- con_2 : *j-th solution was delivered on time, i.e., $t_j - t_0 \leq \sum_{i=1}^{j}(\bar{\Delta}_i + \Psi_i + \Upsilon)$.*
- con_3 : *new proof π' is accepted, i.e., $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m^*_j, \pi', g_j, \Psi_j, t_j, t_0, \boldsymbol{\Delta}, \Upsilon) = 1$. i.e., sets $a := (j, \pi', m^*_j)$, s.t., $m^*_j \notin \mathcal{L}$ w.r.t. proof π'.*
- con_4 : *generates a valid proof, i.e., sets $a := (j, \pi_j, m^*_j)$, s.t., $m^*_j \in \mathcal{L}$ w.r.t. proof π_j.*
- con_5 : *j-th solution was not delivered on time, i.e., $t'_j - t_0 > \sum_{i=1}^{j}(\bar{\Delta}_i + \Psi_i + \Upsilon)$.*
- con_6 : *a valid proof is accepted, even though it was not delivered on time, i.e., $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m^*_j, \pi_j, g_j, \Psi_j, t'_j, t_0, \boldsymbol{\Delta}, \Upsilon) = 1$.*

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

Definition 11 (Fair Payment). *A D-TLP scheme supports fair payment, if for every j (where $1 \leq j \leq z$) and any PPT adversary $\mathcal{A}$, there exists a negligible function $\mu(.)$, such that:*

$$Pr\left[\begin{array}{l|l} & \mathsf{C.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}) \rightarrow (cpk, csk) \\ & \mathcal{A}(1^\lambda, cpk, z) \rightarrow (\boldsymbol{m}, \textit{state}) \\ & \mathsf{C.Delegate}_{\mathsf{DTLP}}(\boldsymbol{m}, cpk, csk) \rightarrow (\boldsymbol{m}^*, t_0) \\ & \mathsf{S.Delegate}_{\mathsf{DTLP}}(\texttt{CEDG}, ToC, S, \boldsymbol{\Delta}, aux, \\ & adr_{\mathcal{TPH}'}, t_0, \Upsilon, \boldsymbol{coins}) \rightarrow adr_{\mathcal{SC}} \\ a_j \neq b_j : & \mathsf{TPH.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk) \\ & \mathsf{GenPuzzle}_{\mathsf{DTLP}}(\boldsymbol{m}^*, cpk, pk, sk, t_0) \rightarrow \hat{p} \\ & \mathcal{A}(cpk, \boldsymbol{m}, \textit{state}, t_0, \texttt{CEDG}, ToC, aux, \boldsymbol{\Psi}, \\ & \boldsymbol{\Delta}, coins, adr_{\mathcal{SC}}, adr_{\mathcal{TPH}'}, pk, \boldsymbol{p}) \rightarrow (s_j, t_j, \pi_j) \\ & \mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m^*_j, \pi_j, g_j, \Psi_j, t_j, t_0, \boldsymbol{\Delta}, \Upsilon) \rightarrow a_j \\ & \mathsf{Pay}_{\mathsf{DTLP}}(a_j, adr_{\mathcal{TPH}'}, \boldsymbol{coins}, j) \rightarrow b_j \end{array}\right] \leq \mu(\lambda)$$

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.*

6.2 Protocol: Efficient Delegated Time-locked Puzzle

In this section, we present Efficient Delegated Time-locked Puzzle (ED-TLP) which is a concrete instantiation of D-TLP that we defined in Section 6.1.

Satisfying the Primary Features. Before explaining the main construction, we briefly explain how ED-TLP satisfies the primary properties.

- *Varied Size Time Intervals and Efficiently Handling Multiple Puzzles.* ED-TLP treats the algorithms of the GC-TLP in a block-box manner. Therefore, ED-TLP inherits GC-TLP's appealing features including the support of varied size time intervals and efficiently dealing with multiple puzzles at once.
- *Privacy.* To ensure plaintext solutions' privacy, $\mathcal{C}$ encrypts all plaintext solutions using symmetric-key encryption and asks $\mathcal{TPH}$ to treat the ciphertexts as puzzles' solutions. $\mathcal{C}$ sends the secret key of the encryption only to $\mathcal{S}$. This approach protects the privacy of plaintext messages from both $\mathcal{TPH}$ and $\mathcal{TPH}'$.
- *Exact-time Recovery of Solutions.* To allow $\mathcal{S}$ to determine exactly when $\mathcal{TPH}'$ will find a solution, ED-TLP enables $\mathcal{S}$ (given the exact computational power of $\mathcal{TPH}'$) to use predicate `CEDG` (.) to calculate the extra time that $\mathcal{TPH}'$ requires to find each j-th solution. Thus, instead of assuming that $\mathcal{TPH}'$ is in the ideal situation and possesses the highest level of computing resources that would allow it to perform the maximum number of squarings S per second and find a solution on time, $\mathcal{S}$ considers the available level of resources to $\mathcal{TPH}'$ (possibly much lower than it would have been in the ideal case) and gives $\mathcal{TPH}'$ extra time to find a solution.
- *Timely Delivery of Solutions and Fair Payment.* To ensure the timely *delivery* of a solution, $\mathcal{S}$ constructs a smart contract and specifies in it exactly when each solution must be delivered, based on the output of `CEDG` (.). $\mathcal{S}$ also deposits a certain amount of coins that $\mathcal{TPH}'$ will receive if it provides to the smart contract a valid (encrypted) solution on time. $\mathcal{S}$ requires the smart contract (1) to check whether j-th (encrypted) solution is valid and delivered on time and (2) to pay $\mathcal{TPH}'$ if the two checks pass. Thus, this approach gives $\mathcal{TPH}'$ the assurance that it will get paid if and only if it provides a valid solution on time. The $\mathcal{SC}$-side verification imposes a low computation cost as it involves the invocation of a hash function a couple of times per solution.

An Overview of ED-TLP. At a high level, ED-TLP works as follows. At the setup, $\mathcal{C}$ encrypts all the plaintext solutions using symmetric-key encryption under a secret key, csk. It sends csk to $\mathcal{S}$ which determines the extra time that $\mathcal{TPH}'$ needs, to find each j-th solution, with the help of predicate `CEDG` (.). Next, $\mathcal{S}$ constructs a smart contract $\mathcal{SC}$ and specifies in it the expected delivery time for each solution. It deploys $\mathcal{SC}$ to the blockchain and deposits enough coins for z valid solutions to $\mathcal{SC}$.

Moreover, $\mathcal{C}$ sends the ciphertexts to $\mathcal{TPH}$ which (i) generates all required secret and public keys on $\mathcal{C}$'s behalf and (ii) builds puzzles on the ciphertexts (instead of the plaintext solutions in GC-TLP). It sends all puzzles to $\mathcal{TPH}'$ which checks the deposit

and the parameters in $\mathcal{SC}$. If $\mathcal{TPH}'$ agrees to proceed, then it solves each puzzle and generates a proof asserting the correctness of the solution. It sends the solution and proof to $\mathcal{SC}$ which checks whether (i) the solution-proof pair has been delivered on time and (ii) the solution is valid with the help of the proof. If the two checks pass, then $\mathcal{SC}$ sends a portion of the deposit to $\mathcal{TPH}'$. Given a valid encrypted solution stored in $\mathcal{SC}$ and secret key psk, $\mathcal{S}$ locally decrypts the ciphertext to retrieve a plaintext solution.

Detailed Description of ED-TLP. Below, we present ED-TLP in detail.

1. Client-side Setup. $\mathsf{C.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}) \rightarrow (cpk, csk)$
   This phase involves the client, $\mathcal{C}$.
   - call $\mathtt{SKE.keyGen}(1^\lambda) \rightarrow csk$, to generate a secret key, for symmetric-key encryption.
   - set $cpk = \boldsymbol{\Delta}$ and sends it do $\mathcal{S}$.
2. Client-side Delegation. $\mathsf{C.Delegate}_{\mathsf{DTLP}}(\boldsymbol{m}, csk) \rightarrow (\boldsymbol{m}^*, t_0)$
   This phase involves $\mathcal{C}$.
   (a) encrypt each element of vector $\boldsymbol{m}$ consisting of plaintext solutions as follows. $\forall i, 1 \leq i \leq z : \mathsf{Enc}(csk, m_i) \rightarrow m_i^*$.
   (b) send vector $\boldsymbol{m}^* = [m_1^*, .., m_z^*]$ to $\mathcal{TPH}$ and value t_0 to $\mathcal{TPH}$ and $\mathcal{S}$.
3. Server-side Delegation. $\mathsf{S.Delegate}_{\mathsf{DTLP}}(\mathtt{CEDG}, ToC, S, \boldsymbol{\Delta}, aux, adr_{\mathcal{TPH}'}, t_0, \Upsilon, \boldsymbol{coins}) \rightarrow adr_{\mathcal{SC}}$
   This phase involves $\mathcal{S}$.
   (a) determine the extra delay Ψ_j that $\mathcal{TPH}'$ will have, to find every j-th solution, where $1 \leq j \leq z$. To do that, call $\mathtt{CEDG}(ToC, S, \bar{\Delta}_j, aux) \rightarrow \Psi_j$.
   (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 $coins_j \in \boldsymbol{coins}$. Let $adr_{\mathcal{SC}}$ be the address of the deployed $\mathcal{SC}$.
   (c) set the delivery time for j-th solution as $T_j = t_0 + \sum_{i=1}^{j}(\bar{\Delta}_i + \Psi_i + \Upsilon)$ for all j where $1 \leq j \leq z$.
   (d) register $\boldsymbol{T} = [T_1, ..., T_z]$ and $adr_{\mathcal{TPH}'}$ in $\mathcal{SC}$.
   (e) send $adr_{\mathcal{SC}}$ to $\mathcal{TPH}'$.
4. Helper-side Setup. $\mathsf{TPH.Setup}_{\mathsf{DTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk)$
   This phase involves the third-party helper, $\mathcal{TPH}$.
   - call $\mathsf{Setup}_{\mathsf{GMTLP}}(1^\lambda, \boldsymbol{\Delta}, S, z) \rightarrow (pk, sk)$, to generate public and private parameters for z puzzles.
5. Helper-side Puzzle Generation. $\mathsf{GenPuzzle}_{\mathsf{DTLP}}(\boldsymbol{m}^*, pk, sk, t_0) \rightarrow \hat{p}$
   This phase involves $\mathcal{TPH}$.
   (a) call $\mathsf{GenPuzzle}_{\mathsf{GMTLP}}(\boldsymbol{m}^*, pk, sk) \rightarrow \hat{p}$, to generate z puzzles and their commitments. Recall, $\hat{p} := (\boldsymbol{p}, \boldsymbol{g})$, where $\boldsymbol{p}$ is a vector of puzzles and $\boldsymbol{g}$ is a vector of commitments.
   (b) at time t_0, send $\boldsymbol{p}$ to $\mathcal{TPH}'$ and $\boldsymbol{g}$ to $\mathcal{SC}$.
6. Solve Puzzle. $\mathsf{Solve}_{\mathsf{DTLP}}(\boldsymbol{\Psi}, aux, pk, \boldsymbol{p}, adr_{\mathcal{SC}}, coins', coins) \rightarrow (\boldsymbol{s}, q)$
   This phase involves $\mathcal{TPH}'$.

(a) check the amount of deposit: $coins$ and elements of vector $\boldsymbol{\Psi}$ in $\mathcal{SC}$. If the amount of deposit is less than its expectation (i.e., $coins < coins'$) or elements of $\boldsymbol{\Psi}$ are not large enough (do not match aux's parameters), set $q = False$ and halt.
(b) call $\mathsf{Solve}_{\mathsf{GMTLP}}(pk, \boldsymbol{p}) \rightarrow (\boldsymbol{s}, \boldsymbol{m}^*)$, to solve z puzzles.

7. Prove. $\mathsf{Prove}_{\mathsf{DTLP}}(pk, s_j) \rightarrow \pi_j$
This phase involves $\mathcal{TPH}'$.
- call $\mathsf{Prove}_{\mathsf{GMTLP}}(pk, s_j) \rightarrow \pi_j$, to generate a proof, upon discovering a solution $s_j \in \boldsymbol{s}$.

8. Register Puzzle. $\mathsf{Register}_{\mathsf{DTLP}}(s_j, \pi_j, adr_{\mathcal{SC}}) \rightarrow t_j$
This phase involves $\mathcal{TPH}'$.
- send j-th encoded message m_j^* (where $m_j^* \in s_j$) and its proof π_j to $\mathcal{SC}$, such that they can be registered in $\mathcal{SC}$ by time t_j.

9. Verify. $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, m_j^*, \pi_j, g_j, \Psi_j, t_j, t_0, \boldsymbol{\Delta}, \Upsilon) \rightarrow a_j$
This phase involves $\mathcal{SC}$.
(a) if this is the first time that it is invoked (when $j = 1$), set two vectors $\boldsymbol{a} = [a_1, ..., a_z]$ and $\boldsymbol{b} = [b_1, ..., b_z]$ whose elements are initially set to empty ϵ.
(b) read from $\mathcal{SC}$'s state and check whether pair (s_j, π_j) was delivered to $\mathcal{SC}$ on time, by ensuring:

$$t_j \leq T_j = t_0 + \sum_{i=1}^{j} (\bar{\Delta}_i + \Psi_i + \Upsilon)$$

(c) call $\mathsf{Verify}_{\mathsf{GMTLP}}(pk, j, m_j^*, \pi_j, g_j) \rightarrow a'_j \in \{0, 1\}$, to check a proof's validity, where $g_j \in \boldsymbol{g}$.
(d) set $a_j = 1$, if both checks in steps 9b and 9c pass; set $a_j = 0$, otherwise.

10. Pay. $\mathsf{Pay}_{\mathsf{DTLP}}(a_j, adr_{\mathcal{TPH}'}, \boldsymbol{coins}, j) \rightarrow b_j$
This phase involves $\mathcal{SC}$. It can be invoked by $\mathcal{C}$ or $\mathcal{S}$.
- if $a_j = 1$, then:
  (a) if $b_j \neq 1$, for j-th puzzle, send $coins_j$ coins to $\mathcal{TPH}'$, where $coins_j \in \boldsymbol{coins}$.
  (b) set $b_j = 1$. This ensures one will not be paid multiple times for a solution.
- if $a_j = 0$, then send $coins_j$ coins back to $\mathcal{S}$ and set $b_j = 0$.

11. Retrieve. $\mathsf{Retrieve}_{\mathsf{DTLP}}(pk, csk, m_j^*) \rightarrow m_j$
This phase involves $\mathcal{S}$.
(a) read m_j^* from $\mathcal{SC}$.
(b) locally decrypt m_j^* as: $\mathsf{Dec}(csk, m_j^*) \rightarrow m_j$.

In the ED-TLP (and definition D-TLP) we assumed $\mathcal{TPH}'$ is a rational adversary, to only ensure the timely delivery of the solution through our incentivisation mechanism. However, the privacy of the scheme holds even against a fully malicious $\mathcal{TPH}'$.

Theorem 5. *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 [16], and the underlying signature satisfies "existential unforgeability under chosen message attacks") and the smart contract's correctness holds, then ED-TLP is secure, w.r.t. Definition 12.*

7 Delegated VDF

In this section, we will briefly discuss how the Verifiable Delay Function (VDF) research line can benefit from the ideas behind D-TLP. At a high level, VDF involves three algorithms (i) $\texttt{Setup}(.)$ that returns system parameters, (ii) $\texttt{Eval}(.)$ that returns a value y and proof that the value has been constructed correctly, and (iii) $\texttt{Verify}(.)$ that returns 1 if the proof is valid and returns 0 otherwise.

By definition, the execution of $\texttt{Eval}(.)$ demands considerable computation resources for a predefined period of time (that can be long). Therefore, a thin client with limited resources may not be able to satisfy the resource requirements demanded by $\texttt{Eval}(.)$. In this case, following the ideas behind D-TLP, the client can delegate the execution of $\texttt{Eval}(.)$ to a third-party helper. The helper produces the result y and proves to the smart contract that it has computed y correctly. Subsequently, the contract pays the helper if the helper provides *valid proof on time*.

Although the original schemes presented by Wesolowski [31] and Pietrzak [25] required interactions between the verifier and the prover, they demonstrated that the verification could be made non-interactive based on Fiat-Shamir heuristic [14]. Consequently, in these schemes, a smart contract can act as the verifier, aligning with the concept of delegation.

8 Security Analysis of ED-TLP

In this section, we prove the security of ED-TLP, i.e., Theorem 5.

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 $\mathcal{TPH}$ and $\mathcal{TPH}'$, w.r.t. Case 1 in Definition 9.*

Proof. The messages that $\mathcal{TPH}$ receives include a vector $\boldsymbol{m}^*$ of ciphertexts (i.e., encrypted plaintext solutions), and public parameters in set $A = \{cpk, \texttt{CEDG}, ToC, S, \boldsymbol{\Delta}, aux, coins, t_0, adr_{\mathcal{TPH}'}, adr_{\mathcal{SC}}\}$. Other parameters that are given to $\mathcal{A}_1$ in the experiment (i.e., parameters in set $B = \{pk, \boldsymbol{s}, \hat{p}\}$) are generated by $\mathcal{TPH}$ itself and may seem redundant. However, we have given these parameters to $\mathcal{A}_1$ to cover the case where $\mathcal{TPH}'$ 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. Also, due to the semantic security of the used symmetric-key encryption (i.e., meets IND-CPA), the vector $\boldsymbol{m}^*$ of ciphertext reveals no information about each m_i. Specifically, in the experiment (i.e., Case 1 in Definition 9), the probability that $\mathcal{A}_1$ can tell whether a ciphertext $m^*_{b_i,i} \in \boldsymbol{m}^*$ is an encryption of message $m_{0,i}$ or $m_{1,i}$, both of which were initially chosen by $\mathcal{A}_1$, is at most $\frac{1}{2} + \mu(\lambda)$.

Now, we turn our attention to the case where $\mathcal{TPH}'$ is corrupt. The messages that $\mathcal{TPH}'$ receives include a vector $\boldsymbol{m}^*$ of ciphertexts and parameters in set $A+B$. As long as the symmetric-key encryption meets IND-CPA, the vector $\boldsymbol{m}^*$ of ciphertext reveals no information about each m_i. Hence, when $\mathcal{TPH}'$ is corrupt, the probability that $\mathcal{A}_1$

(in Case 1 in Definition 9) can tell whether a ciphertext $m^*_{b_i,i} \in \boldsymbol{m}^*$ 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 $\frac{1}{2} + \mu(\lambda)$.

Furthermore, as we discussed above, the public parameters in A were generated independently of the plaintext solutions $m_1, \ldots, m_z$, they do not reveal anything about each m_i. The same holds for the public parameter $pk \in B$. Moreover, parameters $\boldsymbol{s}$ and $\hat{p}$ in B have been generated by executing algorithms $\mathsf{GenPuzzle}_{\mathsf{DTLP}}$ and $\mathsf{Solve}_{\mathsf{DTLP}}$ on the ciphertexts in vector $\boldsymbol{m}^*$. Thus, $\boldsymbol{s}$ 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 $\mathcal{TPH}'$, w.r.t. Case 2 in Definition 9.*

Proof. Before solving the puzzles, the messages that $\mathcal{TPH}'$ or $\mathcal{S}$ receives include the elements of sets $A = \{pk, \hat{p}\}$ and $B = \{\boldsymbol{\Psi}, \texttt{CEDG}, t_0, coins, adr_{\mathcal{TPH}'}, adr_{\mathcal{SC}}, ToC, aux\}$. The elements of set A are identical to what $\mathcal{S}$ receives in GC-TLP. Moreover, the elements of set B are independent of the plaintext solutions and the puzzles' secret and public parameters. Therefore, the knowledge of B does not help $\mathcal{S}$ or $\mathcal{TPH}'$ learn the plaintext solutions before solving the puzzles. More formally, given $A + B$, in the experiment (Case 2 in Definition 9), the probability that $\mathcal{A}_3$ can tell whether a puzzle $p_{b_i,i}$ (where $p_{b_i,i} \in \boldsymbol{p} \in \hat{p}$) 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 2. ■

Now, we move on to the solution's validity, w.r.t. Definition 10.

Claim 3 *If GC-TLP satisfies solution-validity (w.r.t. Definition 5), the blockchain is secure (i.e., it meets persistence and liveness properties [16], and the signature satisfies existential unforgeability under chosen message attacks) and the smart contract's correctness holds, then ED-TLP preserves a solution validity, w.r.t. Definition 10.*

First, we focus on event $\mathrm{I} = (con_1 \wedge con_2 \wedge con_3)$. It considers the case where a prover submits proof on time and passes the verification even though the proof contains 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 $\mathcal{TPH}'$) learns in ED-TLP includes parameters in set $A + B$. Nevertheless, these parameters are independent of the plaintext messages and those parameters that are used for the commitment. Therefore, the solution's validity is reduced to the solution validity of GC-TLP; i.e., to the security of the hash-based commitment scheme. Specifically, given the commitment $g_j = \mathtt{G}(m_j, d_j)$ and an opening $\pi := (m_j, d_j)$, for an adversary to break the solution validity, it must generate (m'_j, d'_j), such that $\mathtt{G}(m'_j, d'_j) = g_j$, where $m_j \neq m'_j$. However, this is infeasible for a PPT adversary, as $\mathtt{G}(.)$ 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 turn our attention to event $\mathrm{II} = (con_4 \wedge con_5 \wedge con_6)$ which captures the case where the adversary has generated a valid proof and passed the verification despite it registered 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). Also, 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; therefore, 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, $\mathsf{Verify}_{\mathsf{DTLP}}(.)$ outputs 0 except for a negligible probability, $\mu(\lambda)$. Thus, 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 (i.e., it meets persistence and liveness properties [16]), then ED-TLP offers fair payment, w.r.t. Definition 11.*

The proof boils down to the security of the blockchain (and smart contracts). Specifically, due to the persistence and liveness properties, (i) the state of a smart contract cannot be tampered with and (ii) 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 $\mathsf{Verify}_{\mathsf{DTLP}}(pk, j, \pi_j, g_j, \Psi_j, t_j, t_0, \boldsymbol{\Delta}, \Upsilon) \rightarrow a_j \in \{0, 1\}$, then (1) the intact a_j is passed on to $\mathsf{Pay}_{\mathsf{DTLP}}(a_j, adr_{\mathcal{TPH}'}, \boldsymbol{coins}, j)$ as input (because the smart contract generates and maintains a_j and passes it to $\mathsf{Verify}_{\mathsf{DTLP}}$) and (2) $b_j = a_j$, 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). □

9 Evaluation

In this section, we analyse the overheads of GC-TLP and ED-TLP. We also compare their costs with the original RSA-based TLP of Rivest *et al.* [27] and the C-TLP of Abadi and Kiayias [1]. Table 2 summarises the costs comparison between the four schemes. In our cost evaluation, for the sake of simplicity, we do not include the output of $\mathtt{CEDG}(.)$, as the output is a fixed value and remains the same for all the schemes we analyse in this section. Table 3 compares the features of the four schemes.

9.1 Features

Verification. TLP in [27] does not support verification, whereas the C-TLP, our GC-TLP, and ED-TLP efficiently support that, in the sense that they allow a third party to check whether a solver has honestly solved every puzzle.

Delegation. TLP, C-TLP, and GC-TLP do not support secure delegations, where the client and the server could delegate their resource-demanding tasks to third-party helpers who can potentially be passive (or even rational) adversaries. Nevertheless, our ED-TLP can efficiently do so while ensuring that the third-party helpers will learn no information about the plaintext solutions while inheriting the appealing features of C-TLP and GC-TLP. Our ED-TLP also offers timely delivery of solutions and fair payments.

Table 2. Asymptotic costs comparison. 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 GC-TLP and ED-TLP. $\bar{\Delta}_i < \Delta_i$ when $i > 1$.

Scheme	Operation	Algorithms Complexity										Total Complexity	
		Setup		Delegate		GenPuzzle		Solve		Verify	Retrieve	Comp.	Comm.
		$\mathcal{C}$	$\mathcal{TPH}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{C}$	$\mathcal{TPH}$	$\mathcal{S}$	$\mathcal{TPH}'$				
Our ED-TLP	Exp.	–	$O(z)$	–	–	–	$O(z)$	–	$O(S\sum_{i=1}^{z}\bar{\Delta}_i)$	–	–	$O(S\sum_{i=1}^{z}\bar{\Delta}_i)$	$O(z)$
	Add./Mul.	–	–	–	$O(z)$	–	$O(z)$	–	$O(z)$	–	–		
	Hash	–	–	–	–	–	$O(z)$	–	–	$O(z)$	–		
	Sym. Enc	–	–	$O(z)$	–	$O(z)$	$O(z)$	–	$O(z)$	–	$O(z)$		
Our GC-TLP	Exp.	$O(z)$				$O(z)$		$O(S\sum_{i=1}^{z}\bar{\Delta}_i)$		–		$O(S\sum_{i=1}^{z}\bar{\Delta}_i)$	$O(z)$
	Add./Mul.	–				$O(z)$		$O(z)$		–			
	Hash	–				$O(z)$		–		$O(z)$			
	Sym. Enc	–				$O(z)$		$O(z)$		–			
C-TLP in [1]	Exp.	1				$O(z)$		$O(zS\Delta)$		–		$O(zS\Delta)$	$O(z)$
	Add./Mul.	–				$O(z)$		$O(z)$		–			
	Hash	—				$O(z)$		–		$O(z)$			
	Sym. Enc	–				$O(z)$		$O(z)$		–			
TLP in [27]	Exp.	$O(z)$				$O(z)$		$O(S\sum_{i=1}^{z}\Delta_i)$				$O(S\sum_{i=1}^{z}\Delta_i)$	$O(z)$
	Add./Mul.	–				$O(z)$		$O(z)$					
	Hash	–				–		–					
	Sym. Enc	–				$O(z)$		$O(z)$					

Table 3. Comparison of the features offered by different schemes.

Schemes	Features					
	Multi-Puzzle	Varied-size Intervals	Verification	Delegation	Exact-time Solution Recovery	Fair Payment
Our ED-TLP	✓	✓	✓	✓	✓	✓
Our GC-TLP	✓	✓	✓	×	×	×
C-TLP in [1]	✓	×	✓	×	×	×
TLP in [27]	×	✓	×	×	×	×

Multi-Puzzle and Varied-size Time Intervals. The original TLP in [27] was not designed to efficiently support multiple puzzles, if it is used in the multi-puzzle setting, then it would demand a high level of computation and processors. However, C-LTP in [1] and our GC-TLP and ED-TLP can efficiently handle multi-puzzle.

In the case where TLP in [27] is used naively in the multi-puzzle setting, then it could support varied-size time intervals between the appearance of two consecutive messages, as each puzzle is generated independently of other puzzles. Nevertheless, this is not the case for C-TLP in [1], as it assumes that the time intervals have an identical size. Nevertheless, our GC-TLP and ED-TLP can support varied-size time intervals. Figure 1 illustrates differences between these four schemes in terms of supporting multi-puzzle and varied-size time intervals.

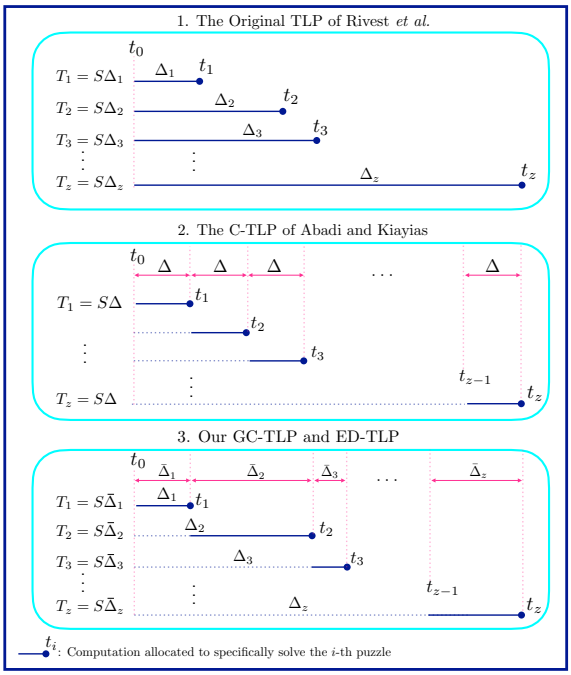

Fig. 1. Comparing time-lock puzzle schemes from supporting multi-puzzle and varied-size time intervals perspectives.

9.2 Asymptotic Cost Analysis

Computation Cost. Below, we analyse the computation complexity of each of the above four schemes

- *RSA-based Time-Lock Puzzle (TLP) of Rivest et al. [27].* Client $\mathcal{C}$ in the setup phase performs modular exponentiation (over $\bmod \phi(N)$) for each puzzle. Thus, for z puzzles it performs $O(z)$ modular exponentiation in this phase. $\mathcal{C}$ in the puzzle generation phase, for each puzzle, invokes an instance of symmetric-key encryption, performs a single modular exponentiation, and a modular addition. Thus, for z puzzles, $\mathcal{C}$ invokes each of the above three operations (i.e., symmetric-key encryption, exponentiation, and addition) $O(z)$ times. Server $\mathcal{S}$ in the puzzle-solving phase, to solve i-th puzzle (a) performs $T_i = S\Delta_i$ number of squaring modulo N, (b) performs a single modular subtraction, and (c) single symmetric-key decryption. Thus, for z puzzles, $\mathcal{S}$ performs $O(S\sum_{i=1}^{z}\Delta_i)$ modular squaring/exponentiation, and $O(z)$ modular subtraction and symmetric-key decryption.
- *Chained Time-Lock Puzzle (C-TLP) in [1].* $\mathcal{C}$ in the setup, performs single exponentiation (over $\bmod \phi(N)$) to generate parameters for z puzzles. In the puzzle generation phase, $\mathcal{C}$ invokes $O(z)$ instances of symmetric-key encryption, performs $O(z)$ modular exponentiation over $\mathbb{Z}_N$ and carries out $O(z)$ modular addition. In the same phase, $\mathcal{C}$ invokes $O(z)$ instances of the hash function. Thus, the total com-

putation complexity of $\mathcal{C}$ in the puzzle generation phase is $O(z)$. For $\mathcal{S}$ to solve z puzzles, it performs $O(zS\Delta)$ modular squaring over $\mathbb{Z}_N$, carries out $O(z)$ modular addition, and invokes $O(z)$ instances of symmetric-key decryption. Therefore, the total computation complexity of $\mathcal{S}$ is $O(zS\Delta)$. The verification cost mainly involves invoking $O(z)$ instances of the hash function.

- *Our GC-TLP*. $\mathcal{C}$ in the setup phase (Phase 1), performs z exponentiation (over $\mathrm{mod}\phi(N)$) to generate the parameters (i.e., $a_1, ..., a_z$) for z puzzles. In the generate puzzle phase (Phase 2), $\mathcal{C}$ invokes $O(z)$ instances of symmetric-key encryption, performs $O(z)$ modular exponentiation over $\mathbb{Z}_N$ and conducts $O(z)$ modular addition. In this phase, $\mathcal{C}$ invokes $O(z)$ instances of the hash function. So, its total computation complexity in the puzzle generation phase is $O(z)$. For $\mathcal{S}$ to solve z puzzles (in Phase 3), it performs $O(S\sum_{i=1}^{z}\bar{\Delta}_i)$ modular squaring over $\mathbb{Z}_N$, carries out $O(z)$ modular addition, and invokes $O(z)$ instances of symmetric-key decryption, where $\bar{\Delta}_i < \Delta_i$ when $i > 1$. Hence, $\mathcal{S}$'s total computation complexity is $O(S\sum_{i=1}^{z}\bar{\Delta}_i)$. The prove phase (Phase 4) imposes a negligible cost to $\mathcal{S}$. The verify phase (Phase 5) cost involves invoking $O(z)$ instances of the hash function.
- *Our ED-TLP*. $\mathcal{C}$'s cost in the setup phase (Phase 1) is negligible, as it involves generating a single secret key. $\mathcal{C}$ in the client-side delegation phase (Phase 2) invokes $O(z)$ instances of symmetric-key encryption. $\mathcal{TPH}$ in the helper-side setup phase (Phase 4), performs z exponentiation (over $\mathrm{mod}\ \phi(N)$) to generate the parameters for z puzzles. $\mathcal{TPH}$ in the helper-side puzzle generation phase (Phase 5), invokes $O(z)$ instances of symmetric-key encryption, performs $O(z)$ modular exponentiation over $\mathbb{Z}_N$ and carries out $O(z)$ modular addition. In this phase, $\mathcal{TPH}$ also invokes $O(z)$ instances of the hash function. $\mathcal{S}$'s computation cost in the delegation phase (Phase 3) mainly involves $O(z)$ additions. Its cost in the retrieve phase (Phase 11) mainly involves $O(z)$ invocations of the decryption algorithm of symmetric key encryption.

  The cost of $\mathcal{TPH}'$ in the solve puzzle phase (Phase 6) includes $O(S\sum_{i=1}^{z}\bar{\Delta}_i)$ modular exponentiation, $O(z)$ modular addition, and $O(z)$ symmetric key decryption. Its cost in the prove and register phases (Phases 7 and 8) is negligible. The cost of $\mathcal{SC}$ in the verify phase (Phase 9) involves $O(z)$ addition and $O(z)$ invocations of the hash function. Its cost in the pay phase (Phase 10) is negligible.

Hence, (a) ED-TLP has the lowest client-side setup cost, (b) ED-TLP has the lowest server-side cost, (c) TLP, C-TLP, our GC-TLP, and ED-TLP have the same computation complexity in the puzzle-solving phase (but the last three schemes impose much lower costs than TLP, in the multi-client setting), and (d) C-TLP, our GC-TLP, and ED-TLP have the same verification cost.

Communication Cost.

- *TLP of Rivest et al. [27]*. The total communication cost of the TLP is $O(z)$, as $\mathcal{C}$ for each puzzle sends to $\mathcal{S}$ two values, (i) an element of $\mathbb{Z}_N$ (whose size is about

2048 bits) and (ii) ciphertext of symmetric-key encryption (whose size is about 256 bits).

- *C-TLP in [1] and our GC-TLP*. In the C-TLP and GC-TLP, for each puzzle, $\mathcal{C}$ sends to $\mathcal{S}$ a commitment (whose size is about 256 bits), an element of $\mathbb{Z}_N$ and ciphertext of symmetric-key encryption. In the proving phase, $\mathcal{S}$ sends an opening of a commitment (with the total size of $256 + |m|$, where $|m|$ is the plaintext solution's size). Thus, the total communication cost for z puzzles in the C-TLP and GC-TLP is $O(z)$.
- *Our ED-TLP*. In the ED-TLP, $\mathcal{C}$ in the client-side delegation phase (Phase 2) for each puzzle sends a ciphertext of symmetric-key encryption to $\mathcal{TPH}$. $\mathcal{S}$ in the server-side delegation phase (Phase 3) deploys to the blockchain a smart contract $\mathcal{SC}$. It also sends to $\mathcal{SC}$: (1) z values $T_1, ..., T_z$, where each value is a few bits long, and (2) a single address $adr_{\mathcal{SC}}$ to $\mathcal{TPH}'$. $\mathcal{TPH}$ in the helper-side puzzle generation phase (Phase 5) sends to $\mathcal{TPH}'$ a vector $\boldsymbol{p}$ of z puzzles, where each puzzle is a pair containing an element of $\mathbb{Z}_N$ and a ciphertext of symmetric-key encryption. In the same phase, it sends to $\mathcal{SC}$ a vector $\boldsymbol{g}$ of z commitments. $\mathcal{TPH}'$ in the register puzzle phase (Phase 8) sends to $\mathcal{SC}$ a vector of pairs where each pair contains a ciphertext of symmetric-key encryption and a random value (of size 256-bit). Thus, ED-TLP's total communication cost in the z-puzzle setting is $O(z)$.

Hence, in the z-puzzle setting, all four schemes' overall communication complexity is $O(z)$, while ED-TLP has the lowest (i) client-side and (ii) server-side communication costs, because in this scheme $\mathcal{C}$ and $\mathcal{S}$ send messages of much shorter length than they do in the other three schemes.

9.3 Concrete Run-time Analysis

Setting the Parameters. To conduct a comprehensive evaluation and compare our protocols against previous work, we implemented not only our ED-TLP and GC-TLP but also the C-TLP from [1] and the TLP from [27]. Through a series of carefully designed benchmarks, we performed a concrete analysis of the run-time differences. For those interested, the Python 3 implementation has been made open-source and is available at [26]. It utilises the high-performance multiple precision integer library GMP [18] via the gmpy2 [19] Python module.

The implementation strictly adheres to the protocol descriptions for TLP, C-TLP, and GC-TLP. However, for ED-TLP, there are slight deviations since it does not directly interact with a blockchain-based smart contract. Instead, it interfaces with a minimal mock of the necessary functions found in a smart contract. In cases where protocols required a commitment scheme, we employed a SHA512 hash-based commitment with 128-bit witness/random values.

We performed the run-time evaluation on consumer hardware, a MacBook Pro 2021 equipped with an Apple M1 Pro CPU. To determine the number of repeated squaring operations that the CPU can perform per second (i.e., the concrete value of S in $T = S\Delta$) benchmarking was performed using two methods: timing the duration to execute a fixed number of operations and counting the operations performed within a fixed period. Both methods have been included in the open-source implementation and produced similar performance evaluations. The benchmarking results exhibit that the CPU

is capable of up to 2,260,000 repeated squaring operations per second in a 1024-bit RSA group and up to 845,000 operations per second in a 2048-bit group. These results align with the findings of previous work in [29].

Table 4. Concrete run time comparison. The benchmarks were run using 2048-bit moduli and the intervals are set to 1 second, except in TLP where puzzles are generated using cumulative durations to ensure puzzles cannot be solved earlier than equivalent multi-instance puzzles, even if solved out of order.

Scheme	Number of instances	Algorithms Run Time										
		Setup		Delegate		GenPuzzle		Solve		Verify	Retrieve	Total time
		$\mathcal{C}$	$\mathcal{TPH}$	$\mathcal{C}$	$\mathcal{S}$	$\mathcal{C}$	$\mathcal{TPH}$	$\mathcal{S}$	$\mathcal{TPH}'$			
Our ED-TLP	10	< 0.001	0.045	< 0.001	< 0.001	–	0.017	–	10.008	< 0.001	< 0.001	10.071
	100	< 0.001	0.049	0.002	< 0.001	–	0.174	–	101.248	< 0.001	0.002	101.476
	1000	< 0.001	0.076	0.020	< 0.001	–	1.743	–	1030.211	0.004	0.021	1032.117
	10000	< 0.001	0.358	0.202	0.001	–	17.650	–	10312.387	0.042	0.203	10331.043
Our GC-TLP	10	0.046				0.018		10.180		< 0.001		10.243
	100	0.049				0.181		101.211		< 0.001		101.441
	1000	0.076				1.777		1016.077		0.004		1017.936
	10000	0.353				17.851		10304.015		0.039		10322.280
C-TLP in [1]	10	0.046				0.019		10.086		< 0.001		10.151
	100	0.046				0.176		102.197		< 0.001		102.420
	1000	0.057				1.741		1015.815		0.004		1017.616
	10000	0.156				17.475		10316.014		0.039		10334.022
TLP in [27]	10	0.215				0.019		55.162				55.396
	100	3.315				0.181		5055.681				5059.177
	1000	33.799				1.792		N/A				N/A
	10000	326.502				17.948		N/A				N/A

Run-time Comparison. As Table 4 demonstrates, ED-TLP always imposes the lowest costs to $\mathcal{C}$ (in total in the Setup, Delegate, and GenPuzzle phases). For instance, when $z = 100$, then ED-TLP outperforms:

- GC-TLP by a factor of 76.
- C-TLP by a factor of 74.
- TLP by a factor of 1165.

The above factors grow in ED-TLP's favour when the number of puzzles increases. For instance, when $z = 1000$, then ED-TLP outperforms (i) GC-TLP by a factor of 88, (ii) C-TLP by a factor of 85, and (iii) TLP by a factor of 1694.

Moreover, ED-TLP always imposes the lowest costs to $\mathcal{S}$ in the Solve phase. For example, when $z = 100$, then ED-TLP outperforms:

- GC-TLP by a factor of 101.
- C-TLP by a factor of 102.
- TLP by a factor of 5055.

The length of the time intervals does not appear to meaningfully affect the time required for the setup and puzzle generation algorithms. To better inform the discussion

on communication cost, we measure the size of the setup output of C-TLP and GC-TLP. Table 5 shows the breakdown and differences between the two protocols. It shows that both protocols have a linear increase in the size of random generators r, and witness values d, as one of each is required per puzzle. Additionally, GC-TLP introduces a linear increase in the size of the number of squaring operations T and secret exponents a as it requires a value per puzzle instance. The size of the RSA modulus used also impacts the size of the outputs. The size of the setup using a 2048-bit modulus is approximately 50% larger than when using a 1024-bit modulus. The length of the time intervals does not affect the size of the output, as computations happen in a finite group.

Table 5. Size in memory of setup output of C-TLP and GC-TLP. The table shows the size in bytes of each component, totals, and the differences between the C-TLP and GC-TLP protocols across 1, 100, and 10000 puzzle instances of 10 seconds each, with an RSA key size of 2048 bits.

Instances	**1**			**100**			**10000**		
	C-TLP	GC-TLP	Diff.	C-TLP	GC-TLP	Diff.	C-TLP	GC-TLP	Diff.
Auxiliary information	175	175	0	175	175	0	175	175	0
Modulus N	296	296	0	296	296	0	296	296	0
Squaring operations T	56	136	80	56	28 784	28 488	56	588 240	587 944
Public key	823	903	80	1063	29 551	28 488	3 601 523	9 213 499	5 611 976
Random generators r	377	377	0	29 820	29 820	0	2 975 176	2 975 176	0
Secret exponent a	296	384	88	296	30 520	30 224	296	3 045 176	3 044 880
Witnesses d	137	137	0	5820	5820	0	5 700 984	5 700 984	0
Secret key	810	898	88	35 936	66 160	30 224	3 550 648	6 595 528	3 044 880
Total	1633	1801	168	36 999	95 711	58 712	3 551 711	7 184 535	3 632 824

Acknowledgements

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. Steven J. Murdoch was supported by REPHRAIN and The Royal Society under grant UF160505.

References

1. Abadi, A., Kiayias, A.: Multi-instance publicly verifiable time-lock puzzle and its applications. In: FC (2021)
2. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: CRYPTO (2017)
3. Baum, C., David, B., Dowsley, R., Nielsen, J.B., Oechsner, S.: TARDIS: A foundation of time-lock puzzles in UC. In: EUROCRYPT (2021)

4. Blum, M., Santis, A.D., Micali, S., Persiano, G.: Noninteractive zero-knowledge. SIAM J. Comput. 20(6) (1991)
5. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO'18
6. Boneh, D., Bünz, B., Fisch, B.: A survey of two verifiable delay functions. IACR Cryptol. ePrint Arch. (2018)
7. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000
8. Brakerski, Z., Döttling, N., Garg, S., Malavolta, G.: Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In: TCC'19,
9. Chen, H., Deviani, R.: A secure e-voting system based on RSA time-lock puzzle mechanism. In: BWCCA'12 ,
10. Chvojka, P., Jager, T., Slamanig, D., Striecks, C.: Generic constructions of incremental and homomorphic timed-release encryption. IACR Cryptol. ePrint Arch.
11. Chvojka, P., Jager, T., Slamanig, D., Striecks, C.: Versatile and sustainable timed-release encryption and sequential time-lock puzzles (extended abstract). In: ESORICS 2021. Springer (2021)
12. Ephraim, N., Freitag, C., Komargodski, I., Pass, R.: Continuous verifiable delay functions. In: EUROCRYPT (2020)
13. Feige, U., Lapidot, D., Shamir, A.: Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In: 31st Annual Symposium on Foundations of Computer Science. IEEE Computer Society (1990)
14. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO (1986)
15. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02
16. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: EUROCRYPT (2015)
17. Garay, J.A., Kiayias, A., Panagiotakos, G.: Iterated search problems and blockchain security under falsifiable assumptions. IACR Cryptology ePrint Archive (2019)
18. Granlund, T., the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library (2023), http://gmplib.org/
19. Horsen, C.V., the GMPY 2 development team: gmpy2 (2023), https://github.com/aleaxit/gmpy
20. Kavousi, A., Abadi, A., Jovanovic, P.: Timed secret sharing. Cryptology ePrint Archive (2023)
21. Malavolta, G., Thyagarajan, S.A.K.: Homomorphic time-lock puzzles and applications. In: CRYPTO'19
22. May, T.C.: Timed-release crypto (1993), https://cypherpunks.venona.com/date/1993/02/msg00129.html
23. Medley, L., Loe, A.F., Quaglia, E.A.: Sok: Delay-based cryptography. Cryptology ePrint Archive, Paper 2023/687 (2023), https://eprint.iacr.org/2023/687, https://eprint.iacr.org/2023/687
24. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91
25. Pietrzak, K.: Simple verifiable delay functions. In: 10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
26. Ristea, D.: Source code for delegated time-lock puzzle (2023), https://github.com/danrr/Generic-MITLP
27. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep. (1996)

28. Srinivasan, S., Loss, J., Malavolta, G., Nayak, K., Papamanthou, C., Thyagarajan, S.A.K.: Transparent batchable time-lock puzzles and applications to byzantine consensus. In: PKC (2023)
29. Thyagarajan, S.A.K., Bhat, A., Malavolta, G., Döttling, N., Kate, A., Schröder, D.: Verifiable timed signatures made practical. In: CCS (2020)
30. Thyagarajan, S.A.K., Gong, T., Bhat, A., Kate, A., Schröder, D.: Opensquare: Decentralized repeated modular squaring service. In: CCS (2021)
31. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT'19
32. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014)

A Symmetric-key Encryption Scheme

A symmetric-key encryption scheme consists of three algorithms $(\texttt{SKE.keyGen}, \texttt{Enc}, \texttt{Dec})$, defined as follows. (1) $\texttt{SKE.keyGen}(1^\lambda) \to k$ is a probabilistic algorithm that outputs a symmetric key k. (2) $\texttt{Enc}(k, m) \to c$ takes as input k and a message m in some message space and outputs a ciphertext c. (3) $\texttt{Dec}(k, c) \to m$ 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, it holds that

$$\Pr\left[\ \texttt{Dec}(k, \texttt{Enc}(k, m)) = m : \texttt{SKE.keyGen}(1^\lambda) \to k\right] = 1\ .$$

The symmetric-key encryption scheme satisfies *indistinguishability against chosen-plaintext attacks (IND-CPA)*, if any PPT adversary $\mathcal{A}$ has no more than $\frac{1}{2} + \mathsf{negl}(\lambda)$ probability in winning the following game: the challenger generates a symmetric key $\texttt{SKE.keyGen}(1^\lambda) \to k$. The adversary $\mathcal{A}$ is given access to an encryption oracle $\texttt{Enc}(k, \cdot)$ 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 $\texttt{Enc}(k, m_b) \to c_b$. Upon receiving c_b, $\mathcal{A}$ continues to have access to $\texttt{Enc}(k, \cdot)$ and wins if its guess b' is equal to b.

B Sequential and Iterated Functions

Definition 13 ($\Delta, \delta(\Delta)$)-Sequential function). *For a function: $\delta(\Delta)$, time parameter: Δ and security parameter: $\lambda = O(\log(|X|))$, $f : X \to Y$ 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:*

$$Pr\left[y_A = f(x) \middle| y_A \xleftarrow{\$} \mathcal{A}(\lambda, x), x \xleftarrow{\$} X\right] \leq negl(\lambda)$$

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 $f(k, x) = \beta^{(k)}(x) = \overbrace{\beta \circ \beta \circ ... \circ \beta}^{k \text{ Times}}$ is an iterated sequential function, with round function β, if for all $k = 2^{o(\lambda)}$ the 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 $\mu(.)$ such that:*

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

C Further Discussion on the Network Delay Parameter

As discussed in [16], liveness states that an honestly generated transaction will eventually be included more than κ blocks deep in an honest party's blockchain. It is parameterised 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 [2], 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 solver) propagates its transaction (containing the proof) all honest parties will see it on their chain after at most: $\Upsilon = WindowSize + u$ time period.