symmetry

*Article*

# Practical Verifiable Time-Lock Puzzle: Pre- and Post-Solution Verification

**Zheyi Wu, Haolin Liu and Lei Wang ***

School of Electronics Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China; wuzheyi@sjtu.edu.cn (Z.W.); liuhaolinsjtu@sjtu.edu.cn (H.L.)
* Correspondence: wanglei_hb@sjtu.edu.cn

**Abstract:** A time-lock puzzle encapsulates a secret message such that the receiver needs to perform a sequential computation, which takes a specified amount of time, to recover the message. Time-lock puzzles can be used in various scenarios, such as sealed-bid auctions, fair contract signing, and so on. The time required to generate a time-lock puzzle and the time needed to solve it are asymmetric, making the verification of a time-lock puzzle crucial. Before solving the puzzle, the solver needs to verify the validity of the puzzle to avoid computing invalid time-lock puzzles. After the puzzle has been solved, it is essential for a third party to confirm the correctness of the solution. This paper proposes a framework for time-lock puzzles, providing both pre-verification and post-verification functionalities, and outlines the security requirements of this framework. Furthermore, we present a practical construction based on iterated squaring in the RSA group and analyze the security of the specific construction. Finally, we implement this construction in Python and demonstrate its efficiency in different settings when implemented in practice.

**Keywords:** iterated squaring; post-verification; pre-verification; time-lock puzzle

check for updates

**Citation:** Wu, Z.; Liu, H.; Wang, L. Practical Verifiable Time-Lock Puzzle: Pre- and Post-Solution Verification. *Symmetry* **2024**, *16*, 1347. https://doi.org/10.3390/sym16101347

Academic Editor: Wiesław Leonski

Received: 15 August 2024
Revised: 27 September 2024
Accepted: 9 October 2024
Published: 11 October 2024

## 1. Introduction

The concept of time-lock puzzles was initially proposed by Rivest et al. [1] in 1996. A time-lock puzzle addresses the challenge of sending a secret message to the future, ensuring that the receiver cannot access the message until a predetermined amount of time has passed. This challenge was first identified by May [2] in the Cyberpunks community, where a solution using trusted agents as third parties to preserve secret messages was proposed. Time-lock puzzles are proposed as an alternative approach to address this problem without relying on a trusted third party. Loosely speaking, in a time-lock puzzle scheme, the sender encapsulates a secret message within a time-lock puzzle that requires the receiver to engage in a sequential computation to reveal the secret message. The evaluation time of this sequential computation is determined by a time parameter in the time-lock puzzle. Even with substantial parallel processing capabilities, the receiver cannot significantly reduce the required time.

Time-lock puzzles have a wide range of applications, including sealed-bid auctions [3–6], fair contract signing [7–9], electronic voting [10,11], zero-knowledge argument [12], and non-malleable commitment [13]. For instance, a sealed-bid auction consists of two phases. In the bidding phase, all bidders submit their encapsulated bids, which cannot be accessed by other bidders. In the opening phase, all bidders reveal their bids. However, the robustness of the protocol can be compromised if a bidder refuses or fails to reveal its bid during the opening phase. Time-lock puzzle provides a method to handle this problem [14,15]. During the bidding phase, bidders encapsulate their bids into time-lock puzzles, which guarantees the confidentiality of these bids since the puzzles cannot be solved before the specified time. In the subsequent opening phase, bidders are required to reveal their bids. If a malicious bidder refuses to do so, its bid can still be revealed by solving the corresponding time-lock puzzle.

We categorize the verification process of the time-lock puzzle scheme into two stages, depending on when it occurs: pre-verification and post-verification. Pre-verification takes place before the protocol's execution. For instance, before a message is used in the protocol execution, the message receiver can authenticate the identity of the sender using a signature scheme. On the other hand, post-verification is executed after the execution of the protocol. It involves the capacity of the verifier to verify the execution outcome. For example, the verifier confirms that the obtained output is accurately evaluated by the protocol and satisfies the claimed properties.

This paper focuses on the verifiability of time-lock puzzle schemes, which is essential to ensure that the protocol behaves as intended and maintains its expected properties, particularly in the presence of potential adversarial activities. In time-lock puzzle schemes, there is an asymmetry between the time required to generate the puzzle and the time needed to solve it. The generation of the time-lock puzzle is efficient, whereas solving it is time-consuming. Consequently, the adversary may exploit this by transmitting invalid time-lock puzzles, thereby depleting the computational resources of the receiver.

### *1.1. Our Contributions*

The initial construction of the time-lock puzzle [1] lacks a verification algorithm, resulting in a lack of verifiability. The verifiability of a time-lock puzzle is important when applied in sealed-bid auctions. When a time-lock puzzle is used in sealed-bid auctions, it is necessary for the receiver to verify the legitimacy of a time-lock puzzle once a bidder has encapsulated its bid. This verification process is essential to detect malicious behavior that would prevent the time-lock puzzles from being opened correctly in the opening phase. On the other hand, after the time-lock puzzle is solved, the revealed bid must be publicly verifiable, ensuring that everyone can verify the correctness of this bid.

A typical method of providing pre-verification for the time-lock puzzle is to sign the puzzle. The receiver can verify the identity of the sender by checking the signature. After solving the puzzle, the receiver needs to generate a proof to convince a third party that the solution is correct. However, the validity of the puzzle itself should be verified, since a malicious puzzle generator could send invalid time-lock puzzles to the receiver. Moreover, generating a proof for post-verification brings an additional cost for the receiver.

In this paper, we propose a framework that addresses the two challenges outlined above. Since the receiver needs to check the identity of the sender, a signature scheme is required in the time-lock puzzle (implicitly). In our framework, rather than signing the puzzle itself, the sender signs the commitment of the secret value within the time-lock puzzle. This commitment can be used to provide pre-verification of the puzzle, since they are related to the same value. After solving the puzzle, a proof is necessary to help a third party to verify the correctness of the solution. In our framework, the signed commitment serves as this proof, as it is authenticated by the sender. Formally, for a secret message, $m$, the puzzle generator encapsulates it into a time-lock puzzle, $z$. It also generates a commitment value, $x$, for the message, $m$, and then signs it to produce the signature, $\sigma$. The generator sends the puzzle, $z$, the commitment value, $x$, and the signature, $\sigma$, to the receiver. If the puzzle generator is honest, then the commitment value, $x$, and the puzzle, $z$, will correspond to the same message. Consequently, the receiver can verify the validity of the puzzle before it starts solving the puzzle. Once the puzzle passes the pre-verification, the receiver solves it by spending a certain amount of sequential computation. For the secret message, $m$, revealed from the puzzle, a third party can verify its correctness using the commitment value, $x$, eliminating the need for the receiver to generate additional proof.

In simple terms, the framework should satisfy the following properties:

- *Correctness*: For all correctly generated time-lock puzzles, they can pass pre-verification and can be resolved to the corresponding secret message.
- *Sequentiality*: All receivers need to spend a certain amount of time to solve the time-lock puzzle, even with substantial parallel processing capabilities.

- *Unforgeability*: The probability that the adversary can generate a forged time-lock puzzle (or a forged solution) that can pass the pre-verification (or post-verification) is negligible.
- *Verifiability*: The validity of the time-lock puzzle and the solution should be verifiable before and after solving the puzzle, respectively.

A widely used method to construct a time-lock puzzle is based on iterated squaring. So, according to our proposed framework, we present a specific construction based on iterated squaring on an RSA group. In our framework, the signature scheme is used to sign on the commitment of the secret value within the time-lock puzzle. This signature provides the identity of the sender, ensuring that the message has not been modified during the transformation. In our framework, we do not restrict the specific signature scheme. A hash-then-sign structure signature can be used in our framework. In addition, we demonstrate that the construction satisfies the desired properties and analyze its efficiency theoretically. Furthermore, we implement the time-lock puzzle construction in Python and analyze its performance in different settings. We will demonstrate that the solving time of the puzzle increases as the time parameter increases, while the generation time of the puzzle, the verification time of the puzzle, and the verification time of the solution remain relatively unchanged. Additionally, the solving time of the puzzle is significantly larger than the other times. For example, when the security parameter is 2048 and the time parameter is $2^{21}$, the receiver needs 21.7 s to solve the puzzle, while it takes 1.25 s to verify the validity of the puzzle.

## *1.2. Related Works*

### 1.2.1. Timed Commitments

In a timed commitment scheme, the committer publishes a timed commitment, which can be forced open if the committer refuses to open the commitment. The receiver should be able to verify the "well-formedness" of the message before it starts a time-consuming algorithm. This problem was first considered in timed commitment [7] and following works [16,17]. In a timed commitment scheme, the committer generates a commitment string $c$ and a time-related element, $u$. The commitment string, $c$, can be revealed by the receiver without the help of the committer. The receiver can calculate a string from a time-related element, $u$, and use the string to reveal the commitment. The committer generates a proof, $\pi$, to prove that the time-related element, $u$, is correctly generated and sends the tuple, $(c, u, \pi)$, to the receiver. The receiver checks that, $u$, is exactly the value declared by the proof, $\pi$. However, the relationship between $c$ and $u$ is not verified, an attacker can change $c$ to another value, $c'$, while keeping the elements $u, \pi$ as the same as the correct one. This faked tuple $(c', u, \pi)$ can also pass the verification, but the commitment has already been changed. Hence, this verification is insufficient to convince the receiver before executing the forced open algorithm. In EuroS&P 2022, Manevich et al. [18] considered the construction of attribute verifiable timed commitment. It requires the committer to generate an interactive proof to convince the receiver that the committed value satisfies some attributes.

### 1.2.2. Homomorphic Time-Lock Puzzle

In ESORICS 2022, Liu et al. [19] considered the pre-verification for homomorphic time-lock puzzles. The puzzle generator provides a zero-knowledge proof to show the existence of a solution for the puzzle. It requires a trusted setup to generate the public parameters. The relationship between the time-related elements in public parameters is implicitly considered in their construction. However, this construction becomes insecure when considering the scenario in which the puzzle generator generates these public parameters. A malicious generator may generate the wrong elements to defraud the receiver. So, it is imperative to verify the correctness of the public parameters *explicitly* by the puzzle receiver. In PKC 2023, Srinivasan et al. [20] considers batchable time-lock puzzle to improve scalability when there are a large number of time-lock puzzles that need to be solved.

### 1.2.3. Verifiable Timed Signatures

In ACM CCS 2020, Thyagarajan et al. [21] proposed verifiable timed signature and utilized a cut-and-choose mechanism to provide verifiability. The signature is sent to the receiver using a $k$-out-of-$n$ secret sharing scheme, with each share encapsulated within a time-lock puzzle. Subsequently, the receiver requires the generator to open $k-1$ randomly selected time-lock puzzles to show that these shares are correctly encapsulated. If all $k-1$ puzzles pass verification, then the receiver proceeds to solve the remaining $n-k+1$ time-lock puzzles to retrieve the signature based on $k$ or more shares. However, this protocol suffers a security-efficiency trade-off. A potential threat arises from a malicious sender generating $k-1$ correct shares, while the remaining shares are incorrect. The protocol's vulnerability lies in the small probability that the receiver selects all $k-1$ pieces as the correctly generated ones, resulting in successful verification but an incorrect signature retrieval. Consequently, it requires a large $n$ to improve the security bound, resulting in a long time verification. In ESORICS 2022, Thyagarajan et al. [22] proposed a verifiable timed linkable ring signature. This construction can be used to hide a linkable ring signature for a predetermined amount of time in a verifiable way. This is a special time signature and can be used to enhance the privacy in cryptocurrency.

### 1.2.4. Verifiable Delay Functions

In verifiable delay functions (VDFs) [23–26], the output is evaluated from the input after computation for a certain amount of time. After the evaluation, the evaluator has obtained the output and a proof. A verifier can use the proof to *efficiently* verify that the output is correctly generated. The verifiable delay functions can be used to provide post-verification for a time-lock puzzle scheme. The post-verification for a time-lock puzzle scheme means that a third party not solving the puzzle should be able to verify the correctness of the solution. A naive method of verifying the correctness is to repeat the solving process; this is an inefficient approach. So, the running time for the post-verification should be much less than the evaluation time. This property is considered in [27–29] as public verifiability. The constructions of publicly verifiable time-lock puzzles utilize the VDFs: the puzzle-solving process can be regarded as the execution of a VDF. So, aside from the solution of the puzzle, the puzzle solver can generate an additional proof to prove the correctness of the solution.

## 2. Preliminaries

In this section, we provide the necessary cryptographic foundations and formal definitions that utilized in our work.

### *2.1. Time-Lock Puzzle*

A time-lock puzzle [1,20,27,29–34] is used to encapsulate a message such that nobody can obtain it before spending computation for a certain amount of time. For conceptual simplicity, we consider schemes with binary messages. Specifically, a time-lock puzzle scheme, Puz, contains the following algorithms.

- $z \leftarrow \mathsf{Encaps}(s,t)$ : For a message, $s \in \{0,1\}$, and time parameter, $t$, the puzzle generator runs this *randomized* encapsulation algorithm to encapsulate $s$ into a time-lock puzzle, $z$.
- $m \leftarrow \mathsf{Sol}(z)$ : The receiver runs this *deterministic* puzzle-solving algorithm to solve the puzzle, $z$, and obtain the message, $s$.

**Definition 1.** *(Correctness) For all security parameter $\lambda$, all polynomials $t$ in $\lambda$ and all messages $s \in \{0,1\}$, it holds that* $\mathsf{Sol}(\mathsf{Encaps}(s,t)) = s$.

**Definition 2.** *(Security) A time-lock puzzle is sequential with gap $\varepsilon < 1$, if there exists a polynomial $\hat{\mathcal{T}}(\cdot)$. For all polynomial $\mathcal{T}(\cdot) \geq \hat{\mathcal{T}}(\cdot)$ and for all polynomial time adversary $\mathcal{A}$ with running times* *of less than* $\mathcal{T}^{\varepsilon}(\lambda)$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all security parameters,* $\lambda$*, it holds that:*

$$\Pr[b \leftarrow \mathcal{A}(z) : z \leftarrow \mathsf{Encaps}(b,t)] \leq 1/2 + \mathsf{negl}(\lambda). \tag{1}$$

**Definition 3.** *(Efficiency) For all messages* $s \in \{0,1\}$*, the encapsulation algorithm* $\mathsf{Encaps}(s,t)$ *can be computed in time* $\mathsf{poly}(\log t, \lambda)$*, while the puzzle-solving algorithm,* $\mathsf{Sol}(z)$*, can be computed in time* $t \cdot \mathsf{poly}(\lambda)$*.*

### 2.2. Signature Scheme

A signature scheme [35] consists of the following three probabilistic polynomial time (PPT) algorithms: $\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify}$:

- $(vk, sk) \leftarrow \mathsf{KGen}(1^{\lambda})$ : The *randomized* key generation algorithm takes the security parameter $1^{\lambda}$ as input and outputs the verification key and signing key pairs $(vk, sk)$. The message space is denoted as $\mathcal{M}$.
- $\sigma \leftarrow \mathsf{Sign}(sk, m)$ : The signing algorithm takes the signing key, $sk$, and a message $m \in \mathcal{M}$ as input, and outputs the signature $\sigma$.
- $0/1 \leftarrow \mathsf{Verify}(vk, m, \sigma)$ : This *deterministic* verification algorithm takes the verification key, $vk$, the message, $m \in \mathcal{M}$, and the signature, $\sigma$, as input. If $\sigma$ is the valid signature on $m$, then the verification algorithm outputs 1; otherwise, it outputs 0.

**Definition 4.** *(Correctness) For all* $1^{\lambda}$*, all* $(vk, sk)$ *generated by* $\mathsf{KGen}(1^{\lambda})$ *and all message* $m \in \mathcal{M}$*, it holds that* $\mathsf{Verify}(vk, m, \mathsf{Sign}(sk, m)) = 1$*.*

**Definition 5.** *(Existentially unforgeable under a chosen-message attack (EUF-CMA)) All PPT adversaries,* $\mathcal{A}$*, can query messages* $\{m_1, m_2, \dots\}$*, obtaining the corresponding signatures* $\{\sigma_1, \sigma_2, \dots\}$*. Denote* $\mathcal{Q}$ *as the set of messages queried by* $\mathcal{A}$*; here, there exists a negligible function,* $\mathsf{negl}(\cdot)$*, such that:*

$$\Pr[1 \leftarrow \mathsf{Verify}(vk, m, \sigma) \wedge m \notin \mathcal{Q} : (m, \sigma) \leftarrow \mathcal{A}(\{m_i, \sigma_i\}_{i=1,2,\dots})] \leq \mathsf{negl}(1^{\lambda}). \tag{2}$$

### 2.3. Non-Interactive Zero-Knowledge Proofs of Discrete Logarithm Knowledge

Given four group elements $g, \hat{g}, h, \hat{h}$, a zero-knowledge proof $EQLOG(g, \hat{g}, h, \hat{h})$ is used to demonstrate the knowledge of a secret value $\alpha$, such that $\hat{g} = g^{\alpha}$ and $\hat{h} = h^{\alpha}$ without revealing any secret information about $\alpha$. Chaum and Pedersen presented an interactive proof construction in [36]; this can be transformed into a non-interactive version as follows:

- The prover selects a random value $\beta$, and sends $g_1 = g^{\beta}$ and $h_1 = h^{\beta}$ to the verifier.
- The prover computes $e = \mathcal{H}_e(g, \hat{g}, h, \hat{h}, g_1, h_1)$ by a hash function $\mathcal{H}_e$ and sends $z = \beta - \alpha e$ to the verifier.
- If both $g_1 = g^z \hat{g}^e$ and $h_1 = h^z \hat{h}^e$ hold, then the verifier accepts this proof.

### 2.4. Non-Interactive Proofs of Sequential Computation

In a sequential computation, the evaluator needs to generate a sequential proof $\Pi(x, y, t)$ to prove that the output value $y$ is exactly evaluated based on the input $x$ by $t$ sequential computations. Boneh et al. [7] proposed a construction of non-interactive sequential proof $\Pi(x, y, t)$ for iterated squaring based on an RSA group $\mathbb{Z}_N^*$ when the time parameter $t = 2^{\tau}$ for a positive integer $\tau$. That is, $N = pq$ for two distinct primes, $p, q$ and $y = x^{2^{2^{\tau}}} \mod N$.

Denote $b_i = g^{2^{2^i}} \mod N$; the prover generates a vector $W = \langle b_0, b_1, \dots, b_{\tau} \rangle$. For each $i = 0, \dots, \tau - 1$, the prover shows that the tuple $(g, b_i, b_{i+1})$ satisfy the relationship $(g, g^a, g^{a^2})$ for some $a$ by a NIZK proof $EQLOG(g, g^a, g^a, g^{a^2})$. By verifying that the last element in $w$ is $b_{\tau} = y$, the verifier is assured that $y$ is exactly $x^{2^{2^{\tau}}} \mod N$. These $\tau$ proofs can be generated by the prover in parallel, and the length of the total proof $\Pi(x, y, t)$ is $O(\log t)$ for the time parameter, $t$.

### *2.5. Sequential Squaring Assumption*

The sequentiality of iterated squaring on an RSA group is based on the following assumption:

**Definition 6.** *(Sequential Squaring Assumption [32]) Let N be a uniform strong RSA integer, g be a generator of $\mathbb{J}_N$, and $\mathcal{T}(\cdot)$ be a polynomial. Then, there exists some $0 < \varepsilon < 1$, such that, for every polynomial-size adversary $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ whose depth is bounded from above by $\mathcal{T}^{\varepsilon}(\lambda)$, there exists a negligible function* $\mathsf{negl}(\cdot)$*:*

$$\Pr\left[ b \leftarrow \mathcal{A}(N, g, \mathcal{T}(\lambda), x, y) : \begin{array}{r} x \xleftarrow{\$} \mathbb{J}_N;\ b \xleftarrow{\$} \{0,1\} \\ \text{if } b = 0 \text{ then } y \xleftarrow{\$} \mathbb{J}_N \\ \text{if } b = 1 \text{ then } y = x^{2^{\mathcal{T}(\lambda)}} \end{array} \right] \leq 1/2 + \mathsf{negl}(\lambda). \tag{3}$$

Note that the restriction of $x$ and $y$ to $\mathbb{J}_N$ is to avoid trivial attack where the adversary computes the Jacobi symbol of the group elements.

## 3. The Framework

As introduced in Section 1, when a time-lock puzzle scheme is implemented in practice, it is imperative to perform verification both before and after the execution of the puzzle solution. Before solving the puzzle, the pre-verification prevents the solver from spending time-consuming operations on invalid time-lock puzzles. After solving the puzzle, the post-verification can convince a third party of the correctness of the solution.

In this section, we present an overview and the syntax of our framework, which is designed to perform verification both before and after solving a time-lock puzzle. Additionally, we discuss the adversary model and the properties that the framework should satisfy.

### *3.1. Overview of the Framework*

The framework is given in Figure 1. A time-lock puzzle is executed between two participants: the sender $\mathcal{S}$ and the receiver $\mathcal{R}$. The sender is the generator of the puzzle, it encapsulates the secret message into the time-lock puzzle. The receiver is also the solver of the puzzle, it first checks the validity of the puzzle and solves the valid puzzle by spending computation for a certain amount of time. As shown in Figure 1, the encapsulation for the message is denoted as a time-related relationship, $F_t(\cdot)$. Here, $t$ is the time parameter which determines that the receiver must spend *at least* time $t$ to solve the puzzle. $F(\cdot)$ is a commitment scheme and the signing algorithm is denoted as $\mathsf{Sign}_{sk}(\cdot)$, where $sk$ is the signing key of the sender $\mathcal{S}$.

$$\begin{tikzcd}
m \arrow[r, "F"] \arrow[d, "F_t"'] & x \arrow[r, "\mathsf{Sign}_{sk}"] \arrow[d, dashed, leftrightarrow, "F'_t"] & \sigma \\
z \arrow[r, dashed, "F'"'] & x' &
\end{tikzcd}$$

**Figure 1.** The framework.

For a secret message $m$, the time-lock puzzle is generated as $z = F_t(m)$. The sender evaluates the commitment scheme $F$ on the message to obtain the value $x = F(m)$, and then signs on $x$ by its signing key, $sk$, to produce the signature $\sigma = \mathsf{Sign}_{sk}(x)$. After receiving the tuple $(z, x, \sigma)$ from the sender, the receiver verifies the validity of the tuple.

Since the sender does not sign on the puzzle, $z$, directly, the receiver cannot verify the validity of the puzzle by verifying the signature. This signature can ensure the validity of the commitment value, $x$. The dotted line in Figure 1 shows the design idea of our framework: $F'(\cdot)$ is a function related to $F(\cdot)$ that maps the time-lock puzzle, $z$, to an intermediate value, $x'$. If the puzzle, $z$, is correctly generated, then the two values $x$ and $x'$ should satisfy a corresponding relationship, $F_t'(\cdot)$, which can be verified by the receiver. If the puzzle passes the verification, then the receiver is convinced that the time-lock puzzle, $z$, is correctly generated.

After a solution, $m'$, is revealed by the receiver from the puzzle, $z$, a third party can verify the correctness of this solution, $m'$, by verifying the relationship between the message, $m'$, and the commitment value, $x$. The validity of $x$ is guaranteed by the signature, $\sigma$, and a third party verifies whether $x$ is the commitment to the message $m'$.

### 3.2. The Syntax

This section presents the syntax of the framework. The main idea of the framework is that the receiver generates an intermediate value $x'$ based on the puzzle, $z$, and verifies the relationship between $x$ and $x'$. For a smooth reading experience, the table of variables is presented in Table A1. Formally, the framework contains the following algorithms:

- $(pk, sk) \leftarrow \mathsf{Setup}(1^\lambda)$ : The *randomized* setup algorithm takes the security parameter $1^\lambda$ as input, and outputs the public verification key, $pk$, and the private signing key, $sk$, for the sender. By convenience, the security parameter specifies the message space, $\mathcal{M}$.
- $x \leftarrow \mathsf{Commit}(m)$ : The commitment algorithm takes the message $m \in \mathcal{M}$ in the message space as input and outputs a commitment value, $x$.
- $\sigma \leftarrow \mathsf{Sign}(sk, x)$ : The signing algorithm takes the signing key of the sender, $sk$, and the commitment value, $x$, as inputs, and outputs the signature, $\sigma$.
- $z \leftarrow \mathsf{Encaps}(m, t)$ : The *randomized* encapsulation algorithm is used to encapsulate the message into a time-lock puzzle. It takes the message, $m$, and the time parameter, $t$, as inputs, and outputs the time-lock puzzle, $z$. The running time of the encapsulation algorithm should be much less than $t$.
- $\pi \leftarrow \mathsf{ProofGen}(m, x, z, \sigma)$ : The proof generation algorithm is used to output a non-interactive proof, $\pi$, which can help the receiver *efficiently* verify the validity of the puzzle, $z$. It takes the secret message, $m$, the commitment value, $x$, the puzzle, $z$, and the signature, $\sigma$, as inputs, and outputs a proof, $\pi$. The output $\pi$ can be represented as an empty string NULL if no such proof is required.
- $0/1 \leftarrow \mathsf{PuzVerify}(pk, z, x, \sigma, \pi)$ : The *deterministic* puzzle verification algorithm takes the puzzle, $z$, the commitment value, $x$, the signature, $\sigma$, and proof $\pi$ as inputs. It outputs 1 if $z$ is a valid time-lock puzzle and outputs 0 otherwise. The running time of this puzzle verification algorithm should be much less than $t$.
- $m' \leftarrow \mathsf{Solve}(z)$ : If the puzzle verification algorithm outputs 1, then the receiver runs this puzzle-solving algorithm to reveal the message from the puzzle. The puzzle-solving algorithm takes the time-lock puzzle, $z$, as input and outputs a solution $m'$. It requires *at least* time $t$ for the receiver to run this puzzle-solving algorithm.
- $0/1 \leftarrow \mathsf{SolVerify}(pk, m', z, x, \sigma, \pi)$ : The *deterministic* solution verification takes the public verification key, $pk$, the solution, $m'$, the commitment value, $x'$, the puzzle, $z$, the signature, $\sigma$, and the proof, $\pi$, as inputs. It outputs 1 if the solution, $m'$, is exactly the message encapsulated in the puzzle, $z$, and outputs 0 otherwise. The running time of this solution verification algorithm should be much less than $t$.

### 3.3. Property Requirements

The goal of an adversary is to deviate the protocol from its security requirements. For example, after obtaining a valid tuple of outputs, a passive adversary may want to solve the puzzle faster than the required time, and an active adversary may try to forge a puzzle. The active adversary tries to generate a forged puzzle, such that it can pass the verification of the verifier but cannot be revealed as the correct message. In this paper, we assume that all adversaries are probabilistic polynomial time, which means that the adversary cannot break the underlying cryptographic primitive.

**Definition 7.** *(Correctness) For all security parameters $1^\lambda$, all key pairs $(pk, sk)$ generated by* $\mathsf{Setup}(1^\lambda)$*, all messages, $m$, and all time parameters, $t$, if the signature is $\sigma \leftarrow \mathsf{Sign}(sk, \mathsf{Commit}(m))$, the puzzle is $z \leftarrow \mathsf{Encaps}(m, t)$, and the proof is $\pi \leftarrow \mathsf{ProofGen}(m, \mathsf{Commit}(m), z, \sigma)$, then it holds that $\mathsf{PuzVerify}(z, \mathsf{Commit}(m), \sigma, \pi) = 1$, $m \leftarrow \mathsf{Solve}(z)$ and $1 \leftarrow \mathsf{SolVerify}(pk, m, z, \sigma)$.*

**Definition 8.** *(Sequentiality) For all security parameters, $1^\lambda$, all time parameters, $t$, all messages, $m$, and all tuples, $(z, x, \sigma, \pi)$, generated by the sender, if the puzzle verification algorithm $\mathsf{PuzVerify}(pk, z, x, \sigma, \pi)$ outputs $1$, then, for $0 < \varepsilon < 1$ and the running time of an adversary $\mathcal{A}$ which is less than $\varepsilon \cdot t$, there exists a negligible function $\mathsf{negl}(\cdot)$, such that*

$$\Pr\left[m' = m : \begin{matrix} z \leftarrow \mathsf{Encaps}(m, t) \\ m' \leftarrow \mathcal{A}(z, x, \sigma, \pi) \end{matrix}\right] \le \mathsf{negl}(1^\lambda). \tag{4}$$

**Definition 9.** *(Unforgeability of puzzle) For any probabilistic polynomial time adversary $\mathcal{A}$, it can query the set of messages $\{m_i\}_{i=1,2,\ldots}$ and obtains the corresponding set of tuples $\mathcal{T} = \{(z_i, x_i, \sigma_i, \pi_i)\}_{i=1,2,\ldots}$. Denote the set of all required puzzles as $\mathcal{Z}$, there exists a negligible function $\mathsf{negl}(\cdot)$, such that:*

$$\Pr\left[\begin{matrix} 1 \leftarrow \mathsf{PuzVerify}(z', x', \sigma', \pi') \\ z' \notin \mathcal{Z} \end{matrix} : (z', x', \sigma', \pi') \leftarrow \mathcal{A}(\mathcal{T})\right] \le \mathsf{negl}(1^\lambda). \tag{5}$$

In Definition 9, an active adversary wants to forge a tuple $(z', x', \sigma', \pi')$ that can pass the puzzle verification algorithm but cannot be correctly solved by the receiver. The puzzle $z'$ in the faked tuple should be different from the correct one, $z$, otherwise, the receiver can still retrieve the message encapsulated in the puzzle. So, it requires that the puzzle $z'$ has not been queried before.

**Definition 10.** *(Unforgeability of solution) For all security parameters, $1^\lambda$, all time parameters, $t$, and all messages, $m$, if the tuples $(z, x, \sigma, \pi)$ are generated correctly, then, for all probabilistic polynomial time adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$, such that:*

$$\Pr\left[\begin{matrix} m' \neq m \\ 1 \leftarrow \mathsf{SolVerify}(pk, m', z, x, \sigma, \pi) \end{matrix} : \begin{matrix} z \leftarrow \mathsf{Encaps}(m, t) \\ m' \leftarrow \mathcal{A}(z, x, \sigma, \pi) \end{matrix}\right] \le \mathsf{negl}(1^\lambda). \tag{6}$$

**Definition 11.** *(Verifiability of puzzle) After receiving the tuple $(z, x, \sigma, \pi)$ from the sender, the receiver should be able to verify the validity of the time-lock puzzle, $z$, by the puzzle verification algorithm, $\mathsf{PuzVerify}(z, x, \sigma, \pi)$. The running time of the puzzle verification algorithm should be much less than time $t$.*

**Definition 12.** *(Public verifiability of solution) After the receiver reveals its solution, $m'$, on the time-lock puzzle, $z$, everyone can verify the correctness of the solution by the solution verification algorithm $\mathsf{SolVerify}(pk, m', z, \sigma)$. The running time of the solution verification algorithm should be much less than time $t$.*

Note that the verification for the puzzle is not required to be performed publicly, since only the receiver needs to verify the validity of the puzzle before spending a large amount of sequential computation. The verifiability of the solution should be public so that everyone can verify the correctness of the result.

## 4. Iterated-Squaring-Based Construction

According to the framework presented in Section 3, we give a specific construction based on iterated squaring on an RSA group and discuss its properties in this section.

## *4.1. The Construction*

The sender randomly chooses two distinct safe primes, $p = 2p' + 1$ and $q = 2q' + 1$, where $p'$ and $q'$ are also primes, and then it computes $N = pq$ and chooses a random generator, $g \in \mathbb{Z}_N^*$. (A zero-knowledge proof $\pi_N$ can be generated to prove that $N$ is the product of two safe primes [37]). The time-lock puzzle is constructed on $\mathbb{Z}_N^*$: for a large time parameter, $t$, the sender computes $h = g^{2^t} \bmod N$ *efficiently* by the trapdoor $\varphi(N)$; it first computes $e = 2^t \bmod \varphi(N)$ and computes $h = g^e \bmod N$, which is used as a mask to encapsulate the message, $m$. The time-lock puzzle is generated as $z = m \cdot h \bmod N$. After receiving the puzzle, $z$, the receiver computes $h$ based on the element, $g$, and the time parameter, $t$. Since the receiver does not know the trapdoor, $\varphi(N)$, it must compute $h$ by $t$ sequential computations. The receiver can reveal the message, $m$, from the puzzle, $z$, as $m = z \cdot h^{-1} \bmod N$.

Based on the above time-lock puzzle construction, we combine it with the framework in Section 3 to verify the secret message. To facilitate the generation of the proof, we require that the time parameter, $t$, can be presented as $t = 2^\tau$ for a positive integer $\tau$. For a secret message, $m \in \mathbb{Z}_N^*$, the time-lock puzzle, $z$, is generated as the protocol in the above section. The sender chooses a random value $a \in \mathbb{Z}_{\varphi(N)}^*$ and computes $x = m^a \bmod N$. It signs on $x$ to obtain the signature $\sigma$. Additionally, the sender generates a sequential proof $\pi_t = \Pi(g^a, (g^a)^{2^t}, t)$ to help the receiver check the validity of the puzzle efficiently. The receiver is given the values $x$, $a$, the proof, $\pi_t$, and the signature, $\sigma$.

As a summary, the process of the sender is present in Algorithm 1. For simplicity, we denote all values except the time-lock puzzle sent by the sender as auxiliary information.

**Algorithm 1:** The sender's process based on iterated squaring

```
Setup: The RSA number, $N$, the group, $\mathbb{Z}_N^*$, the public verification key, $pk$, and the
  secret signing key, $sk$;
Input: A secret message $m \in \mathbb{Z}_N^*$;
Choose a random element $g \in \mathbb{Z}_N^*$;
Generate the puzzle as $z = m \cdot g^{2^t} \bmod N$;
// The sender can compute $g^{2^t} \bmod N$ efficiently as $g^{2^t \bmod \varphi(N)} \bmod N$
Choose a random value $a \in \mathbb{Z}_{\varphi(N)}^*$;
Compute the commitment of the message as $x = m^a \bmod N$;
Generate the sequential proof $\pi_t = \Pi(g^a, (g^a)^{2^t} \bmod N, t)$;
Sign on $x$: $\sigma = \mathsf{Sign}(sk, x)$;
Output: The time-lock puzzle, $z$, and auxiliary information $g, a, x, \sigma, \pi_t$.
```

After receiving the outputs from the sender, the receiver first checks the validity of the signature, $\sigma$, to make sure that $x$ is correctly generated by the sender, then the receiver needs to verify the relationship between $z$ and $x$. The receiver computes $x' = z^a \bmod N$ and computes $x'' = x' \cdot x^{-1} \bmod N$. Then, it verifies the time relationship between $g^a$ and $x''$ by the proof $\pi_t$. If the value $x''$ passes the verification, then the receiver is convinced that it has obtained a value $x'' = (g^{2^t})^a \bmod N$. Since the operation to obtain the value $x''$ is the exponent on the puzzle, $z$, the receiver is convinced that the puzzle is $m \cdot g^{2^t}$, and the revealed value $m$ is already committed by the sender.

The algorithm of the receiver is shown in Algorithm 2:

**Algorithm 2:** The receiver's process based on iterated squaring

```
Input: The time-lock puzzle, $z$, and auxiliary information $g, a, x, \sigma, \pi_t$;
// Check the validity of the signature $\sigma$
if $\mathsf{Verify}(pk, x, \sigma) = 0$ then
  | Output $\perp$
end
Compute $x' = z^a \bmod N$ and $x'' = x' \cdot x^{-1} \bmod N$;
if $x'' \neq (g^a)^{2^t} \bmod N$ then
  | Output $\perp$
end
// This verification can be performed by the sequential proof $\pi_t$
Compute $h' = g^{2^t} \bmod N$;
// The receiver needs to compute $h'$ by iterated squaring
Compute $m' = z \cdot h'^{-1} \bmod N$;
Output: A message $m' \in \mathbb{Z}_N^*$.
```

The security of the RSA group is based on the assumption that the adversary cannot find the factorization of the RSA number efficiently. Another group used in iterated squaring is the class group of an imaginary quadratic field [38,39]. This group has a property that there is no efficient algorithm to compute the order of such group.

An alternative method for constructing a time-lock puzzle is based on modular square roots. For a prime, $p$, and a quadratic residue $a$ modulo $p$, a sequential time $O(\log p)$ is required to compute the square roots of $a$. In this method, the time of sequential evaluation is fixed for a given prime, $p$. In contract, with an iterated-squaring-based construction, the time can be easily adjusted by changing the number of iterations.

### *4.2. Security Analysis*

As outlined in Section 3, the construction should satisfy the following properties.

#### Correctness

The correctness requires that if the outputs of the sender are correctly generated, then the time-lock puzzle must pass the verification of the receiver, and the puzzle can be revealed as the same message encapsulated by the sender.

**Theorem 1.** *For a secret message m, if the sender follows the protocol and sends the time-lock puzzle, z, and auxiliary information $g, a, x, \sigma, \pi_t$ to the receiver, then the puzzle, z, can pass the puzzle verification algorithm and be revealed as the secret message, m, by the receiver.*

**Proof.** If the sender and the receiver both follow the protocol, the sender uses $h = g^{2^t \bmod \varphi(N)} \bmod N$ to encapsulate the secret message, $h$, and the receiver computes $h' = g^{2^t} \bmod N$ by iterated squaring. Since the order of $\mathbb{Z}_N^*$ is $\varphi(N)$, there is $h' = h$. Hence, the time-lock puzzle can be correctly solved by the receiver.

Additionally, if the time-lock puzzle and auxiliary information are generated correctly, it holds that $x = m^a \bmod N$, $z = m \cdot g^{2^t} \bmod N$, and $\pi_t = \Pi(g^a, (g^a)^{2^t}, t)$. The receiver computes $x'' = z^a \cdot x^{-1} \bmod N$ and can use the sequential proof $\pi_t = \Pi(g^a, x'', t)$ to verify that $x''$ is exactly $(g^{2^t})^a \bmod N$. So, the receiver can be convinced that the puzzle is correctly generated as $z = m \cdot g^{2^t} \bmod N$. After verifying the validity of the puzzle, $z$, the receiver can solve the puzzle as $m = z \cdot (g^{2^t})^{-1} \bmod N$. □

#### Sequentiality

The sequentiality requires all probabilistic polynomial time adversaries to spend a certain amount of time to solve the puzzle.

**Theorem 2.** *In the construction in Section 4, for $0 < \varepsilon < 1$ and any adversary $\mathcal{A}$ with a running time less than $\varepsilon \cdot t$, there exists a negligible function* $\mathsf{negl}(\cdot)$*, such that the probability that the receiver can solve the puzzle faster than time $\varepsilon \cdot t$ after receiving the puzzle, $z$, from the sender is* $\mathsf{negl}(\lambda)$.

**Proof.** The sequentiality of the time-lock puzzle is based on Definition 6. For any probabilistic polynomial time adversary $\mathcal{A}$, it does not know the factorization of $N$; hence, $\mathcal{A}$ cannot compute $h' = g^{2^t} \bmod N$ faster than time $t$.

Except for the time-lock puzzle, $z$, $\mathcal{A}$ is also given $x$, which is also related to the secret message, $m$. The secret message, $m$, can be computed from $x$ as $m = x^{1/a} \bmod N$. Since $\mathcal{A}$ does not know the factorization of $N$, the probability of success for $\mathcal{A}$ is $\mathsf{negl}(\lambda)$.

Additionally, the proof $\pi_t$ is related to the sequential computation and the adversary wants to compute $h' = g^{2^t} \bmod N$ faster from $\pi_t$. The specific construction of $\pi_t$ is given in Section 2, the values given in $\pi_t$ can be presented as $b_i = g^{2^{2^i} \cdot a} \bmod N$ for $i = 1, 2, \ldots, \tau$, where $t = 2^\tau$. The closest value to $h'$ is $b_{\tau-1} = g^{2^{t/2} \cdot a} \bmod N$. Since $a \in \mathbb{Z}^*_{\varphi(N)}$, there is $a < 2^\lambda$. Hence, it requires at least $t/2 - \lambda$ times iterated squaring for the adversary to compute $h'$ from $b_{\tau-1}$. In practice, $t$ is chosen as $2^{30}$ and $\lambda = 2^{10}$, so $\lambda \ll t/2$.

Hence, after receiving the puzzle, $z$, from the sender, the receiver should spend at least time $\varepsilon \cdot t$ to solve the puzzle, where $\varepsilon < 1/2$. □

Unforgeability

The unforgeability of the puzzle requires that, for any probabilistic polynomial time adversary, the probability that it can generate a forged puzzle that can pass the puzzle verification algorithm is negligible.

**Theorem 3.** *In the construction in Section 4, for any probabilistic polynomial time adversary $\mathcal{A}$, it can query the message it chooses and obtain the corresponding outputs of the sender. There exists a negligible function,* $\mathsf{negl}(\cdot)$*, and the probability that the adversary can forge a new puzzle that can pass the verification of the receiver is* $\mathsf{negl}(\lambda)$.

**Proof.** The adversary $\mathcal{A}$ can query the set of messages $\{m_i\}_{i=1,2,\ldots}$ and obtains the corresponding set of tuples $\mathcal{T} = \{(z_i, x_i, \sigma_i, \pi_i)\}_{i=1,2,\ldots}$. Denote the set of all obtained puzzles as $\mathcal{Z}$. If $\mathcal{A}$ can forge a new puzzle $z' \notin \mathcal{Z}$ and passes the pre-verification, then it must satisfy one of the following two situations.

The first situation is that the adversary generates a tuple by choosing a new message $m'$, and generate the puzzle $z'$, the commitment value $x'$ by itself. It then tries to forge the signature $\sigma'$, on the commitment value $x'$. The unforgeability of the signature scheme ensures that the probability of $\mathcal{A}$ can forge a signature on a new commitment value $x'$ successfully is $\mathsf{negl}(\lambda)$.

Another situation is to bypass the forgery of the signature scheme. The adversary $\mathcal{A}$ reuses the commitment value $x_i$ of some tuple that has been queried before. In this case, if the adversary wants to generate another puzzle $z'_i \neq z_i$, then it tries to find another message $m'_i \neq m_i$ but $F(m'_i) = F(m_i)$. The construction in Section 4 sets $F(\cdot)$ as $F(x) = x^a \bmod N$ for a random $a \in \mathbb{Z}^*_{\varphi(N)}$, since $a$ is coprime with $\varphi(N)$, the function $F$ is a permutation on $\mathbb{Z}^*_N$. Hence, the adversary cannot find such $m'_i$.

In summary, the probability that a PPT adversary can forge a valid puzzle that can pass the pre-verification is negligible. □

The unforgeability of solution requires that the probability that a receiver can forge a solution that can pass the solution verification is negligible.

**Theorem 4.** *In the construction in Section 4, for any secret message, $m$, after receiving the corresponding valid tuple $z, x, \sigma, \pi$ from the sender, there exists a negligible function,* $\mathsf{negl}(\cdot)$*, and the probability that the adversary can forge a solution, $m' \neq m$, that can pass the solution verification is* $\mathsf{negl}(\lambda)$.

**Proof.** After obtaining the solution, $m'$, from the receiver, the third party first checks whether $\sigma$ is the signature of the value, $x$, by the public key of the puzzle generator, and then verifies whether $x$ is the commitment on $m'$. If the receiver can succeed forge a solution $m' \neq m$ that can pass the verification of the third party, it must satisfy one of the following two situations.

The first situation is that the receiver generates a commitment $x'$ on the forged solution, $m'$, and then forges the signature $\sigma'$, on $x'$. The unforgeability of the signature scheme guarantees that the probability that this situation happens is negligible. Another situation requires the commitment $x$ can be revealed as the forged solution, $m'$. The probability of this situation is negligible according to the binding property of the commitment scheme.

In summary, the probability that a receiver can forge a solution, $m' \neq m$, and pass the post-verification step, is negligible. $\square$

## 4.3. Efficiency Analysis

According to Algorithm 1, in order to generate the time-lock puzzle, the sender first computes $e = 2^t \bmod \varphi(N)$ and then computes $h = g^e \bmod N$. These are two modular exponentiation operations, which require $O(\log t) + O(\lambda)$ multiplications over $\mathbb{Z}_N^*$. In the puzzle verification algorithm, the receiver needs to compute $x' = z^a \bmod N$, $x'' = x' \cdot x^{-1} \bmod N$ and verify that $x''$ is exactly $(g^{2^t})^a \bmod N$ by the proof $\pi_t$. The proof $\pi_t$ contains $\log t$ small proofs which can be verified in parallel. The verification time complexity of each small proof is $O(\lambda)$. So the time complexity of this verification process is $O(\lambda)$. For the solution verification algorithm, a third party checks the correctness of the solution by verifying the signature and the commitment, which results in a time complexity of $O(\lambda)$. As proved in Theorem 2, the time complexity of solving the time-lock puzzle for all receiver is $O(t)$.

In practice, the time parameter $t$ is a polynomial of the security parameter $\lambda$. This indicates that the generation and the verification of the time-lock puzzle are much faster than solving the puzzle. Hence, the pre-verification and the post-verification are both efficient.

## 4.4. Construction for Long Messages

In the above construction, it requires that the message, $m$, lies in the group $\mathbb{Z}_N^*$. In practice, $N$ is chosen as a 2048 bits number; hence, the length of $m$ is at most 2048 bits, which is shorter than the length of message in a real network. A long message can still be treated as a group element in the protocol, as in the case of a short message. In this method, a group with the same length as the message needs to be generated during the setup phase of the protocol. This will increase the evaluation time of each group operation, resulting in the poor efficiency of the protocol. Additionally, this method is not flexible if the length of the message changes. A new group must be generated for a longer message, which is inconvenient in practice implementation.

A common approach to handle a long message is to segment the message into short consecutive blocks: $m = m_1||m_2||\ldots||m_n$. Here, $||$ represents the concatenation of two strings, and each block $m_i$ is $\kappa$ bits. Here, $\kappa$ is a parameter related to the security parameter $\lambda$. For example, $\kappa = \lambda - 1$. This ensures that each block $m_i$ can be represented as a group element in $\mathbb{Z}_N^*$ with overwhelming probability. If the last block $m_n$ is less than $\kappa$ bits, a padding scheme can be used to fill the last block.

The element $h = g^{2^t} \bmod N$ we used to encapsulate a short message is a group element in $\mathbb{Z}_N^*$, which is at most $\lambda$ bits. It can be used to encapsulate only one block each time. A naive idea is to compute different $h_i = g_i^{2^t} \bmod N$ to mask different blocks, i.e., the puzzle is generated as $z = h_1 \cdot m_1||h_2 \cdot m_2||...||h_n \cdot m_n$. In this case, the verification for each block is the same as the short message case, and the receiver needs to solve $n$ different short time-lock puzzles to reveal the message. Another method to generate the puzzle uses only one time-delay element. The sender computes $h = g^{2^t} \bmod N$ and generates the puzzle as $z = h \cdot m_1||h^2 \cdot m_2||...||h^{2^{n-1}} \cdot m_n$. In this case, the receiver needs to solve one short time-lock puzzle to reveal the message.

## 5. Experiment and Analysis

This section presents the efficiency of our construction. When implemented in practice, it is necessary to require that the verification time of the receiver to check the validity of the time-lock puzzle is much less than solving the time-lock puzzle. Additionally, a third party should be able to verify the correctness of the solution efficiently. We analyze the efficiency of the protocol from both theoretical and practical aspects. Moreover, we implement the construction in Python and emulate the running time in different settings.

### *5.1. Experiment Setup*

We implement our construction in Python and use cryptography libraries for fundamental cryptographic and arithmetic operations. Since we do not care about the specific communication method between the sender and the receiver, we implement the sender and the receiver in one laptop.

The experimental environment is presented in the following table (Table 1).

**Table 1.** The experiment setup.

| Component | Description |
|---|---|
| CPU | Intel Core i5-10210U (1.60 GHz) |
| Number of threads | 1 |
| RAM | 16 GB |
| System | Ubuntu 22.04 LTS |

### *5.2. Comparison of Solving Time and Verification Time*

In the following experiment, the security parameter is 2048, i.e., $N$ is a 2048-bit number. First, we investigate how the running time of solving a time-lock puzzle relates to the time parameter, $t$, by conducting experiments with $t$ ranging from $2^{16}$ to $2^{21}$, while maintaining the other conditions the same. The results are depicted in Figure 2 using logarithmic coordinates, indicating linear correlations between the average running time to solve a time-lock puzzle and the time parameter, $t$. As a comparison, the verification time for the receiver to perform the pre-verification and a third party to perform the post-verification are given in Table 2.

To minimize single measurement errors, each of the above values is the average of ten measurements taken with the same settings. The experimental data indicate that the running time for the sender to generate the time-lock puzzle and the verification time for the receiver to check the validity of the time-lock puzzle are on the order of milliseconds and increase slowly with the time parameter. For example, when the time parameter is $2^{21}$, the time it takes for the receiver to verify the validity of the puzzle is about 1/17 of the time it takes to solve the puzzle. Hence, it implicates the efficiency of our construction.

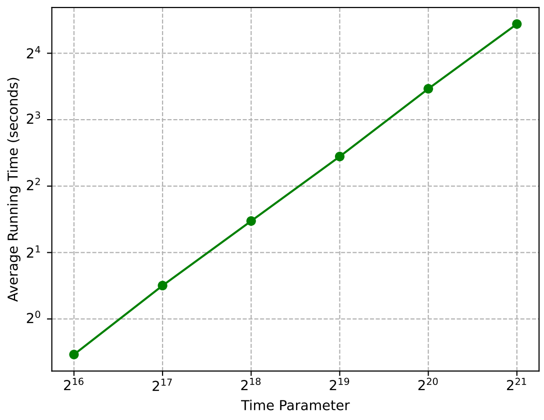

**Figure 2.** Average solving time under different time parameters.

**Table 2.** Average running time under different time parameters.

| Time Para. | Puzzle Gen. (ms) | Puzzle Verif. (Sequential) (ms) | Puzzle Verif. (Parallel) (ms) | Solution Verif. (ms) |
|---|---|---|---|---|
| $2^{16}$ | 26.6 | 975.2 | 61.0 | 27.6 |
| $2^{17}$ | 27.3 | 1037.9 | 61.0 | 28.3 |
| $2^{18}$ | 27.9 | 1090.3 | 60.6 | 28.2 |
| $2^{19}$ | 29.3 | 1141.8 | 60.1 | 27.0 |
| $2^{20}$ | 34.8 | 1236.7 | 61.8 | 27.8 |
| $2^{21}$ | 40.0 | 1257.9 | 59.9 | 27.8 |

We compare our experimental results with those in [18] in Table 3. The verification time of their construction is 38 ms and it takes 21.725 s for the receiver to solve the puzzle. A third party takes 8.673 ms to verify the solution. Their implementation also shows that the verification time is in the order of milliseconds and is much smaller than the time to solve the puzzle.

**Table 3.** Comparison with related work.

| | Puzzle Gen. (ms) | Puzzle Verif. (ms) | Solution Verif. (ms) | Puzzle Sol. (s) |
|---|---|---|---|---|
| [18] | 37.98 | 38 | 8.673 | 21.725 |
| Our work | 40 | 59.9 | 27.8 | 21.7 |

### *5.3. Communication Cost*

In our implementation, the sender sends the time-lock puzzle, the signature, the commitment, and the proof to the receiver. When the security parameter is 2048, both the puzzle and the commitment are 256 bytes in length. The length of the signature depends on the specific signature scheme. We use ECDSA with secp256r1 as the underlying signature scheme; the length of the signature is 64 bytes. The length of the proof is related to the time parameter, when the time parameter is $2^{20}$, the proof length is 15 KB.

### *5.4. Discussion about Solving Time*

The difference in computational power between different solvers always exists when a time-lock puzzle is implemented in practice [40]. It is an inherent issue since adversaries can always gain a time advantage if they possess more sequential computational power than honest nodes. This difference in computation power brings different solving times for the same time-lock puzzle. According to the performance data given by Intel [41], the difference between updated server-grade and updated custom-grade chips is approximately 10. Hardware performance also doubles roughly every four years, as indicated by the performance growth rate given in [42].

Generating a time-lock puzzle is much more efficient than solving the puzzle, so the differences in hardware do not significantly affect puzzle generation. However, if adversaries were to use updated server-grade chips while honest nodes employed four-year-old custom-grade chips, then the adversaries would be able to solve the time-lock puzzle 20 times faster than the honest nodes. Consequently, when a time-lock puzzle protocol is used in practice, honest nodes should periodically upgrade their hardware to make sure that the disparity of computation power remains within a reasonable range. This represents a trade-off between the security and the hardware requirements of honest nodes.

## 6. Conclusions and Discussion

This paper considers the verifiability of a time-lock puzzle scheme, which is an important property of a time-lock puzzle scheme when implemented in practice. Before the receiver starts solving the time-lock puzzle, it should be able to verify the validity of the puzzle to avoid invalid computation. After the puzzle is solved, a third party should verify the correctness of the solution without resolving the puzzle.

In this paper, we propose a framework of time-lock puzzles to provide both pre-verification and post-verification by combining a signature scheme. In this framework, the receiver of the time-lock puzzle can verify the validity of the puzzle before it starts solving the puzzle. After receiving the puzzle from the puzzle generator, the receiver computes an intermediate value based on the values it receives and checks the validity of the puzzle. It can be convinced that the secret value behind the time-lock puzzle has already been signed by the puzzle generator if the puzzle passes the verification. After the time-lock puzzle is solved, a third party can also verify the correctness of the solution by the signature efficiently.

Based on the framework, we present a specific construction based on iterated squaring over the RSA group and give security proofs for these constructions. We have also implemented the construction in Python. The experimental results show the efficiency of the construction. The running time for the generation of the time-lock puzzle, the verification of the validityof this approach, and the solution of the time-lock puzzle are all in the order of milliseconds, while the running time for the solver is in the order of seconds when the construction is implemented in practice. Consequently, the pre-verification and the post-verification for the time-lock puzzles are both efficient.

Our construction becomes insecure when considering quantum attack, since it is based on iterated squaring on an RSA group. There exists an efficient algorithm to find the factorization of an RSA number on quantum computers. There are four methods used to design post-quantum protocols: lattice-based, code-based, hash-based, and multivariate polynomial. In CRYPTO 2024, Agrawalr et al. [34] present a time-lock puzzle construction from lattice. Hence, it remains a future work to extend our construction with their construction or design a verifiable time-lock puzzle using the other three methods.

In this paper, we only consider the implementation when the time-lock puzzle is used on a small scale. When the time-lock puzzle is used in electronic voting protocol with a large number of voters, it requires a lot of calculations to open each encapsulated vote individually, which effects the efficiency of the protocol. There are two methods to improve the scalability of the time-lock puzzle when it is implemented. One method is using the same time-delay element to generate several time-lock puzzles. In this method, the receiver only needs to solve one time-lock puzzle to obtain the time-delay element and then use this value to solve other time-lock puzzles. Another method is using a homomorphic technique. The receiver only needs to solve one homomorphic time-lock puzzle to obtain the evaluation results of the solution to these puzzles. Moreover, it is a challenge to design scalable quantum-resistance verifiable time-lock puzzles when we both consider the scalability problem and quantum attack.

**Author Contributions:** Software, H.L.; Validation, Z.W.; Formal analysis, Z.W.; Data curation, H.L.; Writing—original draft, Z.W.; Writing—review and editing, L.W.; Supervision, L.W. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported by National Key Research and Development (2019YFB2101600).

**Data Availability Statement:** The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

**Conflicts of Interest:** The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

## Appendix A. Table of Variables

**Table A1.** The table of variables.

| Variable | Meaning |
|---|---|
| $\lambda$ | security parameter |
| $(pk, sk)$ | public/private key pairs |
| $m$ | message to be encapsulated |
| $x$ | commitment to the message |
| $\sigma$ | signature on the commitment |
| $t$ | time parameter |
| $z$ | time-lock puzzle on the message |
| $\pi$ | proof used in pre-verification |
| $x'$ | intermediate value computed by the receiver |
| $m'$ | solved message |

## References

1. Rivest, R.L.; Shamir, A.; Wagner, D.A. *Time-Lock Puzzles and Timed-Release Crypto*; Massachusetts Institute of Technology: Cambridge, MA, USA, 1996.
2. May, T.C. Timed-Release Crypto. 1993. Available online: https://cypherpunks.venona.com/date/1993/02/msg00129.html (accessed on 28 June 2023).
3. Elyakime, B.; Laffont, J.J.; Loisel, P.; Vuong, Q. First-price sealed-bid auctions with secret reservation prices. *Ann. D'economie Stat.* **1994**, *34*, 115–141. [CrossRef]
4. Athey, S.; Levin, J.; Seira, E. Comparing open and sealed bid auctions: Evidence from timber auctions. *Q. J. Econ.* **2011**, *126*, 207–257. [CrossRef]
5. Bag, S.; Hao, F.; Shahandashti, S.F.; Ray, I.G. SEAL: Sealed-bid auction without auctioneers. *IEEE Trans. Inf. Forensics Secur.* **2019**, *15*, 2042–2052. [CrossRef]
6. Chvojka, P.; Jager, T.; Slamanig, D.; Striecks, C. Versatile and Sustainable Timed-Release Encryption and Sequential Time-Lock Puzzles (Extended Abstract). In *Proceedings of the ESORICS 2021, Part II*; Bertino, E., Shulman, H., Waidner, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12973, pp. 64–85. [CrossRef]
7. Boneh, D.; Naor, M. Timed Commitments. In *Proceedings of the CRYPTO 2000*; Bellare, M., Ed.; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1880, pp. 236–254. [CrossRef]
8. Wang, G. An abuse-free fair contract signing protocol based on the RSA signature. In Proceedings of the 14th International Conference on World Wide Web, Chiba, Japan, 10–14 May 2005; pp. 412–421.
9. Ben-Or, M.; Goldreich, O.; Micali, S.; Rivest, R.L. A fair protocol for signing contracts. *IEEE Trans. Inf. Theory* **1990**, *36*, 40–46. [CrossRef]
10. Kohno, T.; Stubblefield, A.; Rubin, A.D.; Wallach, D.S. Analysis of an electronic voting system. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 9–12 May 2004; pp. 27–40.
11. Gritzalis, D.A. *Secure Electronic Voting*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012; Volume 7.
12. Dwork, C.; Naor, M. Zaps and Their Applications. In Proceedings of the 41st FOCS, Redondo Beach, CA, USA, 12–14 November 2000; IEEE Computer Society Press: Piscataway, NJ, USA, 2000; pp. 283–293. [CrossRef]
13. Lin, H.; Pass, R.; Soni, P. Two-Round and Non-Interactive Concurrent Non-Malleable Commitments from Time-Lock Puzzles. In Proceedings of the 58th FOCS, Berkeley, CA, USA, 15–17 October 2017; Umans, C., Ed.; IEEE Computer Society Press: Piscataway, NJ, USA, 2017; pp. 576–587. [CrossRef]
14. Tyagi, N.; Arun, A.; Freitag, C.; Wahby, R.; Bonneau, J.; Mazières, D. Riggs: Decentralized Sealed-Bid Auctions. In Proceedings of the CCS, Copenhagen, Denmark, 26–30 November 2023; ACM: New York, NY, USA, 2023; pp. 1227–1241.
15. Zhang, M.; Yang, M.; Shen, G. SSBAS-FA: A secure sealed-bid e-auction scheme with fair arbitration based on time-released blockchain. *J. Syst. Archit.* **2022**, *129*, 102619. [CrossRef]
16. Garay, J.A.; Jakobsson, M. Timed Release of Standard Digital Signatures. In *Proceedings of the FC 2002*; Blaze, M., Ed.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2357, pp. 168–182.
17. Garay, J.A.; Pomerance, C. Timed Fair Exchange of Standard Signatures: [Extended Abstract]. In *Proceedings of the FC 2003*; Wright, R., Ed.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2742, pp. 190–207.
18. Manevich, Y.; Akavia, A. Cross chain atomic swaps in the absence of time via attribute verifiable timed commitments. In Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P), Genoa, Italy, 6–10 June 2022; pp. 606–625.
19. Liu, Y.; Wang, Q.; Yiu, S.M. Towards Practical Homomorphic Time-Lock Puzzles: Applicability and Verifiability. In *Proceedings of the ESORICS 2022, Part I*; Atluri, V., Di Pietro, R., Jensen, C.D., Meng, W., Eds.; Springer: Berlin/Heidelberg, Germany, 2022; Volume 13554, pp. 424–443. [CrossRef]

20. Srinivasan, S.; Loss, J.; Malavolta, G.; Nayak, K.; Papamanthou, C.; Thyagarajan, S.A. Transparent batchable time-lock puzzles and applications to byzantine consensus. In *Proceedings of the IACR International Conference on Public-Key Cryptography*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 554–584.
21. Thyagarajan, S.A.K.; Bhat, A.; Malavolta, G.; Döttling, N.; Kate, A.; Schröder, D. Verifiable Timed Signatures Made Practical. In *Proceedings of the ACM CCS 2020*; Ligatti, J., Ou, X., Katz, J., Vigna, G., Eds.; ACM Press: New York, NY, USA, 2020; pp. 1733–1750. [CrossRef]
22. Thyagarajan, S.A.; Malavolta, G.; Schmid, F.; Schröder, D. Verifiable timed linkable ring signatures for scalable payments for monero. In *Proceedings of the European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 467–486.
23. Mahmoody, M.; Moran, T.; Vadhan, S.P. Publicly verifiable proofs of sequential work. In *Proceedings of the ITCS 2013*; Kleinberg, R.D., Ed.; ACM: New York, NY, USA, 2013; pp. 373–388. [CrossRef]
24. Boneh, D.; Bonneau, J.; Bünz, B.; Fisch, B. Verifiable Delay Functions. In *Proceedings of the CRYPTO 2018, Part I*; Shacham, H., Boldyreva, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10991, pp. 757–788. [CrossRef]
25. Pietrzak, K. Simple Verifiable Delay Functions. In *Proceedings of the ITCS 2019*; Blum, A., Ed.; LIPIcs: St. Louis, MO, USA, 2019; Volume 124, pp. 60:1–60:15. [CrossRef]
26. Wesolowski, B. Efficient Verifiable Delay Functions. In *Proceedings of the EUROCRYPT 2019, Part III*; Ishai, Y., Rijmen, V., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11478, pp. 379–407. [CrossRef]
27. Freitag, C.; Komargodski, I.; Pass, R.; Sirkin, N. Non-malleable Time-Lock Puzzles and Applications. In *Proceedings of the TCC 2021, Part III*; Nissim, K., Waters, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2021; Volume 13044, pp. 447–479. [CrossRef]
28. Abadi, A.; Kiayias, A. Multi-instance Publicly Verifiable Time-Lock Puzzle and Its Applications. In *Proceedings of the FC 2021, Part II*; Borisov, N., Díaz, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12675, pp. 541–559. [CrossRef]
29. Katz, J.; Loss, J.; Xu, J. On the Security of Time-Lock Puzzles and Timed Commitments. In *Proceedings of the TCC 2020, Part III*; Pass, R., Pietrzak, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12552, pp. 390–413. [CrossRef]
30. Mahmoody, M.; Moran, T.; Vadhan, S.P. Time-Lock Puzzles in the Random Oracle Model. In *Proceedings of the CRYPTO 2011*; Rogaway, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6841, pp. 39–50. [CrossRef]
31. Bitansky, N.; Goldwasser, S.; Jain, A.; Paneth, O.; Vaikuntanathan, V.; Waters, B. Time-Lock Puzzles from Randomized Encodings. In *Proceedings of the ITCS 2016*; Sudan, M., Ed.; ACM: New York, NY, USA, 2016; pp. 345–356. [CrossRef]
32. Malavolta, G.; Thyagarajan, S.A.K. Homomorphic Time-Lock Puzzles and Applications. In *Proceedings of the CRYPTO 2019, Part I*; Boldyreva, A., Micciancio, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11692, pp. 620–649. [CrossRef]
33. Hiraga, D.; Hara, K.; Tezuka, M.; Yoshida, Y.; Tanaka, K. Security Definitions on Time-Lock Puzzles. In *Proceedings of the ICISC 20*; Hong, D., Ed.; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12593, pp. 3–15. [CrossRef]
34. Agrawalr, S.; Malavolta, G.; Zhang, T. Time-Lock Puzzles from Lattices. In Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 18–22 August 2024; Springer: Berlin/Heidelberg, Germany, 2024; pp. 425–456.
35. Katz, J. *Digital Signatures*; Springer: Berlin/Heidelberg, Germany, 2010; Volume 1.
36. Chaum, D.; Pedersen, T.P. Transferred Cash Grows in Size. In *Proceedings of the EUROCRYPT'92*; Rueppel, R.A., Ed.; Springer: Berlin/Heidelberg, Germany, 1993; Volume 658, pp. 390–407. [CrossRef]
37. Camenisch, J.; Michels, M. Proving in Zero-Knowledge that a Number Is the Product of Two Safe Primes. In *Proceedings of the EUROCRYPT'99*; Stern, J., Ed.; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1592, pp. 107–122. [CrossRef]
38. Buchmann, J.; Hamdy, S. A survey on IQ cryptography. In Proceedings of the Public-Key Cryptography and Computational Number Theory, Warsaw, Poland, 11–15 September 2001; pp. 1–15.
39. Hafner, J.L.; McCurley, K.S. A rigorous subexponential algorithm for computation of class groups. *J. Am. Math. Soc.* **1989**, *2*, 837–850. [CrossRef]
40. Schindler, P.; Judmayer, A.; Hittmeir, M.; Stifter, N.; Weippl, E.R. RandRunner: Distributed Randomness from Trapdoor VDFs with Strong Uniqueness. In Proceedings of the NDSS 2021, online, 21–25 February 2021; The Internet Society: Reston, VA, USA, 2021.
41. Intel. Export Compliance Metrics for Intel Microprocessors. 2022. Available online: https://www.intel.com/content/www/us/en/support/articles/000005755/processors.html (accessed on 22 November 2022).
42. Hennessy, J.L.; Patterson, D.A. *Computer Architecture: A Quantitative Approach*; Elsevier: Amsterdam, The Netherlands, 2011.