Tempora-Fusion: Time-Lock Puzzle with Efficient Verifiable Homomorphic Linear Combination

Aydin Abadi*

Newcastle University

Abstract. To securely transmit sensitive information into the future, Time-Lock Puzzles (TLPs) have been developed. Their applications include scheduled payments, timed commitments, e-voting, and sealed-bid auctions. Homomorphic TLP is a key variant of TLP that enables computation on puzzles from different clients. This allows a solver/server to tackle only a single puzzle encoding the computation's result. However, existing homomorphic TLPs lack support for *verifying* the correctness of the computation results. We address this limitation by introducing Tempora-Fusion, a TLP that allows a server to perform homomorphic linear combinations of puzzles from different clients while ensuring verification of computation correctness. This scheme avoids asymmetric-key cryptography for verification, thus paving the way for efficient implementations. We discuss our scheme's application in various domains, such as federated learning, scheduled payments in online banking, and e-voting.

1 Introduction

Time-Lock Puzzles (TLPs) are elegant cryptographic primitives that enable the transmission of information to the future, without relying on a trusted third party. They allow a party to lock a message in such a way that no one else can unlock it until a certain time has elapsed. TLPs have various applications, including scheduled payments in cryptocurrencies [46], timed commitments [31], e-voting [20], sealed-bid auctions [42], byzantine broadcast [50], zero-knowledge proofs [25], timed secret sharing [32], and verifiable delay functions [13], and contract signing [16].

In a TLP, upon receipt of a message, the server persistently engages in computation until the solution is discovered. Since its introduction by Rivest *et al.* [42], the TLPs have evolved, giving rise to an important variant, homomorphic TLPs. Malavolta *et al.* [37] proposed the notion of fully homomorphic TLPs, enabling the execution of arbitrary functions over puzzles prior to their resolution. Broadly, fully homomorphic TLPs address scenarios involving n clients, each generating and transmitting a puzzle encoding its respective solution to a server. The server then executes a homomorphic function across these puzzles, producing a unified puzzle. The solution to this puzzle represents the output of the function evaluated across all individual solutions. To achieve efficiency, partially homomorphic TLPs have also been proposed, including those that facilitate homomorphic linear combinations or the multiplication of puzzles [37].

Homomorphic TLPs have found applications in various areas, such as verifiable timed signatures [46], atomic swaps [47], and payment channels [48]. These applications surpass the original motivations for designing homomorphic TLPs, which primarily revolved around their use in e-voting and sealed-bid auctions.

Nevertheless, state-of-the-art homomorphic TLPs lack support for verifying computation results. They operate under the assumption of the server's honest computation, a presumption that might be overly optimistic, especially in scenarios involving a potentially malicious server. For example, in e-voting or sealed bid auctions, the server responsible for tallying votes or managing bids could exclude or tamper with certain puzzles (representing votes or bids) or fail to execute the function honestly. Without a verification mechanism in place, parties involved cannot detect the server's misbehavior, leading to blind acceptance of results.

* aydin.abadi@newcastle.ac.uk

1.1 Our Solutions

Partially Homomorphic TLP. To overcome the lack of support for verification in (partially) homomorphic TLPs, this work introduces Tempora-Fusion[1] and provides a formal definition of it. Tempora-Fusion is a TLP protocol that enables a party to perform homomorphic linear combinations of puzzles while ensuring the ability to verify the correctness of the computation result.

Consider the scenario where there are n independent clients, each with a coefficient q_i and a secret solution m_i. These clients are not aware of each other. In this setting, Tempora-Fusion enables each client to *independently* generate its puzzles and send them to a server or publish the puzzles. Upon receiving each puzzle, the server begins working to solve it. After publishing the puzzles, the clients can delete any local copy of the secret solutions. The clients will not need to download and locally access the plaintext solution at any point in the future before the server solves the puzzles.[2]

Crucially, at a later stage (before the server discovers any puzzle solution), the clients can convene and engage with the server to perform a homomorphic linear combination of their puzzles, yielding a single puzzle. In this scenario, they authorize the server to discover the solution to the computation after a designated period. Once the server computes the result $\sum_{j=1}^{n} q_i \cdot m_i$, and publishes it, anyone can efficiently verify its correctness. During this period, while the solution to each client's puzzle remains undiscovered, the clients can request the server to perform a homomorphic linear combination of their puzzles an *unlimited number of times*. They can use different coefficients $[q'_1, \ldots, q'_n]$ or select various subsets of puzzles.

Later on, when the server discovers a client's puzzle solution, it can also efficiently demonstrate the correctness of the solution to any party. In Tempora-Fusion, the verification mechanisms employed (for checking the computation's result and verifying a client's puzzle solution) are lightweight, avoiding the use of public-key cryptographic-based proofs, like zero-knowledge proof systems, which typically incur high costs.

In devising Tempora-Fusion, we employ several techniques previously unexplored in TLP research, including (i) using a polynomial representation of a message, (ii) employing an unforgeable encrypted polynomial, (iii) switching blinding factors via oblivious linear function evaluation, and (iv) using a small-sized field for homomorphic operations. Tempora-Fusion achieves its objectives without relying on a trusted setup.

This scheme does not require clients to know or interact with each other during the setup phase when they prepare their initial puzzles. The clients only need to interact with each other and with the server once, later when they decide to ask the server to perform a homomorphic linear combination of their puzzles. After this delegation phase, the clients can go back offline. The feature of not requiring clients to know or interact with each other when preparing their initial puzzles is significant for several reasons:

- *Independence*: Each client can generate their puzzle at their own convenience without having to coordinate with others, leading to less complexity and more flexibility.
- *Asynchronous Participation*: Clients can join the scheme at different times without needing to wait for others. This flexibility is particularly valuable in environments where clients might be distributed across different time zones or have varying availability.
- *Dynamic Client Base*: The scheme can easily accommodate a changing number of clients, as new clients can prepare their puzzles independently and join the homomorphic combination phase later.

[1] This term combines "tempora", meaning time in Latin, with "fusion", conveying the merging aspect of the homomorphic linear combination support in our protocol.

[2] The advantage of not needing to access the plaintext solution locally before the puzzles are discovered is multi-faceted: (i) Enhanced Security: By allowing clients to delete the plaintext solutions immediately after publishing the puzzles, the risk of these solutions being compromised is minimized. This reduces the likelihood of unauthorized access or leakage of sensitive information, and (ii) Compliance with Regulations: In scenarios where data protection regulations require minimizing the retention of sensitive data, this approach helps clients comply by ensuring that secret solutions are not retained longer than necessary.

- *Scalability*: Since the clients do not need to interact during the setup phase, the system can efficiently support a large number of clients and puzzles. In the subsequent computation phase, only a subset of these clients interested in the computation must interact with the server for puzzle processing. This approach enables the system to scale effectively, handling extensive initial client participation while managing server load during computation.

Tempora-Fusion ensures that even if a malicious server gains access to a subset of clients' secret keys, the privacy of non-corrupt clients and the validity of a solution and computation's result will still be upheld.

1.2 Applications

Timed Secure Aggregation in Federated Learning. Federated Learning (FL) is a machine learning framework where multiple parties collaboratively build machine learning models without revealing their sensitive input to their counterparts [54,39,3]. The process involves training a global model via collaborative learning on local data, and only the model updates are sent to the server. To allow the server to compute sums of model updates from clients in a secure manner, Bonawitz *et al*. [12] developed a *secure aggregation* mechanism. The scheme relies on a trusted party and a public-key-based verification mechanism to detect the server misbehaviors.

Tempora-Fusion can serve as a substitute for this secure aggregation in scenarios where the server must learn the aggregation result after a period. It offers two additional features. Firstly, it operates without requiring a trusted setup leading to relying on a weaker security assumption. Secondly, it utilizes symmetric-key-based verification mechanisms which can be more efficient compared to public-key-based verification methods.

Transparent Scheduled Payments in Online Banking. Insider attacks pose imminent threats to many organizations and their clients, including financial institutions and their customers. Insiders may collaborate with external fraudsters to obtain highly valuable data [35,21]. Investment strategies scheduled by individuals or companies, through financial institutions, contain sensitive information that could be exploited by insiders [49]. Tempora-Fusion can enable individuals and businesses to schedule multiple payments and investments through their online banking without the need to disclose each transfer's amounts before the scheduled transfer time. With the support of a homomorphic linear combination, Tempora-Fusion allows the bank to learn the average or total amount of transfers ahead of time, to ensure (i) the bank can facilitate the transfers and (ii) the average or total amount of transfers complies with the bank's policy and regulations [8,52,53].

Verifiable E-Voting and Sealed-Bid Auction Systems. E-voting and sealed-bid auction systems are applications in which ensuring that the voting or bidding process remains secure and transparent is of utmost importance. Researchers suggested that homomorphic TLPs can be utilized in such systems to enable secure computations without compromising the privacy of individual votes or bids [37]. By implementing Tempora-Fusion in e-voting and sealed-bid auction systems, an additional benefit in terms of verifiability can be achieved. This allows anyone to verify the correctness of computations, ensuring that their votes or bids are tallied correctly while maintaining their privacy.

2 Related Work

Timothy May [38] was the first to propose the idea of sending information into the future, i.e., time-lock puzzle/encryption. A basic property of a time-lock scheme is that generating a puzzle takes less time than solving it. Since the scheme that Timothy May proposed uses a trusted agent that 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.* [42] proposed an RSA-based TLP. This scheme does not require a trusted agent, relies on sequential (modular) squaring, and is secure against a receiver who may have many computation resources that run in parallel.

Since the introduction of the RSA-based TLP, various variants of it have been proposed. For instance, researchers such as Boneh *et al.* [16] and Garay *et al.* [27] have proposed TLPs that consider the setting where a client can be malicious and need to prove (in zero-knowledge) to a server that the correct solution will be recovered after a certain time. Also, Baum *et al.* [9] have developed a composable TLP that can be defined and proven in the universal composability framework.

2.1 Homomorphic Time-lock Puzzles

Malavolta and Thyagarajan *et al.* [37] proposed the notion of homomorphic TLPs, which let an arbitrary function run over puzzles before they are solved. The schemes use the RSA-based TLP and fully homomorphic encryption. To achieve efficiency, partially homomorphic TLPs have also been proposed, including those that facilitate homomorphic linear combinations or the multiplication of puzzles [37,36]. Partially homomorphic TLPs do not rely on fully homomorphic encryption resulting in more efficient implementations than fully homomorphic TLPs. Unlike the partially homomorphic TLP in [37], the ones in [36] allow a verifier to (1) ensure puzzles have been generated correctly and (2) ensure the server provides a correctness solution for a single client's puzzle (but not a solution related to homomorphic computation). It uses a public-key-based proof, initially proposed in [51].

Later, Srinivasan *et al.* [45] observed that existing homomorphic TLPs support a limited number of puzzles when it comes to batching solving; thus, solving one puzzle results in discovering all batched solutions. Accordingly, they proposed a scheme that allows an unlimited number of puzzles from various clients to be homomorphically combined into a single one, whose solution will be found by a server. The construction is based on indistinguishability obfuscation and puncturable pseudorandom function. To improve the efficiency of this scheme, Dujmovic *et al.* [24] proposed a new approach, without using indistinguishability obfuscation. Instead, the new scheme relies on pairings and learning with errors. The above two schemes assume that all initial puzzles will be solved at the same time.

All of the aforementioned homomorphic TLPs, except the one proposed in [45], require a trusted setup. Additionally, none of the homomorphic TLPs facilitates verification of the computation's correctness.

2.2 Verifiable Delay Function (VDF)

A VDF enables a prover to provide publicly verifiable proof stating that it has performed a pre-determined number of sequential computations [13,51,14,40]. VDFs have many applications, such as in decentralized systems to extract reliable public randomness from a blockchain.

VDF was first formalized by Boneh *et al.* in [13]. 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 SNARK-based approaches require a trusted setup. Later, Wesolowski [51] and Pietrzak [40] 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 TLPs. However, the converse is not necessarily the case, because VDFs are not suitable for encoding an arbitrary private message and they take a public message as input, whereas TLPs have been designed to conceal a private input message.

3 Preliminaries

3.1 Notations and Informal Threat Model

We define Δ_u as the period within which client c_u would like its puzzle's solution m_u to remain secret. We define U as the universe of a solution m_u. In this paper, $\ddot{t}$ refers to the total number of leaders. We set $\bar{t} = \ddot{t} + 2$. We denote by $\lambda \in \mathbb{N}$ the security parameter. For certain system parameters, we use polynomial $poly(\lambda)$ to state the parameter is a polynomial function of λ. We define a public set X as $X = \{x_1, \ldots, x_n\}$, where $x_i \neq x_j$, $x_i \neq 0$, and $x_i \notin U$.

We define a hash function $\mathtt{G} : \{0,1\}^* \rightarrow \{0,1\}^{poly(\lambda)}$ that maps an arbitrary-length message to a message of length $poly(\lambda)$. We denote a null value or set by $\perp$. By $||v||$ we mean the bit-size of v and by $||\vec{v}||$ we mean the total bit-size of elements of $\vec{v}$. We denote by p a large prime number, where $\log_2(p)$ is the security parameter, e.g., $\log_2(p) = 128$.

To ensure generality in the definition of our verification algorithms, we adopt notations from zero-knowledge proof systems [11,26]. Let R_{cmd} be an efficient binary relation that consists of pairs of the form (stm_{cmd}, wit_{cmd}), where stm_{cmd} is a statement and wit_{cmd} is a witness. Let $\mathcal{L}_{cmd}$ be the language (in $\mathcal{NP}$) associated with R_{cmd}, i.e., $\mathcal{L}_{cmd} = \{stm_{cmd} | \ \exists wit_{cmd}$ s.t. $R(stm_{cmd}, wit_{cmd}) = 1\}$. A (zero-knowledge) proof for $\mathcal{L}_{cmd}$ allows a prover to convince a verifier that $stm_{cmd} \in \mathcal{L}_{cmd}$ for a common input stm_{cmd} (without revealing wit_{cmd}). In this paper, two main types of verification occur (1) verification of a single client's puzzle solution, in this case, $cmd =$ clientPzl, and (2) verification of a linear combination, in this case, $cmd =$ evalPzl.

We assume parties interact with each other through a secure channel. Moreover, we consider a strong malicious server (or active adversary) and semi-honest clients. A malicious server is considered strong because it can act arbitrarily and access the secret keys and parameters of a subset of clients. As previously stated, the scheme designates (at random) a subset of clients as leaders. Let $\mathcal{I}$ be this subset, containing $\ddot{t}$ leaders. We also allow the malicious server to gain access to the secret keys and parameters of some of these leaders.

We proceed to elaborate on this. Let set $P = \{s, c_1, \ldots, c_n\}$ contain all the parties involved in the scheme. We allow the adversary to adaptively corrupt a subset $\mathcal{W}$ of P. It will fully corrupt s and act arbitrarily on its behalf. It will also retrieve the secret keys of a subset of clients in P. Specifically, we define a threshold t and require the number of non-corrupted leaders (i.e., the parties in $\mathcal{I}$) to be at least t. For instance, when $|P| = 100$, and the total number of leaders is 5 (i.e., $\ddot{t} = 5$), and $t = 2$, then the adversary may corrupt 98 parties in P (i.e., $|\mathcal{W}| = 98$), as long as at most 3 parties from $\mathcal{I}$ are in $\mathcal{W}$, i.e., $|\mathcal{W} \cap \mathcal{I}| \leq t$. Section 4 presents a formal definition of security.

3.2 Pseudorandom Function

Informally, a pseudorandom function is a deterministic function that takes a key of length λ and an input; and outputs a value. The security of PRF states that the output of PRF is indistinguishable from that of a truly random function. In this paper, we use pseudorandom functions: $\mathtt{PRF} : \{0,1\}^* \times \{0,1\}^{poly(\lambda)} \rightarrow \mathbb{F}_p$. In practice, a pseudorandom function can be obtained from an efficient block cipher [30]. In this work, we use PRF to derive pseudorandom values to blind (or encrypt) secret messages.

3.3 Oblivious Linear Function Evaluation

Oblivious Linear function Evaluation (OLE) is a two-party protocol that involves a sender and receiver. In OLE, the sender has two inputs $a, b \in \mathbb{F}_p$ and the receiver has a single input, $c \in \mathbb{F}_p$. The protocol allows the receiver to learn only $s = a \cdot c + b \in \mathbb{F}_p$, while the sender learns nothing. Ghosh *et al.* [28] proposed an efficient OLE that has $O(1)$ overhead and involves mainly symmetric-key operations.[3]

Later, in [29] an enhanced OLE, called $\mathtt{OLE}^+$, was proposed. The latter ensures that the receiver cannot learn anything about the sender's inputs, even if it sets its input to 0. $\mathtt{OLE}^+$ is also accompanied by an efficient symmetric-key-based verification mechanism that enables a party to detect its counterpart's misbehavior during the protocol's execution. In this paper, we use $\mathtt{OLE}^+$ to securely switch the blinding factors of secret messages (encoded in the form of puzzles) held by a server. We refer readers to Appendix A, for the construction of $\mathtt{OLE}^+$.

[3] The scheme uses an Oblivious Transfer (OT) extension as a subroutine. However, the OT extension requires only a constant number of public-key-based OT invocations. The rest of the OT invocations are based on symmetric-key operations. The exchanged messages in the OT extension are defined over a small-sized field, e.g., a field of size 128-bit [7].

3.4 Polynomial Representation of a Message

In general, encoding a message m as a polynomial $\boldsymbol{\pi}(x)$ allows us to impose a certain structure on the message. Polynomial representation has been used in various contexts, such as in secret sharing [44], private set intersection [33], or error-correcting codes [41]. There are two common approaches to encode m in $\boldsymbol{\pi}(x)$:

1. setting m as the constant terms of $\boldsymbol{\pi}(x)$, e.g., $m + \sum_{j=1}^{n} x^j \cdot a_j \bmod p$.
2. setting m as the root of $\boldsymbol{\pi}(x)$, e.g., $\boldsymbol{\pi}(x) = (x - m) \cdot \boldsymbol{\tau}(x) \bmod p$.

In this paper, we employ both approaches. The former enables us to perform a linear combination of the constant terms of different polynomials. Meanwhile, we utilize the latter to insert a secret random root into the polynomials encoding the messages. Consequently, the resulting polynomial representing the linear combination encompasses this specific root, facilitating the verification of the computations' correctness.

Point-Value Form. Polynomials can be represented in the "point-value form". Specifically, a polynomial $\boldsymbol{\pi}(x)$ of degree n can be represented as a set of l ($l > n$) point-value pairs $\{(x_1, \pi_1), \ldots, (x_l, \pi_l)\}$ such that all x_i are distinct non-zero points and $\pi_i = \boldsymbol{\pi}(x_i)$ for all i, $1 \leq i \leq m$. A polynomial in this form can be converted into coefficient form via polynomial interpolation, e.g., via Lagrange interpolation [6].

Arithmetic of Polynomials in Point-Value Form. Arithmetic of polynomials in point-value representation can be done by adding or multiplying the corresponding y-coordinates of polynomials.

Let a be a scalar and $\{(x_1, \pi_1), \ldots, (x_l, \pi_l)\}$ be (y, x)-coordinates of a polynomial $\boldsymbol{\pi}(x)$. Then, the polynomial $\boldsymbol{\theta}$ interpolated from $\{(x_1, a \cdot \pi_1), \ldots, (x_l, a \cdot \pi_l)\}$ is the product of a and polynomial $\boldsymbol{\pi}(x)$, i.e., $\boldsymbol{\theta}(x) = a \cdot \boldsymbol{\pi}(x)$.

3.5 Unforgeable Encrypted Polynomial with a Hidden Root

An interesting property of encrypted polynomials has been stated in [22]. Informally, it can be described as follows. Let us consider a polynomial $\boldsymbol{\pi}(x)$ (where $\boldsymbol{\pi}(x) \in \mathbb{F}_p[x]$) that has a random secret root β. We can represent $\boldsymbol{\pi}(x)$ in the point-value form and then encrypt its y-coordinates. We give all the x-coordinates and encrypted y-coordinates to an adversary and we locally delete all the y-coordinates. The adversary may modify any subset of the encrypted y-coordinates and send back to us the encrypted y-coordinates (some of which might have been modified). If we decrypt all the y-coordinates sent by the adversary and then interpolate a polynomial $\boldsymbol{\pi}'(x)$, the probability that $\boldsymbol{\pi}'(x)$ will have the root β is negligible in the security parameter $\lambda = \log_2(p)$. Below, we formally state it.

Theorem 1 (Unforgeable Encrypted Polynomial with a Hidden Root). *Let $\boldsymbol{\pi}(x)$ be a polynomial of degree n with a random root β, and $\{(x_1, \pi_1), \ldots, (x_l, \pi_l)\}$ be point-value representation of $\boldsymbol{\pi}(x)$, where $l > n$, p denote a large prime number, $\log_2(p) = \lambda'$ is the security parameter, $\boldsymbol{\pi}(x) \in \mathbb{F}_p[x]$, and $\beta \xleftarrow{\$} \mathbb{F}_p$. Let $o_i = w_i \cdot (\pi_i + z_i) \bmod p$ be the encryption of each y-coordinate π_i of $\boldsymbol{\pi}(x)$, using values w_i and r_i chosen uniformly at random from $\mathbb{F}_p$. Given $\{(x_1, o_1), \ldots, (x_l, o_l)\}$, the probability that an adversary (which does not know $(w_1, r_1), \ldots, (w_l, r_l), \beta$) can forge $[o_1, \ldots, o_l]$ to arbitrary $\vec{\ddot{o}} = [\ddot{o}_1, \ldots, \ddot{o}_l]$, such that: (i) $\exists \ddot{o}_i \in \vec{\ddot{o}}, \ddot{o}_i \neq o_i$, and (ii) the polynomial $\boldsymbol{\pi}'(x)$ interpolated from unencrypted y-coordinates $\{\big(x_1, (w_1 \cdot \ddot{o}_1) - z_l\big), \ldots, \big(x_l, \big(w_l \cdot \ddot{o}_l\big) - z_l\big)\}$ will have root β is negligible in λ', i.e.,*

$$Pr[\boldsymbol{\pi}'(\beta) \bmod p = 0] \leq \mu(\lambda')$$

We refer readers to [22, p.160] for the proof of Theorem 1. In this paper, we use the concept of the unforgeable encrypted polynomial with a hidden root to detect a server's misbehaviors.

3.6 Commitment Scheme

A commitment scheme comprises a sender and a receiver, and it encompasses two phases: commitment and opening. During the commitment phase, the sender commits to a message, using the algorithm $\texttt{Com}(m, r) = com$, where m is the message and r is a secret value randomly chosen from $\{0,1\}^{poly(\lambda)}$. Once the commitment phase concludes, the sender forwards the commitment com to the receiver.

In the opening phase, the sender transmits the pair $\hat{m} := (m, r)$ to the receiver, who proceeds to verify its correctness using the algorithm $\texttt{Ver}(com, \hat{m})$. The receiver accepts the message if the output is equal to 1. A commitment scheme must adhere to two properties. (1) Hiding: this property ensures that it is computationally infeasible for an adversary, in this case, the receiver, to gain any knowledge about the committed message m until the commitment com is opened. (2) Binding: this property guarantees that it is computationally infeasible for an adversary, which in this context is the sender, to open a commitment com to different values $\hat{m}' := (m', r')$ than the ones originally used during the commit phase. In other words, it should be infeasible to find an alternative pair $\hat{m}'$ such that $\texttt{Ver}(com, \hat{m}) = \texttt{Ver}(com, \hat{m}') = 1$ while $\hat{m} \neq \hat{m}'$.

There is a well-known efficient hash-based commitment scheme. It involves computing $\texttt{Q}(m||r) = com$ during the commitment. The verification step requires confirming whether $\texttt{Q}(m||r) \stackrel{?}{=} com$. Here $\texttt{Q} : \{0,1\}^* \rightarrow \{0,1\}^{poly(\lambda)}$ is a collision-resistant hash function, meaning that the probability of finding two distinct values m and m' such that $\texttt{Q}(m) = \texttt{Q}(m')$ is negligible regarding the security parameter λ. In this paper, we use this commitment scheme to detect a server's misbehaviors.

3.7 Time-Lock Puzzle

In this section, we restate the TLP's formal definition, taken from [5].

Definition 1. *A TLP consists of three algorithms:* ($\mathsf{Setup}_{\mathsf{TLP}}$, $\mathsf{GenPuzzle}_{\mathsf{TLP}}$, $\mathsf{Solve}_{\mathsf{TLP}}$) *defined below. It meets completeness and efficiency properties. TLP involves a client c and a server s.*

* *Algorithms:*

- $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, max_{ss}) \rightarrow (pk, sk)$. *A probabilistic algorithm run by c. It takes as input a security parameter,* 1^λ*, time parameter* Δ *that specifies how long a message must remain hidden in seconds, and time parameter* max_{ss} *which is the maximum number of squaring that a solver (with the highest level of computation resources) can perform per second. It outputs a pair* (pk, sk) *that contains public and private keys.*
- $\mathsf{GenPuzzle}_{\mathsf{TLP}}(m, pk, sk) \rightarrow o$. *A probabilistic algorithm run by c. It takes as input a solution* m *and* (pk, sk)*. It outputs a puzzle* o*.*
- $\mathsf{Solve}_{\mathsf{TLP}}(pk, o) \rightarrow s$. *A deterministic algorithm run by s. It takes as input pk and o. It outputs a solution* s*.*

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

* *Efficiency. The run-time of* $\mathsf{Solve}_{\mathsf{TLP}}(pk, o)$ *is upper-bounded by* $poly(\Delta, \lambda)$*.*

The security of a TLP requires that the puzzle's solution stay confidential from all adversaries running in parallel within the time period, Δ. 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 TLP is secure if for all* λ *and* Δ*, all probabilistic polynomial time (PPT) adversaries* $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$ *where* $\mathcal{A}_1$ *runs in total time* $O(poly(\Delta, \lambda))$ *and* $\mathcal{A}_2$ *runs in time* $\delta(\Delta) < \Delta$ *using at most* $\xi(\Delta)$ *parallel processors, there is a negligible function* $\mu()$*, such that:*

$$Pr\left[\begin{array}{l}\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, max_{ss}) \rightarrow (pk, sk)\\ \mathcal{A}_1(1^\lambda, pk, \Delta) \rightarrow (m_0, m_1, state)\\ b \xleftarrow{\$} \{0,1\}\\ \mathsf{GenPuzzle}_{\mathsf{TLP}}(m_b, pk, sk) \rightarrow o\\ \hline \mathcal{A}_2(pk, o, state) \rightarrow b\end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

We refer readers to Appendix B for the description of the original RSA-based TLP, which is the core of the majority of TLPs. By definition, TLPs are sequential functions. Their construction requires that a sequential function, such as modular squaring, is invoked iteratively a fixed number of times. The sequential function and iterated sequential functions notions, in the presence of an adversary possessing a polynomial number of processors, are formally defined in [13]. We restate the definitions in Appendix C.

4 Definition of Verifiable Homomorphic Linear Combination TLP

In general, the basic functionality $\mathcal{F}^{\text{PLC}}$ that any n-party Private Linear Combination (PLC) computes takes as input a pair of coefficient q_j and plaintext value m_j from j-th party (for every $j, 1 \leq j \leq n$), and returns their linear combination $\sum_{j=1}^{n} q_j \cdot m_j$ to each party. More formally, $\mathcal{F}^{\text{PLC}}$ is defined as:

$$\mathcal{F}^{\text{PLC}}\big((q_1, m_1), \ldots, (q_n, m_n)\big) \rightarrow (q_1 \cdot m_1 + \ldots + q_n \cdot m_n) \tag{1}$$

Note, $\mathcal{F}^{\text{PLC}}$ implies that corrupt parties that collude with each other can always deduce the linear combination of non-colluding parties' inputs from the output of $\mathcal{F}^{\text{PLC}}$. The aforementioned point holds for any scheme that realizes this functionality (including the ones in [37,18] and ours) regardless of the primitives used.

Next, we present a definition of Verifiable Homomorphic Linear Combination TLP ($\mathcal{VHLC}$-$\mathcal{TLP}$), by initially presenting the syntax followed by the security and correctness definitions.

4.1 Syntax

Definition 3 (Syntax). *A Verifiable Homomorphic Linear Combination TLP ($\mathcal{VHLC}$-$\mathcal{TLP}$) scheme consists of six algorithms: $\mathcal{VHLC}$-$\mathcal{TLP}$ = ($\mathsf{S.Setup}$, $\mathsf{C.Setup}$, $\mathsf{GenPuzzle}$, $\mathsf{Evaluate}$, Solve, Verify), defined below.*

- $\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_s$. *It is an algorithm run by the server s. It takes security parameters $1^\lambda, \ddot{t}$, and t. It generates a pair $K_s := (sk_s, pk_s)$, that includes a set of secret parameters sk_s and a set of public parameters pk_s. It returns K_s. Server s publishes pk_s.*
- $\mathsf{C.Setup}(1^\lambda) \rightarrow K_u$. *It is a probabilistic algorithm run by a client c_u. It takes security parameter 1^λ as input. It returns a pair $K_u := (sk_u, pk_u)$ of secret key sk_u and public key pk_u. Client c_u publishes pk_u.*
- $\mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$. *It is an algorithm run by c_u. It takes as input plaintext message m_u, key pair K_u, server's public parameters set pk_s, time parameter Δ_u determining the period which m_u should remain private, and the maximum number max_{ss} of sequential steps (e.g., modular squaring) per second that the strongest solver can perform. It returns a puzzle vector $\vec{o}_u$ (representing a single puzzle) along with a pair $prm_u := (sp_u, pp_u)$ of secret parameter sp_u and public parameter pp_u of the puzzle, which may include the index of c_u. Client c_u publishes $(\vec{o}_u, pp_u)$.*
- $\mathsf{Evaluate}(\langle s(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), c_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(Evl)})$. *It is an (interactive) algorithm run by s (and each client in $\{c_1, \ldots, c_n\}$). When no interaction between s and the clients is required, the clients' inputs will be null $\perp$. Server s takes as input vector $\vec{o}$ of n clients' puzzles, time parameter Δ within which the evaluation result should remain private (where $\Delta < min(\Delta_1, \ldots, \Delta_n)$), max_{ss}, $\vec{pp} = [pp_1, \ldots, pp_n]$, $\vec{pk} = [pk_1, \ldots, pk_n]$, and pk_s. Each c_u takes as input $\Delta, max_{ss}, K_u, prm_u$, coefficient q_u, and pk_s. It returns a vector of public parameters $\vec{pp}^{(Evl)}$ and a puzzle vector $\vec{g}$, representing a single puzzle. Server s publishes $\vec{g}$ and the clients publish $\vec{pp}^{(Evl)}$.*

- $\mathsf{Solve}(\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd) \rightarrow (m, \zeta)$. *It is a deterministic algorithm run by s. It takes as input a single client c_u's puzzle vector $\vec{o}_u$, the puzzle's public parameter pp_u, a vector $\vec{g}$ representing the puzzle that encodes evaluation of all clients' puzzles, a vector of public parameter $\vec{pp}^{(Evl)}$, pk_s, and a command cmd, where cmd $\in$ {clientPzl, evalPzl}. When cmd = clientPzl, it solves puzzle $\vec{o}_u$ (which is an output of* $\mathsf{GenPuzzle}()$*), this yields a solution m. In this case, input $\vec{g}$ can be null $\perp$. When cmd = evalPzl, it solves puzzle $\vec{g}$ (which is an output of* $\mathsf{Evaluate}()$*), this results in a solution m. In this case, input $\vec{o}_u$ can be $\perp$. Depending on the value of cmd, it generates a proof ζ (asserting that $m \in \mathcal{L}_{cmd}$). It outputs plaintext solution m and proof ζ. Server s publishes (m, ζ).*

- $\mathsf{Verify}(m, \zeta, \vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd) \rightarrow \ddot{v} \in \{0, 1\}$. *It is a deterministic algorithm run by a verifier. It takes as input a plaintext solution m, proof ζ, puzzle $\vec{o}_u$ of a single client c_u, public parameters pp_u of $\vec{o}_u$, a puzzle $\vec{g}$ for a linear combination of puzzles, public parameters $\vec{pp}^{(Evl)}$ of $\vec{g}$, server's public key pk_s (where $pk_s \in K_s$), and command cmd that determines whether the verification corresponds to c's single puzzle or linear combination of puzzles. It returns 1 if it accepts the proof. It returns 0 otherwise.*

To be more precise, in the above, the prover is required to generate a witness/proof ζ for the language $\mathcal{L}_{cmd} = \{stm_{cmd} = (pp, m) |\ R_{cmd}(stm_{cmd}, \zeta) = 1\}$, where if cmd = clientPzl, then $pp = pp_u$ and if cmd = evalPzl, then $pp = \vec{pp}^{(\text{Evl})}$.

4.2 Security Model

A $\mathcal{VHLC}$-$\mathcal{TLP}$ scheme must satisfy *security* (i.e., privacy and solution-validity), *completeness*, *efficiency*, and *compactness* properties. Each security property of a $\mathcal{VHLC}$-$\mathcal{TLP}$ scheme is formalized through a game between a challenger $\mathcal{E}$ that plays the role of honest parties and an adversary $\mathcal{A}$ that controls the corrupted parties. In this section, initially, we define a set of oracles that will strengthen the capability of $\mathcal{A}$. After that, we provide formal definitions of $\mathcal{VHLC}$-$\mathcal{TLP}$'s properties.

Oracles. To enable $\mathcal{A}$ to adaptively choose plaintext solutions and corrupt parties, we provide $\mathcal{A}$ with access to two oracles: (i) puzzle generation $\mathsf{O.GenPuzzle}()$ and (ii) evaluation $\mathsf{O.Evaluate}()$. Furthermore, to enable $\mathcal{A}$ to have access to the messages exchanged between the corrupt and honest parties during the execution of $\mathsf{Evaluate}()$, we define an oracle called $\mathsf{O.Reveal}()$. Below, we define these oracles.

- $\mathsf{O.GenPuzzle}(st_{\mathcal{E}}, GeneratePuzzle, m_u, \Delta_u) \rightarrow (\vec{o}_u, pp_u)$. It is executed by the challenger $\mathcal{E}$. It receives a query $(GeneratePuzzle, m_u, \Delta_u)$, where $GeneratePuzzle$ is a string, m_u is a plaintext message, and Δ_u is a time parameter. $\mathcal{E}$ retrieves (K_u, pk_s, max_{ss}) from its state $st_{\mathcal{E}}$ and then executes $\mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$, where $prm_u := (sp_u, pp_u)$. It returns $(\vec{o}_u, pp_u)$ to $\mathcal{A}$.

- $\mathsf{O.Evaluate}(\mathcal{W}, \mathcal{I}, t, st_{\mathcal{E}}, evaluate) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$. It is executed interactively between the corrupt parties specified in $\mathcal{W}$ and the challenger $\mathcal{E}$. If $|\mathcal{W} \cap \mathcal{I}| > t$, then $\mathcal{E}$ returns $(\perp, \perp)$ to $\mathcal{A}$. Otherwise, it interacts with the corrupt parties specified in $\mathcal{W}$ to run $\mathsf{Evaluate}(\langle \hat{s}(input_s), \hat{c}_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, \hat{c}_n(\Delta, max_{ss},\ K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$, where the inputs of honest parties are retrieved by $\mathcal{E}$ from $st_{\mathcal{E}}$ and if $\hat{s} \in \mathcal{W}$, then $\hat{s}$ may provide arbitrary inputs $input_s$ during the execution of $\mathsf{Evaluate}$. However, when $\hat{s} \notin \mathcal{W}$ then $\hat{s}$ is an honest server (i.e., $\hat{s} = s$) and $input_s = (\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s)$. Moreover, when $\hat{c}_j \notin \mathcal{W}$ then $\hat{c}_j$ is an honest client (i.e., $\hat{c}_j = c_j$). $\mathcal{E}$ returns $(\vec{g}, \vec{pp}^{(\text{Evl})})$ to $\mathcal{A}$.

- $\mathsf{O.Reveal}(\mathcal{W}, \mathcal{I}, t, st_{\mathcal{E}}, reveal^{(\text{Evl})}, \vec{g}, \vec{pp}^{(\text{Evl})}) \rightarrow transcript^{(\text{Evl})}$. It is run by $\mathcal{E}$ which is provided with a set $\mathcal{W}$ of corrupt parties, a set of parties in $\mathcal{I}$, and a state $st_{\mathcal{E}}$. It receives a query $(reveal^{(\text{Evl})}, \vec{g}, \vec{pp}^{(\text{Evl})})$, where $Reveal^{(\text{Evl})}$ is a string, and pair $(\vec{g}, \vec{pp}^{(\text{Evl})})$ is an output pair (previously) returned by an instance of $\mathsf{Evaluate}()$. If $|\mathcal{W} \cap \mathcal{I}| > t$ or the pair $(\vec{g}, \vec{pp}^{(\text{Evl})})$ was never generated, then the challenger sets $transcript^{(\text{Evl})}$ to $\emptyset$ and returns $transcript^{(\text{Evl})}$ to $\mathcal{A}$. Otherwise, the challenger retrieves from $st_{\mathcal{E}}$ a set of messages that honest parties sent to corrupt parties in $\mathcal{W}$ during the execution of the specific instance of $\mathsf{Evaluate}()$. It appends these messages to $transcript^{(\text{Evl})}$ and returns $transcript^{(\text{Evl})}$ to $\mathcal{A}$.

Properties. Next, we formally define the primary properties of a $\mathcal{VHLC}$-$\mathcal{TLP}$ scheme, beginning with the privacy property. Informally, *privacy* states that the solution m to a single puzzle must remain concealed for a predetermined duration from any adversaries equipped with a polynomial number of processors. More precisely, an adversary with a running time of $\delta(T)$ (where $\delta(T) < T$) is unable to discover a message significantly earlier than $\delta(\Delta)$. This requirement also applies to the result of the evaluation. Specifically, the plaintext message representing the linear evaluation of messages must remain undisclosed to the previously described adversary within a predefined period.

To capture privacy, we define an experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^{\lambda}, n, \ddot{t}, t)$, that involves a challenger $\mathcal{E}$ which plays honest parties' roles and a pair of adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. This experiment considers a strong adversary that has access to two oracles, puzzle generation $\mathsf{O.GenPuzzle}()$ and evaluation $\mathsf{O.Evaluate}()$. It may adaptively corrupt a subset $\mathcal{W}$ of the parties involved P, i.e., $\mathcal{W} \subset P = \{s, c_1, \ldots, c_n\}$, and retrieve secret keys of corrupt parties (as shown in lines 9–13). Given the set of corrupt parties, their secret keys, having access to $\mathsf{O.GenPuzzle}()$ and $\mathsf{O.Evaluate}()$, $\mathcal{A}_1$ outputs a pair of messages $(m_{0,u}, m_{1,u})$ for each client c_u (line 14). Next, $\mathcal{E}$ for each pair of messages provided by $\mathcal{A}_1$ picks a random bit b_u and generates a puzzle and related public parameter for the message with index b_u (lines 15–18).

Given all the puzzles, related parameters, and access to the two oracles, $\mathcal{A}_1$ outputs a state (line 19). With this state as input, $\mathcal{A}_2$ guesses the value of bit b_u for its chosen client (line 20). The adversary wins the game (i.e., the experiment outputs 1) if its guess is correct for a non-corrupt client (line 21). It is important to note that during this phase, the experiment excludes corrupt clients because the adversary can always correctly identify the bit chosen for a corrupt client, given the knowledge of the corrupt client's secret key (by decrypting the puzzle quickly and identifying which message was selected by $\mathcal{E}$).

The experiment proceeds to the evaluation phase, imposing a constraint (line 22) on the number t of certain parties the adversary corrupts when executing $\mathsf{Evaluate}()$. This constraint is defined such that $\mathcal{E}$ selects a set $\mathcal{I}$ of parties (lines 6–8), where $|\mathcal{I}| = \ddot{t}$. The size of $\mathcal{I}$ can be small. The experiment returns 0 and halts if $\mathcal{A}$ corrupts more than t parties in $\mathcal{I}$.

Next, the corrupt parties and $\mathcal{E}$ interactively execute $\mathsf{Evaluate}()$ (line 23). During the execution of $\mathsf{Evaluate}()$, $\mathcal{A}_1$ has access to the private keys of corrupt parties (as stated in lines 24 and 25). The experiment also enables $\mathcal{A}_1$ to learn about the messages exchanged between honest and corrupt parties, by providing $\mathcal{A}_1$ with access to an oracle called $\mathsf{O.Reveal}$ (); given this transcript, $\mathcal{A}_1$ outputs a state (line 26). Having access to this state and the output of $\mathsf{Evaluate}()$, adversary $\mathcal{A}_2$ guesses the value of bit b_u for its chosen client (line 27). The adversary wins the game, if its guess is correct for a non-corrupt client (line 28).

Definition 4 (Privacy). *A $\mathcal{VHLC}$-$\mathcal{TLP}$ scheme is privacy-preserving if for any security parameter λ, any difficulty parameter $T = \Delta_l \cdot max_{ss}$ (where $\Delta_l \in \{\Delta, \Delta_1, \ldots, \Delta_n\}$ is the period, polynomial in λ, within which a message m must remain hidden and max_{ss} is a constant in λ), any plaintext input message $m_1, \ldots, m_n$ and coefficient $q_1, \ldots, q_n$ (where each m_u and q_u belong to the plaintext universe U), any security parameters $\ddot{t}, t$ (where $1 \leq \ddot{t}, t \leq n$ and $\ddot{t} \geq t$), and any polynomial-size adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in time $O(poly(T, \lambda))$ and $\mathcal{A}_2$ runs in time $\delta(T) < T$ using at most $\bar{poly}(T)$ parallel processors, there exists a negligible function $\mu()$ such that for any experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^{\lambda}, n, \ddot{t}, t)$:*

$\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^\lambda,\ n,\ \ddot{t},\ t)$

1: $\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_s := (sk_s, pk_s)$
2: For $u = 1, \ldots, n$ do :
3: $\quad \mathsf{C.Setup}(1^\lambda) \rightarrow K_u := (sk_u, pk_u)$
4: $state \leftarrow \{pk_s, pk_1, \ldots, pk_n\}, \mathcal{W} \leftarrow \emptyset, \mathcal{I} \leftarrow \emptyset$
5: $K \leftarrow \emptyset, cont \leftarrow True, counter \leftarrow 0, \vec{b} \leftarrow \mathbf{0}$
6: For $1, \ldots, \ddot{t}$ do :
7: $\quad$ Select a from $\{1, \ldots, n\}$
8: $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{\mathcal{B}_a\}$
9: While $(cont = True)$ do :
10: $\quad \mathcal{A}_1(state, \mathsf{O.GenPuzzle}(), \mathsf{O.Evaluate}(), K, \mathcal{I}, \Delta_1, \ldots, \Delta_n, \Delta, max_{ss}) \rightarrow (state, cont, \mathcal{B}_j)$
11: $\quad$ If $cont = True$, then
12: $\quad\quad \mathcal{W} \leftarrow \mathcal{W} \ \cup \ \{\mathcal{B}_j\}$
13: $\quad\quad K \leftarrow K_{\mathcal{B}_j}$
14: $\mathcal{A}_1(state, K, \mathsf{O.GenPuzzle}(), \mathsf{O.Evaluate}(), \mathcal{W}) \rightarrow \vec{m} = [(m_{0,1}, m_{1,1}), \ldots, (m_{0,n}, m_{1,n})]$
15: For $u = 1, \ldots, n$ do :
16: $\quad b_u \xleftarrow{\$} \{0, 1\}$
17: $\quad \vec{b}[u] \leftarrow b_u$
18: $\quad \mathsf{GenPuzzle}(m_{b_u,u}, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$
19: $\mathcal{A}_1(state, K, \mathcal{W}, \mathsf{O.GenPuzzle}(), \mathsf{O.Evaluate}(), \vec{o}_1, \ldots, \vec{o}_n, \vec{pp}_1, \ldots, \vec{pp}_n) \rightarrow state$
20: $\mathcal{A}_2(\vec{o}_1, \ldots, \vec{o}_n, \vec{pp}_1, \ldots, \vec{pp}_n, state) \rightarrow (b'_u, u)$
21: If $(b'_u = \vec{b}[u]) \wedge (\mathcal{B}_u \notin \mathcal{W})$, then return 1
22: If $|\mathcal{W} \cap \mathcal{I}| > t$, then return 0
23: $\mathsf{Evaluate}(\langle \hat{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), \hat{c}_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, \hat{c}_n(\Delta, max_{ss},$
$K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(Evl)}), s.t.$
24: $\quad$ If $\hat{s} \in \mathcal{W}$, then $\hat{s}$ has access to $state$ and K, else $\hat{s}$ is an honest server, i.e., $\hat{s} = s$
25: $\quad$ If $\hat{c}_j \in \mathcal{W}$, then $\hat{c}_j$ has access to $state$ and K, else $\hat{c}_j$ is an honest client, i.e., $\hat{c}_j = c_j$
26: $\mathcal{A}_1(state, K, \mathsf{O.Reveal}(), \mathcal{W}, \vec{g}, \vec{pp}^{(Evl)}) \rightarrow state$
27: $\mathcal{A}_2(\vec{o}, \vec{pp}, \vec{g}, \vec{pp}^{(Evl)}, state) \rightarrow (b'_u, u)$
28: If $(b'_u = \vec{b}[u]) \wedge (\mathcal{B}_u \notin \mathcal{W})$, then return 1, else return 0

it holds that:

$$Pr\left[\,\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^\lambda, n, \ddot{t}, t) \rightarrow 1\,\right] \leq \frac{1}{2} + \mu(\lambda)$$

In simple terms, *solution validity* requires that it should be infeasible for a probabilistic polynomial time (PPT) adversary to come up with an invalid solution (for a single client's puzzle or a puzzle encoding a linear combination of messages) and successfully pass the verification process. To capture solution validity, we define an experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{val}}(1^\lambda, n, \ddot{t}, t)$ that involves $\mathcal{E}$ which plays honest parties' roles and an adversary $\mathcal{A}$. Given the three types of oracles previously defined, $\mathcal{A}$ may corrupt a set of parties and learn their secret keys (lines 9–13).

Given the corrupt parties, their secret keys, and having access to $\mathsf{O.GenPuzzle}()$ and $\mathsf{O.Evaluate}()$, $\mathcal{A}$ outputs a message m_u for each client c_u (line 14). The experiment proceeds by requiring $\mathcal{E}$ to generate a puzzle for each message that $\mathcal{A}$ selected (lines 15 and 16).

The experiment returns 0 and halts, if $\mathcal{A}$ corrupts more than t parties in $\mathcal{I}$ (line 17). Otherwise, it allows the corrupt parties and $\mathcal{E}$ to interactively execute $\mathsf{Evaluate}()$ (line 18). During the execution of $\mathsf{Evaluate}()$, $\mathcal{A}$ has access to the private keys of corrupt parties (as stated in lines 19 and 20). Given the output of $\mathsf{Evaluate}()$ which is itself a puzzle, $\mathcal{E}$ solves the puzzle and outputs the solution (line 21).

The experiment enables $\mathcal{A}$ to learn about the messages exchanged between honest and corrupt parties during the execution of $\mathsf{Evaluate}()$, by providing $\mathcal{A}$ with access to an oracle called $\mathsf{O.Reveal}$ (); given this transcript,

the output of Evaluate(), and the plaintext solution, $\mathcal{A}$ outputs a solution and proof (line 22). $\mathcal{E}$ proceeds to check the validity of the solution and proof provided by $\mathcal{A}$. It outputs 1 (and $\mathcal{A}$ wins) if $\mathcal{A}$ persuades $\mathcal{E}$ to accept an invalid evaluation result (line 23).

$\mathcal{E}$ solves every client's puzzle (lines 24 and 25). Given the puzzles, solutions, and access to oracles O.GenPuzzle() and O.Evaluate(), $\mathcal{A}$ provides a solution and proof for its chosen client (line 26). The experiment proceeds by requiring $\mathcal{E}$ to check the validity of the solution and proof provided by $\mathcal{A}$. The experiment outputs 1 (and $\mathcal{A}$ wins) if $\mathcal{A}$ persuades $\mathcal{E}$ to accept an invalid message for a client's puzzle (line 27).

Definition 5 (Solution-Validity). *A $\mathcal{VHLC}$-$\mathcal{TLP}$ scheme preserves a solution validity, if for any security parameter λ, any difficulty parameter $T = \Delta_l \cdot max_{ss}$ (where $\Delta_l \in \{\Delta, \Delta_1, \ldots, \Delta_n\}$ is the period, polynomial in λ, within which a message m must remain hidden and max_{ss} is a constant in λ), any plaintext input message $m_1, \ldots, m_n$ and coefficient $q_1, \ldots, q_n$ (where each m_u and q_u belong to the plaintext universe U), any security parameters $\ddot{t}, t$ (where $1 \leq \ddot{t}, t \leq n$ and $\ddot{t} \geq t$), and any polynomial-size adversary $\mathcal{A}$ that runs in time $O(poly(T, \lambda))$, there exists a negligible function $\mu()$ such that for any experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathrm{val}}(1^\lambda, n, \ddot{t}, t)$:*

$\mathsf{Exp}^{\mathcal{A}}_{\mathrm{val}}(1^\lambda,\ n,\ \ddot{t},\ t)$

1: $\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_s := (sk_s, pk_s)$
2: For $u = 1, \ldots, n$ do :
3: $\quad \mathsf{C.Setup}(1^\lambda) \rightarrow K_u := (sk_u, pk_u)$
4: $state \leftarrow \{pk_s, pk_1, \ldots, pk_n\}, \mathcal{W} \leftarrow \emptyset, \mathcal{I} \leftarrow \emptyset$
5: $K \leftarrow \emptyset, cont \leftarrow True, counter \leftarrow 0$
6: For $1, \ldots, \ddot{t}$ do :
7: $\quad$ Select a from $\{1, \ldots, n\}$
8: $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{\mathcal{B}_a\}$
9: While $(cont = True)$ do :
10: $\quad \mathcal{A}(state, \mathsf{O.GenPuzzle}(), \mathsf{O.Evaluate}(), K, \mathcal{I}, \Delta_1, \ldots, \Delta_n, \Delta, max_{ss}) \rightarrow (state, cont, \mathcal{B}_j)$
11: $\quad$ If $cont = True$, then
12: $\quad\quad \mathcal{W} \leftarrow \mathcal{W} \cup \{\mathcal{B}_j\}$
13: $\quad\quad K \leftarrow K_{\mathcal{B}_j}$
14: $\mathcal{A}(state, K, \mathcal{W}, \mathsf{O.GenPuzzle}(), \mathsf{O.Evaluate}(), \mathcal{W}) \rightarrow \vec{m} = [m_1, \ldots, m_n]$
15: For $u = 1, \ldots, n$ do :
16: $\quad \mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$
17: If $|\mathcal{W} \cap \mathcal{I}| > t$, then return 0
18: $\mathsf{Evaluate}(\langle \hat{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), \hat{c}_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, \hat{c}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(Evl)}), s.t.$
19: $\quad$ If $\hat{s} \in \mathcal{W}$, then $\hat{s}$ has access to $state$ and K, else $\hat{s}$ is an honest server, i.e., $\hat{s} = s$
20: $\quad$ If $\hat{c}_j \in \mathcal{W}$, then $\hat{c}_j$ has access to $state$ and K, else $\hat{c}_j$ is an honest client, i.e., $\hat{c}_j = c_j$
21: $\mathsf{Solve}(.,.,\vec{g}, \vec{pp}^{(Evl)}, pk_s, evalPzl) \rightarrow (m, \zeta)$
22: $\mathcal{A}(state, K, \mathcal{W}, \mathsf{O.Reveal}(), m, \zeta, \vec{m}, \vec{o}_1, \ldots, \vec{o}_n, \vec{pp}_1, \ldots, \vec{pp}_n, \vec{g}, \vec{pp}^{(Evl)}) \rightarrow (m', \zeta')$
23: If $\mathsf{Verify}(m', \zeta', ., ., \vec{g}, \vec{pp}^{(Evl)}, pk_s, evalPzl) \rightarrow 1, s.t.\ m' \notin \mathcal{L}_{evalPzl}$, then return 1
24: For $u = 1, \ldots, n$ do :
25: $\quad \mathsf{Solve}(\vec{o}_u, pp_u, ., ., pk_s, clientPzl) \rightarrow (\bar{m}_u, \zeta_u)$
26: $\mathcal{A}(state, K, \mathcal{W}, \mathsf{O.GenPuzzle}(), \mathsf{O.Evaluate}(), (\bar{m}_1, \zeta_1), \ldots, (\bar{m}_u, \zeta_u), (m, \zeta), \vec{m}, \vec{o}_1, \ldots, \vec{o}_n, \vec{pp}_1, \ldots, \vec{pp}_n) \rightarrow (m'_u, \zeta'_u, u)$
27: If $\mathsf{Verify}(m'_u, \zeta'_u, \vec{o}_u, pp_u, ., ., pk_s, clientPzl) \rightarrow 1, s.t.\ m'_u \notin \mathcal{L}_{clientPzl}$, then return 1

it holds that:

$$Pr\left[\mathsf{Exp}^{\mathcal{A}}_{\mathrm{val}}(1^\lambda, n, \ddot{t}, t) \rightarrow 1\right] \leq \mu(\lambda)$$

Informally, *completeness* considers the behavior of the algorithms in the presence of honest parties. It asserts that a correct solution will always be retrieved by Solve() and Verify() will always return 1, given an honestly

generated solution. Since $\mathsf{Solve}()$ is used to find both a solution for (i) a single puzzle generated by a client and (ii) a puzzle that encodes linear evaluation of messages, we separately state the correctness concerning this algorithm for each case. For the same reason, we state the correctness concerning $\mathsf{Verify}()$ for each case. In the following definitions, since the experiments' description is relatively short, we integrate the experiment into the probability. Accordingly, we use the notation $\Pr\left[\frac{\mathsf{Exp}}{\mathsf{Cond}}\right]$, where Exp is an experiment, and Cond is the set of the corresponding conditions under which the property must hold.

Definition 6 (Completeness). *A $\mathcal{VHLC}$-$\mathcal{TLP}$ is complete if for any security parameter λ, any plaintext input message $m_1, \ldots, m_n$ and coefficient $q_1, \ldots, q_n$ (where each m_u and q_u belong to the plaintext universe U), any security parameters $\ddot{t}, t$ (where $1 \leq \ddot{t}, t \leq n$ and $\ddot{t} \geq t$), any difficulty parameter $T = \Delta_l \cdot max_{ss}$ (where Δ_l is the period, polynomial in λ, within which m must remain hidden and max_{ss} is a constant in λ), the following conditions are met.*

1. $\mathsf{Solve}(\vec{o}_u, pp_u, ., ., pk_s, cmd)$*, that takes a puzzle $\vec{o}_u$ encoding plaintext solution m_u and its related parameters, always returns m_u:*

$$Pr\left[\begin{array}{l}\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_s \\ \mathsf{C.Setup}(1^\lambda) \rightarrow K_u \\ \underline{\mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)} \\ \mathsf{Solve}(\vec{o}_u, pp_u, ., ., pk_s, cmd) \rightarrow (m_u, .)\end{array}\right] = 1$$

where $pp_u \in prm_u$ and $cmd = clientPzl$.

2. $\mathsf{Solve}(., ., \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd)$*, that takes (i) a puzzle $\vec{g}$ encoding linear combination $\sum_{u=1}^{n} q_u \cdot m_u$ of n messages, where each m_u is a plaintext message and q_u is a coefficient and (ii) their related parameters, always returns $\sum_{u=1}^{n} q_u \cdot m_u$:*

$$Pr\left[\begin{array}{l}\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_s \\ \mathsf{For}\ u = 1, \ldots, n\ \mathsf{do}: \\ \quad \mathsf{C.Setup}(1^\lambda) \rightarrow K_u \\ \quad \mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u) \\ \mathsf{Evaluate}\big(\langle s(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), c_1(\Delta, max_{ss}, K_1, prm_1, \\ \underline{q_1, pk_s), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle\big) \rightarrow (\vec{g}, \vec{pp}^{(Evl)})} \\ \mathsf{Solve}(., ., \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd) \rightarrow (\sum_{u=1}^{n} q_u \cdot m_u, .)\end{array}\right] = 1$$

where $\vec{o} = [\vec{o}_1, \ldots, \vec{o}_n]$, $\vec{pp} = [pp_1, \ldots, pp_n]$, $\vec{pk} = [\vec{pk}_1, \ldots, \vec{pk}_n]$, $pk_u \in K_u$, $pk_s \in K_s$, and $cmd = evalPzl$.

3. $\mathsf{Verify}(m_u, \zeta, \vec{o}_u, pp_u, ., ., pk_s, cmd)$*, that takes a solution for a client's puzzle, related proof, and public parameters, always returns 1:*

$$Pr\left[\begin{array}{l}\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_s \\ \mathsf{C.Setup}(1^\lambda) \rightarrow K_u \\ \mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u) \\ \underline{\mathsf{Solve}(\vec{o}_u, pp_u, ., ., pk_s, cmd) \rightarrow (m_u, \zeta)} \\ \mathsf{Verify}(m_u, \zeta, \vec{o}_u, pp_u, ., ., pk_s, cmd) \rightarrow 1\end{array}\right] = 1$$

where $cmd = clientPzl$.

4. $\mathsf{Verify}(m, \zeta, ., ., \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd)$*, that takes a solution for a puzzle that encodes a linear combination of n messages, related proof, and public parameters, always returns 1:*

$$Pr\left[\begin{array}{l}\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \to K_s \\ \mathsf{For}\ u = 1, \ldots, n\ \mathsf{do}: \\ \quad \mathsf{C.Setup}(1^\lambda) \to K_u \\ \quad \mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \to (\vec{o}_u, prm_u) \\ \mathsf{Evaluate}(\langle s(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), c_1(\Delta, max_{ss}, K_1, prm_1, \\ q_1, pk_s), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \to (\vec{g}, \vec{pp}^{(Evl)}) \\ \mathsf{Solve}(., ., \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd) \to (m, \zeta) \\ \hline \mathsf{Verify}(m, \zeta, ., ., \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd) \to 1\end{array}\right] = 1$$

where $cmd = evalPzl$.

Intuitively, *efficiency* states that (1) $\mathsf{Solve}()$ returns a solution in polynomial time, i.e., polynomial in the time parameter T, (2) $\mathsf{GenPuzzle}()$ generates a puzzle faster than solving it, with a running time of at most logarithmic in T, and (3) the running time of $\mathsf{Evaluate}()$ is faster than solving any puzzle involved in the evaluation, that should be at most logarithmic in T [24]. We proceed to define it formally.

Definition 7 (Efficiency). *A $\mathcal{VHLC}$-$\mathcal{TLP}$ is efficient if the following two conditions are satisfied:*

1. *The running time of* $\mathsf{Solve}(\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd)$ *is upper bounded by* $T \cdot poly(\lambda)$*, where* $poly()$ *is a fixed polynomial.*
2. *The running time of* $\mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss})$ *is upper bounded by* $poly'(\log T, \lambda)$*, where* $poly'()$ *is a fixed polynomial.*
3. *The running time of* $\mathsf{Evaluate}(\langle s(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), c_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \to (\vec{g}, \vec{pp}^{(Evl)})$ *is upper bounded by* $poly''\Big(\log T, \lambda, \mathcal{F}^{PLC}\big((q_1, m_1), \ldots, (q_n, m_n)\big)\Big)$*, where* $poly''()$ *is a fixed polynomial and* $\mathcal{F}^{PLC}()$ *is the functionality that computes a linear combination of messages (as stated in Relation 1) .*

Informally, *compactness* requires that the size of evaluated ciphertexts is independent of the complexity of the evaluation function $\mathcal{F}^{\text{PLC}}$.

Definition 8 (Compactness). *A $\mathcal{VHLC}$-$\mathcal{TLP}$ is compact if for any security parameter λ, any difficulty parameter $T = \Delta_l \cdot max_{ss}$, any plaintext input message $m_1, \ldots, m_n$ and coefficient $q_1, \ldots, q_n$ (where each m_u and q_u belong to the plaintext universe U), and any security parameters $\ddot{t}, t$ (where $1 \leq \ddot{t}, t \leq n$ and $\ddot{t} \geq t$), always* $\mathsf{Evaluate}()$ *outputs a puzzle (representation) whose bit-size is independent of $\mathcal{F}^{PLC}$'s complexity $O(\mathcal{F}^{PLC})$:*

$$Pr\left[\begin{array}{l}\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \to K_s \\ \mathsf{For}\ u = 1, \ldots, n\ \mathsf{do}: \\ \quad \mathsf{C.Setup}(1^\lambda) \to K_u \\ \quad \mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \to (\vec{o}_u, prm_u) \\ \hline \mathsf{Evaluate}\big(\langle s(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), c_1(\Delta, max_{ss}, K_1, prm_1, \\ q_1, pk_s), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle\big) \to (\vec{g}, \vec{pp}^{(Evl)}) \\ s.t. \\ ||\vec{g}|| = poly\Big(\lambda, ||\mathcal{F}^{PLC}\big((q_1, m_1), \ldots, (q_n, m_n)\big)||\Big)\end{array}\right] = 1$$

Definition 9 (Security). *A $\mathcal{VHLC}$-$\mathcal{TLP}$ is secure if it satisfies privacy and solution validity as outlined in Definitions 4 and 5.*

5 Tempora-Fusion

In this section, we present Tempora-Fusion, a protocol that realizes $\mathcal{VHLC}$-$\mathcal{TLP}$ and supports a homomorphic linear combination of puzzles. Briefly, it allows (1) each party to independently generate a puzzle and verify the correctness of a solution and (2) a server to verifiably compute the linear combination of plaintext solutions before the solutions are discovered. We must address several challenges to develop an efficient scheme. These challenges and our approaches to tackling them are outlined in Section 5.1. Subsequently, we explain Tempora-Fusion in Section 5.2.

5.1 Challenges to Overcome

Defining an Identical Group for all Puzzles. To facilitate correct computation on puzzles (or ciphertexts), the puzzles must be defined over the same group or field. For instance, in the case of the RSA-based time-lock puzzle, it can be over the same group $\mathbb{Z}_N$. There are a few approaches to deal with it. Below, we briefly discuss them.

1. *Jointly computing N*: Through this approach, all clients agree on two sufficiently large prime numbers p_1 and p_2 and then compute $N = p_1 \cdot p_2$, where $\log_2(N)$ is a security parameter. However, this approach will not be secure if one of the clients reveals the secret key $\phi(N) = (p_1 - 1) \cdot (p_2 - 1)$ to the server s. In this case, s can immediately retrieve the honest party's plaintext message without performing the required sequential work. Another approach is to require all participants (i.e., the server and all clients) to use secure multi-party computation (e.g., the solution proposed in [15]) to compute N, ensuring that no one learns $\phi(N)$. However, this approach necessitates that all clients are known and interact with each other at the outset of the scheme, which (a) is a strong assumption avoided in time-lock puzzle literature, and (b) significantly limits the scheme's flexibility and scalability. Furthermore, through this approach, it is not clear how each client can efficiently generate its puzzle without the knowledge of $\phi(N)$ and without relying on a trusted setup.
2. *Using a trusted setup*: Via this approach a fully trusted third party generates $N = p_1 \cdot p_2$ and publishes only N. As shown in [37], in the trusted setup setting, it is possible to efficiently generate puzzles without knowing $\phi(N)$. However, fully trusting the third party and assuming that it will not collude with and not reveal the secret to s may be considered a strong assumption, and not be desirable in scenarios where the solutions are highly sensitive. The homomorphic TLPs proposed in [37,24] rely on a trusted setup.
3. *Using the class group of an imaginary quadratic order*: Through this approach, one can use the class group of an imaginary quadratic order [19]. However, employing this approach will hamper the scheme's efficiency as the puzzle generation phase will no longer be efficient [37]. Hence, it will violate Requirement 2 in Definition 7, i.e., the efficiency of generating puzzles.

To address this challenge, we propose and use the following new technique, which has been simplified for the sake of presentation. We require the server s, only once to generate and publish a sufficiently large prime number p, e.g., $\log_2(p) \geq 128$.

We allow each client to *independently* choose its p_1 and p_2 and compute $N = p_1 \cdot p_2$. Thus, different clients will have different values of N. As in the original RSA-based TLP [42], each client generates $a = 2^T \bmod \phi(N)$ for its time parameter T, picks a random value r, and then generates $mk = r^a \bmod N$.

However, now, each client derives two pseudorandom values from mk as:

$$k = \texttt{PRF}(1, mk), \quad s = \texttt{PRF}(2, mk)$$

and then encrypts/masks its message using the derived values as one-time pads. For instance, the puzzle of a client with solution m is now:

$$o = k \cdot (m + s) \bmod p$$

As it is evident, now all clients' puzzles are defined over the identical field, $\mathbb{F}_p$. Given each puzzle o (as well as public parameters N, p, and r), a server s can find the solution, by computing mk where $mk = r^{2^T} \bmod N$ through repeated squaring of r modulo N, deriving pseudorandom values k and s from mk, and then decrypting o to get m.

Supporting Homomorphic Linear Combination. Establishing a field within which all puzzles are defined, we briefly explain the new techniques we utilize to facilitate a homomorphic linear combination of the puzzles. Recall that each client uses different random values (one-time pads) to encrypt its message. Therefore, naively applying linear combinations to the puzzles will not yield a correct result.

To maintain the result's correctness, we require the clients to *switch* their blinding factors to new ones when they decide to let s find a linear combination on the puzzles. To this end, a small subset of the clients (as leaders) independently generates new blinding factors. These blinding factors are generated in such a way that after a certain time when s solves puzzles related to the linear combination, s can remove these blinding factors. Each leader sends (the representation of some of) the blinding factors to the rest of the clients.

To switch the blinding factors securely, each client participates in an instance of $\texttt{OLE}^+$ with s. In this case, the client's input is (some of) the new blinding factors and the inverse of the old ones while the input of s is the client's puzzle. $\texttt{OLE}^+$ returns the output with refreshed blinding factors to s.

To ensure that s will learn only the linear combination of honest clients' solutions, without learning a solution for a client's puzzle, we require the leaders to generate and send to the rest of the clients some (of the keys used to generate) zero-sum pseudorandom values such that if all these values are summed up, they will cancel out each other. Each client also inserts these pseudorandom values into the instance of $\texttt{OLE}^+$ that it invokes with s, such that the result returned by $\texttt{OLE}^+$ to s will also be blinded by these pseudorandom values.

Efficient Verification of the Computation. It is essential for a homomorphic time-lock puzzle to enable a verifier to check whether the result computed by s is correct. However, this is a challenging goal to achieve for several reasons; for instance, (1) each client does not know other clients' solutions, (2) each client has prepared its puzzle independently of other clients, (3) server s may exclude or modify some of the clients' puzzles before, during, or after the computation, and (4) server s may corrupt some of the (leader) clients and learn their secrets, thereby aiding in compromising the correctness of the computation.

To achieve the above goal without using computationally expensive primitives (such as zero-knowledge proofs), we rely on the following novel techniques. Instead of using plaintext message m as a solution, we represent m as a polynomial $\boldsymbol{\pi}(x)$, and use $\boldsymbol{\pi}(x)$'s point-value representation (as described in Section 3.4) to represent m.

Now, we require each leader client to pick a random root and insert it into its outsourced puzzle (which is now a blinded polynomial), during the invocation of $\texttt{OLE}^+$ with s. It commits to this root (using a random value that s can discover when it solves a puzzle related to the linear combination) and publishes the commitment.

Each leader client sends (a blinded representation of) the root to the rest of the clients that insert it into their outsourced puzzle during the invocation of $\texttt{OLE}^+$. Server s sums all the outputs of $\texttt{OLE}^+$ instances and publishes the result. To find the plaintext result, s needs to solve a small set of puzzles. We will shortly explain why we are considering a set of puzzles rather than just a single puzzle.

If s follows the protocol's instruction, all roots selected by the leader clients will appear in the resulting polynomial that encodes the linear combination of all clients' plaintext solutions. However, if s misbehaves then the honest clients' roots will not appear in the result (according to Theorem 1). Therefore, a verifier can detect it.

For s to prove the computation's correctness, after it solves the puzzles related to the computation, it removes the blinding factors. Then, it extracts and publishes (i) the computation result (i.e., the linear combination

of the plaintext solutions), (ii) the roots, and (iii) the randomness used for the commitments. Given this information and public parameters, anyone can check the correctness of the computation's result.

We proceed to briefly explain how the four main challenges laid out above are addressed.

- Challenge (1): *each client does not know other clients' solutions*: during the invocation of $\texttt{OLE}^+$ they all agree on and insert certain roots to it, this ensures that all clients' polynomials have the same set of common roots. Thus, now they know what to expect from a correctly generated result.
- Challenge (2): *each client has prepared its puzzle independently of other clients*: during the invocation of $\texttt{OLE}^+$ they switch their old blinding factors to new ones that are consistent with other clients.
- Challenge (3): server s *may exclude or modify some of the clients' puzzles before, during, or after the computation*: the solutions to Challenges (1) and (2), the support of $\texttt{OLE}^+$ for verification, and the use of zero-sum blinding factors (that requires s sums all outputs of $\texttt{OLE}^+$ instances) ensure that such misbehavior will be detected by a verifier. Moreover, requiring s to open the commitments for the roots chosen by the leader clients ensures that s cannot exclude all parties' inputs, and come up with its choice of inputs encoding arbitrary roots.
- Challenge (4): server s *may corrupt some of the (leader) clients and learn their secrets, thereby aiding in compromising the correctness of the computation*: the messages each client receives from leader clients are blinded and reveal no information about their plaintext messages including the roots. Also, each leader client adds a layer of encryption (i.e., blinding factor) to the result. Thus, even if the secret keys of all leaders except one are revealed to the adversary, the adversary cannot find the solution sooner than intended. This is because s still needs to solve the puzzle of the honest party to remove the associated blinding factor from the result.

5.2 The Protocol

We present Tempora-Fusion in three tiers of detail, high-level overview, intermediate-level description, and detailed construction. Figure 1 outlines the workflow of Tempora-Fusion.

High-Level Overview. At this level, we provide a broad perspective on the protocol. Initially, each client generates a puzzle encoding a secret solution (or message) and sends the puzzle to a server s. Each client may generate and send its puzzle to s at different times. At this point, the client can locally delete its solution and go offline. Upon receiving each puzzle, s will work on it to find the solution at a certain time. The time that each client's solution is found can be independent of and different from that of other clients. Along with each solution, s generates a proof asserting the correctness of the solution. Anyone can efficiently verify the proof.

Later, possibly long after they have sent their puzzles to s, some clients whose puzzles have not been discovered yet, may get together and ask s to homomorphically combine their puzzles. The combined puzzles will encode the linear combination of their solutions, a.k.a., the computation result. The computation result will be discovered by s after a certain time, possibly before any of their puzzles will be discovered.

To enable s to impose a certain structure on its outsourced puzzle and homomorphically combine these puzzles, each client interacts with s. Each client also sends a short message to other clients. At this point, each client can go back offline. After a certain time, s finds the solution for the puzzle encoding the computation result. It also generates proof asserting the correctness of the solution. Anyone can efficiently check this proof, by ensuring that the result preserves a certain structure.

Server s eventually finds each client's single puzzle's solution. In this case, it publishes the solution and a proof that allows anyone to check the validity of the solution.

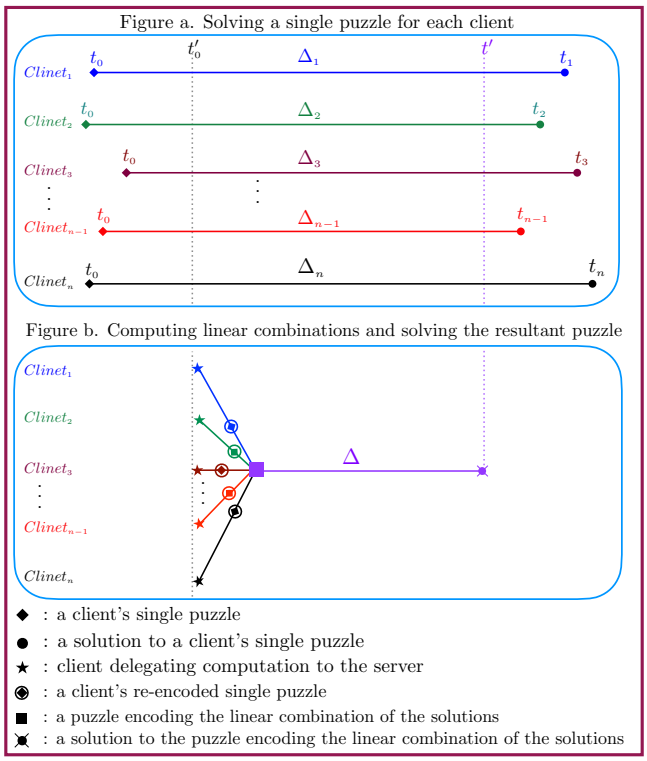


Fig. 1: Outline of the workflow of Tempora-Fusion. In the figure, t_0 refers to the time when a server receives a puzzle instance from a client, t'_0 is the time when clients delegate the homomorphic linear combination of their puzzles' solutions to the server, t_i is the time when the solution to $Cleint_i$'s puzzle is found, t' is the time when the solution to a puzzle encoding the linear combination is found, Δ_i is the period after which the solution to $Client_i$'s puzzle is found, and Δ is the period after which a solution to the puzzle encoding the linear combination is discovered.

Intermediate-Level Description. Next, we will delve deeper into the description of Tempora-Fusion, elucidating its key mechanisms and components across various phases.

1. **Setup**. Initially, server s generates a set of public parameters, without requiring it to generate any secret keys. The public parameters include a large prime number p and a set $X = \{x_1, \ldots, x_{\bar{t}}\}$ of distinct and non-zero elements. The elements in X can be considered as x-coordinates and will help each client to represent its message as a polynomial in point-value form, consistent with other clients' polynomials.

2. **Key Generation**. Each client independently generates a secret key and public key N_u. It publishes its public key.

3. **Puzzle Generation**. Using its secret key and time parameter T_u that determines how long a solution must remain concealed, each client c_u generates a master key mk_u and a set of public parameters pp_u. Given pp_u, anyone who will solve this client's puzzle will be able to find mk_u in the future, after a certain time. The client uses mk_u to derive pseudorandom values $(z_{i,u}, w_{i,u})$ for each element x_i of X.

   The client represents its secret solution m_u as a polynomial in point-value form. This results in a vector of y-coordinates: $[\pi_{1,u}, \ldots, \pi_{\bar{t},u}]$. It encrypts each y-coordinate using the related pseudorandom values: $o_{i,u} = w_{i,u} \cdot (\pi_{i,u} + z_{i,u}) \bmod p$. These encrypted y-coordinates $\vec{o}_u = [o_{1,u}, \ldots, o_{\bar{t},u}]$ represent its puzzle.

   The reason that the client represents its solution m_u as a polynomial, is to facilitate future homomorphic computation and efficient verification of the computation (as explained in Section 5.1).

To enable anyone to verify the correctness of the solution that s will find, the client commits $com_u = \texttt{Com}(m_u, mk_u)$ to the solution m_u using mk_u as the randomness used in the commitment. The client publishes (or sends to s) com_u and the puzzle $\vec{o}_u$. If s solves the client's puzzle, it can find both m_u and mk_u, and prove they match the commitment.

4. **Linear Combination**. Prior to this phase, each client c_u has already created and transmitted a puzzle for its solution m_u to s. Within this phase, the clients produce specific messages that will enable s to find, after time Δ, a linear combination of the clients' plaintext solutions: $\sum_{\forall c_u \in C} q_u \cdot m_u$, where each q_u is a coefficient picked by each client c_u.

   The clients initially collaborate to designate a small subset $\mathcal{I}$ of themselves as leaders. These leaders are picked at random. Each leader client c_u, using its secret key and time parameter Y (which determines how long the result of the computation must remain private), generates a *temporary* master key tk_u along with some public parameters $pp_u^{(\text{Evl})}$. Anyone who solves a puzzle for the computation will be able to find tk_u, after time Δ. Each leader client uses tk_u to derive new pseudorandom values $(z'_{i,u}, w'_{i,u})$ for each element x_i of X. It also uses its secret key to *regenerate* the pseudorandom values $(z_{i,u}, w_{i,u})$ used to encrypt each y-coordinate related to its solution m_u.

   The leader client picks a random root: $root_u$. It commits to this root, using tk_u as the randomness: $com'_u = \texttt{Com}(root_u, tk_u)$. This approach will ensure that s (i) cannot come up with its own root, and (ii) will find the commitment's opening itself if it solves the leader's puzzle.

   Each leader client also represents $root_u$ as a polynomial in point-value form. This yields a vector of y-coordinates: $[\gamma_{1,u}, \ldots, \gamma_{\bar{t},u}]$. It encrypts each y-coordinate using the related new pseudorandom values as: $\gamma'_{i,u} = \gamma_{i,u} \cdot w'_{i,u} \bmod p$ and sends the encrypted y-coordinates to the rest of the clients. This (encrypted) random root of each leader will be inserted by every client into its outsourced puzzle to give a certain structure to the computation result, facilitating future verification.

   For every client, for instance c_l, the leader client selects a fresh key f_l and sends it to that client. This key is used by each c_l and the leader client to generate zero-sum pseudorandom values. These values are generated such that if those generated by each c_l and the leader client are summed, they will cancel out each other. They are used to ensure that s can only learn the linear combination of the clients' messages.

   Each leader client participates in an instance of $\texttt{OLE}^+$ with s, for each y-coordinate. Broadly speaking, each leader client's input includes the y-coordinate of the random root, the new pseudorandom values $(z'_{i,u}, w'_{i,u})$, the inverse of the old pseudorandom values (so ultimately the old ones can be replaced with the new ones), its coefficient q_u, and the pseudorandom values derived from f_l, and $\prod_{\forall c_l \in \mathcal{I} \setminus c_u} \gamma'_{i,l} \bmod p$. The input of s is the client's puzzle. Each instance of $\texttt{OLE}^+$ returns to s an encrypted y-coordinate. Each leader client publishes its public parameters $pp_u^{(\text{Evl})}$.

   Each non-leader client also participates in an instance of $\texttt{OLE}^+$ with s, for each y-coordinate. Each non-leader client's input is similar to a leader client's input, with the main difference being that (i) it does not include $z'_{i,u}$ and (ii) instead of inserting the y-coordinate of a random root and $w'_{i,u}$, it inserts $\prod_{\forall c_l \in \mathcal{I}} \gamma'_{i,l} \bmod p$, where each $\gamma'_{i,l} = \gamma_{i,u} \cdot w'_{i,u} \bmod p$ has been sent to it by a leader client. In this case, each instance of $\texttt{OLE}^+$ also returns to s an encrypted y-coordinate.

   Server s sums the outputs of $\texttt{OLE}^+$ component-wise, which yields a vector of encrypted y-coordinates, $\vec{g} = [g_1, \ldots, g_{\bar{t}}]$. It publishes $\vec{g}$. Note that each g_i has $|\mathcal{I}|$ layers of blinding factors, each of which is inserted by a leader client. This multi-layer encryption ensures that even if some of the leader clients' secret keys are disclosed to server s, the server cannot find the computation result significantly earlier than the predefined time, Δ.

5. **Solving a Puzzle**. Server s operates as follows when it wants to find the result of the computation. Given public parameters $pp_u^{(\text{Evl})}$ of each leader client, it solves each leader client's puzzle to find temporary key tk_u which allows it to remove a layer of encryption from each g_i.

   By removing all the layers of encryption, s will obtain a set of y-coordinates. It uses these y-coordinates and the x-coordinates in X to interpolate a polynomial $\boldsymbol{\theta}$. It retrieves the roots of $\boldsymbol{\theta}$. It publishes each root and tk_u that match a published commitment com'_u. It also retrieves the computation result (i.e., the linear combination of the clients' solutions) from $\boldsymbol{\theta}$ and publishes it.

   Server s takes the following steps when it wants to find a solution for a single client's puzzle (note that solving a single client's puzzle is independent of computing homomorphic linear computation of all clients' puzzles). Given public parameters pp_u and puzzle vector $[o_{1,u}, \ldots, o_{\bar{t},u}]$ (generated in Phase 3) for a client c_u, server s after time Δ_u finds the master key mk_u. Using mk_u, s removes the blinding factors from each $o_{i,u}$, that yields a vector of y-coordinates. It uses them and x-coordinates in X to interpolate a polynomial $\boldsymbol{\pi}_u$ and retrieves message m_u from $\boldsymbol{\pi}_u$. It publishes m_u and mk_u that match the published commitment com_u, generated in Phase 3.

6. **Verification**. When verifying a solution related to the linear combination, a verifier (i) checks whether every opening (root and tk_u) matches the published commitment com'_u, and (ii) unblinds the elements of $\vec{g}$ using every tk_u, interpolates a polynomial $\boldsymbol{\theta}(x)$, and checks if $\boldsymbol{\theta}(x)$ possesses all the roots. When verifying a solution related to a single client's puzzle, a verifier checks if the opening (m_u, mk_u) matches the commitment com_u.

Detailed Construction. We proceed to provide a detailed description of the Tempora-Fusion protocol.

1. **Setup**. $\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow (., pk_s)$

   The server s (or any party) only once takes the following steps:

   (a) *Setting a field's parameter*: generates a sufficiently large prime number p, where $\log_2(p)$ is security parameter, e.g., $\log_2(p) \geq 128$.

   (b) *Generating public x-coordinates*: let $\ddot{t}$ be the total number of leader clients. It sets $\bar{t} = \ddot{t} + 2$ and $X = \{x_1, \ldots, x_{\bar{t}}\}$, where $x_i \neq x_j$, $x_i \neq 0$, and $x_i \notin U$.

   (c) *Publishing public parameters*: publishes $pk_s = (p, X, t)$.

2. **Key Generation**. $\mathsf{C.Setup}(1^\lambda) \rightarrow K_u$

   Each party c_u in $C = \{c_1, \ldots, c_n\}$ takes the following steps:

   (a) *Generating RSA public and private keys*: computes $N_u = p_1 \cdot p_2$, where p_i is a large randomly chosen prime number, where $\log_2(p_i)$ is a security parameter, e.g., $\log_2(p_i) \geq 2048$. Next, it computes Euler's totient function of N_u, as: $\phi(N_u) = (p_1 - 1) \cdot (p_2 - 1)$.

   (b) *Publishing public parameters*: locally keeps secret key $sk_u = \phi(N_u)$ and publishes public key $pk_u = N_u$.

3. **Puzzle Generation**. $\mathsf{GenPuzzle}(m_u, K_u, pk_s, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$

   Each c_u independently takes the following steps to generate a puzzle for a message m_u.

   (a) *Checking public parameters*: checks the bit-size of p and elements of X in pk_s, to ensure $\log_2(p) \geq 128$, $x_i \neq x_j, x_i \neq 0$, and $x_i \notin U$. If it does not accept the parameters, it returns $(\perp, \perp)$ and does not take further action.

   (b) *Generating secret keys*: generates a master key mk_u and two secret keys k_u and s_u as follows:

i. sets exponent a_u:

$$a_u = 2^{T_u} \bmod \phi(N_u)$$

where $T_u = \Delta_u \cdot max_{ss}$ and $\phi(N_u) \in K_u$.

ii. selects a base uniformly at random: $r_u \xleftarrow{\$} \mathbb{Z}_{N_u}$ and then sets a master key mk_u as follows:

$$mk_u = r_u^{a_u} \bmod N_u$$

iii. derive two keys from mk_u:

$$k_u = \mathtt{PRF}(1, mk_u), \quad s_u = \mathtt{PRF}(2, mk_u)$$

(c) *Generating blinding factors*: generates $2 \cdot \bar{t}$ pseudorandom blinding factors using k_u and s_u:

$$\forall i, 1 \le i \le \bar{t}: \quad z_{i,u} = \mathtt{PRF}(i, k_u), \quad w_{i,u} = \mathtt{PRF}(i, s_u)$$

(d) *Encoding plaintext message*:

i. represents plaintext message m_u as a polynomial, such that the polynomial's constant term is the message. Specifically, it computes polynomial $\boldsymbol{\pi}_u(x)$ as:

$$\boldsymbol{\pi}_u(x) = x + m_u \bmod p$$

ii. computes $\bar{t}$ y-coordinates of $\boldsymbol{\pi}_u(x)$:

$$\forall i, 1 \le i \le \bar{t}: \quad \pi_{i,u} = \boldsymbol{\pi}_u(x_i) \bmod p$$

where $x_i \in X$ and $p \in pk_s$.

(e) *Encrypting the message*: encrypts the y-coordinates using the blinding factors as follows:

$$\forall i, 1 \le i \le \bar{t}: \quad o_{i,u} = w_{i,u} \cdot (\pi_{i,u} + z_{i,u}) \bmod p$$

(f) *Committing to the message*: commits to the plaintext message:

$$com_u = \mathtt{Com}(m_u, mk_u)$$

(g) *Managing messages*: publishes $\vec{o}_u = [o_{1,u}, \ldots, o_{\bar{t},u}]$ and $pp_u = (com_u, T_u, r_u, N_u)$. It locally keeps secret parameters $sp_u = (k_u, s_u)$ and deletes everything else, including $m_u, \boldsymbol{\pi}_u(x), \pi_{1,u}, \ldots, \pi_{\bar{t},u}$. It sets $prm_u = (sp_u, pp_u)$.

4. **Linear Combination**. $\mathsf{Evaluate}(\langle s(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), c_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\mathrm{Evl})})$

In this phase, the parties produce certain messages that allow s to find a linear combination of the clients' plaintext message after time Δ.

(a) *Randomly selecting leaders*: all parties in C agree on a random key $\hat{r}$, e.g., by participating in a coin tossing protocol [10]. Each c_u deterministically finds index of $\ddot{t}$ leader clients: $\forall j, 1 \le j \le \ddot{t} : idx_j = \mathtt{G}(j||\hat{r})$. Let $\mathcal{I}$ be a vector contain these $\ddot{t}$ clients.

(b) *Granting the computation by each leader client*: each leader client c_u in $\mathcal{I}$ takes the following steps.

i. *Generating temporary secret keys*: generates a temporary master key tk_u and two secret keys k'_u and s'_u for itself. Also, it generates a secret key f_l for each client. To do that, it takes the following steps. It computes the exponent:

$$b_u = 2^Y \bmod \phi(N_u)$$

where $Y = \Delta \cdot max_{ss}$. It selects a base uniformly at random: $h_u \xleftarrow{\$} \mathbb{Z}_{N_u}$ and then sets a temporary master key tk_u:

$$tk_u = h_u^{b_u} \bmod N_u$$

It derives two keys from tk_u:

$$k'_u = \texttt{PRF}(1, tk_u), \quad s'_u = \texttt{PRF}(2, tk_u)$$

It picks a random key f_l for each client c_l excluding itself, i.e., $f_l \xleftarrow{\$} \{0,1\}^{poly(\lambda)}$, where $c_l \in C \setminus c_u$. It sends f_l to each c_l.

ii. *Generating temporary blinding factors*: derives $\bar{t}$ pseudorandom values from s'_u:

$$\forall i, 1 \leq i \leq \bar{t}: \quad w'_{i,u} = \texttt{PRF}(i, s'_u)$$

iii. *Generating an encrypted random root*: picks a random root: $root_u \xleftarrow{\$} \mathbb{F}_p$. It represents $root_u$ as a polynomial, such that the polynomial's root is $root_u$. Specifically, it computes polynomial $\boldsymbol{\gamma}_u(x)$ as:

$$\boldsymbol{\gamma}_u(x) = x - root_u \bmod p$$

Then, it computes $\bar{t}$ y-coordinates of $\boldsymbol{\gamma}_u(x)$:

$$\forall i, 1 \leq i \leq \bar{t}: \quad \gamma_{i,u} = \boldsymbol{\gamma}_u(x_i) \bmod p$$

It encrypts each y-coordinate $\gamma_{i,u}$ using blinding factor $w'_{i,u}$:

$$\forall i, 1 \leq i \leq \bar{t}: \quad \gamma'_{i,u} = \gamma_{i,u} \cdot w'_{i,u} \bmod p$$

It sends $\vec{\gamma}'_u = [\gamma'_{1,u}, \ldots, \gamma'_{\bar{t},u}]$ to the rest of the clients.

iv. *Generating blinding factors*: receives $(\bar{f}_l, \vec{\gamma}'_l)$ from every other client which are in $\mathcal{I}$.

It regenerates its original blinding factors:

$$\forall i, 1 \leq i \leq \bar{t}: \quad z_{i,u} = \texttt{PRF}(i, k_u), \quad w_{i,u} = \texttt{PRF}(i, s_u)$$

where k_u and s_u are in $\vec{prm}_u$ and were generated in step 3(b)iii. It also generates new ones:

$$\forall i, 1 \leq i \leq \bar{t}: \quad z'_{i,u} = \texttt{PRF}(i, k'_u)$$

It sets values $v_{i,u}$ and $y_{i,u}$ as follows. $\forall i, 1 \leq i \leq \bar{t}$:

$$v_{i,u} = \gamma'_{i,u} \cdot \prod_{\forall c_l \in \mathcal{I} \setminus c_u} \gamma'_{i,l} \bmod p$$

$$y_{i,u} = -\sum_{\forall c_l \in C \setminus c_u} \texttt{PRF}(i, f_l) + \sum_{\forall c_l \in \mathcal{I} \setminus c_u} \texttt{PRF}(i, \bar{f}_l) \bmod p$$

where $c_u \in \mathcal{I}$.

v. *Re-encoding outsourced puzzle*: obliviously, without having to access a plaintext solution, prepares the puzzle (held by s) for the computation. To do that, it participates in an instance of $\texttt{OLE}^+$ with s, for every i, where $1 \leq i \leq \bar{t}$. The inputs of c_u to i-th instance of $\texttt{OLE}^+$ are:

$$e_i = q_u \cdot v_{i,u} \cdot (w_{i,u})^{-1} \bmod p$$
$$e'_i = -(q_u \cdot v_{i,u} \cdot z_{i,u}) + z'_{i,u} + y_{i,u} \bmod p$$

The input of s to i-th instance of $\texttt{OLE}^+$ is c_u's encrypted y-coordinate: $e''_i = o_{i,u}$ (where $o_{i,u} \in \vec{o}$). Accordingly, i-th instance of $\texttt{OLE}^+$ returns to s:

$$\begin{aligned} d_{i,u} &= e_i \cdot e''_i + e'_i \\ &= q_u \cdot v_{i,u} \cdot \pi_{i,u} + z'_{i,u} + y_{i,u} \bmod p \\ &= q_u \cdot \gamma_{i,u} \cdot w'_{i,u} \cdot (\prod_{\forall c_l \in \mathcal{I} \setminus c_u} \gamma_{i,l} \cdot w'_{i,l}) \cdot \pi_{i,u} + z'_{i,u} + y_{i,u} \bmod p \end{aligned}$$

where q_u is the party's coefficient. If c_u detects misbehavior during the execution of $\texttt{OLE}^+$, it sends a special symbol $\perp$ to all parties and halts.

vi. *Committing to the root*: computes $com'_u = \texttt{Com}(root_u, tk_u)$.

vii. *Publishing public parameters*: publishes $pp_u^{(\text{Evl})} = (h_u, com'_u, N_u, Y)$. Note that all $c_u \in \mathcal{I}$ use identical Y. Let $\vec{pp}^{(\text{Evl})}$ contain all the triples $pp_u^{(\text{Evl})}$ published by c_u, where $c_u \in \mathcal{I}$.

(c) *Granting the computation by each non-leader client*: each non-leader client c_u takes the following steps.

i. *Generating blinding factors*: receives $(\bar{f}_l, \vec{\gamma}'_l)$ from every other client which is in $\mathcal{I}$.

It regenerates its original blinding factors:

$$\forall i, 1 \leq i \leq \bar{t}: \quad z_{i,u} = \texttt{PRF}(i, k_u), \quad w_{i,u} = \texttt{PRF}(i, s_u)$$

It set values $v_{i,u}$ and $y_{i,u}$ as follows. $\forall i, 1 \leq i \leq \bar{t}$:

$$v_{i,u} = \prod_{\forall c_l \in \mathcal{I}} \gamma'_{i,l} \bmod p$$
$$y_{i,u} = \sum_{\forall c_l \in \mathcal{I}} \texttt{PRF}(i, \bar{f}_l) \bmod p$$

ii. *Re-encoding outsourced puzzle*: participates in an instance of $\texttt{OLE}^+$ with the server s, for every i, where $1 \leq i \leq \bar{t}$. The inputs of c_u to i-th instance of $\texttt{OLE}^+$ are:

$$e_i = q_u \cdot v_{i,u} \cdot (w_{i,u})^{-1} \bmod p$$
$$e'_i = -(q_u \cdot v_{i,u} \cdot z_{i,u}) + y_{i,u} \bmod p$$

The input of s to i-th instance of $\texttt{OLE}^+$ is c_u's encrypted y-coordinate: $e''_i = o_{i,u}$. Accordingly, i-th instance of $\texttt{OLE}^+$ returns to s:

$$\begin{aligned} d_{i,u} &= e_i \cdot e''_i + e'_i \\ &= q_u \cdot v_{i,u} \cdot \pi_{i,u} + y_{i,u} \bmod p \\ &= q_u \cdot (\prod_{\forall c_l \in \mathcal{I} \setminus c_u} \gamma_{i,l} \cdot w'_{i,l}) \cdot \pi_{i,u} + y_{i,u} \bmod p \end{aligned}$$

where q_u is the party's coefficient. If c_u detects misbehavior during the execution of $\texttt{OLE}^+$, it sends a special symbol $\perp$ to all parties and halts.

(d) *Computing encrypted linear combination*: server s sums all of the outputs of $\texttt{OLE}^+$ instances that it has invoked, $\forall i, 1 \leq i \leq \bar{t}$:

$$\begin{aligned} g_i &= \sum_{\forall c_u \in C} d_{i,u} \bmod p \\ &= (\prod_{\forall c_u \in \mathcal{I}} \underbrace{\gamma_{i,u} \cdot w'_{i,u}}_{v_{i,u}} \cdot \sum_{\forall c_u \in C} q_u \cdot \pi_{i,u}) + \sum_{\forall c_u \in \mathcal{I}} z'_{i,u} \bmod p \end{aligned}$$

(e) *Disseminating encrypted result*: server s publishes $\vec{g} = [g_1, \ldots, g_{\bar{t}}]$.

5. **Solving a Puzzle**. $\mathsf{Solve}(\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_s, cmd) \rightarrow (m, \zeta)$

Server s takes the following steps.

Case 1. when solving a puzzle related to the linear combination, i.e., when $cmd =$ evalPzl:

(a) *Finding secret keys*: for each $c_u \in \mathcal{I}$:

i. finds tk_u (where $tk_u = h_u^{2^Y} \bmod N_u$) through repeated squaring of h_u modulo N_u, such that $(h_u, Y, N_u) \in \vec{pp}^{(\text{Evl})}$.

ii. derives two keys from tk_u:

$$k'_u = \texttt{PRF}(1, tk_u), \quad s'_u = \texttt{PRF}(2, tk_u)$$

(b) *Removing blinding factors*: removes the blinding factors from $[g_1, \ldots, g_{\bar{t}}] \in \vec{g}$.

$\forall i, 1 \leq i \leq \bar{t}$:

$$\begin{aligned} \theta_i &= \big(\prod_{\forall c_u \in \mathcal{I}} \underbrace{\texttt{PRF}(i, s'_u)}_{w'_{i,u}}\big)^{-1} \cdot \big(g_i - \sum_{\forall c_u \in \mathcal{I}} \overbrace{\texttt{PRF}(i, k'_u)}^{z'_{i,u}}\big) \bmod p \\ &= (\prod_{\forall c_u \in \mathcal{I}} \gamma_{i,u}) \cdot \sum_{\forall c_u \in C} q_u \cdot \pi_{i,u} \bmod p \end{aligned}$$

(c) *Extracting a polynomial*: interpolates a polynomial $\boldsymbol{\theta}$, given pairs $(x_1, \theta_1), \ldots, (x_{\bar{t}}, \theta_{\bar{t}})$. Note that $\boldsymbol{\theta}$ will have the following form:

$$\boldsymbol{\theta}(x) = \prod_{\forall c_u \in \mathcal{I}} (x - root_u) \cdot \sum_{\forall c_u \in C} q_u \cdot (x + m_u) \bmod p$$

We can rewrite $\boldsymbol{\theta}(x)$ as follows:

$$\boldsymbol{\theta}(x) = \boldsymbol{\psi}(x) + \prod_{\forall c_u \in \mathcal{I}} (-root_u) \cdot \sum_{\forall c_u \in C} q_u \cdot m_u \bmod p$$

where $\boldsymbol{\psi}(x)$ is a polynomial of degree $\ddot{t} + 1$ whose constant term is 0.

(d) *Extracting the linear combination*: retrieves the final result (which is the linear combination of the messages $m_1, \ldots, m_n$) from polynomial $\boldsymbol{\theta}(x)$'s constant term: $cons = \prod_{\forall c_u \in \mathcal{I}} (-root_u) \cdot \sum_{\forall c_u \in C} q_u \cdot m_u$ as follows:

$$\begin{aligned} res &= cons \cdot (\prod_{\forall c_u \in \mathcal{I}} (-root_u))^{-1} \bmod p \\ &= \sum_{\forall c_u \in C} q_u \cdot m_u \end{aligned}$$

(e) *Extracting valid roots*: extracts the roots of $\boldsymbol{\theta}$. Let set R contain the extracted roots. It identifies the valid roots, by finding every $root_u$ in R, such that $\texttt{Ver}(com'_u, (root_u, tk_u)) = 1$. Note that s performs the check for every c_u in $\mathcal{I}$.

(f) *Publishing the result*: publishes the solution $m = res$ and the proof $\zeta = \big\{(root_u, tk_u)\big\}_{\forall c_u \in \mathcal{I}}$.

Case 2. when solving a puzzle of single client c_u, i.e., when $cmd =$ clientPzl:

(a) *Finding secret keys*: finds mk_u where $mk_u = r_u^{2^{T_u}} \bmod N_u$ through repeated squaring of r_u modulo N_u, where $(T_u, r_u) \in pp_u$. Then, it derives two keys from mk_u:

$$k_u = \texttt{PRF}(1, mk_u), \quad s_u = \texttt{PRF}(2, mk_u)$$

(b) *Removing blinding factors*: re-generates $2 \cdot \bar{t}$ pseudorandom values using k_u and s_u:

$$\forall i, 1 \leq i \leq \bar{t}: \quad z_{i,u} = \texttt{PRF}(i, k_u), \quad w_{i,u} = \texttt{PRF}(i, s_u)$$

Then, it uses the blinding factors to unblind $[o_{1,u}, \ldots, o_{\bar{t},u}]$:

$$\forall i, 1 \leq i \leq \bar{t}: \quad \pi_{i,u} = \big((w_{i,u})^{-1} \cdot o_{i,u}\big) - z_{i,u} \bmod p$$

(c) *Extracting a polynomial*: interpolates a polynomial $\boldsymbol{\pi}_u$, given pairs $(x_1, \pi_{1,u}), \ldots, (x_{\bar{t}}, \pi_{\bar{t},u})$.

(d) *Publishing the solution*: considers the constant term of $\boldsymbol{\pi}_u$ as the plaintext solution, m_u. It publishes the solution $m = m_u$ and the proof $\zeta = mk_u$.

6. **Verification**. $\mathsf{Verify}(m, \zeta, ., pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_s, cmd) \rightarrow \ddot{v} \in \{0, 1\}$

A verifier (that can be anyone, not just $c_u \in C$) takes the following steps.

Case 1. when verifying a solution related to the linear combination, i.e., when $cmd =$ evalPzl:

(a) *Checking the commitments' openings*: verifies the validity of every $(root_u, tk_u) \in \zeta$, provided by s in Case 1, step 5f:

$$\forall c_u \in \mathcal{I}: \quad \texttt{Ver}\big(com'_u, (root_u, tk_u)\big) \stackrel{?}{=} 1$$

where $com'_u \in \vec{pp}^{(\text{Evl})}$. If all of the verifications pass, it proceeds to the next step. Otherwise, it returns $\ddot{v} = 0$ and takes no further action.

(b) *Checking the resulting polynomial's valid roots*: checks if the resulting polynomial contains all the roots in ζ, by taking the following steps.

i. derives two keys from tk_u:

$$k'_u = \texttt{PRF}(1, tk_u), \quad s'_u = \texttt{PRF}(2, tk_u)$$

ii. removes the blinding factors from $[g_1, \ldots, g_{\bar{t}}] \in \vec{g}$ that were provided by s in step 4e.

$\forall i, 1 \leq i \leq \bar{t}:$

$$\begin{aligned} \theta_i &= \Big(\prod_{\forall c_u \in \mathcal{I}} \texttt{PRF}(i, s'_u)\Big)^{-1} \cdot \Big(g_i - \sum_{\forall c_u \in \mathcal{I}} \texttt{PRF}(i, k'_u)\Big) \bmod p \\ &= \prod_{\forall c_u \in \mathcal{I}} \gamma_{i,u} \cdot \sum_{\forall c_u \in C} q_u \cdot \pi_{i,u} \bmod p \end{aligned}$$

iii. interpolates a polynomial $\boldsymbol{\theta}$, given pairs $(x_1, \theta_1), \ldots, (x_{\bar{t}}, \theta_{\bar{t}})$, similar to step 5c. This yields a polynomial $\boldsymbol{\theta}$ having the form:

$$\begin{aligned}\boldsymbol{\theta}(x) &= \prod_{\forall c_u \in \mathcal{I}} (x - root_u) \cdot \sum_{\forall c_u \in C} q_u \cdot (x + m_u) \bmod p \\ &= \boldsymbol{\psi}(x) + \prod_{\forall c_u \in \mathcal{I}} (-root_u) \cdot \sum_{\forall c_u \in C} q_u \cdot m_u \bmod p\end{aligned}$$

where $\boldsymbol{\psi}(x)$ is a polynomial of degree $\ddot{t}+1$ whose constant term is 0.

iv. if the following checks pass, it will proceed to the next step. It checks if every $root_u$ is a root of $\boldsymbol{\theta}$, by evaluating $\boldsymbol{\theta}$ at $root_u$ and checking if the result is 0, i.e., $\boldsymbol{\theta}(root_u) \stackrel{?}{=} 0$. Otherwise, it returns $\ddot{v} = 0$ and takes no further action.

(c) *Checking the final result*: retrieves the final result (i.e., the linear combination of the messages $m_1, \ldots, m_n$) from polynomial $\boldsymbol{\theta}(x)$'s constant term: $cons = \prod_{\forall c_u \in \mathcal{I}} (-root_u) \cdot \sum_{\forall c_u \in C} q_u \cdot m_u$ as follows:

$$\begin{aligned}res' &= cons \cdot (\prod_{\forall c_u \in \mathcal{I}} (-root_u))^{-1} \bmod p \\ &= \sum_{\forall c_u \in C} q_u \cdot m_u\end{aligned}$$

It checks $res' \stackrel{?}{=} m$, where $m = res$ is the result that s sent to it.

(d) *Accepting or rejecting the result*: If all the checks pass, it accepts m and returns $\ddot{v} = 1$. Otherwise, it returns $\ddot{v} = 0$.

Case 2. when verifying a solution of single puzzle belonging to c_u, i.e., when cmd = clientPzl:

(a) *Checking the commitment' opening*: checks whether opening pair $m = m_u$ and $\zeta = mk_u$ matches the commitment:

$$\texttt{Ver}\big(com_u, (m_u, mk_u)\big) \stackrel{?}{=} 1$$

where $com_u \in pp_u$.

(b) *Accepting or rejecting the solution*: accepts the solution m and returns $\ddot{v} = 1$ if the above check passes. It rejects the solution and returns $\ddot{v} = 0$, otherwise.

Theorem 2. *If the sequential modular squaring assumption holds, factoring N is a hard problem,* PRF, $\texttt{OLE}^+$, *and the commitment schemes are secure, then the protocol presented above is a secure $\mathcal{VHLC}$-$\mathcal{TLP}$, w.r.t. Definition 9.*

Shortly, in Section 5.3, we present the proof of Theorem 2.

Remark 1. $\texttt{OLE}^+$ ensures that the homomorphic operation can be securely operated sequentially multiple times, regardless of the distribution of the input messages to $\texttt{OLE}^+$. Specifically, one may try to use the following naive approach. Each client, for each i-th y-coordinate $o_{i,u}$, directly sends the following values to the server: $e_i = q_u \cdot v_{i,u} \cdot (w_{i,u})^{-1} \bmod p$, $e'_i = -(q_u \cdot v_{i,u} \cdot z_{i,u}) + y_{i,u} \bmod p$. Each client asks the server to compute $e_i \cdot o_{i,u} + e'_i$. This will yield $q_u \cdot \gamma_{i,u} \cdot w'_{i,u} \cdot (\prod_{\forall c_l \in \mathcal{I} \setminus c_u} \gamma_{i,l} \cdot w'_{i,l}) \cdot \pi_{i,u} + z'_{i,u} + y_{i,u} \bmod p$, for a client which is in $\mathcal{I}$. However, this approach is not secure if the homomorphic linear combination must be computed multiple times. Because within this approach the security of each message e_i relies on the randomness of

$(w_{i,u})^{-1}$.[4] In scenarios where the homomorphic linear combination has to be computed multiple times, the same $(w_{i,u})^{-1}$ will be included in e_i, meaning that a one-time pad is used multiple times, yielding leakage.

Remark 2. In the above protocol, the number of elements in X is $\bar{t}$ for the following reason. Each client's outsourced polynomial (that represents its puzzle) is of degree 1. During Phase 4 (Linear Combination), this polynomial is multiplied by $\ddot{t}$ polynomials each representing a random root and is of degree 1. Thus, the resulting polynomial will have degree $\ddot{t}+1$. Hence, $\bar{t} = \ddot{t}+2$ (y,x)-coordinate pairs are sufficient to interpolate the polynomial.

Remark 3. One interesting aspect of Tempora-Fusion is its flexible approach to time-locking messages. Each encrypted message $\vec{o}_u$ from a client c_u, which is either published or transmitted to server s, need not be necessarily disclosed after a specified period. Despite this, it retains the capability to support verifiable homomorphic linear combinations. In essence, Tempora-Fusion offers clients the option to apply time-lock mechanisms to their solutions. Some clients may employ time locks on their encrypted messages, while others may opt for straightforward encryption of their solutions. Nevertheless, the clients can still allow s to learn the result of homomorphic linear combinations on their encrypted messages after a certain period. To encrypt a message without a time-lock, the client can employ the same encryption method utilized during the Puzzle Generation phase (Phase 3) with the sole distinction being the omission of the base r_u publication in step 3g of Phase 3.

5.3 Proof of Theorem 2

In this section, we prove the security of Tempora-Fusion, i.e., Theorem 2.

Proof. In the proof of Theorem 2, we consider a strong adversary that *always corrupts* s and some clients. Thus, the proof considers the case where corrupt s learns the secret inputs, secret parameters, and the messages that corrupt clients receive from honest clients. The messages that an adversary $\mathcal{A}$ receives are as follows.

- by the end of the puzzle generation phase, it learns:

$$Set_1 = \Big\{max_{ss}, \{N_u, \Delta_u, T_u, r_u, com_u, \vec{o}_u\}_{\forall u, 1\leq u\leq n}, \{K_j\}_{\forall \mathcal{B}_j \in \mathcal{W}}\Big\}$$

  where $\mathcal{W}$ is a set of corrupt parties, including server s.

- by the end of the linear combination phase (before any puzzle is fully solved), it also learns:

$$Set_2 = \Big\{trans_s^{\mathtt{OLE}_u^+}, Y, \{g_1, \ldots, g_{\bar{t}}\}, \{com'_u, h_u, d_{1,u}, \ldots, d_{\bar{t},u}\}_{\forall u, 1\leq u\leq n}, \{\bar{f}_l, \vec{\gamma}'_l\}_{\forall c_l \in \{\mathcal{W}\cap\mathcal{I}\}}\Big\}$$

  where $trans_s^{\mathtt{OLE}_u^+}$ is a set of messages sent to s during the execution of $\mathtt{OLE}^+$.

We initially prove that Tempora-Fusion is privacy-preserving, w.r.t. Definition 4.

Lemma 1. *If the sequential modular squaring assumption holds, factoring N is a hard problem,* PRF *is secure,* $\mathtt{OLE}^+$ *is secure (i.e., privacy-preserving), and the commitment scheme satisfies the hiding property, then Tempora-Fusion is privacy-preserving, w.r.t. Definition 4.*

[4] Note that, in this case, we cannot rely on the random value $w'_{i,u}$ or w'_{i,c_l} to guarantee the privacy of each message, as the message of every client contains the same $w'_{i,u}$ and w'_{i,c_l}.

Proof. We will argue that the probability that adversary $\mathcal{A}_2 \in \mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ outputs correct value of b_u (in the experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 4) is at most $\frac{1}{2} + \mu(\lambda)$. Since parameters $(max_{ss}, N_u, \Delta_u, T_u, Y, r_u, h_u)$ have been picked independently of the plaintext messages/solutions, they reveal nothing about the messages. Each y-coordinate $\pi_{i,u}$ in $\vec{o}_{i,u}$ has been masked (or encrypted) with a fresh output of PRF (where $o_{i,u} \in \vec{o}_u$). Due to the security of PRF, outputs of PRF are computationally indistinguishable from the outputs of a random function. As a result, a message blinded by an output PRF does not reveal anything about the message, except for the probability $\mu(\lambda)$. Also, each client c_u picks its secret keys and accordingly blinding factors independent of other clients. Therefore, knowing corrupt clients' secret keys $\{K_j\}_{\forall \mathcal{B}_j \in \mathcal{W}}$ does not significantly increase the adversary's probability of winning the game in the experiment, given honest parties' puzzles and corresponding parameters. Because of the hiding property of the commitment scheme, commitments com_u and com'_u reveal no information about the committed value.

Due to the security of $\texttt{OLE}^+$, a set $trans_s^{\texttt{OLE}^+_u}$ of messages that $\mathcal{A}$ (acting on behalf of corrupt parties) receives during the execution of $\texttt{OLE}^+$ are (computationally) indistinguishable from an ideal model where the parties send their messages to a trusted party and receive the result. This means that the exchanged messages during the execution of $\texttt{OLE}^+$ reveal nothing about the parties' inputs, that include the encoded plaintext solution (i.e., y-coordinate $\pi_{i,u}$) and PRF's output used to encrypt $\pi_{i,u}$.

Each $d_{i,u}$ is an output of $\texttt{OLE}^+$. Due to the security of $\texttt{OLE}^+$, it reveals to $\mathcal{A}$ nothing about the input of each honest client c_u to $\texttt{OLE}^+$, even if s inserts 0. Moreover, each $d_{i,u}$ has been encrypted with $y_{i,u}$ which is a sum of fresh outputs of PRF. Recall that each $y_{i,u}$ has one of the following the forms:

- $y_{i,u} = -\sum_{\forall c_l \in C \setminus c_u} \texttt{PRF}(i, f_l) + \sum_{\forall c_l \in \mathcal{I} \setminus c_u} \texttt{PRF}(i, \bar{f}_l) \bmod p$, when client c_u is one of the leaders, i.e., $c_u \in \mathcal{I}$.
- $y_{i,u} = \sum_{\forall c_l \in \mathcal{I}} \texttt{PRF}(i, \bar{f}_l) \bmod p$, when client c_u is not one of the leaders, i.e., $c_u \notin \mathcal{I}$.

Due to the security of PRF, given secret keys $\{\bar{f}_l\}_{\forall c_l \in \{\mathcal{C} \cap \mathcal{I}\}}$, it will be infeasible for $\mathcal{A}$ to learn anything about secret blinding factor used by each honest party, as long as the number of corrupt leaders is smaller than the threshold t, i.e., $|\mathcal{W} \cap \mathcal{I}| < t$. Therefore, given $\{\bar{f}_l\}_{\forall c_l \in \{\mathcal{C} \cap \mathcal{I}\}}$, $\mathcal{A}$ learns nothing about each honest client c_u y-coordinate $\pi_{i,u}$ (as well as $z'_{i,u}$ when $c_u \in \mathcal{I}$) in $d_{i,u}$, except with the negligible probability in λ, meaning that $d_{1,u}, \ldots, d_{\bar{t},u}$ are computationally indistinguishable from random values, for $u, 1 \leq u \leq n$.

Each puzzle g_i that encodes a y-coordinate for the linear combination, uses the sum of $z'_{i,u}$ (and $w'_{i,u}$) to encrypt the y-coordinate, where each honest client's $z'_{i,u}$ is a fresh output of PRF and unknown to $\mathcal{A}$. Given corrupt clients' secret keys $\{K_j\}_{\forall \mathcal{B}_j \in \mathcal{W}}$, $\mathcal{A}$ can remove the blinding factors $z'_{i,u}$ for the corrupt parties. However, due to the security of PRF and accordingly due to the indistinguishability of each $d_{i,u}$ from random values, it cannot remove $z'_{i,u}$ of honest parties from g_i (before attempting to solve the puzzle) expect with the negligible probability in λ.

Due to the security of PRF, given the encrypted y-coordinates of the roots $\{\vec{\gamma}'_l\}_{\forall c_l \in \{\mathcal{W} \cap \mathcal{I}\}}$ received by corrupt clients and the corrupt parties' secret keys $\{K_j\}_{\forall \mathcal{B}_j \in \mathcal{W}}$, $\mathcal{A}$ cannot learn the y-coordinates of the random root chosen by each honest client (accordingly it cannot learn the random root), except for a negligible probability in λ.

Thus, given Set_1 and Set_2, if the sequential modular squaring assumption holds and factoring problem is hard, $\mathcal{A}_2$ that runs in time $\delta(T_u) < T_u$ using at most $\bar{poly}(T_u)$ parallel processors, cannot find a solution m_u (from $\vec{o}_u$) significantly earlier than $\delta(\Delta_u)$, except with negligible probability $\mu(\lambda)$. This means that it cannot output the correct value of b_u (in line 20 of experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 4) with a probability significantly greater than $\frac{1}{2}$.

Recall that each $d_{i,u}$ is blinded with a blinding factor $y_{i,u}$. These blinding factors will be cancelled out, if all $d_{i,u}$ (of different clients) are summed up. Given the above discussion, knowing the elements of Set_1 and Set_2, if the sequential modular squaring assumption holds and the factoring problem is hard, $\mathcal{A}_2$ that runs in time $\delta(Y) < Y$ using at most $\bar{poly}(Y)$ parallel processors, cannot output the correct value of b_u (in the

line 27 of experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 4) with a probability significantly greater than $\frac{1}{2}$.

This means, $\mathcal{A}$ cannot find a solution $\sum_{\forall c_u \in C} q_u \cdot m_u$ (from $g_1, \ldots, g_{\bar{t}}$) significantly earlier than $\delta(\Delta)$, except with negligible probability $\mu(\lambda)$. Accordingly, $\mathcal{A}$ can only learn the linear *combination of all honest clients' messages*, after solving puzzles $g_1, \ldots, g_{\bar{t}}$. ■

We proceed to prove that Tempora-Fusion preserves a solution validity, w.r.t. Definition 5.

Lemma 2. *If the sequential modular squaring assumption holds, factoring N is a hard problem,* PRF *is secure,* $\texttt{OLE}^+$ *is secure (i.e., offers result validity and is privacy-preserving), and the commitment scheme meets the binding and hiding properties, then Tempora-Fusion preserves a solution validity, w.r.t. Definition 5.*

Proof. We will demonstrate the probability that a PPT adversary $\mathcal{A}$ outputs an invalid solution but passes the verification (in the experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{val}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 5) is negligible in the security parameter, i.e., $\mu(\lambda)$.

In addition to (Set_1, Set_2), the messages that an adversary $\mathcal{A}$ receives are as follows:

- by the end of the puzzle-solving phase for a puzzle related to the linear combination, it learns:

$$Set_3 = \Big\{ \{root_u, tk_u\}_{\forall c_u \in \mathcal{I}}, m = res \Big\}$$

- by the end of the puzzle-solving phase for a puzzle of a single honest client c_u, it also learns:

$$Set_4 = \Big\{ mk_u, m = m_u \Big\}$$

  where $\mathcal{A}$ learns Set_4 for any client (long) after it learns Set_3.

Due to the binding property of the commitment scheme, the probability that $\mathcal{A}$ can open every commitment related to a valid root in $\{root_u\}_{\forall c_u \in \mathcal{I}}$ to an invalid root (e.g., $root'$, where $root' \neq root_u$) and pass all the verifications, is $(\mu(\lambda))^{|\mathcal{I}|}$. Thus, this is detected in step 6a of the protocol with a high probability, i.e., $1 - (\mu(\lambda))^{|\mathcal{I}|}$.

As discussed in the proof of Lemma 1, before the puzzles of honest parties are solved, $\mathcal{A}$ learns nothing about the blinding factors of honest parties or their random roots (due to the hiding property of the commitment scheme, the privacy property of $\texttt{OLE}^+$, security of PRF, and under the assumptions that sequential modular squaring holds and factoring N is a hard problem).

Due to Theorem 1 (unforgeable encrypted polynomial with a hidden root), any modification by $\mathcal{A}$ to the inputs $\{o_{1,u}, \ldots, o_{\bar{t},u}\}_{\forall c_u \notin \mathcal{W}}$ and outputs $\{d_{1,u}, \ldots, d_{\bar{t},u}\}_{\forall c_u \notin \mathcal{W}}$ of $\texttt{OLE}^+$, makes the resulting polynomial $\boldsymbol{\theta}$ not contain every root in $\{root_u\}_{\forall c_u \notin \mathcal{W}}$. The same applies to the generation of $\vec{g} = [g_1, \ldots, g_{\bar{t}}]$. Specifically, if each g_i is not the sum of all honest parties $d_{i,u}$, then their blinding factors will not be cancelled out, making the resulting polynomial $\boldsymbol{\theta}$ not have every root in $\{root_u\}_{\forall c_u \notin \mathcal{W}}$, according to Theorem 1. Thus, this can be detected with a high probability (i.e., at least $1 - (\mu(\lambda))^t$) at step 6(b)iv of the protocol.

$\mathcal{A}$ will eventually learn the elements of Set_3 for honest parties. However, this knowledge will not help it cheat without being detected, as $\mathcal{A}$ has already published the output of the evaluation, e.g., $\vec{g} = [g_1, \ldots, g_{\bar{t}}]$.

Due to the security of $\texttt{OLE}^+$ (specifically result validity), any misbehavior of $\mathcal{A}$ (corrupting s) during the execution of $\texttt{OLE}^+$ will not be detected only with a negligible probability $\mu(\lambda)$, in steps 4(b)v and 4(c)ii of the protocol.

Hence, $\mathcal{A}$ cannot persuade $\mathcal{E}$ to return 1 on an invalid output of Evaluate() (in line 23 of the experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathrm{val}}(1^{\lambda}\, n, \ddot{t}, t)$ defined in Definition 5) with a probability significantly greater than $\mu(\lambda)$.

By solving a single client's puzzle (after all invocations of Evaluate()), $\mathcal{A}$ will also learn Set_4 for each (honest) client c_u. Due to the binding property of the commitment scheme, the probability that $\mathcal{A}$ can open every commitment corresponding to the single message m_u of each client c_u in $\{c_1, \ldots, c_n\}$ to an invalid message (e.g., m', where $m' \neq m_u$) and pass the verification in steps 6a and 6b of the protocol is negligible, $\mu(\lambda)$. Note that the elements of set $\{root_u, tk_u\}_{\forall c_u \in \mathcal{I}} \in Set_3$ have been selected uniformly at random and independent of each solution m_u. Thus, knowing elements $\{root_u, tk_u\}_{\forall c_u \in \mathcal{I}}$ will not increase the probability of the adversary to persuade a verifier to accept an invalid message m', in steps 6a and 6b of the protocol.

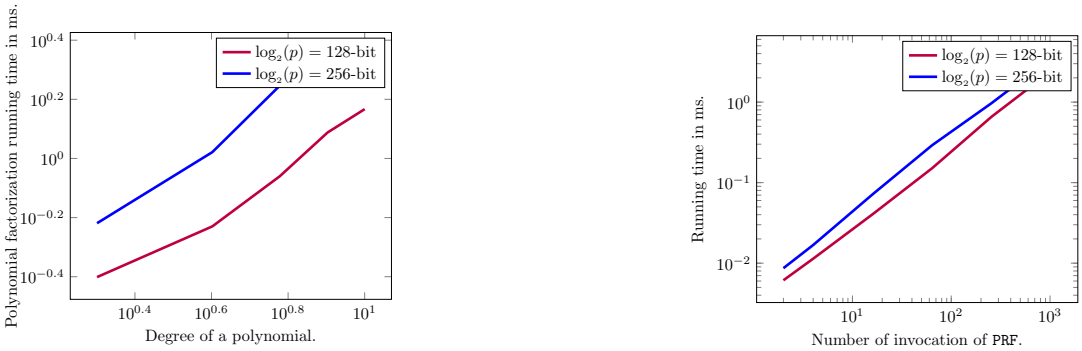
Thus, $\mathcal{A}$ cannot win and persuade $\mathcal{E}$ to return 1 on an invalid solution (in line 27 of the experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathrm{val}}(1^{\lambda}, n, \ddot{t}, t)$) with a probability significantly greater than $\mu(\lambda)$. ■

We have demonstrated that Tempora-Fusion is privacy-preserving (w.r.t. Definition 4) and preserves solution validity (w.r.t. Definition 5). Hence, Tempora-Fusion is secure, according to Definition 9.

This concludes the proof of Theorem 2. □

6 Cost Analysis

In this section, we begin by examining the asymptotic costs of our scheme before analyzing its concrete costs. In addition to the standard time-lock puzzle execution, the primary concrete costs in our scheme arise from invoking PRF, polynomial factorization, and $\mathtt{OLE}^{+}$ execution. We will demonstrate that the computational overheads associated with PRF and polynomial factorization of varying degrees are minimal in our scheme. We will also assert that $\mathtt{OLE}^{+}$ running time is low and has been used (and implemented) in various schemes. Figure 2 and Tables 2 and 3 provide detailed information on the actual running times for polynomials factorization and PRF invocations. In contrast, Table 1 summarizes the asymptotic costs of our scheme.



(a) Running time of factorizing polynomials.

(b) Running Time of PRF Invocations.

Fig. 2: Performance of polynomial factorizations and PRF. Figure 2a, depicts the performance of polynomial factorizations across polynomial degrees ranging from 2 to 10 over fields of 128 and 256 bits, i.e., $\log_2(p) = 128$ and $\log_2(p) = 256$. Figure 2b, showcases the performance of PRF across 2 to 1024 invocations, with output sizes of 128 and 256 bits.

6.1 Asymptotic Cost

Client's Computation Cost. In the Puzzle Generation phase (Phase 3), in each step 3(b)i and 3(b)ii, a client c_u performs a modular exponentiation over $\phi(N_u)$ and N_u respectively. In steps 3(b)iii and 3c, in total the

Table 1: Complexities of Tempora-Fusion. In the figure, n is the total number of clients, $\ddot{t}$ is the number of leaders, $\bar{t} = \ddot{t} + 2$, Δ is the period between granting the computation and when a linear combination of solutions is learned by server s, max_{ss} is the maximum number of squarings that a solver can perform per second, and $Y = max_{ss} \cdot \Delta$.

Schemes	Parties	Computation Cost	Communication Cost
Tempora-Fusion	Client	$O(\bar{t})$	$O(\bar{t} \cdot n)$
	Verifier	$O(\ddot{t}^2 + \ddot{t})$	—
	Server	$O(\ddot{t}^2 + \bar{t} \cdot n + \ddot{t} \cdot Y)$	$O(\bar{t} \cdot n)$

client invokes $2\bar{t} + 2$ instances of PRF. In step 3(d)i, it performs a single modular addition. In step 3(d)ii, it evaluates the polynomial at $\bar{t}$ x-coordinates, which will involve $\bar{t}$ modular additions, using Horner's method [23]. In step 3e, the client also performs $\bar{t}$ additions and $\bar{t}$ multiplications to encrypt the y-coordinates. In step 3f, the client invokes the hash function once to commit to its message.

In the Linear Combination Phase (Phase 4), we will focus on the cost of a leader client, as its overall cost is higher than a non-leader one. In step 4a, a client invokes a hash function $\ddot{t}$ times. In step 4(b)i, it performs two modular exponentiations, one over $\phi(N_u)$ and the other over N_u. In the same step, it invokes PRF twice to generate two temporary keys. In step 4(b)ii, it invokes $\bar{t}$ instances of PRF. In step 4(b)iii, it performs $\bar{t}$ additions and $\bar{t}$ multiplications. In step 4(b)iv, it invokes $3 \cdot \bar{t}$ instances of PRF and performs $\bar{t} + 1$ multiplications.

In step 4(b)v, the client performs $2 \cdot \bar{t}$ additions $4 \cdot \bar{t}$ multiplications. In the same step, it invokes $\bar{t}$ instances of $\texttt{OLE}^+$. In step 4(b)vi, it invokes the hash function once to commit to the random root. Thus, the client's complexity is $O(\bar{t})$.

Verifier's Computation Cost. In the Verification phase (Phase 6), the computation cost of a verifier in Case 1 is as follows. In step 6a, it invokes $\ddot{t}$ instances of the hash function (to check the opening of $\ddot{t}$ commitments). In step 6b it invokes $2 \cdot (\bar{t} \cdot \ddot{t} + 1)$ instances of PRF. In step 6(b)ii, it performs $\bar{t} \cdot \ddot{t} + 1$ additions and $\bar{t} \cdot \ddot{t}$ multiplications. In step 6(b)iii, it interpolates a polynomial of degree $\ddot{t} + 1$ that involves $O(\ddot{t})$ addition and $O(\ddot{t})$ multiplication operations.

In step 6(b)iv, it evaluates a polynomial of degree $\ddot{t} + 1$ at $\ddot{t}$ points, resulting in $\ddot{t}^2 + \ddot{t}$ additions and $\ddot{t}^2 + \ddot{t}$ multiplications. In step 6c, it performs $\ddot{t} + 1$ multiplication. In the Verification phase (Phase 6), the computation cost of a verifier in Case 2 involves only a single invocation of the hash function to check the opening of a commitment. Thus, the verifier's complexity is $O(\ddot{t}^2 + \ddot{t})$.

Server's Computation Cost. In step 4(b)v, server s engages $\bar{t}$ instances of $\texttt{OLE}^+$ with each client. In step 4d, server s performs $\bar{t} \cdot n$ modular addition. During the Solving Puzzles phase (Phase 5), in Case 1 step 5a, server s performs Y repeated modular squaring and invokes two instances of PRF for each client in $\mathcal{I}$. In step 5b, s it performs $\bar{t} \cdot \ddot{t} + 1$ additions and $\bar{t} \cdot \ddot{t}$ multiplications.

In step 5c, it interpolates a polynomial of degree $\ddot{t} + 1$ that involves $O(\ddot{t})$ addition and $O(\ddot{t})$ multiplication operations. In step 5d, it performs $\ddot{t} + 1$ modular multiplications. In step 5e, it factorizes a polynomial of degree $\ddot{t} + 1$ to find its root, which will cost $O(\ddot{t}^2)$. In the same step, it invokes the hash function $\ddot{t}$ times to identify the valid roots. Thus, the complexity of s in Case 1 is $O(\ddot{t}^2 + \bar{t} \cdot n + \ddot{t} \cdot Y)$.

In Case 2, the costs of server s for each client c_u involves the following operations. s performs T_u modular squaring to find master key mk_u. It invokes $\bar{t} + 2$ instances of PRF. It performs $\bar{t}$ addition and $\bar{t}$ multiplication to decrypt y-coordinates. It interpolates a polynomial of degree $\ddot{t} + 1$ that involves $O(\ddot{t})$ addition and $O(\ddot{t})$ multiplication operations. Therefore, the complexity of s in Case 2 is $O(\ddot{t} + \bar{t} + T_u)$. Note that in all schemes relying on modular squaring a server performs $O(T_u)$ squaring.

Now we proceed to the parties' communication costs. We first concentrate on each client's cost.

Client's Communication Cost. In the following analysis, we consider the communication cost of a leader client, as it transmits more messages than non-leader clients. In the Key Generation phase (Phase 2) step 2b, the client publishes a single public key of size about 2048 bits. In the Puzzle Generation phase (Phase 3) step 3g, the client publishes $\bar{t}+4$ values. In the Linear Combination phase (Phase 4), step 4b, the leader client transmits to each client a key for PRF.

In step 4(b)iii, it sends $\bar{t}$ encrypted y-coordinates of a random root to the rest of the clients. In step 4(b)v, it invokes $\bar{t}$ instances of $\texttt{OLE}^+$ where each instance imposes $O(1)$ communication cost. In step 4(b)vii, the leader client publishes four elements. Thus, the leader client's communication complexity is $O(\bar{t} \cdot n)$. Note that the size of the majority of messages transmitted by the client in the above steps is 128 bits.

Server's Communication Cost. In the Setup phase (Phase 1), the server publishes $\bar{t}+1$ messages. In the Linear Combination phase (Phase 4) step 4(b)v, it invokes $\bar{t}$ instances of $\texttt{OLE}^+$ with each client, where each instance imposes $O(1)$ communication cost. In step 4e, it publishes $\bar{t}$ messages.

In the Solving a Puzzle phase (Phase 5), Case 1, step 5f, it publishes $\ddot{t}+1$ messages. In Case 2 step 5d, the server publishes two messages. The size of each message it publishes in the last three steps is 128 bits. Therefore, the communication complexity of the server is $O(\bar{t} \cdot n)$.

6.2 Concrete Cost

Having addressed the concrete communication costs of the scheme in the previous section, we now shift our focus to the concrete computation costs. As previously discussed, the three primary operations that impose costs to the participants of our scheme are polynomial factorization, invocations of PRF, and $\texttt{OLE}^+$ execution. In this section, we analyze their concrete costs.

Implementation Environment. To evaluate the performance of polynomial factorization and PRF, we have developed prototype implementations written in C++. They can be found in [1,2]. We utilize the NTL library[5] for polynomial factorization, the GMP library[6] for modular multiple precision arithmetic, and the CryptoPP library[7] for implementing PRF based on AES. All experiments were conducted on a MacBook Pro, equipped with a 2-GHz Quad-Core Intel processor and a 16-GB RAM. With did not take advantage of parallelization. To estimate running times, we run the experiments for at least 100 times.

Choice of Parameters. Since the performance of polynomial factorization and PRF can be influenced by the size $\log_2(p)$ of the field over which polynomials are defined and the output size (also referred to as $\log_2(p)$, respectively, we use two different field sizes: 128 and 256 bits. Furthermore, in Tempora-Fusion, since increasing the total number $\ddot{t}$ of leader clients will increase the resulting polynomial's degree and the complexity of polynomial factorization is quadratic with the polynomial's degree, we run the experiment on different polynomial degrees, ranging from 2 to 10. It is worth noting that even within this range of $\ddot{t}$, the total number of clients can be very high, as discussed in Section 4.2.

Result. Increasing the polynomial's degree from 2 to 10 results in the following changes in the running time of factorization: (i) from 0.3 to 1.4 milliseconds (ms) when the field size is 128 bits, and (ii) from 0.6 to 2.2 ms when the field size is 256 bits. We observed that doubling the size of the field results in the polynomial factorization's running time increasing by a factor of approximately 1.66. Table 2 and Figure 2a elaborate on the performance of polynomial factorizations.

[5] https://libntl.org
[6] https://gmplib.org
[7] https://cryptopp.com

Table 2: Concrete runtime of polynomial factorizations, measured in milliseconds.

Field size $\log_2(p)$	Polynomial degree				
	2	4	6	8	10
128-bit	0.3	0.5	0.8	1.2	1.4
256-bit	0.6	1	1.7	2.1	2.2

Table 3: Concrete runtime of PRF invocation, measured in milliseconds.

Output size $\log_2(p)$	Number of PRF invocation					
	2	4	16	64	256	1024
128-bit	0.006	0.011	0.04	0.15	0.658	2.424
256-bit	0.008	0.016	0.071	0.29	0.97	3.534

Moreover, as we increase the number of PRF invocations from 2 to 1024, the running time (a) grows from 0.006 to 2.424 ms when the output size is 128 bits and (b) increases from 0.008 to 3.534 ms when the output size is 256 bits. Table 3 and Figure 2b elaborate on the concrete performance of PRF.

We observe that the running time of $\mathtt{OLE}^+$ is low. For instance, Boyle *et al.* CCS'18 [17], proposed an efficient generalization of OLE called vector OLE, secure against malicious adversaries. Vector OLE allows the receiver to learn any linear combination of two vectors held by the sender. In various applications of OLE, one can replace a large number of instances of OLE with a smaller number of long instances of vector OLE. The authors estimated the running time of their scheme is about 26.3 ms when the field size is about 128 bits and the input vectors size is about 2^{20}. As another example, Schoppmann *et al.* CCS'19 [43] proposed a variant of vector OLE called pseudorandom vector OLE, secure against semi-honest adversaries. This variant with the input vectors of 2^{14} elements can be run in less than 1 second.

Therefore, based on our experimental variations in polynomial degrees, field sizes, and PRF's output sizes, we project the total added concrete costs of our schemes to range between 3.007 and 3.012 seconds, factoring in an additional 2 seconds for other operations such as modular arithmetic and hash function invocations.

7 Conclusion and Future Work

Time lock puzzles (TLPs) are elegant cryptographic protocols with applications across various domains, including e-voting, timed secret sharing, timed commitments, and zero-knowledge proofs. In this work, we present a novel time lock puzzle scheme that simultaneously supports (1) partially homomorphic computation (i.e., linear combination) of different clients' puzzles and (2) efficient verification of the computation's correctness. This scheme employs a set of techniques not previously applied in the TLP context and is robust against a strong malicious server that may gain access to a subset of clients' secret keys. We demonstrate that it is possible to define the puzzles over a finite field (of relatively short size) without relying on a trusted third party. Furthermore, we have identified several applications for the proposed scheme in federated learning, online banking, and e-voting. Our analysis of the scheme's asymptotic and concrete costs confirms its efficiency. Future work could explore:

- *Post-Quantum Secure, Verifiable Homomorphic TLP*: There have been efforts to develop post-quantum secure TLPs, such as the one proposed in [34]. However, existing post-quantum secure TLPs do not

support verifiable homomorphic operations on different puzzles. Therefore, it would be compelling to upgrade these post-quantum secure TLPs to support verifiable homomorphic operations, enhancing their functionality and broadening their potential applications.

- *Scalability Improvements*: Explore methods to enhance the scalability of the proposed scheme, ensuring it can handle a large number of users and datasets without significantly compromising performance. This could involve distributed computing approaches or could replace $\texttt{OLE}^{+}$ with a more efficient and scalable primitive.
- *Real-World Implementation*: Conduct real-world implementation and testing of our scheme in various domains like online banking and federated learning. This would involve collaboration with industry partners to identify practical challenges and refine the scheme based on empirical data.

References

1. Abadi, A.: Source code for polynomial factorizations (2024), https://github.com/AydinAbadi/Tempora-Fusion/tree/main/Polynomial-Factorization
2. Abadi, A.: Source code for pseudorandom invocations (2024), https://github.com/AydinAbadi/Tempora-Fusion/tree/main/PRF%20Invocations
3. Abadi, A., Doyle, B., Gini, F., Guinamard, K., Murakonda, S.K., Liddell, J., Mellor, P., Murdoch, S.J., Naseri, M., Page, H., Theodorakopoulos, G., Weller, S.: Starlit: Privacy-preserving federated learning to enhance financial fraud detection. IACR Cryptol. ePrint Arch. (2024)
4. Abadi, A., Kiayias, A.: Multi-instance publicly verifiable time-lock puzzle and its applications. In: FC (2021)
5. Abadi, A., Ristea, D., Murdoch, S.J.: Delegated time-lock puzzle. arXiv preprint arXiv:2308.01280 (2023)
6. Aho, A.V., Hopcroft, J.E.: The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1974)
7. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: CCS'13 (2013)
8. Barclays Bank: Daily payment limits in online banking (2024), https://www.barclays.co.uk/help/payments/payment-information/online-banking-limits/
9. Baum, C., David, B., Dowsley, R., Nielsen, J.B., Oechsner, S.: TARDIS: A foundation of time-lock puzzles in UC. In: EUROCRYPT (2021)
10. Blum, M.: Coin flipping by telephone - A protocol for solving impossible problems. In: COMPCON'82, Digest of Papers, Twenty-Fourth IEEE Computer Society International Conference, San Francisco, California, USA, February 22-25, 1982. pp. 133–137. IEEE Computer Society (1982)
11. Blum, M., Santis, A.D., Micali, S., Persiano, G.: Noninteractive zero-knowledge. SIAM J. Comput. **20**(6) (1991)
12. Bonawitz, K.A., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H.B., Patel, S., Ramage, D., Segal, A., Seth, K.: Practical secure aggregation for privacy-preserving machine learning. In: CCS (2017)
13. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO'18
14. Boneh, D., Bünz, B., Fisch, B.: A survey of two verifiable delay functions. IACR Cryptol. ePrint Arch. (2018)
15. Boneh, D., Franklin, M.K.: Efficient generation of shared RSA keys (extended abstract). In: CRYPTO (1997)
16. Boneh, D., Naor, M.: Timed commitments. In: ACRYPTO. Springer (2000)
17. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: CCS (2018)
18. Brakerski, Z., Döttling, N., Garg, S., Malavolta, G.: Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In: TCC'19,
19. Buchmann, J., Williams, H.C.: A key-exchange system based on imaginary quadratic fields. J. Cryptol. (1988)
20. Chen, H., Deviani, R.: A secure e-voting system based on RSA time-lock puzzle mechanism. In: BWCCA'12 ,
21. Department of Justice–U.S. Attorney's Office: Former jp morgan chase bank employee sentenced to four years in prison for selling customer account information (2018), https://www.justice.gov/usao-edny/pr/former-jp-morgan-chase-bank-employee-sentenced-four-years-prison-selling-customer
22. Dong, C., Abadi, A., Terzis, S.: VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In: FC (2016)
23. Dorn, W.S.: Generalizations of horner's rule for polynomial evaluation. IBM Journal of Research and Development (1962)
24. Dujmovic, J., Garg, R., Malavolta, G.: Time-lock puzzles with efficient batch solving. In: EUROCRYPT. Springer-Verlag (2024)

25. Dwork, C., Naor, M.: Zaps and their applications. In: FoCS (2000)
26. 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)
27. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02
28. Ghosh, S., Nielsen, J.B., Nilges, T.: Maliciously secure oblivious linear function evaluation with constant overhead. In: ASIACRYPT (2007)
29. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: EUROCRYPT (2019)
30. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
31. Katz, J., Loss, J., Xu, J.: On the security of time-lock puzzles and timed commitments. In: Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part III. Lecture Notes in Computer Science (2020)
32. Kavousi, A., Abadi, A., Jovanovic, P.: Timed secret sharing. Cryptology ePrint Archive (2023)
33. Kissner, L., Song, D.X.: Privacy-preserving set operations. In: CRYPTO 2005, 25th International Cryptology Conference. pp. 241–257 (2005)
34. Lai, R.W.F., Malavolta, G.: Lattice-based timed cryptography. IACR Cryptol. ePrint Arch. (2024)
35. Leigh, D., Ball, J., Garside, J., Pegg, D.: Hsbc files timeline: From swiss bank leak to fallout. The Guardian **12** (2015)
36. Liu, Y., Wang, Q., Yiu, S.M.: Towards practical homomorphic time-lock puzzles: Applicability and verifiability. In: ESORICS (2022)
37. Malavolta, G., Thyagarajan, S.A.K.: Homomorphic time-lock puzzles and applications. In: CRYPTO'19
38. May, T.C.: Timed-release crypto (1993), `https://cypherpunks.venona.com/date/1993/02/msg00129.html`
39. McMahan, H.B., Moore, E., Ramage, D., y Arcas, B.A.: Federated learning of deep networks using model averaging. CoRR **abs/1602.05629** (2016)
40. 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)
41. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the society for industrial and applied mathematics (1960)
42. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep. (1996)
43. Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-ole: Improved constructions and implementation. In: CCS (2019)
44. Shamir, A.: How to share a secret. Commun. ACM (1979)
45. 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)
46. Thyagarajan, S.A.K., Bhat, A., Malavolta, G., Döttling, N., Kate, A., Schröder, D.: Verifiable timed signatures made practical. In: CCS (2020)
47. Thyagarajan, S.A.K., Malavolta, G., Moreno-Sanchez, P.: Universal atomic swaps: Secure exchange of coins across all blockchains. In: IEEE Symposium on Security and Privacy, SP (2022)
48. Thyagarajan, S.A.K., Malavolta, G., Schmidt, F., Schröder, D.: Paymo: Payment channels for monero. IACR Cryptol. ePrint Arch. (2020)
49. United States Attorney's Office: Ex-morgan stanley adviser pleads guilty in connection with data breach (2015), `https://www.reuters.com/article/idUSKCN0RL229`
50. Wan, J., Xiao, H., Devadas, S., Shi, E.: Round-efficient byzantine broadcast under strongly adaptive and majority corruptions. In: TCC. Lecture Notes in Computer Science (2020)
51. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT'19 (2019)
52. WorldRemit Content Team: Bank transfer limits uk: a guide to transferring large amounts of money (2023), `https://www.worldremit.com/en/blog/finance/bank-transfer-limit-uk`
53. WorldRemit Content Team: Bank of america online banking service agreement (2024), `https://www.bankofamerica.com/online-banking/service-agreement.go`
54. Yang, Q., Liu, Y., Chen, T., Tong, Y.: Federated machine learning: Concept and applications. ACM Trans. Intell. Syst. Technol. (2019)

A The Enhanced OLE's Ideal Functionality and Protocol

The PSIs proposed in [29] use an enhanced version of the $\texttt{OLE}$. The enhanced $\texttt{OLE}$ ensures that the receiver cannot learn anything about the sender's inputs, in the case where it sets its input to 0, i.e., $c = 0$. The enhanced $\texttt{OLE}$'s protocol (denoted by $\texttt{OLE}^+$) is presented in Figure 3.

1. Receiver (input $c \in \mathbb{F}$): Pick a random value, $r \xleftarrow{\$} \mathbb{F}$, and send $(\texttt{inputS}, (c^{-1}, r))$ to the first $\mathcal{F}_{\texttt{OLE}}$.
2. Sender (input $a, b \in \mathbb{F}$): Pick a random value, $u \xleftarrow{\$} \mathbb{F}$, and send $(\texttt{inputR}, u)$ to the first $\mathcal{F}_{\texttt{OLE}}$, to learn $t = c^{-1} \cdot u + r$. Send $(\texttt{inputS}, (t + a, b - u))$ to the second $\mathcal{F}_{\texttt{OLE}}$.
3. Receiver: Send $(\texttt{inputR}, c)$ to the second $\mathcal{F}_{\texttt{OLE}}$ and obtain $k = (t + a) \cdot c + (b - u) = a \cdot c + b + r \cdot c$. Output $s = k - r \cdot c = a \cdot c + b$.

Fig. 3: Enhanced Oblivious Linear function Evaluation ($\texttt{OLE}^+$) [29].

B The Original RSA-Based TLP

Below, we restate the original RSA-based time-lock puzzle proposed in [42].

1. Setup: $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, max_{ss})$.
   (a) pick at random two large prime numbers, q_1 and q_2. Then, compute $N = q_1 \cdot q_2$. Next, compute Euler's totient function of N as follows, $\phi(N) = (q_1 - 1) \cdot (q_2 - 1)$.
   (b) set $T = max_{ss} \cdot \Delta$ the total number of squaring needed to decrypt an encrypted message m, where max_{ss} 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., $\texttt{SKE.keyGen}(1^\lambda) \rightarrow k$.
   (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: $o_1 = \texttt{SKE.Enc}(k, m)$.
   (b) encrypt the symmetric-key encryption key k, as follows: $o_2 = k + r^a \bmod N$.
   (c) set $o := (o_1, o_2)$ as puzzle and output the puzzle.
3. Solve Puzzle: $\mathsf{Solve}_{\mathsf{TLP}}(pk, o)$.
   (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., $k = o_2 - b \bmod N$.
   (c) decrypt the message's ciphertext, i.e., $m = \texttt{SKE.Dec}(k, o_1)$. Output the solution, m.

The security of the RSA-based TLP relies on the hardness of the factoring problem, the security of the symmetric key encryption, and the sequential squaring assumption. We restate its formal definition below and refer readers to [4] for the proof.

Theorem 3. *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 TLP.*

C Sequential and Iterated Functions

Definition 10 ($\Delta, \delta(\Delta)$)-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(Δ, λ) 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 11 (Iterated Sequential function). *Let $\beta : X \to X$ be a $(\Delta, \delta(\Delta))$-sequential function. A function $f : \mathbb{N} \times X \to X$ defined as $f(k, x) = \beta^{(k)}(x) = \overbrace{\beta \circ \beta \circ ... \circ \beta}^{k \ \ Times}$ is an iterated sequential function, with round function β, if for all $k = 2^{o(\lambda)}$ the function $h : X \to X$ 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:*

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