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, with applications including scheduled payments, e-voting, and sealed-bid auctions. Two key variants of TLP are the homomorphic TLP and the multi-instance TLP. Homomorphic TLPs enable computation on puzzles from different clients, allowing a solver/server to tackle only a single puzzle encoding the computation's result. Conversely, multi-instance TLPs let a server *efficiently* find solutions to different puzzles provided by a client at once. This approach allows a server to deal with puzzles sequentially, instead of handling them simultaneously. However, two limitations persist: current homomorphic TLPs lack support for *verifying* computation results, and existing multi-instance TLPs do not facilitate (verifiable) *homomorphic computation.*

We address both limitations. Firstly, we introduce Tempora-Fusion, a TLP that allows a server to perform homomorphic linear combinations of puzzles from different clients, while ensuring verification of computation correctness. Secondly, we propose a multi-instance TLP that supports verifiable homomorphic linear combinations on puzzles. It enables sequential puzzle-solving by a server while ensuring anyone can check computations and solutions' correctness. Both schemes avoid asymmetric-key cryptography for verification, thus paving the way for efficient implementations.

We discuss our schemes' applications in various domains, such as federated learning, scheduled payments in online banking, and verifiable e-voting.

CCS CONCEPTS

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

KEYWORDS

Time-Lock Encryption, Privacy, Secure Multi-Party Computation

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 [17], sealed-bid auctions [41], byzantine broadcast [49], zero-knowledge proofs [24], and verifiable delay functions [11].

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.* [41], the TLPs have evolved, giving rise to at least two main variants: homomorphic TLPs and multi-instance TLPs. Our work contributes to both of these variants.

Malavolta *et al.* [35] 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 [35].

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.

In a separate line of research, multi-instance TLPs were introduced in [1]. A multi-instance TLP setting entails a single client generating n puzzles and simultaneously transmitting them to the server. This concept serves as a natural extension of the initial single-puzzle paradigm established in [41]. In the multi-instance TLP, each puzzle's solution is found by the server at a different time. Multi-instance TLPs allow the server to deal with each puzzle *sequentially* rather than simultaneously handling them. Such an approach leads to notable reductions in computational overhead for the server. Multi-instance TLPs have found application in diverse domains, including the gradual release of confidential documents (by journalists or whistleblowers), sequential revealing of secret keys, and continuous verification of cloud service availability [1].

1.1 Limitations of TLP Schemes

1.1.1 Homomorphic TLPs. 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.

1.1.2 Multi-Instance TLPs. Current multi-instance TLPs lack support for any (verifiable) homomorphic operations on puzzles. These TLPs only facilitate the server's learning of each puzzle's solution after a predetermined time and allow a verifier to confirm the correctness of the solution obtained by the server. This limitation can hinder the real-world applicability of multi-instance TLPs.

*aydin.abadi@newcastle.ac.uk

1.2 Our Solutions

1.2.1 Partially Homomorphic TLP. To overcome the lack of support for verification in (partially) homomorphic TLPs, this work introduces Tempora-Fusion 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 clients, each with a coefficient q_i and a secret solution m_i. Tempora-Fusion lets each client 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. Upon discovering a solution, it can efficiently demonstrate the correctness of the solution to any party.

Crucially, at a later stage, 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 after a designated period. Once the server computes the result $\sum\limits_{j=1}^{n} q_i \cdot m_i$, and publish it, anyone can efficiently verify its correctness. The verification mechanisms employed in Tempora-Fusion 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 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.

It 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.

1.2.2 Multi-Instance Single-Client Partially Homomorphic TLP. To address the limitation of multi-instance TLPs, we introduce Multi-Instance Tempora-Fusion, the first multi-instance TLP that supports efficient verifiable homomorphic linear combinations on puzzles. It enables a client to generate many puzzles and transmit them to the server at once. The server does not need to simultaneously handle them; instead, it can solve them sequentially.

It enables the client to return online at a later point to grant computation on its puzzles. The server will *learn the linear combination of puzzles' solutions* after a certain time. This scheme also supports public verification for (1) a single puzzle's solution, and (2) the computation's result. Inspired by previous works, we rely on the idea of chaining puzzles, such that when the server solves one puzzle it will obtain enough information to work on the next puzzle. However, we introduce a new technique for chaining puzzles that also facilitates homomorphic linear combinations. This method enables the client to derive the base for the next puzzle from the current puzzle's master key, without altering the structure of the underlying solution.

Tempora-Fusion and Multi-Instance Tempora-Fusion can be combined into a single protocol, incorporating the features of both.

1.3 Applications

1.3.1 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 [36, 53]. 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.* [10] 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.

1.3.2 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 [20, 33]. Investment strategies scheduled by individuals or companies, through financial institutions, contain sensitive information that could be exploited by insiders.

Multi-Instance 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, Multi-Instance 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 [6, 51, 52].

1.3.3 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 [35] .

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

Rivest *et al.* [41] initially introduced an RSA-based TLP. It 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, such as those in [7, 13, 26]. In this section, we delve into variants closely related to our research. Also, we discuss the concept of verifiable delay function in Appendix A.

2.1 Homomorphic Time-lock Puzzles

Malavolta and Thyagarajan *et al.* [35] 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 [34, 35]. Partially homomorphic TLPs do not rely on fully homomorphic encryption resulting in more efficient implementations. Unlike the partially homomorphic TLP in [35], the ones in [34] allow a verifier to ensure (1) puzzles have been generated correctly and (2) 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, proposed in [50].

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. The construction is based on indistinguishability obfuscation and puncturable pseudorandom function. To improve the efficiency of this scheme, Dujmovic *et al.* [23] proposed a new approach, without using indistinguishability obfuscation. It relies on pairings and learning with errors.

Except for the TLP proposed in [45], all mentioned homomorphic TLPs require a trusted setup and none offers a means to verify computation corectness.

2.2 Multi-instance Time Lock Puzzle

Researchers also considered the setting where a server may receive multiple puzzles/instances at once from a client and need to solve all of them. The TLPs proposed in [1, 18] deal with such a setting. The scheme in [18] uses an asymmetric-key encryption scheme, as opposed to a symmetric-key one used in the original TLP. Also, this scheme offers no verification. To address these limitations Abadi and Kiayias [1] proposed a scheme that uses efficient hash-based verification to let the server prove that it found the correct solutions. It efficiently operates only for identical-size time intervals. Later, this limitation was addressed in [40]. Nevertheless, none of these TLPs support (verifiable) homomorphic computations on puzzles.

3 PRELIMINARIES

3.1 Notations and Assumptions

We define Δ_u as the period within which client $\mathbf{c}_u$ wants its puzzle's solution m_u to remain secret. We define U as the universe of a solution m_u. Also, $\ddot{t}$ refers to the total number of leaders. We set $\bar{t} = \ddot{t} + 2$. We denote by $\lambda \in \mathbb{N}$ the security parameter. We use the security parameter $poly(\lambda)$ to state the parameter is a polynomial function of λ. We define a public set 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 $\mathsf{G} : \{0,1\}^* \rightarrow \{0,1\}^{poly(\lambda)}$, that maps 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.

To ensure generality in our verification algorithms' definition, we adopt notations from zero-knowledge proof systems [9, 25]. Let R_{cmd} be an efficient binary relation consisting 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 $\mathcal{NP}$ language associated with R_{cmd}, i.e., $\mathcal{L}_{cmd} = \{stm_{cmd} |\ \exists wit_{cmd} \text{ s.t. } R(stm_{cmd}, wit_{cmd}) = 1\}$. A proof for $\mathcal{L}_{cmd}$ lets a prover 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 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. We consider a strong malicious (or active) adversary that will corrupt the server and may gain access to secret keys and parameters of a subset of the clients.

3.2 Pseudorandom Function

Informally, a pseudorandom function is a deterministic function that takes an arbitrary input and a key of length $poly(\lambda)$. It 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: $\mathsf{PRF} : \{0,1\}^* \times \{0,1\}^{poly(\lambda)} \rightarrow \mathbb{F}_p$. 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.* [27] proposed an efficient OLE that has $O(1)$ cost and involves mainly symmetric-key operations.[1]

Later, in [28] an enhanced OLE, called OLE^+, was proposed. The OLE^+ ensures that the receiver cannot learn anything about the sender's inputs, even if it sets its input to 0. OLE^+ is accompanied by an efficient symmetric-key-based verification mechanism that enables a party to detect its counterpart's misbehavior during the protocol's execution. We use OLE^+ to securely switch the blinding factors of solutions (in puzzles) held by a server. We refer readers to Appendix B, for the construction of OLE^+.

3.4 Polynomial Representation of a Message

Encoding a message m as a polynomial $\boldsymbol{\pi}(x)$ allows us to impose a certain structure on the message. This representation has been used in various contexts, such as in secret sharing [43], private set intersection [32], and error-correcting codes [39].

There as two common approaches to encode m in $\boldsymbol{\pi}(x)$: (1) setting m as the constant terms of $\boldsymbol{\pi}(x)$, e.g., $m + \sum_{j=1}^{n} x^j \cdot a_j \bmod p$ and (2) setting m as the root of $\boldsymbol{\pi}(x)$, e.g., $\boldsymbol{\pi}(x) = (x - m) \cdot \boldsymbol{\tau}(x) \bmod p$. We employ both approaches. The former enables us to perform a linear combination of the constant terms of different polynomials. 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 computations' correctness.

[1]The scheme uses an Oblivious Transfer (OT) extension as a subroutine. The OT extension requires 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 [5].

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, $1 \leq i \leq l$. A polynomial in this form can be converted into coefficient form via polynomial interpolation, e.g., via Lagrange interpolation [2].

3.4.1 Arithmetic of Polynomials in Point-Value Form. Arithmetic of polynomials in point-value form 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, polynomial $\boldsymbol{\theta}(x)$ interpolated from $\{(x_1, a \cdot \pi_1), \ldots, (x_l, a \cdot \pi_l)\}$ has the form: $\boldsymbol{\theta}(x) = a \cdot \boldsymbol{\pi}(x)$.

3.5 Unforgeable Encrypted Polynomial

An interesting property of encrypted polynomials has been stated in [21]. It can be described as follows. Let us consider a polynomial $\boldsymbol{\pi}(x)$ that has a random secret root β. We represent $\boldsymbol{\pi}(x)$ in the point-value form and encrypt its y-coordinates. We give all the x-coordinates and encrypted y-coordinates to an adversary. 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 the y-coordinates sent by the adversary and interpolate a polynomial $\boldsymbol{\pi}'(x)$, then the probability that $\boldsymbol{\pi}'(x)$ will have root β is negligible. Below, we formally state it.

Theorem 3.1 (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 a 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 $o_i = w_i \cdot (\pi_i + z_i)$* mod 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 $\{(x_1, (w_1^{-1} \cdot \ddot{o}_1) - z_l), \ldots, (x_l, (w_l^{-1} \cdot \ddot{o}_l) - z_l)\}$ will have root β is negligible in λ', i.e.,*

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

We refer readers to [21, p.160] for the proof of Theorem 3.1. 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 involves a sender and a receiver. It includes two phases, commit and open. In the commit phase, the sender commits to a message m as $\texttt{Com}(m, r) = com$, that involves a secret value, r. In the open phase, the sender sends the opening $\hat{m} := (m, r)$ to the receiver which verifies its correctness: $\texttt{Ver}(com, \hat{m}) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy: (a) *hiding*: it is infeasible for an adversary to learn any information about the committed message m, until the commitment com is opened, and (b) *binding*: it is infeasible for a malicious sender to open a commitment com to different values than that was used in the commit phase. In this paper, we use a commitment scheme to detect a server's misbehaviors and rely on the standard efficient hash-based commitment scheme. Appendix C provides further detail.

3.7 Time-Lock Puzzle

In this section, we restate the TLP's formal definition. A TLP consists of three algorithms: $(\mathsf{Setup}_{\mathsf{TLP}}, \mathsf{GenPuzzle}_{\mathsf{TLP}}, \mathsf{Solve}_{\mathsf{TLP}})$ defined below. TLP involves a client **c** and a server **s**.

- $\mathsf{Setup}_{\mathsf{TLP}}(1^\lambda, \Delta, max_{ss}) \rightarrow (pk, sk)$. 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 parameter max_{ss} which is the maximum number of squarings that a solver can perform per second. It outputs 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 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.

TLP meets completeness and efficiency. Completeness ensures that for any honest **c** and **s**, it always holds: $\mathsf{Solve}_{\mathsf{TLP}}(pk, o) = m$. Efficiency ensures that 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 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. Appendix D provides a formal security definition of a TLP scheme and a description of the original RSA-based TLP realizing this definition.

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 [11]. We restate the definitions in Appendix E.

4 DEFINITION OF TLP WITH VERIFIABLE HOMOMORPHIC LINEAR COMBINATION

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 the j-th party (for every $j, 1 \leq j \leq n$), and returns their linear combination $\sum\limits_{j=1}^{n} q_j \cdot m_j$ to each party. More formally, $\mathcal{F}^{\text{PLC}}$ is defined as:

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

Note that $\mathcal{F}^{\text{PLC}}$ implies that corrupt parties colluding with each other can always deduce the linear combination of non-colluding parties' inputs from the output of $\mathcal{F}^{\text{PLC}}$. This point remains valid for any scheme that realizes this functionality (including the ones in [15, 35] and ours) irrespective of the primitives employed.

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

4.1 Syntax

Definition 4.1 (Syntax). A Verifiable Homomorphic Linear Combination TLP ($\mathcal{VHLC}$-$\mathcal{TLP}$) scheme consists of six algorithms: $\mathcal{VHLC}$-$\mathcal{TLP}$ = (S.Setup, C.Setup, GenPuzzle, Evaluate, Solve, Verify), as defined below.

- $\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_{\mathbf{s}}$. It is an algorithm run by the server **s**. It takes security parameters 1^λ, $\ddot{t}$, and t. It generates a pair $K_{\mathbf{s}} :=$

(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 $\mathbf{s}$ publishes pk_s.

- $\mathsf{C.Setup}(1^\lambda) \rightarrow K_u$. It is a probabilistic algorithm run by a client $\mathbf{c}_u$. It takes 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 $\mathbf{c}_u$ 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 $\mathbf{c}_u$. It takes as input 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 take. It returns a vector $\vec{o}_u$ (representing a puzzle) and 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 $\mathbf{c}_u$. Client $\mathbf{c}_u$ publishes $(\vec{o}_u, prm_u)$.
- $\mathsf{Evaluate}(\langle \mathbf{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), c_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$. It is an (interactive) algorithm run by $\mathbf{s}$ (and each client in $\{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$). When no interaction between $\mathbf{s}$ and the clients is required, the clients' inputs will be $\perp$. $\mathbf{s}$ 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 $\mathbf{c}_u$ takes as input $\Delta, max_{ss}, K_u, prm_u$, coefficient q_u, and pk_s. It returns a vector of public parameters $\vec{pp}^{(\text{Evl})}$ and a puzzle vector $\vec{g}$, representing a puzzle. $\mathbf{s}$ publishes $\vec{g}$ and the clients publish $\vec{pp}^{(\text{Evl})}$.
- $\mathsf{Solve}(\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_s, cmd) \rightarrow (m, \zeta)$. It is a deterministic algorithm run by $\mathbf{s}$. It takes as input a single client $\mathbf{c}_u$'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 $\vec{pp}^{(\text{Evl})}$, pk_s, and a command cmd, where $cmd \in \{\text{clientPzl}, \text{evalPzl}\}$. When $cmd = \text{clientPzl}$, it solves puzzle $\vec{o}_u$ (an output of $\mathsf{GenPuzzle}()$), this yields a solution m. In this case, input $\vec{g}$ can be $\perp$. When $cmd = \text{evalPzl}$, it solves puzzle $\vec{g}$ (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 $\mathbf{s}$ publishes (m, ζ).
- $\mathsf{Verify}(m, \zeta, \vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_s, cmd) \rightarrow \ddot{v} \in \{0, 1\}$. It is a deterministic algorithm run by a verifier. It takes as input a solution m, proof ζ, puzzle $\vec{o}_u$ of a single client $\mathbf{c}_u$, public parameters pp_u of $\vec{o}_u$, a puzzle $\vec{g}$ for a linear combination of puzzles, public parameters $\vec{pp}^{(\text{Evl})}$ 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 $\mathbf{c}$'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 $cmd = \text{clientPzl}$, then $pp = pp_u$ and if $cmd = \text{evalPzl}$, then $pp = \vec{pp}^{(\text{Evl})}$.

4.2 Security Model

A $\mathcal{VHLC\text{-}TLP}$ scheme must satisfy *security* (i.e., privacy and solution-validity), *completeness*, *efficiency*, and *compactness* properties. Each security property of a $\mathcal{VHLC\text{-}TLP}$ 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 present the definitions of $\mathcal{VHLC\text{-}TLP}$'s properties.

4.2.1 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 $\mathsf{O.Evaluate}()$. Also, 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 $\mathcal{E}$. 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 run interactively between the corrupt parties in $\mathcal{W}$ and $\mathcal{E}$. 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 $\mathsf{Evaluate}(\langle \hat{\mathbf{s}}(input_s), \hat{\mathbf{c}}_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \ldots, \hat{\mathbf{c}}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$, where the inputs of honest parties are retrieved by $\mathcal{E}$ from $st_{\mathcal{E}}$ and if $\hat{\mathbf{s}} \in \mathcal{W}$, then $\hat{\mathbf{s}}$ may provide arbitrary inputs $input_s$ during the execution of Evaluate. However, when $\hat{s} \notin \mathcal{W}$ then $\hat{\mathbf{s}}$ is an honest server (i.e., $\hat{\mathbf{s}} = \mathbf{s}$) and $input_s = (\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s)$. Moreover, when $\hat{\mathbf{c}}_j \notin \mathcal{W}$ then $\hat{\mathbf{c}}_j$ is an honest client (i.e., $\hat{\mathbf{c}}_j = \mathbf{c}_j$). $\mathcal{E}$ 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 $(\vec{g}, \vec{pp}^{(\text{Evl})})$ is an output (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 $\perp$ and returns $transcript^{(\text{Evl})}$ to $\mathcal{A}$. Otherwise, $\mathcal{E}$ 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 $\mathcal{A}$.

4.2.2 Properties. Next, we formally define the primary properties of a $\mathcal{VHLC\text{-}TLP}$ scheme, beginning with the privacy. Informally, *privacy* states that a solution m to a 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 message that represents the linear evaluation of solutions must remain undisclosed to the aforementioned adversary within a period.

To capture privacy, we define an experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^\lambda, n, \ddot{t}, t)$, that involves $\mathcal{E}$ 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 the three oracles. It may adaptively corrupt a subset $\mathcal{W}$ of the parties involved, i.e., $\mathcal{W} \subset P = \{\mathbf{s}, \mathbf{c}_1, \ldots, \mathbf{c}_n\}$, 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 $\mathbf{c}_u$ (line 14). $\mathcal{E}$ for each pair of messages provided by $\mathcal{A}_1$ selects a random bit b_u and generates a puzzle for the message with index b_u (lines 15–17).

The adversary can also engage with $\mathcal{E}$ to execute Evaluate() and output a state with the help of O.Reveal(). Before executing Evaluate(), the experiment returns 0 and halts, if $\mathcal{A}$ corrupts more than t parties in $\mathcal{I}$. For instance, when $|P| = 100$, $\ddot{t} = 5$, and $t = 2$, the experiment allows the adversary to corrupt 98 parties in P (i.e., $|\mathcal{W}| = 98$), as long as at most 3 parties from $\mathcal{I}$ are in $\mathcal{W}$.

With the above states produced by $\mathcal{A}_1$, $\mathcal{A}_2$ guesses the value of bit b_u for its chosen client (line 19 or 26). The adversary wins the game (i.e., the experiment outputs 1) if its guess is correct for a non-corrupt client (line 20 or 26). Appendix F provides a more detailed explanation of $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^{\lambda}, n, \ddot{t}, t)$.

Definition 4.2 (Privacy). A $\mathcal{VHLC}\text{-}\mathcal{TLP}$ 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: $\mathsf{S.Setup}(1^{\lambda}, \ddot{t}, t) \rightarrow K_{\mathbf{s}} := (sk_{\mathbf{s}}, pk_{\mathbf{s}})$
2: For $u = 1, \ldots, n$ do :
3:     $\mathsf{C.Setup}(1^{\lambda}) \rightarrow K_u := (sk_u, pk_u)$
4: $state \leftarrow \{pk_{\mathbf{s}}, pk_1, \ldots, pk_n\}, \mathcal{W} \leftarrow \emptyset, \mathcal{I} \leftarrow \emptyset$
5: $K \leftarrow \emptyset, cont \leftarrow True, counter \leftarrow 0, \vec{b} \leftarrow \mathbf{0}$
6: For $1, \ldots, \ddot{t}$ do :
7:     Select $a$ from $\{1, \ldots, n\}$
8:     $\mathcal{I} \leftarrow \mathcal{I} \cup \{\mathcal{B}_a\}$
9: While $(cont = True)$ do :
10:    $\mathcal{A}_1(state, \mathsf{O.GenPuzzle}, \mathsf{O.Evaluate}, K, \mathcal{I}, \Delta_1, \ldots, \Delta_n, \Delta,$
         $max_{ss}) \rightarrow (state, cont, \mathcal{B}_j)$
11:    If $cont = True$, then
12:        $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathcal{B}_j\}$
13:        $K \leftarrow K_{\mathcal{B}_j}$
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:    $b_u \overset{\$}{\leftarrow} \{0, 1\}, \quad \vec{b}[u] \leftarrow b_u$
17:    $\mathsf{GenPuzzle}(m_{b_u,u}, K_u, pk_{\mathbf{s}}, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$
18: $\mathcal{A}_1(state, K, \mathcal{W}, \mathsf{O.GenPuzzle}, \mathsf{O.Evaluate}, \vec{o}_1, \ldots, \vec{o}_n, \vec{pp}_1, \ldots,$
         $\vec{pp}_n) \rightarrow state$
19: $\mathcal{A}_2(\vec{o}_1, \ldots, \vec{o}_n, \vec{pp}_1, \ldots, \vec{pp}_n, state) \rightarrow (b'_u, u)$
20: If $(b'_u = \vec{b}[u]) \wedge (\mathcal{B}_u \notin \mathcal{W})$, then return 1
21: If $|\mathcal{W} \cap \mathcal{I}| > t$, then return 0
22: $\mathsf{Evaluate}(\langle \hat{\mathbf{s}}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_{\mathbf{s}}), \hat{\mathbf{c}}_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_{\mathbf{s}}),$
         $\ldots, \hat{\mathbf{c}}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_{\mathbf{s}})\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})}), s.t.$
23:    If $\hat{s} \in \mathcal{W}$, $\hat{s}$ knows $(state, K)$, else $\hat{\mathbf{s}}$ is an honest server, $\hat{\mathbf{s}} = \mathbf{s}$
24:    If $\hat{\mathbf{c}}_j \in \mathcal{W}$, $\hat{\mathbf{c}}_j$ knows $(state, K)$, else $\hat{\mathbf{c}}_j$ is an honest client, $\hat{\mathbf{c}}_j = \mathbf{c}_j$
25: $\mathcal{A}_1(state, K, \mathsf{O.Reveal}, \mathcal{W}, \vec{g}, \vec{pp}^{(\text{Evl})}) \rightarrow state$
26: $\mathcal{A}_2(\vec{o}, \vec{pp}, \vec{g}, \vec{pp}^{(\text{Evl})}, state) \rightarrow (b'_u, u)$
27: If $(b'_u = \vec{b}[u]) \wedge (\mathcal{B}_u \notin \mathcal{W})$, then return 1, else return 0
```

it holds that:

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

Solution validity requires that it should be infeasible for an adversary to come up with an invalid solution and successfully pass the verification process. To capture solution validity, we define experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{val}}(1^{\lambda}, n, \ddot{t}, t)$, involving $\mathcal{E}$ and an adversary $\mathcal{A}$ which may corrupt a set of parties and learn their secret keys.

Given the corrupt parties, their secret keys, and access to the oracles, $\mathcal{A}$ outputs a message m_u for each client $\mathbf{c}_u$. $\mathcal{E}$ generates a puzzle for each message that $\mathcal{A}$ selected. The experiment returns 0 and halts, if $\mathcal{A}$ corrupts more than t parties in $\mathcal{I}$. Otherwise, it allows the corrupt parties and $\mathcal{E}$ to run Evaluate() (line 16). $\mathcal{E}$ solves every client's puzzle and the puzzles for the computation. Given the solutions, $\mathcal{A}$ provides a solution and proof for its chosen client. $\mathcal{E}$ verifies the solution and proof. The experiment outputs 1 (and $\mathcal{A}$ wins) if $\mathcal{A}$ persuades $\mathcal{E}$ to accept an invalid message for a client's puzzle. Appendix G provides a formal definition of $\mathsf{Exp}^{\mathcal{A}}_{\text{val}}(1^{\lambda}, n, \ddot{t}, t)$ and explains it in detail.

Definition 4.3 (Solution-Validity). A $\mathcal{VHLC}\text{-}\mathcal{TLP}$ 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 $\mathsf{Exp}^{\mathcal{A}}_{\text{val}}(1^{\lambda}, n, \ddot{t}, t)$, it holds that:

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

Definition 4.4 (Security). A $\mathcal{VHLC}\text{-}\mathcal{TLP}$ is secure if it satisfies privacy and solution validity as outlined in Definitions 4.2 and 4.3.

Completeness considers the algorithms' behavior 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.

Efficiency states that (1) Solve() returns a solution in polynomial time, i.e., polynomial in the time parameter T, (2) GenPuzzle() generates a puzzle faster than solving it, with a running time of at most logarithmic in T, and (3) the running time of Evaluate() is faster than solving any puzzle involved in the evaluation, that should be at most logarithmic in T [23].

Compactness requires that the size of evaluated ciphertexts is independent of the complexity of the evaluation function $\mathcal{F}^{\text{PLC}}$. Appendix H presents formal definitions of completeness, efficiency, and compactness.

5 Tempora-Fusion

In this section, we present Tempora-Fusion[2], a protocol that realizes $\mathcal{VHLC}\text{-}\mathcal{TLP}$ and supports a homomorphic linear combination of puzzles. We must address several challenges to develop an efficient scheme. These challenges and our approaches for tackling them are outlined below.

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

5.1 Challenges to Overcome

5.1.1 Defining 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 TLP, it can be over the same N. There are a few approaches to deal with it. Below, we briefly discuss them.

Jointly computing N: With this approach, all clients agree on two sufficiently large prime numbers p_1 and p_2 and then compute $N = p_1 \cdot p_2$. However, this approach will not be secure if a client reveals the secret key $\phi(N)$ to the server $\mathbf{s}$. In this case, $\mathbf{s}$ can retrieve the honest party's solution without performing the sequential work.

Using a trusted setup: Via this approach a fully trusted third party generates $N = p_1 \cdot p_2$ and publishes N. However, trusting the third party and assuming that it will not collude with and not reveal the secret to $\mathbf{s}$ may be considered a strong assumption. The homomorphic TLPs proposed in [23, 35] use a trusted setup.

Using the class group of an imaginary quadratic order: through this approach, one can use the class group of an imaginary quadratic order [16]. However, employing this approach will hamper the scheme's efficiency as the puzzle generation phase will no longer be efficient [35]. Hence, it will violate the efficiency of generating puzzles requirement, i.e., Requirement 2 in Definition H.2.

To address this challenge, we propose and use the following new technique, simplified for the sake of presentation. We require the server $\mathbf{s}$ 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 [41], 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: $k = \mathsf{PRF}(1, mk)$, $s = \mathsf{PRF}(2, mk)$, 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: $o = k \cdot (m + s) \bmod p$.

Now all clients' puzzles are defined over identical field $\mathbb{F}_p$. Given each puzzle o (as well as public parameters N, p, and r), a server $\mathbf{s}$ 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 decrypting o to obtain m.

5.1.2 Supporting Homomorphic Linear Combination. Establishing a field within which all puzzles are defined, we briefly explain the techniques we utilize to facilitate a homomorphic linear combination of the puzzles. Recall that each client uses different random values to encrypt its message. Hence, 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 enable $\mathbf{s}$ to 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 $\mathbf{s}$ solves puzzles related to the linear combination, $\mathbf{s}$ can remove these blinding factors. Each leader sends (the representation of some of) the blinding factors to the rest of the clients.

To securely switch the blinding factors, each client participates in an instance of OLE^+ with $\mathbf{s}$. In this case, the client's input is the new blinding factors and the inverse of the old ones while the input of $\mathbf{s}$ is the client's puzzle. OLE^+ returns the output with refreshed blinding factors to $\mathbf{s}$. To ensure that $\mathbf{s}$ will learn only the linear combination of honest clients' solutions, without learning a solution for a client's puzzle, leaders generate and send to the rest of the clients some 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 OLE^+ that it invokes with $\mathbf{s}$, such that the result returned by OLE^+ to $\mathbf{s}$ will also be blinded by these pseudorandom values.

5.1.3 Efficient Verification of the Computation. A homomorphic time-lock puzzle needs to enable a verifier to check if the result computed by $\mathbf{s}$ is correct. 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 $\mathbf{s}$ may exclude or modify some of the clients' puzzles before, during, or after the computation, and (4) $\mathbf{s}$ 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 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 (i.e., a trap) and insert it into its outsourced puzzle (i.e., a blinded polynomial), during the invocation of OLE^+ with $\mathbf{s}$. It commits to this root (using a random value that $\mathbf{s}$ 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 OLE^+. Server $\mathbf{s}$ sums all the outputs of OLE^+ instances and publishes the result. To find the plaintext result, $\mathbf{s}$ 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 $\mathbf{s}$ follows the protocol's instruction, all roots selected by the leaders will appear in the resulting polynomial that encodes the linear combination of all clients' solutions. However, if $\mathbf{s}$ misbehaves then the honest clients' roots will not appear in the result (according to Theorem 3.1). Therefore, a verifier can detect it.

For $\mathbf{s}$ to prove the computation's correctness, after it solves the puzzles for the computation, it removes the blinding factors. It extracts and publishes the (i) linear combination of the solutions, (ii) roots, and (iii) randomness used for the commitments. Given this information and public parameters, anyone can check the correctness of the computation's result. Appendix I explains how the above approaches address the four main challenges laid out above.

5.2 The Protocol

We will present Tempora-Fusion in three tiers of detail, high-level overview, intermediate-level description, and detailed construction.

5.2.1 High-Level Overview. Initially, each client generates a puzzle encoding a secret solution (or message) and sends the puzzle to $\mathbf{s}$. Each client may generate and send its puzzle to $\mathbf{s}$ at different times. At this point, the client can locally delete its solution and go offline.

Upon receiving each puzzle, $\mathbf{s}$ will work on it to find the solution. The time that each client's solution is found can be different from that of other clients. Along with each solution, $\mathbf{s}$ generates a proof asserting the correctness of the solution. Anyone can efficiently verify the proof. Later, possibly long after they 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. 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 homomorphically combine these puzzles, each client will interact with **s**, imposing a structure to its outsourced puzzle. Each client will also send a short message to other clients. After that, each client can return 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. Moreover, **s** may eventually find 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.

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

Setup. Initially, server **s** generates a set of public parameters, including a large prime number p and a set $X = \{x_1, \ldots, x_{\bar{t}}\}$. 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.

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

Puzzle Generation. Using its secret key and time parameter T_u that determines how long a solution must remain concealed, each client $\mathbf{c}_u$ 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, 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 solution m_u as a polynomial in point-value form. This yields a vector of y-coordinates: $[\pi_{1,u}, \ldots, \pi_{\bar{t},u}]$. It encrypts each y-coordinate using the 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.

Note, the client represents solution m_u as a polynomial, 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 m_u using mk_u as the randomness. The client sends to **s** com_u and the puzzle $\vec{o}_u$. If **s** solves the puzzle, it can find both m_u and mk_u, and prove they match the commitment.

Linear Combination. 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: $\sum_{\forall \mathbf{c}_u \in C} q_u \cdot m_u$, where each q_u is a coefficient picked by each client $\mathbf{c}_u$.

The clients initially collaborate to designate a small subset $\mathcal{I}$ of themselves as leaders. Each leader $\mathbf{c}_u$, 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 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.

Each leader selects a random root: $root_u$, and commits to it, using tk_u as: $com'_u = \texttt{Com}(root_u, tk_u)$. This approach will ensure that **s** (i) cannot come up with its root, and (ii) will find the commitment's opening if it solves the leader's puzzle. Each leader 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 values to other clients. This (encrypted) root of each leader will be inserted by every client into its outsourced puzzle to give a certain structure to the computation result.

For every client, for instance, $\mathbf{c}_l$, each leader selects a fresh key f_l and sends it to that client. This key is used by each $\mathbf{c}_l$ and the leader to generate zero-sum pseudorandom values. These values are generated such that if those generated by each $\mathbf{c}_l$ and the leader are summed, they will cancel out each other.

Each leader 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 $\prod_{\forall \mathbf{c}_l \in \mathcal{I} \setminus \mathbf{c}_u} \gamma'_{i,l} \bmod p$ received from other leader clients. **s**'s input is the client's puzzle. Each $\texttt{OLE}^+$'s instance returns to **s** an encrypted y-coordinate. Each leader client publishes public parameters $pp_u^{(\text{Evl})}$.

Each non-leader client also participates in an instance of $\texttt{OLE}^+$ with **s**, for each y-coordinate. The non-leader client's input is similar to a leader's one, 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 $\prod_{\forall \mathbf{c}_l \in \mathcal{I}} \gamma'_{i,l} \bmod p$, where each $\gamma'_{i,l} = \gamma_{i,u} \cdot w'_{i,u} \bmod p$ has been sent to it by a leader client. Each instance of $\texttt{OLE}^+$ 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.

Solving a Puzzle. Server **s** operates as follows when it wants to find the computation's result. Given public parameters $pp_u^{(\text{Evl})}$ of each leader, it solves each leader's puzzle (generated for the computation) to find temporary key tk_u, letting it remove a layer of encryption from each g_i. By removing all encryption layers, **s** obtains a set of y-coordinates. It uses them 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 commitment com'_u. It retrieves 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, independent of the linear combination. Given public parameters pp_u and puzzle vector $[o_{1,u}, \ldots, o_{\bar{t},u}]$ (generated in puzzle generation phase) for a client $\mathbf{c}_u$, server **s** after time Δ_u finds key mk_u. Using mk_u, **s** removes the blinding factors from each $o_{i,u}$, yielding a vector of y-coordinates. It uses them and x-coordinates in X to interpolate a polynomial $\boldsymbol{\pi}_u(x)$ and retrieves message m_u from $\boldsymbol{\pi}_u(x)$. It publishes m_u and mk_u that match the commitment com_u, generated in the puzzle generation phase.

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

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

(1) **Setup**. $\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow (., pk_{\mathbf{s}})$
The server $\mathbf{s}$ (or any party) only once takes the following steps:
  (a) *Setting a field's parameter*: generates a sufficiently large prime number p, where $\log_2(p) \geq 128$ is security parameter.
  (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 $pk_{\mathbf{s}} = (p, X, t)$.

(2) **Key Generation**. $\mathsf{C.Setup}(1^\lambda) \rightarrow K_u$
Each party $\mathbf{c}_u$ in $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$ 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 $\log_2(p_i) \geq 2048$ is a security parameter. 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**. $\mathsf{GenPuzzle}(m_u, K_u, pk_{\mathbf{s}}, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$
Each $\mathbf{c}_u$ 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 $pk_{\mathbf{s}}$, 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 as: $a_u = 2^{T_u} \bmod \phi(N_u)$, 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: $mk_u = r_u^{a_u} \bmod N_u$
    (iii) derive 2 keys from mk_u: $k_u = \mathsf{PRF}(1, mk_u)$, $s_u = \mathsf{PRF}(2, mk_u)$.
  (c) *Generating blinding factors*: generates $2 \cdot \bar{t}$ pseudorandom blinding factors using k_u and s_u:
$$\forall i, 1 \leq i \leq \bar{t}: \quad z_{i,u} = \mathsf{PRF}(i, k_u), \quad w_{i,u} = \mathsf{PRF}(i, s_u)$$
  (d) *Encoding plaintext message*:
    (i) represents plaintext message m_u as a polynomial, by computing polynomial $\boldsymbol{\pi}_u(x)$ as: $\boldsymbol{\pi}_u(x) = x + m_u \bmod p$.
    (ii) computes $\bar{t}$ y-coordinates of $\boldsymbol{\pi}_u(x)$:
$$\forall i, 1 \leq i \leq \bar{t}: \qquad \pi_{i,u} = \boldsymbol{\pi}_u(x_i) \bmod p$$
  (e) *Encrypting the message*: encrypts y-coordinates as follows.
$$\forall i, 1 \leq i \leq \bar{t}: \qquad o_{i,u} = w_{i,u} \cdot (\pi_{i,u} + z_{i,u}) \bmod p$$
  (f) *Committing to the message*: commits to the solution:
$$com_u = \mathsf{Com}(m_u, mk_u)$$
  (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**. $\mathsf{Evaluate}(\langle \mathbf{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_{\mathbf{s}}), c_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_{\mathbf{s}}), \ldots, c_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_{\mathbf{s}}) \rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$
In this phase, the parties produce certain messages that let $\mathbf{s}$ find a linear combination of the clients' solutions 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 [8]. Each $\mathbf{c}_u$ deterministically finds index of $\ddot{t}$ leader clients: $\forall j, 1 \leq j \leq \ddot{t} : idx_j = \mathsf{G}(j||\hat{r})$. Let $\mathcal{I}$ be a vector contain these $\ddot{t}$ clients.
  (b) *Granting the computation by each leader client*: each leader $\mathbf{c}_u$ 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:
$$k'_u = \mathsf{PRF}(1, tk_u), \qquad s'_u = \mathsf{PRF}(2, tk_u)$$
It selects a random key f_l for each client $\mathbf{c}_l$ excluding itself, i.e., $f_l \xleftarrow{\$} \{0,1\}^{poly(\lambda)}$. It sends f_l to each $\mathbf{c}_l$.
    (ii) *Generating temporary blinding factors*: derives $\bar{t}$ pseudorandom values from s'_u: $\forall i, 1 \leq i \leq \bar{t}: \ w'_{i,u} = \mathsf{PRF}(i, s'_u)$
    (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 $\boldsymbol{\gamma}_u(x)$:
$$\forall i, 1 \leq i \leq \bar{t}: \qquad \gamma_{i,u} = \boldsymbol{\gamma}_u(x_i) \bmod p$$
It encrypts each $\gamma_{i,u}$ using blinding factor $w'_{i,u}$:
$$\forall i, 1 \leq i \leq \bar{t}: \qquad \gamma'_{i,u} = \gamma_{i,u} \cdot w'_{i,u} \bmod p$$
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 in $\mathcal{I}$. It regenerates its original blinding factors:
$$\forall i, 1 \leq i \leq \bar{t}: \quad z_{i,u} = \mathsf{PRF}(i, k_u), \quad w_{i,u} = \mathsf{PRF}(i, s_u)$$
where k_u and s_u are in $\vec{prm}_u$ and were generated in step 3(b)iii. It also generates new ones:
$$\forall i, 1 \leq i \leq \bar{t}: \qquad z'_{i,u} = \mathsf{PRF}(i, k'_u)$$
It sets values $v_{i,u}$ and $y_{i,u}$ as follows. $\forall i, 1 \leq i \leq \bar{t}$:
$$v_{i,u} = \gamma'_{i,u} \cdot \prod_{\forall \mathbf{c}_l \in \mathcal{I} \setminus \mathbf{c}_u} \gamma'_{i,l} \bmod p$$
$$y_{i,u} = - \sum_{\forall \mathbf{c}_l \in C \setminus \mathbf{c}_u} \mathsf{PRF}(i, f_l) + \sum_{\forall \mathbf{c}_l \in \mathcal{I} \setminus \mathbf{c}_u} \mathsf{PRF}(i, \bar{f}_l) \text{mod}$$
where $\mathbf{c}_u \in \mathcal{I}$.

(v) *Re-encoding outsourced puzzle*: obliviously prepares the puzzle (held by $\mathbf{s}$) for the computation. To do that, it participates in an instance of OLE^+ with $\mathbf{s}$, for every i, where $1 \leq i \leq \bar{t}$. The inputs of $\mathbf{c}_u$ to the i-th instance of 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 $\mathbf{s}$ to the i-th instance of OLE^+ is $\mathbf{c}_u$'s encrypted y-coordinate: $e''_i = o_{i,u}$ (where $o_{i,u} \in \vec{o}$). Accordingly, the i-th instance of OLE^+ returns to $\mathbf{s}$:

$$d_{i,u} = e_i \cdot e''_i + e'_i = q_u \cdot v_{i,u} \cdot \pi_{i,u} + z'_{i,u} + y_{i,u} \bmod p$$
$$= q_u \cdot \gamma_{i,u} \cdot w'_{i,u} \cdot \Big(\prod_{\forall \mathbf{c}_l \in \mathcal{I} \setminus \mathbf{c}_u} \gamma_{i,l} \cdot w'_{i,l} \Big) \cdot \pi_{i,u} + z'_{i,u} + y_{i,u} \bmod p$$

where q_u is the party's coefficient. If $\mathbf{c}_u$ detects misbehavior during OLE^+ execution, it sends $\perp$ to all parties and halts.

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

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

(c) *Granting the computation by each non-leader client*: each non-leader client $\mathbf{c}_u$ takes the following steps.

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

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

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

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

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

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

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

$$d_{i,u} = e_i \cdot e''_i + e'_i = q_u \cdot v_{i,u} \cdot \pi_{i,u} + y_{i,u} \bmod p$$
$$= q_u \cdot \Big(\prod_{\forall \mathbf{c}_l \in \mathcal{I} \setminus \mathbf{c}_u} \gamma_{i,l} \cdot w'_{i,l} \Big) \cdot \pi_{i,u} + y_{i,u} \bmod p$$

where q_u is the $\mathbf{c}_u$'s coefficient. If $\mathbf{c}_u$ detects misbehavior during OLE^+ execution, it sends $\perp$ to all parties and halts.

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

$$g_i = \sum_{\forall \mathbf{c}_u \in C} d_{i,u} \bmod p$$
$$= \Big(\underbrace{\prod_{\forall \mathbf{c}_u \in \mathcal{I}} \gamma_{i,u} \cdot w'_{i,u}}_{v_{i,u}} \cdot \sum_{\forall \mathbf{c}_u \in C} q_u \cdot \pi_{i,u} \Big) + \sum_{\forall \mathbf{c}_u \in \mathcal{I}} z'_{i,u} \bmod p$$

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

(5) **Solving a Puzzle**. $\mathsf{Solve}(\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, cmd) \rightarrow (m, \zeta)$
Server $\mathbf{s}$ 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 $\mathbf{c}_u \in \mathcal{I}$:

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

(ii) derives 2 keys from tk_u: $k'_u = \mathsf{PRF}(1, tk_u), s'_u = \mathsf{PRF}(2, tk_u)$

(b) *Removing blinding factors*: removes the blinding factors from $[g_1, \ldots, g_{\bar{t}}] \in \vec{g}$ as follows. $\forall i, 1 \leq i \leq \bar{t}$:

$$\theta_i = \Big(\prod_{\forall \mathbf{c}_u \in \mathcal{I}} \underbrace{\mathsf{PRF}(i, s'_u)}_{w'_{i,u}} \Big)^{-1} \cdot \Big(g_i - \sum_{\forall \mathbf{c}_u \in \mathcal{I}} \overbrace{\mathsf{PRF}(i, k'_u)}^{z'_{i,u}} \Big) \bmod p$$
$$= \Big(\prod_{\forall \mathbf{c}_u \in \mathcal{I}} \gamma_{i,u} \Big) \cdot \sum_{\forall \mathbf{c}_u \in C} q_u \cdot \pi_{i,u} \bmod p$$

(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: $\boldsymbol{\theta}(x) = \prod_{\forall \mathbf{c}_u \in \mathcal{I}} (x - root_u) \cdot \sum_{\forall \mathbf{c}_u \in C} q_u \cdot (x + m_u) \bmod p$
We can rewrite $\boldsymbol{\theta}(x)$ as follows:

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

where $\boldsymbol{\psi}(x)$'s degree is $\bar{t} + 1$ and its constant term is 0.

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

$$res = cons \cdot \Big(\prod_{\forall \mathbf{c}_u \in \mathcal{I}} (-root_u) \Big)^{-1} \bmod p$$
$$= \sum_{\forall \mathbf{c}_u \in C} q_u \cdot m_u$$

(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 $\mathsf{Ver}(com'_u, (root_u, tk_u)) = 1$. Note that $\mathbf{s}$ performs the check for every $\mathbf{c}_u$ in $\mathcal{I}$.

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

Case 2 when solving a puzzle of single client $\mathbf{c}_u$, i.e., when cmd = clientPzl:

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

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

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

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

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 $\boldsymbol{\pi}_u(x)$, 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**. $\mathsf{Verify}(m, \zeta, ., pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, cmd) \rightarrow \ddot{v} \in \{0, 1\}$
A verifier (anyone, not just $\mathbf{c}_u \in C$) 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 $\mathbf{s}$ in Case 1, step 5f:

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

If 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 has all the roots in ζ, as follows.

(i) derives 2 keys from tk_u: $k'_u = \mathsf{PRF}(1, tk_u)$, $s'_u = \mathsf{PRF}(2, tk_u)$

(ii) removes the blinding factors from $[g_1, \ldots, g_{\bar{t}}] \in \vec{g}$ that were provided by **s** in step 4e. $\forall i, 1 \leq i \leq \bar{t}$:

$$\theta_i = \big(\prod_{\forall \mathbf{c}_u \in \mathcal{I}} \mathsf{PRF}(i, s'_u)\big)^{-1} \cdot \big(g_i - \sum_{\forall \mathbf{c}_u \in \mathcal{I}} \mathsf{PRF}(i, k'_u)\big) \bmod p$$

$$= \prod_{\forall \mathbf{c}_u \in \mathcal{I}} \gamma_{i,u} \cdot \sum_{\forall \mathbf{c}_u \in C} q_u \cdot \pi_{i,u} \bmod p$$

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

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

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

where $\boldsymbol{\psi}(x)$'s degree is $\ddot{t} + 1$ with constant term being 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 checking $\boldsymbol{\theta}(root_u) \stackrel{?}{=} 0$. Otherwise, it returns $\ddot{v} = 0$ and stops.

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

$$res' = cons \cdot \big(\prod_{\forall \mathbf{c}_u \in \mathcal{I}} (-root_u)\big)^{-1} \bmod p$$

$$= \sum_{\forall \mathbf{c}_u \in C} q_u \cdot m_u$$

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

(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 a single puzzle belonging to $\mathbf{c}_u$, i.e., when cmd = clientPzl:

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

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

(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 5.1. *If the sequential modular squaring assumption holds, factoring N is a hard problem,* PRF, OLE^+, *and the commitment schemes are secure, then the protocol presented above is a secure* $\mathcal{VHLC}$-$\mathcal{TLP}$, *w.r.t. Definition 4.4.*

Appendix J presents the proof of Theorem 5.1. Also, Appendix K explains (i) the rationale for using OLE^+, (ii) the justification for setting X's size to $\bar{t} = \ddot{t}+2$, and (iii) an interesting aspect of Tempora-Fusion, which provides *flexibility in time-locking messages.*

6 MULTI-INSTANCE TLP WITH VERIFIABLE HOMOMORPHIC LINEAR COMBINATION

Now, we present Multi-Instance Tempora-Fusion, a (single-client) multi-instance TLP supporting a verifiable homomorphic linear combination. To formalize the problem, we let client **c** have a vector of messages: $\vec{m} = [m_1, \ldots, m_z]$. It wants server **s** to learn each m_i at time $time_i \in \vec{time}$, where $\vec{time} = [time_1, \ldots, time_z]$, $\bar{\Delta}_j = time_j - time_{j-1}$, $\vec{\bar{\Delta}} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, $T_j = max_{ss} \cdot \bar{\Delta}_j$ and $1 \leq j \leq z$. In this setting, **c** will be online rarely; for instance, at an earlier times $time_0$ and $time'_0$, where $time_0, time'_0 < time_1$. Client **c** wants also **s** to learn the linear combination of the messages: $\sum_{j=1}^{z} q_j \cdot m_j$, where q_j is a coefficient for m_j. Server **s** must learn the computation result at a certain time before any puzzle is solved. Client **c** may return online when it wants to grant computation on its puzzles. After that, **s** will learn the result after time Δ, where $\Delta < \bar{\Delta}_1$. Appendix L, briefly discusses a formal definition of Multi-Instance Tempora-Fusion.

6.1 Multi-Instance Tempora-Fusion

We build Multi-Instance Tempora-Fusion upon Tempora-Fusion. We utilize the chaining technique (from [1]), to chain different puzzles, such that when **s** solves one puzzle it will obtain enough information to work on the next puzzle. However, we develop a new technique for chaining puzzles to support homomorphic linear combinations.

In the TLP proposed in [1], during the puzzle generation, the base r_{j+1} for the $(j + 1)$-th puzzle is concatenated with the j-th solution, m_j. Hence, the j-th puzzle is created on solution $m_j||r_{j+1}$. However, this method fails to support homomorphic operations on the puzzle's actual solution m_j, as the solution becomes $m_j||r_{j+1}$.

To address this issue, we take a different approach. In short, we require the client to derive the next puzzle's base from the current puzzle's master key. Recall that in Tempora-Fusion, each j-th puzzle is associated with a master key mk_j, found when the puzzle is solved. Using our new technique, during the puzzle generation, when $j = 1$, client **c** picks a random base r_j, sets $a_j = 2^{T_j} \bmod \phi(N)$, and then sets master key mk_j as $mk_j = r_j^{a_j} \bmod N$. Next, as in Tempora-Fusion, **c** derives some pseudorandom values from mk_j and encrypts the y-coordinates of a polynomial representing m_j.

Nevertheless, when $j > 1$, client **c** derives a base r_j from the *previous master key* as $r_j = \mathsf{PRF}(j||0, mk_{j-1})$. As before, it sets $a_j = 2^{T_j} \bmod \phi(N)$ and sets the current master key mk_j as $mk_j = r_j^{a_j} \bmod N$. It derives some pseudorandom values from mk_j, and encrypts the y-coordinates of a polynomial representing m_j. It repeats this process until it creates a puzzle for the last solution m_z. The client hides all bases except the first one r_1 which is made public.

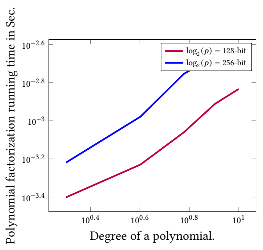
This new approach does not require **c** to modify each solution m_j. Thus, we can now use the techniques we developed in Tempora-Fusion to allow **s** to perform homomorphic linear combinations on the puzzles. In this setting, **s** solves the puzzles in ascending order, starting from the first puzzle, to find a solution and a base for the next puzzle. Once provided with this base, it initiates repeated modular squaring to identify the next solution and base, continuing until it solves the final puzzle. Appendix M explains Multi-Instance Tempora-Fusion in detail. Also, Appendix O outlines how Multi-Instance Tempora-Fusion can be combined with Tempora-Fusion to support multiple clients.

7 COST ANALYSIS

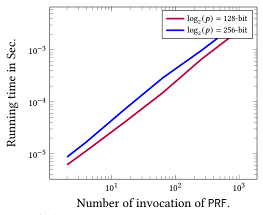
This section examines our schemes' asymptotic and concrete costs.

7.1 Asymptotic Cost

Client's Cost. The client's computation complexity in Tempora- Fusion is $O(\bar{t})$, involving $O(1)$ modular exponentiations, $O(\bar{t})$ PRF invocations, $O(\bar{t})$ modular multiplications and additions, $O(\bar{t})$ OLE^+



(a) Run-time of factorizing.



(b) Run-time of PRF.

Figure 1: Performance of polynomial factorizations and PRF.

Table 1: Complexities of Tempora-Fusion and Multi-Instance Tempora-Fusion. Case 1 refers to computing and solving puzzles for the linear combination while Case 2 refers to solving a client's puzzle.

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	Case 1	$O(\ddot{t}^2 + \bar{t} \cdot n + \ddot{t} \cdot Y)$	$O(\bar{t} \cdot n)$
		Case 2	$O(\ddot{t} + \bar{t} + T_u)$	
Multi-Instace Tempora-Fusion	Client		$O(z)$	$O(z)$
	Verifier		$O(z)$	–
	Server	Case 1	$O(z + Y)$	$O(z)$
		Case 2	$O(max_{ss} \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$	

invocations, and $O(1)$ hash function invocations. A client's communication complexity is $O(\bar{t} \cdot n)$. The client's computation complexity in Multi-Instance Tempora-Fusion is $O(z)$, involving $O(1)$ modular exponentiations, $O(1)$ has function invocations, $O(z)$ PRF invocations, and $O(z)$ modular additions and multiplications. A client's communication complexity is $O(z)$. The size of most messages transmitted by the client in our schemes is 128 bits.

Verifier' Cost. A verifier's computation cost in Tempora-Fusion is $O(\ddot{t}^2 + \ddot{t})$, involving $O(\ddot{t})$ hash function invocations, $O(\bar{t} \cdot \ddot{t})$ PRF invocations, and $O(\ddot{t}^2 + \ddot{t})$ modular additions and multiplications.

Its computation complexity in Multi-Instance Tempora-Fusion is $O(z)$ that includes $O(z)$ hash function invocations, $O(1)$ PRF invocations, and $O(1)$ modular additions and multiplications.

Server's Cost. The computation cost of **s** in Tempora-Fusion, when finding a solution for the linear combination, is $O(\ddot{t}^2 + \bar{t} \cdot n + \ddot{t} \cdot Y)$, that involves $O(\bar{t} \cdot n)$ invocations of OLE^+, $O(\ddot{t})$ PRF invocations, $O(\ddot{t} \cdot Y)$ modular squarings, a polynomial factorization costing $O(\ddot{t}^2)$, $O(\ddot{t})$ hash function invocations, $O(\bar{t} \cdot \ddot{t} + \bar{t} \cdot n)$ additions and $O(\bar{t} \cdot \ddot{t})$ multiplications. The computation complexity of **s**, when finding a solution for a client's puzzle is $O(\ddot{t} + \bar{t} + T_u)$, involving $O(T_u)$ modular squarings, $O(\bar{t})$ invocations of PRF, as well as $O(\bar{t})$ modular additions and multiplications. Note that in all schemes relying on modular squarings a server performs $O(T_u)$ squaring. The communication complexity of **s** in Tempora-Fusion is $O(\bar{t} \cdot n)$.

Its computation cost in Multi-Instance Tempora-Fusion, when finding the linear combination's solution, is $O(z+Y)$ involving $O(Y)$ modular exponentiations, $O(z)$ invocations of OLE^+, $O(z)$ additions, $O(1)$ PRF invocations, $O(1)$ hash function invocations, and $O(1)$ modular multiplications. **s**'s computation cost in Multi-Instance Tempora-Fusion, when finding a solution for a client's puzzle is $O(max_{ss} \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$ involving $O(max_{ss} \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$ modular exponentiations, $O(z)$ PRF invocations, $O(z)$ additions, and $O(z)$ multiplications. **s**'s communication complexity in this scheme is $O(z)$.

Figure 1 summarizes the two schemes' complexities. Appendix P presents a more detailed asymptotic cost analysis.

7.2 Concrete Cost

Having addressed the concrete communication costs of the schemes in the previous section, we now shift our focus to the concrete computation costs. The three primary added operations that impose costs to the participants of our schemes (compared to existing schemes such as those in [1, 34, 35, 35, 40, 41]) are polynomial factorization, invocations of PRF, and OLE^+ execution. In this section, we analyze their concrete costs.

7.2.1 Implementation Environment. To evaluate the performance of polynomial factorization and PRF, we have developed prototype implementations written in C++. They can be found in [3, 4]. We use the NTL library [44] for polynomial factorization, the GMP library [29] for modular multiple precision arithmetic, and the CryptoPP library [19] 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. We run the experiments for at least 100 times to estimate running times.

7.2.2 Choice of Parameters. Since the performance of polynomial factorization and PRF are influenced by the field size $\log_2(p)$ over which polynomials are defined and the output size (also referred to as $\log_2(p)$) respectively, we use different field sizes: 128 and 256 bits. Also, in Tempora-Fusion, since increasing the total number $\ddot{t}$ of leaders 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's 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. Conversely, in Multi-Instance Tempora-Fusion, the degree of the resulting polynomial is always 2.

7.2.3 Result. Increasing the polynomial's degree from 2 to 10 results in the following changes in the running time of factorization: (i) from 0.38 to 1.46 milliseconds (ms) when the field size is 128 bits, and (ii) from 0.6 to 2.24 ms when the field size is 256 bits. Figure 1a, summarizes the performance of polynomial factorizations.

As we increase the number of PRF invocations from 2 to 1024, the running time (a) grows from 0.0061 to 2.42 ms when the output size is 128 bits and (b) increases from 0.0086 to 3.53 ms when the output size is 256 bits. Figure 1b, outlines the performance of PRF.

The running time of OLE^+ is low too. For instance, Boyle *et al.* CCS'18 [14], 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. They 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}. Also, Schoppmann *et al.* CCS'19 [42] proposed a variant of vector OLE, secure against semi-honest adversaries. This variant with the input vectors of 2^{14} elements can be run in less than 1 seconds.

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.

REFERENCES

[1] Aydin Abadi and Aggelos Kiayias. 2021. Multi-instance Publicly Verifiable Time-lock Puzzle and its Applications. In FC.
[2] Alfred V. Aho and John E. Hopcroft. 1974. The Design and Analysis of Computer Algorithms (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[3] Anonymous. 2024. Source Code for Polynomial Factorizations. https://github.com/anonymous2012000/code/blob/main/test.cpp.
[4] Anonymous. 2024. Source Code for Pseudorandom Invocations. https://github.com/anonymous2012000/code-2/blob/main/main.cpp.
[5] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In CCS'13.
[6] Barclays Bank. 2024. Daily payment limits in Online Banking. https://www.barclays.co.uk/help/payments/payment-information/online-banking-limits/.
[7] Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. 2021. TARDIS: A Foundation of Time-Lock Puzzles in UC. In EUROCRYPT.
[8] Manuel Blum. 1982. 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. IEEE Computer Society, 133–137.
[9] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. 1991. Noninteractive Zero-Knowledge. SIAM J. Comput. 20, 6 (1991).
[10] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In CCS.
[11] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable Delay Functions. In CRYPTO'18.
[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2018. A Survey of Two Verifiable Delay Functions. IACR Cryptol. ePrint Arch. (2018).
[13] Dan Boneh and Moni Naor. 2000. Timed Commitments. In ACRYPTO. Springer.
[14] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. 2018. Compressing Vector OLE. In CCS.
[15] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. 2019. Leveraging Linear Decryption: Rate-1 Fully-Homomorphic Encryption and Time-Lock Puzzles. In TCC'19,.
[16] Johannes Buchmann and Hugh C. Williams. 1988. A Key-Exchange System Based on Imaginary Quadratic Fields. J. Cryptol. (1988).
[17] Hsing-Chung Chen and Rini Deviani. 2012. A Secure E-Voting System Based on RSA Time-Lock Puzzle Mechanism. In BWCCA'12.
[18] Peter Chvojka, Tibor Jager, Daniel Slamanig, and Christoph Striecks. 2021. Versatile and Sustainable Timed-Release Encryption and Sequential Time-Lock Puzzles (Extended Abstract). In ESORICS 2021. Springer.
[19] Wei Dai. 2015. CryptoPP Library Multiple Precision Arithmetic Library. https://cryptopp.com.
[20] Department of Justice–U.S. Attorney's Office. 2018. Former JP Morgan Chase Bank Employee Sentenced to Four Years in Prison for Selling Customer Account Information. https://www.justice.gov/usao-edny/pr/former-jp-morgan-chase-bank-employee-sentenced-four-years-prison-selling-customer.
[21] Changyu Dong, Aydin Abadi, and Sotirios Terzis. 2016. VD-PSI: Verifiable Delegated Private Set Intersection on Outsourced Private Datasets. In FC.
[22] William S. Dorn. 1962. Generalizations of Horner's Rule for Polynomial Evaluation. IBM Journal of Research and Development (1962).
[23] Jesko Dujmovic, Rachit Garg, and Giulio Malavolta. 2024. Time-Lock Puzzles with Efficient Batch Solving. In EUROCRYPT. Springer-Verlag.
[24] Cynthia Dwork and Moni Naor. 2000. Zaps and their applications. In FoCS.
[25] Uriel Feige, Dror Lapidot, and Adi Shamir. 1990. 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.
[26] Juan A. Garay and Markus Jakobsson. 2002. Timed Release of Standard Digital Signatures. In FC'02, Matt Blaze (Ed.).
[27] Satrajit Ghosh, Jesper Buus Nielsen, and Tobias Nilges. 2007. Maliciously Secure Oblivious Linear Function Evaluation with Constant Overhead. In ASIACRYPT.
[28] Satrajit Ghosh and Tobias Nilges. 2019. An Algebraic Approach to Maliciously Secure Private Set Intersection. In EUROCRYPT.
[29] GNU Project. 1991. The GNU Multiple Precision Arithmetic Library. https://gmplib.org.
[30] Jonathan Katz and Yehuda Lindell. 2007. Introduction to Modern Cryptography. Chapman and Hall/CRC Press.
[31] Jonathan Katz, Julian Loss, and Jiayu Xu. 2020. 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).
[32] Lea Kissner and Dawn Xiaodong Song. 2005. Privacy-Preserving Set Operations. In CRYPTO 2005, 25th International Cryptology Conference. 241–257.
[33] David Leigh, James Ball, Juliette Garside, and David Pegg. 2015. HSBC files timeline: From Swiss bank leak to fallout. The Guardian 12 (2015).
[34] Yi Liu, Qi Wang, and Siu-Ming Yiu. 2022. Towards practical homomorphic time-lock puzzles: Applicability and verifiability. In ESORICS.
[35] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. 2019. Homomorphic Time-Lock Puzzles and Applications. In CRYPTO'19.
[36] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. 2016. Federated Learning of Deep Networks using Model Averaging. CoRR abs/1602.05629 (2016).
[37] Torben P. Pedersen. 1991. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In CRYPTO.
[38] Krzysztof Pietrzak. 2019. 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.
[39] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. Journal of the society for industrial and applied mathematics (1960).
[40] Dan Ristea, Aydin Abadi, and Steven J Murdoch. 2023. Delegated Time-Lock Puzzle. arXiv preprint arXiv:2308.01280 (2023).
[41] R. L. Rivest, A. Shamir, and D. A. Wagner. 1996. Time-lock Puzzles and Timed-release Crypto. Technical Report.
[42] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. 2019. Distributed Vector-OLE: Improved Constructions and Implementation. In CCS.
[43] Adi Shamir. 1979. How to Share a Secret. Commun. ACM (1979).
[44] Victor Shoup. 1996. NTL: A Library for doing Number Theory. https://libntl.org.
[45] Shravan Srinivasan, Julian Loss, Giulio Malavolta, Kartik Nayak, Charalampos Papamanthou, and Sri Aravinda Krishnan Thyagarajan. 2023. Transparent Batchable Time-lock Puzzles and Applications to Byzantine Consensus. In PKC.
[46] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Giulio Malavolta, Nico Döttling, Aniket Kate, and Dominique Schröder. 2020. Verifiable Timed Signatures Made Practical. In CCS.
[47] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. 2022. Universal Atomic Swaps: Secure Exchange of Coins Across All Blockchains. In IEEE Symposium on Security and Privacy, SP.
[48] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmidt, and Dominique Schröder. 2020. PayMo: Payment Channels For Monero. IACR Cryptol. ePrint Arch. (2020).
[49] Jun Wan, Hanshen Xiao, Srinivas Devadas, and Elaine Shi. 2020. Round-Efficient Byzantine Broadcast Under Strongly Adaptive and Majority Corruptions. In TCC (Lecture Notes in Computer Science).
[50] Benjamin Wesolowski. 2019. Efficient Verifiable Delay Functions. In EUROCRYPT'19, Yuval Ishai and Vincent Rijmen (Eds.).
[51] WorldRemit Content Team. 2023. Bank transfer limits UK: a guide to transferring large amounts of money. https://www.worldremit.com/en/blog/finance/bank-transfer-limit-uk.
[52] WorldRemit Content Team. 2024. Bank of America Online Banking Service Agreement. https://www.bankofamerica.com/online-banking/service-agreement.go.
[53] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. ACM Trans. Intell. Syst. Technol. (2019).

A 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 [11, 12, 38, 50]. 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 [11]. 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 [50] and Pietrzak [38] 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. But, the converse is not necessarily the case. Because, VDFs are not suitable to encode an arbitrary private message and they take a public message as input, whereas TLPs have been designed to conceal a private input message.

B ENHANCED OLE'S IDEAL FUNCTIONALITY AND PROTOCOL

The enhanced 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 OLE's protocol (denoted by OLE^{+}) is presented in Figure 2.

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

Figure 2: Enhanced Oblivious Linear function Evaluation (OLE^{+}) proposed in [28].

C 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 algorithm $\text{Com}(m, r) = com$, 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 $\text{Ver}(com, \hat{m})$. 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 $\text{Ver}(com, \hat{m}) = \text{Ver}(com, \hat{m}') = 1$ while $\hat{m} \neq \hat{m}'$.

Efficient commitment schemes are available in both (a) the standard model, exemplified by the Pedersen scheme [37], and (b) the random oracle model using a hash function Q. In this work, we use the latter for its efficiency. This scheme involves computing $\text{Q}(m||r) = com$ during the commitment. The verification step requires confirming whether $\text{Q}(m||r) \overset{?}{=} com$. Here $\text{Q} : \{0, 1\}^{*} \rightarrow \{0, 1\}^{poly(\lambda)}$ is a collision-resistant hash function, meaning that the probability of finding two distinct values m and m' such that $\text{Q}(m) = \text{Q}(m')$ is negligible regarding the security parameter λ.

D BASIC SECURITY DEFINITION OF TLP AND ITS FIRST REALIZATION

Definition D.1. 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:

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

D.1 The Original RSA-Based TLP

Below, we restate the original RSA-based time-lock puzzle proposed in [41] to realize the TLP's security definition.

(1) Setup: $\text{Setup}_{\text{TLP}}(1^{\lambda}, \Delta, max_{ss})$.
  (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., $\text{SKE.keyGen}(1^{\lambda}) \rightarrow k$.
  (d) choose a uniformly random value r, i.e., $r \overset{\$}{\leftarrow} \mathbb{Z}_N^{*}$.
  (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: $\text{GenPuzzle}_{\text{TLP}}(m, pk, sk)$.
  (a) encrypt the message under key k using the symmetric-key encryption, as follows: $o_1 = \text{SKE.Enc}(k, m)$.
  (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: $\text{Solve}_{\text{TLP}}(pk, o)$.
  (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., $m = \text{SKE.Dec}(k, o_1)$. 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 [1] for the proof.

Theorem D.2. *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.*

E SEQUENTIAL AND ITERATED FUNCTIONS

Definition E.1 ($\Delta, \delta(\Delta)$)-Sequential function). For a function: $\delta(\Delta)$, time parameter: Δ and security parameter: $\lambda = O(\log(|X|))$, $f : X \rightarrow Y$ is a $(\Delta, \delta(\Delta))$-sequential function if the following conditions hold:

- There is an algorithm that for all $x \in X$evaluates f in parallel time Δ, by using $poly(\log(\Delta), \lambda)$ processors.

- For all adversaries $\mathcal{A}$ which execute in parallel time strictly less than $\delta(\Delta)$ with $poly(\Delta, \lambda)$ processors:

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

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

Definition E.2 (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 $f(k, x) = \beta^{(k)}(x) = \overbrace{\beta \circ \beta \circ \ldots \circ \beta}^{k \text{ Times}}$ is an iterated sequential function, with round function β, if for all $k = 2^{o(\lambda)}$ the function $h : X \to X$ 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:

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

F FURTHER DISCUSSION ON PRIVACY EXPERIMENT

In this section, we provide full detail about $\mathsf{Exp}_{\mathrm{prv}}^{\mathcal{A}}(1^{\lambda}, n, \ddot{t}, t)$, presented in Section 4.2. The experiment involves challenger $\mathcal{E}$ which plays honest parties' roles and a pair of adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$.

It considers an adversary that has access to two oracles, puzzle generation O.GenPuzzle() and evaluation O.Evaluate(). It can adaptively corrupt a subset $\mathcal{W}$ of the parties involved P, i.e., $\mathcal{W} \subset P = \{\mathbf{s}, \mathbf{c}_1, \ldots, \mathbf{c}_n\}$, and retrieve secret keys of corrupt parties (as shown in lines 9–13).

Given the set of corrupt parties, their secret keys, and having access to O.GenPuzzle() and O.Evaluate(), $\mathcal{A}_1$ outputs a pair of messages $(m_{0,u}, m_{1,u})$ for each client $\mathbf{c}_u$ (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 15–17).

Given all the puzzles, related parameters, and access to the two oracles, $\mathcal{A}_1$ outputs a state (line 18). With this state as input, $\mathcal{A}_2$ guesses the value of bit b_u for its chosen client (line 19). The adversary wins the game (i.e., the experiment outputs 1) if its guess is correct for a non-corrupt client (line 20). 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 21) on the number t of certain parties the adversary corrupts when executing Evaluate(). This constraint is defined such that $\mathcal{E}$ selects a (possibly small-sized) set $\mathcal{I}$ of parties (lines 6–8), where $|\mathcal{I}| = \ddot{t}$. 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 Evaluate() (line 22). During the execution of Evaluate(), $\mathcal{A}_1$ has access to the private keys of corrupt parties (as stated in lines 23 and 24). 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 O.Reveal (); given this transcript, $\mathcal{A}_1$ outputs a state (line 25).

Having access to this state and the output of Evaluate(), adversary $\mathcal{A}_2$ guesses the value of bit b_u for its chosen client (line 26). The adversary wins the game, if its guess is correct for a non-corrupt client (line 27).

G FURTHER DISCUSSION ON SOLUTION VALIDITY EXPERIMENT

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 $\mathsf{Exp}_{\mathrm{val}}^{\mathcal{A}}(1^{\lambda}, n, \ddot{t}, t)$, 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 9–11).

Given the corrupt parties, their secret keys, and having access to oracles O.GenPuzzle() and O.Evaluate(), $\mathcal{A}$ outputs a message m_u for each client $\mathbf{c}_u$ (line 12). The experiment proceeds by requiring $\mathcal{E}$ to generate a puzzle for each message that $\mathcal{A}$ selected (lines 13 and 14).

The experiment returns 0 and halts, if $\mathcal{A}$ corrupts more than t parties in $\mathcal{I}$ (line 15). Otherwise, it allows the corrupt parties and $\mathcal{E}$ interactively execute Evaluate() (line 16). During the execution of Evaluate(), $\mathcal{A}$ has access to the private keys of corrupt parties (as stated in lines 17 and 18). Given the output of Evaluate() which is a puzzle, $\mathcal{E}$ solves the puzzle and outputs the solution (line 19).

The experiment enables $\mathcal{A}$ to learn about the messages exchanged between honest and corrupt parties during the execution of Evaluate(), by providing $\mathcal{A}$ with access to an oracle called O.Reveal (); given this transcript, the output of Evaluate(), and the plaintext solution, $\mathcal{A}$ outputs a solution and proof (line 20). $\mathcal{E}$ 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 21).

$\mathcal{E}$ solves every client's puzzle (lines 22 and 23). 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 24). 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 25).

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

```
$\text{S.Setup}(1^{\lambda}, \ddot{t}, t) \rightarrow K_{\mathbf{s}} := (sk_{\mathbf{s}}, pk_{\mathbf{s}})$
For $u = 1, \dots, n$ do :
    $\text{C.Setup}(1^{\lambda}) \rightarrow K_u := (sk_u, pk_u)$
$state \leftarrow \{pk_{\mathbf{s}}, pk_1, \dots, pk_n\}, \mathcal{W} \leftarrow \emptyset, \mathcal{I} \leftarrow \emptyset$
$K \leftarrow \emptyset, cont \leftarrow True, counter \leftarrow 0, \vec{b} \leftarrow \mathbf{0}$
For $1, \dots, \ddot{t}$ do :
    Select $a$ from $\{1, \dots, n\}$
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{\mathcal{B}_a\}$
While $(cont = True)$ do :
    $\mathcal{A}(state, \text{O.GenPuzzle}, \text{O.Evaluate}, K, \mathcal{I}, \Delta_1, \dots, \Delta_n, \Delta,$
     $max_{ss}) \rightarrow (state, cont, \mathcal{B}_j)$
    If $cont = True$, then $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathcal{B}_j\}, \quad K \leftarrow K_{\mathcal{B}_j}$
$\mathcal{A}(state, K, \mathcal{W}, \text{O.GenPuzzle}, \text{O.Evaluate}, \mathcal{W}) \rightarrow \vec{m} = [m_1, \dots, m_n]$
For $u = 1, \dots, n$ do :
    $\text{GenPuzzle}(m_u, K_u, pk_{\mathbf{s}}, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$
If $|\mathcal{W} \cap \mathcal{I}| > t$, then return 0
$\text{Evaluate}(\langle \hat{\mathbf{s}}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_{\mathbf{s}}), \hat{\mathbf{c}}_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_{\mathbf{s}}),$
     $\dots, \hat{\mathbf{c}}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_{\mathbf{s}}) \rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})}), s.t.$
    If $\hat{s} \in \mathcal{W}$, $\hat{s}$ knows $(state, K)$, else $\hat{\mathbf{s}}$ is an honest server, $\hat{\mathbf{s}} = \mathbf{s}$
    If $\hat{\mathbf{c}}_j \in \mathcal{W}$, $\hat{\mathbf{c}}_j$ knows $(state, K)$, else $\hat{\mathbf{c}}_j$ is an honest client, $\hat{\mathbf{c}}_j = \mathbf{c}_j$
$\text{Solve}(., ., \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, \text{evalPzl}) \rightarrow (m, \zeta)$
$\mathcal{A}(state, K, \mathcal{W}, \text{O.Reveal}, m, \zeta, \vec{m}, \vec{o}_1, \dots, \vec{o}_n, \vec{pp}_1, \dots, \vec{pp}_n, \vec{g},$
     $\vec{pp}^{(\text{Evl})}) \rightarrow (m', \zeta')$
If $\text{Verify}(m', \zeta', ., ., \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, \text{evalPzl}) \rightarrow 1, m' \notin \mathcal{L}_{\text{evalPzl}}$, return 1
For $u = 1, \dots, n$ do :
    $\text{Solve}(\vec{o}_u, pp_u, ., ., pk_{\mathbf{s}}, \text{clientPzl}) \rightarrow (\bar{m}_u, \zeta_u)$
$\mathcal{A}(state, K, \mathcal{W}, \text{O.GenPuzzle}, \text{O.Evaluate}, (\bar{m}_1, \zeta_1), \dots, (\bar{m}_u, \zeta_u),$
     $(m, \zeta), \vec{m}, \vec{o}_1, \dots, \vec{o}_n, \vec{pp}_1, \dots, \vec{pp}_n) \rightarrow (m'_u, \zeta'_u, u)$
If $\text{Verify}(m'_u, \zeta'_u, \vec{o}_u, pp_u, ., ., pk_{\mathbf{s}}, \text{clientPzl}) \rightarrow 1, m'_u \notin \mathcal{L}_{\text{clientPzl}}$, return 1
```

H FORMAL DEFINITIONS OF COMPLETENESS, EFFICIENCY, AND COMPACTNESS IN $\mathcal{VHLC}$-$\mathcal{TLP}$

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 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 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 $\Pr\left[\frac{\text{Exp}}{\text{Cond}}\right]$, where Exp is an experiment, and Cond is the set of the corresponding conditions under which the property must hold.

Definition H.1 (Completeness). A $\mathcal{VHLC}$-$\mathcal{TLP}$ is correct if for any security parameter λ, any plaintext input message $m_1, \dots, m_n$ and coefficient $q_1, \dots, q_n$ (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) $\text{Solve}(\vec{o}_u, pp_u, ., ., pk_{\mathbf{s}}, cmd)$, that takes a puzzle $\vec{o}_u$ encoding plaintext solution m_u and its related parameters, always returns m_u:

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

where $pp_u \in prm_u$ and $cmd = \text{clientPzl}$.

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

$$Pr\left[\begin{array}{l} \text{S.Setup}(1^{\lambda}, \ddot{t}, t) \rightarrow K_{\mathbf{s}} \\ \text{For } u = 1, \dots, n \text{ do :} \\ \quad \text{C.Setup}(1^{\lambda}) \rightarrow K_u \\ \quad \text{GenPuzzle}(m_u, K_u, pk_{\mathbf{s}}, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u) \\ \text{Evaluate}(\langle \mathbf{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_{\mathbf{s}}), \mathbf{c}_1(\Delta, max_{ss}, K_1, prm_1, \\ q_1, pk_{\mathbf{s}}), \dots, \mathbf{c}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_{\mathbf{s}}) \rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})}) \\ \hline \text{Solve}(., ., \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, cmd) \rightarrow (\sum\limits_{u=1}^{n} q_u \cdot m_u, .) \end{array}\right] = 1$$

where $\vec{o} = [\vec{o}_1, \dots, \vec{o}_n]$, $\vec{pp} = [pp_1, \dots, pp_n]$, $\vec{pk} = [\vec{pk}_1, \dots, \vec{pk}_n]$, $pk_u \in K_u$, $pk_{\mathbf{s}} \in K_{\mathbf{s}}$, and $cmd = \text{evalPzl}$.

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

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

where $cmd = \text{clientPzl}$.

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

$$Pr\left[\begin{array}{l} \text{S.Setup}(1^{\lambda}, \ddot{t}, t) \rightarrow K_{\mathbf{s}} \\ \text{For } u = 1, \dots, n \text{ do :} \\ \quad \text{C.Setup}(1^{\lambda}) \rightarrow K_u \\ \quad \text{GenPuzzle}(m_u, K_u, pk_{\mathbf{s}}, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u) \\ \text{Evaluate}(\langle \mathbf{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_{\mathbf{s}}), \mathbf{c}_1(\Delta, max_{ss}, K_1, prm_1, \\ q_1, pk_{\mathbf{s}}), \dots, \mathbf{c}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_{\mathbf{s}}) \rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})}) \\ \text{Solve}(., ., \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, cmd) \rightarrow (m, \zeta) \\ \hline \text{Verify}(m, \zeta, ., ., \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, cmd) \rightarrow 1 \end{array}\right] = 1$$

where $cmd = \text{evalPzl}$.

Intuitively, efficiency states that (1) Solve() returns a solution in polynomial time, i.e., polynomial in the time parameter T, (2) GenPuzzle() generates a puzzle faster than solving it, with a running time of at most logarithmic in T, and (3) the running time of Evaluate() is faster than solving any puzzle involved in the evaluation, that should be at most logarithmic in T [23].

Definition H.2 (Efficiency). A $\mathcal{VHLC}$-$\mathcal{TLP}$ is efficient if the following two conditions are satisfied:

(1) The running time of $\mathsf{Solve}(\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, cmd)$ is upper bounded by $T \cdot poly(\lambda)$, where $poly()$ is a fixed polynomial.
(2) The running time of $\mathsf{GenPuzzle}(m_u, K_u, pk_{\mathbf{s}}, \Delta_u, max_{ss})$ is upper bounded by $poly'(\log T, \lambda)$, where $poly'()$ is a fixed polynomial.
(3) The running time of $\mathsf{Evaluate}(\langle \mathbf{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_{\mathbf{s}}), \mathbf{c}_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_{\mathbf{s}}), \ldots, \mathbf{c}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_{\mathbf{s}})\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$ is upper bounded by $poly''\Big(\log T, \lambda, \mathcal{F}^{\text{PLC}}\big((q_1, m_1), \ldots, (q_n, m_n)\big)\Big)$, where $poly''$ is a fixed polynomial and $\mathcal{F}^{\text{PLC}}$ is the functionality that computes a linear combination of messages (as stated in Relation 1) .

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

Definition H.3 (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 $1 \le \ddot{t}, t \le n$ and $\ddot{t} \ge t$), always $\mathsf{Evaluate}()$ outputs a puzzle (representation) whose bit-size is independent of $\mathcal{F}^{\text{PLC}}$'s complexity $O(\mathcal{F}^{\text{PLC}})$:

$$Pr\left[\begin{array}{l}\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow K_{\mathbf{s}}\\ \text{For } u = 1, \ldots, n \text{ do :}\\ \quad \mathsf{C.Setup}(1^\lambda) \rightarrow K_u\\ \quad \mathsf{GenPuzzle}(m_u, K_u, pk_{\mathbf{s}}, \Delta_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)\\ \hline \mathsf{Evaluate}\big(\langle \mathbf{s}(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_{\mathbf{s}}), \mathbf{c}_1(\Delta, max_{ss}, K_1, prm_1,\\ q_1, pk_{\mathbf{s}}), \ldots, \mathbf{c}_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_{\mathbf{s}})\rangle\big) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})\\ \text{s.t.}\\ ||\vec{g}|| = poly\Big(\lambda, ||\mathcal{F}^{\text{PLC}}\big((q_1, m_1), \ldots, (q_n, m_n)\big)||\Big)\end{array}\right] = 1$$

I ADDRESSING THE CHALLENGES

We briefly explain how the four main challenges laid out in Section 5.1.3 are addressed.

- Challenge (1): *each client does not know other clients' solutions*: during the invocation of 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 independently prepared its puzzle*: during the invocation of OLE^+ they switch their old blinding factors to new ones that are consistent with other clients.
- Challenge (3): server $\mathbf{s}$ *may exclude or modify some of the clients' puzzles*: the solutions to Challenges (1) and (2), the support of OLE^+ for verification, and the use of zero-sum blinding factors ensure that such misbehavior will be detected by a verifier. Also, requiring $\mathbf{s}$ to open the commitments for the roots chosen by the leader ensures that $\mathbf{s}$ cannot exclude all parties' inputs, and come up with its choice of inputs encoding arbitrary roots.
- Challenge (4): server $\mathbf{s}$ *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 leaders are blinded and reveal no information about their plaintext messages including the roots.
  Also, each leader 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 $\mathbf{s}$ still needs to solve the puzzle of the honest party to remove its blinding factors from the result.

J PROOF OF THEOREM 5.1

In this section, we prove the security of Tempora-Fusion, i.e., Theorem 5.1.

PROOF. In the proof of Theorem 5.1, we consider a strong adversary that *always corrupts* $\mathbf{s}$ and some clients. Thus, the proof considers the case where corrupt $\mathbf{s}$ 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:
  $$Set_1 = \Big\{max_{ss}, \{N_u, \Delta_u, T_u, r_u, com_u, \vec{o}_u\}_{\forall u, 1 \le u \le n}, \{K_j\}_{\forall \mathcal{B}_j \in \mathcal{W}}\Big\}$$
  where $\mathcal{W}$ is a set of corrupt parties, including server $\mathbf{s}$.
- by the end of the linear combination phase (before any puzzle is fully solved), it also learns:
  $$Set_2 = \Big\{trans_{\mathbf{s}}^{\mathsf{OLE}^+_u}, Y, \{g_1, \ldots, g_{\ddot{t}}\}, \{com'_u, h_u, d_{1,u}, \ldots, d_{\ddot{t},u}\}_{\forall u, 1 \le u \le n},$$
  $$\{\bar{f}_l, \vec{\gamma}'_l\}_{\forall \mathbf{c}_l \in \{\mathcal{W} \cap \mathcal{I}\}}\Big\}$$
  where $trans_{\mathbf{s}}^{\mathsf{OLE}^+_u}$ is a set of messages sent to $\mathbf{s}$ during the execution of OLE^+.

We initially prove that Tempora-Fusion is privacy-preserving, w.r.t. Definition 4.2.

LEMMA J.1. *If the sequential modular squaring assumption holds, factoring N is a hard problem,* PRF *is secure,* 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 4.2.*

PROOF. We will argue that the probability that adversary $\mathcal{A}_2 \in \mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ outputs a correct value of b_u (in the experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{prv}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 4.2) 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 $\mathbf{c}_u$ 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 OLE^+, a set $trans_{\mathbf{s}}^{\mathsf{OLE}^+_u}$ of messages that $\mathcal{A}$ (acting on behalf of corrupt parties) receives during the execution of 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 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 OLE^+. Due to the security of OLE^+, it reveals to $\mathcal{A}$ nothing about the input of each honest client $\mathbf{c}_u$ to OLE^+, even if $\mathbf{s}$ 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 forms:

- $y_{i,u} = -\sum\limits_{\forall \mathbf{c}_l \in C \setminus \mathbf{c}_u} \mathsf{PRF}(i, f_l) + \sum\limits_{\forall \mathbf{c}_l \in \mathcal{I} \setminus \mathbf{c}_u} \mathsf{PRF}(i, \bar{f}_l) \bmod p$, when client $\mathbf{c}_u$ is one of the leaders, i.e., $\mathbf{c}_u \in \mathcal{I}$.
- $y_{i,u} = \sum\limits_{\forall \mathbf{c}_l \in \mathcal{I}} \mathsf{PRF}(i, \bar{f}_l) \bmod p$, when client $\mathbf{c}_u$ is not one of the leaders, i.e., $\mathbf{c}_u \notin \mathcal{I}$.

Due to the security of PRF, given secret keys $\{\bar{f}_l\}_{\forall \mathbf{c}_l \in \{C \cap \mathcal{I}\}}$, it will be infeasible for $\mathcal{A}$ to learn anything about the 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 $\{\bar{f}_l\}_{\forall \mathbf{c}_l \in \{C \cap \mathcal{I}\}}$, $\mathcal{A}$ learns nothing about each honest client $\mathbf{c}_u$ y-coordinate $\pi_{i,u}$ (as well as $z'_{i,u}$ when $\mathbf{c}_u \in \mathcal{I}$) 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 $\{\vec{\gamma}'_l\}_{\forall \mathbf{c}_l \in \{\mathcal{W} \cap \mathcal{I}\}}$ 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 the line 19 of experiment $\mathsf{Exp}_{\text{prv}}^{\mathcal{A}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 4.2) 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 26 of experiment $\mathsf{Exp}_{\text{prv}}^{\mathcal{A}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 4.2) with a probability significantly greater than $\frac{1}{2}$.

This means, $\mathcal{A}$ cannot find a solution $\sum\limits_{\forall \mathbf{c}_u \in C} q_u \cdot m_u$ (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 4.3.

Lemma J.2. *If the sequential modular squaring assumption holds, factoring N is a hard problem, PRF is secure, 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 4.3.*

Proof. We will demonstrate the probability that a PPT adversary $\mathcal{A}$ outputs an invalid solution but passes the verification (in the experiment $\mathsf{Exp}_{\text{val}}^{\mathcal{A}}(1^\lambda, n, \ddot{t}, t)$ defined in Definition 4.3) 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:
$$Set_3 = \Big\{\{root_u, tk_u\}_{\forall \mathbf{c}_u \in \mathcal{I}}, m = res\Big\}$$
- by the end of the puzzle-solving phase for a puzzle of a single honest client $\mathbf{c}_u$, 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 $\{root_u\}_{\forall \mathbf{c}_u \in \mathcal{I}}$ 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 the step 6a of the protocol with a high probability, i.e., $1 - (\mu(\lambda))^{|\mathcal{I}|}$.

As discussed in the proof of Lemma J.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 hidden property of the commitment scheme, the privacy property of OLE^+, security of PRF, and under the assumptions that sequential modular squaring holds and factoring N is a hard problem).

Due to Theorem 3.1 (unforgeable encrypted polynomial with a hidden root), any modification by $\mathcal{A}$ to the inputs $\{o_{1,u}, \ldots, o_{\bar{t},u}\}_{\forall \mathbf{c}_u \notin \mathcal{W}}$ and outputs $\{d_{1,u}, \ldots, d_{\bar{t},u}\}_{\forall \mathbf{c}_u \notin \mathcal{W}}$ of OLE^+, makes the resulting polynomial $\boldsymbol{\theta}$ not contain every root in $\{root_u\}_{\forall \mathbf{c}_u \notin \mathcal{W}}$. 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 $\{root_u\}_{\forall \mathbf{c}_u \notin \mathcal{W}}$, according to Theorem 3.1. 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 $\mathtt{OLE}^+$ (specifically result validity), any misbehaviour of $\mathcal{A}$ (corrupting $\mathbf{s}$) during the execution of $\mathtt{OLE}^+$ will not be detected only with a negligible probability $\mu(\lambda)$, in the 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 $\mathsf{Evaluate}()$ (in the line 21 of experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{val}}(1^{\lambda}\ n, \ddot{t}, t)$ defined in Definition 4.3) with a probability significantly greater than $\mu(\lambda)$.

By solving a single client's puzzle (after invocations of $\mathsf{Evaluate}$), $\mathcal{A}$ will also learn Set_4 for each (honest) client $\mathbf{c}_u$. 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 $\mathbf{c}_u$ in $\{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$ to an invalid message (e.g., m', where $m' \neq m_u$) and pass the verification in the steps 6a and 6b of the protocol is negligible, $\mu(\lambda)$. Note that the elements of set $\{root_u, tk_u\}_{\forall \mathbf{c}_u \in \mathcal{I}} \in Set_3$ have been selected uniformly at random and independent of each solution m_u. Thus, knowing elements $\{root_u, tk_u\}_{\forall \mathbf{c}_u \in \mathcal{I}}$ 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 the line 25 of experiment $\mathsf{Exp}^{\mathcal{A}}_{\text{val}}(1^{\lambda}, n, \ddot{t}, t)$) with a probability significantly greater than $\mu(\lambda)$. □

We have demonstrated that Tempora-Fusion is privacy-preserving (w.r.t. Definition 4.2) and preserves solution validity (w.r.t. Definition 4.3). Hence, Tempora-Fusion is secure, w.r.t. Definition 4.4.

This concludes the proof of Theorem 5.1. □

K REMARKS ON Tempora-Fusion

In this section, we provide additional insights into Tempora-Fusion.

K.1 Use of $\mathtt{OLE}^+$

$\mathtt{OLE}^+$ ensures that the homomorphic operation can be securely operated sequentially multiple times, regardless of the distribution of the input messages to $\mathtt{OLE}^+$. 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 $q_u \cdot \gamma_{i,u} \cdot w'_{i,u} \cdot (\prod\limits_{\forall \mathbf{c}_l \in \mathcal{I} \setminus \mathbf{c}_u} \gamma_{i,l} \cdot w'_{i,l}) \cdot \pi_{i,u} + z'_{i,u} + y_{i,u} \bmod p$, 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 $(w_{i,u})^{-1}$.[3] 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 the one-time pad is used multiple times, yielding leakage.

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

K.2 Size of X

In the Tempora-Fusion 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.

K.3 Tempora-Fusion's Flexibility to Time-lock Messages

An interesting aspect of Tempora-Fusion is its flexible approach to time-locking messages. Each encrypted message $\vec{o}_u$ from a client $\mathbf{c}_u$, which is either published or transmitted to server $\mathbf{s}$, need not be 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 choose to employ time locks on their encrypted messages, while others may opt for straightforward encryption of their solutions.

Nevertheless, the clients can still allow $\mathbf{s}$ 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.

L AN OVERVIEW OF FORMAL DEFINITION

We can slightly simplify and use the formal syntax and definitions of $\mathcal{VHLC}\text{-}\mathcal{TLP}$ (presented in Section 4) to capture the required properties of Multi-Instance Tempora-Fusion. In this setting, client $\mathbf{c}$ can play the role of the rest of the clients each having a solution m_j, in $\mathcal{VHLC}\text{-}\mathcal{TLP}$.

However, to make the security meaningful, we slightly limit the capability of the adversary $\mathcal{A}$ in experiments for privacy and solution-validity in Definitions 4.2 and 4.3. In the sense that, since we have only a single client, we require $\mathcal{A}$ to not corrupt the client. Thus, the secret parameters of the client will not be provided to $\mathcal{A}$ in the experiments. Adversary $\mathcal{A}$ still can corrupt the server.

M DETAILED DESCRIPTION OF MULTI-INSTANCE Tempora-Fusion

Before delving into a thorough explanation of the Multi-Instance Tempora-Fusion protocol, we highlight two key differences between Multi-Instance Tempora-Fusion and Tempora-Fusion.

Firstly, since there is only one client, only one random root is selected and inserted into the outsourced polynomials (rather than inserting $\ddot{t}$ roots in Tempora-Fusion). During the verification, a verifier checks whether the evaluation of the resulting polynomial at this root is zero.

Secondly, in Multi-Instance Tempora-Fusion, using only three x-coordinates $X = \{x_1, x_2, x_3\}$ will suffice, because the outsourced puzzle's degree is one, and when it is multiplied by a polynomial representing a random root (during computing the linear combination), the resulting polynomial's degree will become two. Thus, three (y, x)-coordinates is sufficient to interpolate a polynomial of degree two.

(1) **Setup**. $\mathsf{S.Setup}(1^\lambda, \ddot{t}, t) \rightarrow (., pk_s)$
Server **s** 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.
  (b) *Generating public x-coordinates*: sets $X = \{x_1, x_2, x_3\}$, where $x_i \neq x_j, x_i \neq 0$, and $x_i \notin U$.
  (c) *Publishing public parameters*: publishes $pk_s = (p, X)$. Note that, **c** itself can generate (p, X), too.

(2) **Key Generation**. $\mathsf{C.Setup}(1^\lambda) \rightarrow K$
Client **c** takes the flowing steps.
  (a) *Generating RSA public and private keys*: computes $N = p_1 \cdot p_2$, where p_i is a large randomly chosen prime number, e.g., $\log_2(p_i) \geq 2048$. Next, it computes Euler's totient function of N, as: $\phi(N) = (p_1 - 1) \cdot (p_2 - 1)$.
  (b) *Publishing public parameters*: locally keeps secret key $sk = \phi(N_u)$ and publishes public key $pk = N$.

(3) **Puzzle Generation**. $\mathsf{GenPuzzle}(\vec{m}, K, pk_s, \bar{\Delta}, max_{ss}) \rightarrow (\vec{o}, prm)$
Client **c** takes the following steps to generate z puzzles for messages $\vec{m} = [m_1, \dots, m_z]$ and wants **s** to learn each message m_i at time $time_i \in \vec{time}$, where $\vec{time} = [time_1, \dots, time_z]$, $\bar{\Delta}_j = time_j - time_{j-1}$, $\bar{\Delta} = [\bar{\Delta}_1, \dots, \bar{\Delta}_z]$, and $1 \leq j \leq z$.
  (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 vector of master keys $\vec{mk} = [mk_1, \dots, mk_z]$ and two secret keys k_j and s_j for each master key mk_j in $\vec{mk}$ as follows. It constructs an empty vector $\vec{mk}$. Then,
    (i) sets each exponent a_j.
$$\forall j, 1 \leq j \leq z: \quad a_j = 2^{T_j} \bmod \phi(N)$$
    where $T_j = max_{ss} \cdot \bar{\Delta}_j$ is the total number of squaring needed to decrypt an encrypted solution m_j after previous solution m_{j-1} is revealed.
    (ii) computes each master key mk_j as follows. For every j, where $1 \leq j \leq z$:
      - when $j = 1$:
        (A) picks a uniformly random base $r_j \xleftarrow{\$} \mathbb{Z}_N$.
        (B) sets key mk_j as $mk_j = r_j^{a_j} \bmod N$.
        (C) appends mk_j to $\vec{mk}$.
      - when $j > 1$:
        (A) derives a fresh base r_j from the previous master key as $r_j = \mathsf{PRF}(j||0, mk_{j-1})$.
        (B) sets key mk_j as $mk_j = r_j^{a_j} \bmod N$.
        (C) appends mk_j to $\vec{mk}$.
    (iii) derives two secret keys k_j and s_j from each mk_j.
$$\forall j, 1 \leq j \leq z: \quad k_j = \mathsf{PRF}(1, mk_j), \quad s_j = \mathsf{PRF}(2, mk_j)$$
  (c) *Generating blinding factors*: generates six pseudorandom values, by using k_j and s_j. $\forall j, 1 \leq j \leq z$ and $\forall i, 1 \leq i \leq 3$:
$$z_{i,j} = \mathsf{PRF}(i, k_j), \;\; w_{i,j} = \mathsf{PRF}(i, s_j)$$
  (d) *Encoding plaintext message*:
    (i) represents each plaintext solution m_j as a polynomial, such that the polynomial's constant term is the message.
$$\forall j, 1 \leq j \leq z: \quad \boldsymbol{\pi}_j(x) = x + m_j \bmod p$$
    (ii) computes three y-coordinates of each $\boldsymbol{\pi}_j(x)$.
$$\forall j, 1 \leq j \leq z \quad \text{and} \quad \forall i, 1 \leq i \leq 3: \quad \pi_{i,j} = \boldsymbol{\pi}_j(x_i) \bmod p$$
    where $x_i \in X$.
  (e) *Encrypting the solution*: encrypts the y-coordinates using the blinding factors.
$$\forall j, 1 \leq j \leq z \quad \text{and} \quad \forall i, 1 \leq i \leq 3: \quad o_{i,j} = w_{i,j} \cdot (\pi_{i,j} + z_{i,j}) \bmod p$$
  (f) *Committing to the solution*: commits to each plaintext message:
$$\forall j, 1 \leq j \leq z: \quad com_j = \mathsf{Com}(m_j, mk_j)$$
  Let $\vec{com} = [com_1, \dots, com_z]$.
  (g) *Managing messages*: publishes $\vec{o} = \Big[[o_{1,1}, o_{2,1}, o_{3,1}], \dots, [o_{1,z}, o_{2,z}, o_{3,z}]\Big]$ and $pp = (\vec{com}, r_1)$. It locally keeps secret parameters $sp = \vec{mk}$. It sets $prm = (sp, pp)$. It deletes everything else, including $m_j, \boldsymbol{\pi}_j(x), \pi_{1,j}, \pi_{2,j}$, and $\pi_{3,j}$ for every j, $1 \leq j \leq z$.

(4) **Linear Combination**. $\mathsf{Evaluate}(\langle \mathbf{s}(\vec{o}, \Delta, max_{ss}, pp, pk, pk_s), c(\Delta, max_{ss}, K, prm, q_1, pk_s), \dots, c(\Delta, max_{ss}, K, prm, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$
In this phase, the client produces certain messages that let **s** find a linear combination of its plaintext solutions after time Δ.
  (a) *Granting the Computation*: client **c** takes the following steps.
    (i) *Generating temporary secret keys*: generates a temporary master key tk and two secret keys k' and s'. Moreover, it generates $z - 1$ secret key $[f_j, \dots, f_z]$. To do that, it takes the following steps. It computes the exponent:
$$b = 2^Y \bmod \phi(N)$$
    where $Y = \Delta \cdot max_{ss}$. It selects a base uniformly at random: $h \xleftarrow{\$} \mathbb{Z}_N$ and then sets a temporary master key tk:
$$tk = h^b \bmod N$$
    It derives two keys from tk:
$$k' = \mathsf{PRF}(1, tk), \quad s' = \mathsf{PRF}(2, tk)$$
    It picks fresh $z - 1$ random keys $\vec{f} = [f_2, \dots, f_z]$, where $f_j \xleftarrow{\$} \{0, 1\}^{poly(\lambda)}$.
    (ii) *Generating blinding factors*: regenerates its original blinding factors, for each j-th puzzle. Specifically, for every j, derives two secret keys k_j and s_j from mk_j as follow.
$$\forall j, 1 \leq j \leq z: \quad k_j = \mathsf{PRF}(1, mk_j), \quad s_j = \mathsf{PRF}(2, mk_j)$$
    It regenerates pseudorandom values, by using k_j and s_j. $\forall j, 1 \leq j \leq z$ and $\forall i, 1 \leq i \leq 3$:
$$z_{i,j} = \mathsf{PRF}(i, k_j), \;\; w_{i,j} = \mathsf{PRF}(i, s_j)$$
    It also generates new pseudorandom values using keys (k', s'). $\forall i, 1 \leq i \leq 3: \; z'_i = \mathsf{PRF}(i, k'), \; w'_i = \mathsf{PRF}(i, s')$
    It computes new sets of (zero-sum) blinding factors, using the keys in $\vec{f}$, as follows. $\forall j, 1 \leq j \leq z$:

- if $j = 1$:

$$\forall i, 1 \leq i \leq 3 : \quad y_{i,j} = -\sum_{j=2}^{z} \mathsf{PRF}(i, f_j) \bmod p$$

- if $j > 1$:

$$\forall i, 1 \leq i \leq 3 : \quad y_{i,j} = \mathsf{PRF}(i, f_j) \bmod p$$

where $f_j \in \vec{f}$.

(iii) _Generating y-coordinates of a random root_: picks a random root, $root \stackrel{\$}{\leftarrow} \mathbb{F}_p$. It represents $root$ as a polynomial, such that the polynomial's root is $root$. Specifically, it computes polynomial $\boldsymbol{\gamma}(x)$ as $\boldsymbol{\gamma}(x) = x - root \bmod p$. Then, it computes three y-coordinates of $\boldsymbol{\gamma}(x)$:

$$\forall i, 1 \leq i \leq 3 : \quad \gamma_i = \boldsymbol{\gamma}(x_i) \bmod p$$

where $x_i \in X$.

(iv) _Committing to the root_: computes $com' = \mathtt{Com}(root, tk)$.

(v) _Re-encoding outsourced puzzle_: participates in an instance of $\mathtt{OLE}^+$ with $\mathbf{s}$, for every j-th puzzle and every i, where $1 \leq j \leq z$ and $1 \leq i \leq 3$. The inputs of client $\mathbf{c}$ to i-th instance of $\mathtt{OLE}^+$ are:

$$e_{i,j} = \gamma_i \cdot q_j \cdot w'_i \cdot (w_{i,j})^{-1} \bmod p$$

$$e'_{i,j} = -(\gamma_i \cdot q_j \cdot w'_i \cdot z_{i,j}) + z'_i + y_{i,j} \bmod p$$

The input of $\mathbf{s}$ to (i, j)-th instance of $\mathtt{OLE}^+$ is corresponding encrypted y-coordinate: $e''_{i,j} = o_{i,j}$. Accordingly, (i, j)-th instance of $\mathtt{OLE}^+$ returns to $\mathbf{s}$:

$$\begin{aligned} d_{i,j} &= e_{i,j} \cdot e''_{i,j} + e'_{i,j} \\ &= \gamma_i \cdot q_j \cdot w'_i \cdot \pi_{i,j} + z'_i + y_{i,j} \bmod p \end{aligned}$$

where q_j is a coefficient for j-th solution m_j.

(vi) _Publishing public parameters_: publishes $pp^{(\text{Evl})} = (h, com')$.

(b) _Computing encrypted linear combination_: Server $\mathbf{s}$ sums all of the outputs of $\mathtt{OLE}^+$ instances that it has invoked as follows. $\forall i, 1 \leq i \leq 3$:

$$\begin{aligned} g_i &= \sum_{j=1}^{z} d_{i,j} \bmod p \\ &= (w'_i \cdot \gamma_i \cdot \sum_{j=1}^{z} q_j \cdot \pi_{i,j}) + z'_i \bmod p \end{aligned}$$

Note that in $g_{i,j}$ there is no $y_{i,j}$, because $y_{i,j}$ in different $d_{i,j}$ have canceled out each other.

(c) _Disseminating Encrypted Result_: $\mathbf{s}$ publishes $\vec{g} = [g_1, \ldots, g_3]$.

(5) **Solving a Puzzle**. $\mathsf{Solve}(\vec{o}_j, pp, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_{\mathbf{s}}, cmd) \rightarrow (m, \zeta)$ Server $\mathbf{s}$ takes the following steps.

Case 1 when solving a puzzle related to the linear combination.

(a) _Finding secret keys_:

(i) finds temporary key tk, where $tk = h^{2^Y} \bmod N$, via repeated squaring of h modulo N.

(ii) derives two keys from tk:

$$k' = \mathsf{PRF}(1, tk), \quad s' = \mathsf{PRF}(2, tk)$$

(b) _Removing blinding factors_: removes the blinding factors from $[g_1, \ldots, g_3] \in \vec{g}$. $\forall i, 1 \leq i \leq 3$:

$$\begin{aligned} \theta_i &= \underbrace{\left(\mathsf{PRF}(i, s')\right)^{-1}}_{(w'_i)^{-1}} \cdot \big(g_i - \overbrace{\mathsf{PRF}(i, k')}^{z'_i}\big) \bmod p \\ &= \gamma_i \cdot \sum_{j=1}^{z} q_j \cdot \pi_{i,j} \bmod p \end{aligned}$$

(c) _Extracting a polynomial_: interpolates a polynomial $\boldsymbol{\theta}$, given pairs $(x_1, \theta_1), \ldots, (x_3, \theta_3)$. Note that $\boldsymbol{\theta}$ will that the form:

$$\boldsymbol{\theta}(x) = (x - root) \cdot \sum_{j=1}^{z} q_j \cdot (x + m_j) \bmod p$$

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

$$\boldsymbol{\theta}(x) = \boldsymbol{\psi}(x) - root \cdot \sum_{j=1}^{z} q_j \cdot m_j \bmod p$$

where $\boldsymbol{\psi}(x)$ is a polynomial of degree two with constant term being 0.

(d) _Extracting the linear combination_: retrieves the result (i.e., the linear combination of $m_1, \ldots, m_z$) from polynomial $\boldsymbol{\theta}(x)$'s constant term: $cons = -root \cdot \sum_{j=1}^{z} q_j \cdot m_j$ as follows:

$$\begin{aligned} res &= cons \cdot (-root)^{-1} \bmod p \\ &= \sum_{j=1}^{z} q_j \cdot m_j \end{aligned}$$

(e) _Extracting valid roots_: extracts the root(s) of $\boldsymbol{\theta}$. Let set R contain the extracted roots. It identifies the valid root, by finding a root $root$ in R, such that $\mathtt{Ver}(com', (root, tk)) = 1$.

(f) _Publishing the result_: publishes the solution $m = res$ and the proof $\zeta = (root, tk)$.

Case 2 when solving each j-th puzzle $\vec{o}_j$ in $\vec{o}$ of client $\mathbf{c}$ (when cmd = clientPzl), server $\mathbf{s}$ takes the following steps.

(a) _Finding secret bases and keys_: sets r_j and mk_j as follows.

- if $j = 1$: sets the base to r_1, where $r_1 \in pp$. Then, it finds mk_1 where $mk_1 = r_1^{2^{T_1}} \bmod N$ through repeated squaring of r_1 modulo N.
- if $j > 1$: computes base r_j as $r_j = \mathsf{PRF}(j||0, mk_{j-1})$. Next, it finds mk_j where $mk_j = r_j^{2^{T_j}} \bmod N$ through repeated squaring of r_j modulo N.

It derive two keys from mk_j:

$$k_j = \mathsf{PRF}(1, mk_j), \quad s_j = \mathsf{PRF}(2, mk_j)$$

(b) _Removing blinding factors_: re-generates six pseudorandom values using k_j and s_j:

$$\forall i, 1 \leq i \leq 3 : \ z_{i,j} = \mathsf{PRF}(i, k_j), \ w_{i,j} = \mathsf{PRF}(i, s_j)$$

Next, it uses the blinding factors to unblind $\vec{o}_j = [o_{1,j}, \ldots, o_{3,j}]$. $\forall i, 1 \leq i \leq 3 : \ \pi_{i,j} = \left((w_{i,j})^{-1} \cdot o_{i,j}\right) - z_{i,j} \bmod P$

(c) _Extracting a polynomial_: interpolates a polynomial $\boldsymbol{\pi}_j$, given pairs $(x_1, \pi_{1,j}), \ldots, (x_3, \pi_{3,j})$.

(d) _Publishing the solution_: considers the constant term of $\boldsymbol{\pi}_j$ as the plaintext message, m_j. It publishes the solution $m = m_j$ and the proof $\zeta = mk_j$.

(6) **Verification**. $\mathsf{Verify}(m, \zeta, ., pp, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_s, cmd) \rightarrow \ddot{v} \in \{0,1\}$
A verifier (that can be anyone) takes the following steps.

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

(a) *Checking the commitment's opening*: verify the validity of $(root, tk) \in \zeta$, provided by $\mathbf{s}$ in step 5f of Case 1:

$$\mathsf{Ver}\big(com', (root, tk)\big) \overset{?}{=} 1$$

If the verification passes, 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 the root $root$ in ζ, by taking the following steps.

(i) derives two keys from tk:

$$k' = \mathsf{PRF}(1, tk), \quad s' = \mathsf{PRF}(2, tk)$$

(ii) removes the blinding factors from $\vec{g} = [g_1, \ldots, g_3]$ that were provided by server $\mathbf{s}$ in step 4c. Specifically, for every i, $1 \le i \le 3$:

$$\theta_i = \underbrace{\big(\mathsf{PRF}(i, s')\big)^{-1}}_{(w'_i)^{-1}} \cdot \big(g_i - \overbrace{\mathsf{PRF}(i, k')}^{z'_i}\big) \bmod p$$

$$= \gamma_i \cdot \sum_{j=1}^{z} q_j \cdot \pi_{i,j} \bmod p$$

(iii) interpolates a polynomial $\boldsymbol{\theta}$, given $(x_1, \theta_1), \ldots, (x_3, \theta_3)$. Note that $\boldsymbol{\theta}$ will have the form:

$$\boldsymbol{\theta}(x) = (x - root) \cdot \sum_{j=1}^{z} q_j \cdot (x + m_j) \bmod p$$

$$= \boldsymbol{\psi}(x) - root \cdot \sum_{j=1}^{z} q_j \cdot m_j \bmod p$$

where $\boldsymbol{\psi}(x)$ is a polynomial of degree 2 whose constant term is 0.

(iv) checks if $root$ is a root of $\boldsymbol{\theta}$, by evaluating $\boldsymbol{\theta}$ at $root$ and checking if the result is 0, i.e., $\boldsymbol{\theta}(root) \overset{?}{=} 0$. It proceeds to the next step if the check passes. It returns $\ddot{v} = 0$ and takes no further action, otherwise.

(c) *Checking the final result*: retrieves the result (which is the linear combination of $m_1, \ldots, m_n$) from polynomial $\boldsymbol{\theta}(x)$'s constant term: $t = -root \cdot \sum_{j=1}^{z} q_j \cdot m_j$ as follows:

$$res' = -t \cdot root^{-1} \bmod p$$

$$= \sum_{j=1}^{z} q_j \cdot m_j$$

It checks $res' \overset{?}{=} m$, where $m = res$ is the result that $\mathbf{s}$ sent to it, in step 5f of Case 1.

(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 j-th solution of a single puzzle belonging to client $\mathbf{c}$:

(a) *Checking the commitment' opening*: checks whether opening pair (m_j, mk_j), given by $\mathbf{s}$ in step 5d of Case 2, matches the commitment:

$$\mathsf{Ver}\big(com_j, (m_j, mk_j)\big) \overset{?}{=} 1$$

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

Theorem M.1. *If the sequential modular squaring assumption holds, factoring N is a hard problem,* PRF, OLE$^+$*, and the commitment schemes are secure, then the protocol presented above is secure.*

We refer readers to Appendix N for the proof of Theorem M.1.

Remark 1. In step 3(b)iiA, index j is concatenated with 0 to avoid any collision (i.e., generating the same pseudorandom value more than once), because j, as input of PRF, will be used as input in other steps.

N PROOF OF THEOREM M.1

Proof sketch. There will be a significant overlap between the full proof Theorems M.1 and 5.1. The proof of Theorems M.1 differs from that of Theorem 5.1 from a key perspective. Namely, the former requires an additional discussion on the privacy of each base $r_{j'}$ before the $(j'-1)$-th puzzle is solved, for every j', where $2 \le j' \le z$.

Briefly, the additional discussion will rely on the security of standard RSA-based TLP, the hiding property of the commitment, and the security of PRF.

Specifically, before the j-th puzzle is solved the related master key mk_j cannot be extracted except for a probability negligible in the security parameter, $\mu(\lambda)$, if the sequential modular squaring assumption holds, factoring N is a hard problem, and the commitment scheme satisfies the hiding property. This argument holds for any j, where $1 \le j \le z$.

Therefore, with a probably at most $\mu(\lambda)$ an adversary can find a key $mk_{j'-1}$ of PRF to compute the base $r_{j'}$, which has been set as $r_{j'} = \mathsf{PRF}(j'||0, mk_{j'-1})$, for any j', where $2 \le j' \le z$. Furthermore, since $r_{j'}$ is the output of PRF, due to the security of PRF (that its output is indistinguishable from the output of a random function), the probability of correctly computing it is $\mu(\lambda)$, without the knowledge of $mk_{j'-1}$. □

O EXTENSION: MULTI-CLIENT MULTI-INSTANCE Tempora-Fusion

The Multi-Instance Tempora-Fusion can be combined with the Tempora-Fusion protocol to support multiple clients, i.e., a combination of the protocols presented in Section 6.1 and 5.2.

This new version will (i) allow a client to generate multi-puzzles such that the server can solve them sequentially, (ii) the party can ask the server to homomorphically compute a linear combination of its puzzles, (iii) multiple parties get together and ask the server to homomorphically compute a linear combination of some of their puzzles (where the input of computation is a single puzzle from each client), and (iv) enable anyone to verify the correctness of each puzzle's solution and computations' outputs.

In this setting, the server needs to generate $\bar{t}$ x-coordinates (instead of 3 x-coordinates) and each client should use these $\bar{t}$

x-coordinates. Also, there will be two different phases for the linear combination, one to perform a linear combination of a single client's puzzles (as described in Phase 4, Page 20, of Multi-Instance Tempora-Fusion), and another one to perform a linear combination of n different clients' puzzles (as explained in Phase 4, Page 9, of Tempora-Fusion).

P FULL ASYMPTOTIC COST ANALYSIS

In this section, we examine the asymptotic costs of our schemes.

P.1 Tempora-Fusion

We initially focus on each client's computation cost.

Client's Computation Cost. In the Puzzle Generation phase (Phase 3), in each step 3(b)i and 3(b)ii, a client $\mathbf{c}_u$ performs a modular exponentiation over $\phi(N_u)$ and N_u respectively. In steps 3(b)iii and 3c, in total the 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 polynomial at $\bar{t}$ x-coordinates, which will involve $\bar{t}$ modular additions, using Horner's method [22]. 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 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 $\mathbf{s}$ engages $\bar{t}$ instances of OLE^{+} with each client. In step 4d, server $\mathbf{s}$ performs $\bar{t}\cdot n$ modular addition. During the Solving Puzzles phase (Phase 5), in Case 1 step 5a, server $\mathbf{s}$ performs Y repeated modular squaring and invokes two instances of PRF for each client in $\mathcal{I}$. In step 5b, $\mathbf{s}$ 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 $\mathbf{s}$ in Case 1 is $O(\ddot{t}^2+\bar{t}\cdot n+\ddot{t}\cdot Y)$.

In Case 2, the costs of server $\mathbf{s}$ for each client $\mathbf{c}_u$ involves the following operations. $\mathbf{s}$ 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 $\mathbf{s}$ 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, 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 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 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 $O(\bar{t}\cdot n)$.

P.2 Multi-Instance Tempora-Fusion

We initially focus on the client's computation cost.

Client's Computation Cost. In the Puzzle Generation phase (Phase 3), in each step 3(b)i and 3(b)ii, the client performs z modular exponentiations over $\phi(N_u)$ and N_u respectively. In steps 3(b)ii, 3(b)iii, and 3c, in total the client invokes $9\cdot z-1$ instances of PRF.

In step 3(d)i, it performs z modular additions. In step 3(d)ii, it evaluates z polynomials at 3 x-coordinates, which in total will involve $3\cdot z$ modular additions, using Horner's method. In step 3e, the client also performs $3\cdot z$ additions and $3\cdot z$ multiplications to encrypt the y-coordinates. In step 3f, the client invokes the hash function z times to commit to its messages.

In the Linear Combination Phase (Phase 4), step 4(a)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(a)ii, it invokes $9\cdot z+6$ instances of PRF. In step 4(a)iii, it performs 3 additions.

In step 4(a)iv, it invokes the hash function once to commit to the random root. In step 4(a)v, it invokes $3 \cdot z$ instances of OLE^+. Thus, the client's complexity is $O(z)$.

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 a single instance of the hash function (to check the opening of a commitment). In step 6b it invokes 8 instances of PRF.

In step 6(b)ii, it performs 3 multiplications. In step 6(b)iv, it interpolates a polynomial of degree 2 that involves $O(1)$ addition and $O(1)$ multiplication operations. In step 6(b)iii, it evaluates a polynomial of degree 2 at a single point, resulting in 2 additions and 2 multiplications. In step 6c, it performs a single multiplication.

In the Verification phase (Phase 6), the verifier's computation cost in Case 2 includes only a single invocation of the hash function to check the opening of a commitment. Therefore, the verifier's complexity for z puzzles is $O(z)$.

Server's Computation Cost. In step 4(a)v, the server invokes $3 \cdot z$ instances of OLE^+. In step 4b, the server performs $3 \cdot z$ modular additions. 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. In step 5b, **s** it performs 3 additions and 3 multiplications. In the same step, it invokes 6 instances of PRF.

In step 5c, it interpolates a polynomial of degree 2 that involves $O(1)$ addition and $O(1)$ multiplication operations. In step 5d, it performs a single multiplication. In step 5e, it factorizes a polynomial of degree 2 to find its root, which will cost $O(1)$. [4] In the same step, it invokes the hash function once, to identify the valid root. Therefore, the complexity of server **s** in Case 1 is $O(z + Y)$.

In Case 2, the costs of server **s** involve the following operations. In step 5a, the server performs $O(max_{ss} \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$ modular squaring to find z master keys $mk_1, \ldots, mk_z$.

In steps 5a and 5b, in total, it invokes $6 \cdot z + 2$ instances of PRF. It performs $3 \cdot z$ additions and $3 \cdot z$ multiplications to decrypt y-coordinates. In step 5c, in total for z puzzles, it interpolates z polynomials, each with of degree 2, which will involve $O(z)$ additions and $O(z)$ multiplications. Hence, the complexity of server **s** in Case 2 is $O(max_{ss} \cdot \sum_{i=1}^{z} \bar{\Delta}_i)$.

Next, we evaluate the communication cost in Multi-Instance Tempora-Fusion. First, we concentrate on the client's communication cost.

Client's Communication Cost. 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 $4 \cdot z + 1$ messages. In the Linear Combination phase (Phase 4), step 4(a)v, it invokes $3 \cdot z$ instances of OLE^+ where each instance imposes $O(1)$ communication cost. In step 4(a)vi, the client publishes two elements. Hence, the client's communication complexity is $O(z)$. 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 4 messages. In the Linear Combination phase (Phase 4) step 4(a)v, it invokes $3 \cdot z$ instances of OLE^+, where each instance imposes $O(1)$ communication cost. In step 4c, it publishes 3 messages.

In the Solving a Puzzle phase (Phase 5), Case 1, step 5f, it publishes 3 messages. In Case 2 step 5d, the server publishes two messages. The size of each message it publishes in the above steps is 128 bits. Hence, the communication complexity of the server is $O(z)$.

[4]In general the complexity of factorizing a polynomial of degree deg is $O(deg^2)$. However, in the evaluation, the degree of the polynomial is fixed at 2. Consequently, the complexity of factorizing this polynomial can be set to $O(1)$.