Verifiable Homomorphic Linear Combinations in Multi-Instance Time-Lock Puzzles

Aydin Abadi*

Newcastle University

Abstract. Time-Lock Puzzles (TLPs) have been developed to securely transmit sensitive information into the future without relying on a trusted third party. Multi-instance TLP is a *scalable* variant of TLP that enables a server to efficiently find solutions to different puzzles provided by a client at once. Nevertheless, existing multi-instance TLPs lack support for (verifiable) *homomorphic computation*. To address this limitation, we introduce the "Multi-Instance partially Homomorphic TLP" (MH-TLP), a multi-instance TLP supporting efficient verifiable homomorphic linear combinations of puzzles belonging to a client. It ensures anyone can verify the correctness of computations and solutions. Building on MH-TLP, we further propose the "Multi-instance Multi-client verifiable partially Homomorphic TLP" (MMH-TLP). It not only supports all the features of MH-TLP but also allows for verifiable homomorphic linear combinations of puzzles from different clients. Our schemes refrain from using asymmetric-key cryptography for verification and, unlike most homomorphic TLPs, do not require a trusted third party. A comprehensive cost analysis demonstrates that our schemes scale linearly with the number of clients and puzzles.

1 Introduction

Time-Lock Puzzles (TLPs) are interesting cryptographic primitives that enable the transmission of information to the future. They enable a party to encrypt a message in a way that no one else can decrypt it until a certain time has elapsed.¹ TLPs have a wide range of applications, including scheduled payments in cryptocurrencies [38], timed-commitments [23], zero-knowledge proofs [17], e-voting [13], timed secret sharing [24], sealed-bid auctions [34], verifiable delay functions [11], and secure aggregation in federated learning [1].

Since Rivest *et al.* [34] introduced TLPs, they have evolved, leading to the development of two vital variants: *multi-instance* TLPs and *homomorphic* TLPs. Our work advances both of these variants. Multi-instance TLPs were introduced in [3]. The multi-instance TLP setting entails a single client generating n puzzles and simultaneously transmitting them to a server. This concept serves as a natural extension of the initial single-puzzle paradigm established in [34]. 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. Specifically, for a fixed time parameter Δ and z puzzles, it saves $\frac{1}{z} \cdot \sum_{j=1}^{z} j$ times modular squaring. For instance, when z = 100, this approach yields 50 times reduction in modular squaring.

Multi-instance TLPs have applications across various domains. For example, *journalists* or *whistleblowers* in hostile environments can use multi-instance TLPs to schedule the gradual release of sensitive information at the most impactful moments or after ensuring their safety. This method eliminates the need to be online at the time of release and removes the need to trust a third party with sensitive data. In *online education*, multi-instance TLPs can benefit students with unreliable internet connections by allowing them to download multiple exam content in advance, ensuring access when their connection is stable. In this scenario, the exam materials become accessible only at the designated start time. Multi-instance TLPs can also be utilized for the continuous verification of cloud service availabilities [3].

^{*} aydin.abadi@ncl.ac.uk

¹ Certain TLPs rely on third-party assistance to manage the timed release of a secret. However, this category of protocols is not the focus of this paper.

In a separate line of research, Malavolta *et al.* [28] introduced (fully) homomorphic TLP notion, enabling the execution of arbitrary functions over puzzles before their resolution is found. In general, fully homomorphic TLPs consider scenarios where there exist n clients, each generating and transmitting a puzzle encoding its solution to a server. Upon receiving the puzzles, the server executes a homomorphic function across these puzzles, generating a unified puzzle. The solution to this puzzle represents the output of the function evaluated across all individual solutions. To enhance efficiency, partially homomorphic TLPs have also been introduced, including variants that enable homomorphic linear combinations or the multiplication of puzzles [28]. More recently, Abadi [1] proposed a partially homomorphic TLP that also enables efficient verification of the correctness of homomorphic operations performed on the puzzles. Homomorphic TLPs have found applications in numerous areas, including atomic swaps [39], payment channels [40], as well as verifiable e-voting, and secure aggregation in federated learning [1].

1.1 Limitations of State-of-the-Art TLPs

Current TLPs exhibit notable limitations that constrain their practical use. Existing multi-instance TLPs' functionalities are confined to a few basic operations: (i) solution revelation, where the server can access the solution to each puzzle only after a designated time period has elapsed, and (ii) solution verification, allowing a verifier to confirm the correctness of the solution obtained by the server. A key drawback of these multi-instance TLPs is their lack of support for (verifiable) homomorphic operations.

Conversely, the state-of-the-art homomorphic TLPs *lack the essential feature of multi-instance TLPs*. Specifically, when a client employs a homomorphic TLP to generate multiple puzzle instances intended to be solved at different times and submits all these puzzles to a server simultaneously, the server must handle each puzzle instance separately and in parallel. This naive approach is highly resource-intensive and inefficient.

1.2 Our Contributions

Multi-Instance Partially Homomorphic TLP. To address the aforementioned limitations of multiinstance TLPs, we introduce Multi-instance verifiable partially Homomorphic TLP (MH-TLP), 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. In this setting, the server does not need to simultaneously handle them; instead, it can solve them sequentially.

MH-TLP enables the client to come back 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. MH-TLP also supports public verification for (1) a single puzzle's solution, and (2) the computation's result.

Inspired by previous multi-instance TLPs, we also 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.

We formally define our scheme and, for the first time, present the formal definition of multi-instance TLP (in Definition 7). Although the multi-instance TLP idea has been used previously, its core property, efficiently supporting multiple puzzle instances, has not been formally defined until now.

Multi-Instance Multi-Client Partially Homomorphic TLP. To address the limitation of the existing (partially) homomorphic TLPs that do not support multi-instance, we upgrade MH-TLP to a new variant of TLP called Multi-instance Multi-client verifiable partially Homomorphic TLP (MMH-TLP). This new variant offers the features of both partially homomorphic TLP and multi-instance TLP. Specifically, it supports verifiable partially homomorphic operations on the puzzles belonging to single or multiple clients while maintaining the multi-instance feature. MMH-TLP allows each client to independently generate different

puzzles and send them at once to the server. In this case, the server does not have to deal with each client's puzzles simultaneously (offering the multi-instance feature). It enables single or multiple clients to ask the server to perform homomorphic linear combinations of their puzzles (each belonging to a different client in the multi-client setting). This scheme allows anyone to verify whether the server has performed the computation correctly and provided a correct solution. Thus, MMH-TLP bridges the gap between (a) (partially) homomorphic TLPs that support multiple clients (but not multi-instance) and (b) multi-instance (homomorphic) TLPs that support only a single client.

Table 1: Asymptotic costs of existing (partially) homomorphic TLPs. In the figure, n is the number of clients, \ddot{t} is the number of leaders, $\bar{\Delta}_{j,u}$ is the period between the discovery of two consecutive solutions of client C_u , max_{ss} is the maximum number of squaring that the strongest server S can perform per second, z is the total number of puzzles given by a client to S, $\Delta_{j,u}$ is the period when the puzzle of C_u must remain concealed after the related puzzle is given to S, and $Y = max_{ss} \cdot \Delta$ is the period between granting the computation and when a linear combination of solutions is learned by S.

Schemes	Parties	Computation Cost	Communication Cost	
MH-TLP	Client	O(z)	O(z)	
	Verifier	O(z)	-	
	Server	$O(z+Y+max_{ss}\cdot\sum\limits_{j=1}^{z}ar{\varDelta}_{j,u})$	O(z)	
MMH-TLP	Client	$O(\ddot{t} \cdot (z+n))$	$O((\ddot{t}+n)\cdot z)$	
	Verifier	$O(\ddot{t}^2 + \ddot{t} + z)$	-	
	Server	$O(\ddot{t} \cdot (z+n+Y+\ddot{t}) + max_{ss} \cdot \sum_{j=1}^{z} \bar{\Delta}_{j,u})$	$O(\ddot{t}\cdot n+z)$	
[1]	Client	$O(\ddot{t}\cdot n)$	$O(\ddot{t} \cdot n)$	
	Verifier	$O(\ddot{t}^2 + \ddot{t} + z)$	-	
	Server	$O(\ddot{t} \cdot (n+Y+\ddot{t}) + max_{ss} \cdot \sum_{j=1}^{z} \Delta_{j,u})$	$O(\ddot{t}\cdot n)$	
[28]	Client	O(1)	O(1)	
	Verifier	-	-	
	Server	$O(n+max_{ss}\cdot\sum\limits_{j=1}^{z} arDelta_{j,u})$	O(1)	
[27]	Client	O(1)	O(1)	
	Verifier	-	-	
	Server	$O(n + max_{ss} \cdot (\frac{\Delta}{\log(max_{ss} \cdot \Delta)} + \sum_{j=1}^{z} \Delta_{j,u}))$	O(1)	
[16]	Client	O(1)	O(1)	
	Verifier	-	-	
	Server	$O(n^2 + max_{ss} \cdot (\Delta + \sum_{j=1}^{z} \Delta_{j,u}))$	O(1)	

Table 2: Comparing features of different (partially) homomorphic TLPs.

Schemes	Multi-Instance	Multi-Client	Without Trusted-Setup	Supporting Verification		Flexible	Datah Salaina			
				Client's Solution	Linear Combination	1 line raramters	Batch Solving			
MH-TLP	\checkmark	×	\checkmark	\checkmark	\checkmark	\checkmark	×			
MMH-TLP	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	×			
[1]	×	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	×			
[28]	×	\checkmark	×	×	×	×	×			
[27]	×	\checkmark	×	\checkmark	×	×	×			
[16]	×	\checkmark	×	×	×	×	\checkmark			

We compare the computational and communication complexities, as well as the features, of our protocols with those of previous work. Tables 1 and 2 provide a summary of this comparison. The complexities of our protocols are linear with the number of participating clients n and the number of puzzle z that each client possesses. Our schemes support efficient verification of solutions without relying on asymmetric key-based primitives. Moreover, MMH-TLP provides a distinctive set of features that are not simultaneously available in any existing TLP.

1.3 Applications

Scalable, Private, and Compliant Scheduled Payments in Online Banking. Outsider and insider attacks pose an immediate risk to various organizations and their clients, particularly financial institutions and their customers. There have been numerous incidents where customers' names, addresses, credit scores, credit limits, and balances have been stolen and in some cases revealed to the public, e.g., see [26,15,30]. Investment strategies devised by individuals or companies and managed through financial institutions are especially vulnerable, as they contain critical and valuable information that attackers could exploit.

Our schemes enable individuals and businesses to schedule payments and investments privately through their banks, without disclosing the transaction amounts prior to the scheduled transfer time. Leveraging the multiinstance feature, these schemes efficiently manage a high volume of scheduled transactions without placing significant strain on server resources. By utilizing a homomorphic linear combination, our schemes allow banks to precalculate the combined transfer amounts, such as the average or total. This functionality not only ensures compliance with regulatory standards but also enhances risk management. By verifying transfer amounts ahead of time, banks can confirm that all transactions meet regulatory requirements. Additionally, the ability to precalculate transfer amounts helps banks identify and address potential risks before they materialize, preventing issues like overdrafts or breaches of internal limits.

Asynchronous Multi-Model Training in Secure Aggregation as a Service (SAaaS). Federated Learning (FL) is a machine learning framework that allows multiple parties to collaboratively build models without exposing their sensitive data to one another [31]. Unlike traditional centralized methods, where data is collected and processed on a central server, FL enables model training on individual devices or clients, each holding private data. This approach preserves data privacy by ensuring that raw data never leaves the clients. Instead, only model updates are transmitted to a central server.

To securely compute the sum of model updates from clients, Bonawitz *et al.* [10] developed a secure aggregation mechanism. This mechanism uses a trusted setup and a public-key-based verification system to detect any misbehavior by the server. In response to this, the author of verifiable partially homomorphic TLP in [1] proposed an alternative that does not require a trusted setup and supports efficient verification. Our solution, MMH-TLP, can also substitute the TLP in [1] (and the secure aggregation in [10]) offering additional features that are particularly well-suited for more generic, multi-model settings, as explained below.

As the adoption of FL continues to grow, it is anticipated that cloud-based servers offering Secure Aggregation as a Service (SAaaS) will become common. In such scenarios, clients may want to train multiple models over time, in collaboration with the server and various sets of clients. Our MMH-TLP can replace the aforementioned secure aggregation mechanisms while enabling asynchronous multi-model training. Specifically, in this setting, each client can submit to the server its (initial) parameters for different models in a single step, well before the start of each model training round.

MMH-TLP supports (a) asynchronous client participation, allowing clients to create their puzzles and join the computation of linear combinations at different times without waiting for others, and (b) a dynamic client base, enabling new clients to independently join the system, prepare, and submit their puzzles. Consequently, in the context of SAaaS, MMH-TLP facilitates asynchronous and dynamic parameter submissions, ensuring that the server can efficiently and securely aggregate parameters for each model at the appropriate time.

Including the notion of time-lock can also enhance the privacy of FL. By requiring the server to learn the aggregated result after a certain time, the system can allow a sufficient number of clients to submit their model parameters to the server. This is important when there is a possibility of collusion between the clients

and the server. Given the result of the secure aggregation of model parameters: $agr = \sum_{i=1}^{n} a_i$, corrupt parties

that collude with each other can always deduce the linear combination of honest parties' inputs from *agr*. Thus, by defining a time window and assuming enough honest clients submit their updates in this period, we can reduce the chance of colluding adversaries to link input to an honest party.

Sequential Verifiable E-Voting and Sealed-Bid Auction Systems. E-voting and sealed-bid auction systems are critical applications where maintaining data integrity, confidentiality, and system scalability is essential. Researchers have suggested utilizing homomorphic TLPs in these systems, to facilitate secure computations while preserving the privacy of each individual vote or bid [28].

MMH-TLP can be applied more broadly to e-voting and sealed-bid auction systems, where clients participate in multiple instances of voting or bidding managed sequentially by a server. This method minimizes the server's workload by avoiding the need to simultaneously handle each voter's or bidder's puzzle created for a different instance. Additionally, it allows for public verification of the computations, therefore ensuring the result's correctness in the process, in the case where a malicious server may attempt to alter the result.

2 Related Work

Initially, the idea of sending information into the future, i.e., time-lock puzzle/encryption was proposed by Timothy C. May [29]. A basic property of a time-lock scheme is that generating a puzzle takes less time than solving it. May's scheme relies on a trusted agent to release a secret at the appropriate time for a puzzle to be solved, which can be a significant assumption. To address this, Rivest *et al.* [34] proposed an RSA-based TLP that eliminates the need for a trusted agent. This scheme relies on sequential modular squaring and remains secure even against a receiver with extensive computational resources running in parallel.

Following the development of the RSA-based TLP, several new variations have emerged. For instance, Boneh $et \ al. [12]$ and Garay $et \ al. [19]$ introduced TLPs tailored for situations where a potentially malicious client must provide zero-knowledge proofs to assure the server that the correct solution will be revealed after a predetermined time. Additionally, Baum $et \ al. [7]$ devised a composable TLP that can be defined and validated within the universal composability framework. To ensure security against adversaries equipped with quantum computers Agrawal $et \ al. [4]$ proposed a TLP based on lattices, believed to be post-quantum secure. In the remainder of this section, we will discuss two variants that are most closely related to our schemes.

2.1 Multi-instance Time Lock Puzzle

To enhance the scalability of TLPs, researchers have examined scenarios where a server receives multiple puzzles or instances from a client simultaneously and needs to solve all of them. The TLPs proposed in [14,3] address this setting. The scheme in [14] relies on an asymmetric-key encryption method, unlike the symmetric-key approach used in traditional TLPs, and it lacks verification capabilities. To overcome these limitations, a TLP is introduced in [3]. This TLP employs an efficient hash-based verification, enabling the server to prove the correctness of the solutions. However, this scheme is effective only for time intervals of identical size. Later, a multi-instance TLP in [33] addressed this limitation. Nevertheless, none of these multiinstance TLPs support (verifiable) homomorphic computations on puzzles. Their functionality is limited to solving and verifying puzzles.

2.2 Homomorphic Time-lock Puzzles

The notion of homomorphic TLPs was introduced by Malavolta and Thyagarajan *et al.* [28]. It allows arbitrary functions to be applied to puzzles before they are solved. This scheme incorporates the RSA-based TLP and fully homomorphic encryption. To enhance efficiency, partially homomorphic TLPs have been developed,

including those that support homomorphic linear combinations or multiplications of puzzles [28,27]. Unlike fully homomorphic TLPs, partially homomorphic TLPs do not depend on fully homomorphic encryption, leading to more efficient implementations. In contrast to the partially homomorphic TLP described in [28], the TLPs presented in [27] offer additional features. Firstly, they enable a verifier to confirm that puzzles have been correctly generated. Secondly, they allow verification of the server's solution for a single client's puzzle, though they do not support the verification of solutions related to homomorphic computations.

Later, Srinivasan *et al.* [37] noted that existing homomorphic TLPs are limited in their ability to handle a high number of puzzles in a batch, as solving one puzzle leads to discovering all solutions in the batch. To address this limitation, they proposed a scheme based on indistinguishability obfuscation and puncturable pseudorandom functions. To improve the efficiency of this scheme, Dujmovic *et al.* [16] introduced a new method that does not rely on indistinguishability obfuscation but instead uses pairings and learning with errors. Both schemes assume that all initial puzzles' solutions will be discovered simultaneously.

All the homomorphic TLPs discussed, except the scheme proposed in [37], require a fully trusted setup and assume the server will act honestly during computation. Incorporating a trusted third party undermines the core objective of RSA-based TLPs, which is to eliminate the need for such a party. Otherwise, this third party could simply hold the secret and release it to the recipient at the designated time, thereby compromising the design's intent. Recently, Abadi [1] introduced a partially homomorphic TLP that enables anyone to verify the correctness of the server's solution, whether for a single client's puzzle or the computation itself. This scheme operates without relying on a trusted third party. However, all of the above schemes assume each client has only a single puzzle. They lack support for the multi-instance setting. If they are used directly in this setting, they would impose a high computation cost on the server.

3 Preliminaries

3.1 Notations and Assumptions

We consider the case where a server S is given multiple instances of a puzzle (or multiple puzzles) by a single client at once, to solve them. We consider a generic case where client C has a vector of messages: $\vec{m} = [m_1, \ldots, m_z]$. It wants S to learn each message m_i at time $time_i \in time$, where $time = [time_1, \ldots, time_z]$ and $time_{j-1} < time_j$. We define a time interval between two consecutive points in time as $\bar{\Delta}_j = time_j - time_{j-1}$. We define a vector of time intervals as $\vec{\Delta} = [\vec{\Delta}_1, \ldots, \vec{\Delta}_z]$.

The standard parameter max_{ss} denotes the maximum number of squaring that a solver (with the highest level of computation resources) can perform per second. We define a time parameter $T_j = max_{ss} \cdot \bar{\Delta}_j$ and $1 \leq j \leq z$. Time points $time_0$ and $time'_0$ refer to the only times when C is available and online, where $time_0, time'_0 < time_1$. We define Δ_j the time period where the *j*-th message remains hidden; therefore, Δ_j can be written as $\Delta_j = \sum_{i=1}^{j} \bar{\Delta}_i$. Server S must learn the computation result (i.e., a linear combination of messages) after a certain time Δ , where $\Delta < \bar{\Delta}_1$.

We define U as the universe of a solution m_j . We denote by $\lambda \in \mathbb{N}$ the security parameter. For certain system parameters, we use polynomial $poly(\lambda)$ to state the parameter is a polynomial function of λ . We define a public vector \vec{x} as $\vec{x} = [x_1, \dots, x_m]$, where $x_i \neq x_j$, $x_i \neq 0$, and $x_i \notin U$.

We define a hash function $G : \{0,1\}^* \to \{0,1\}^{poly(\lambda)}$, 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, e.g., $\log_2(p) = 128$. A set of leaders are involved in MMH-TLP, we denote the total number of leaders with \vec{t} . We set $\vec{t} = \vec{t} + 2$. For a value v defined over a finite field \mathbb{F}_p (of prime order p), by v^{-1} we mean the multiplicative inverse of v, i.e., $v \cdot v^{-1} \equiv 1 \mod p$.

To ensure generality, we adopt notations from zero-knowledge proof systems [9,18]. Let R_{cmd} be an efficient binary relation that consists of pairs of the form (stm_{cmd}, wit_{cmd}) , where stm_{cmd} is a statement and wit_{cmd}

is a witness. Let \mathcal{L}_{cmd} be the language (in \mathcal{NP}) associated with R_{cmd} , i.e., $\mathcal{L}_{cmd} = \{stm_{cmd} | \exists wit_{cmd} \text{ s.t.} R(stm_{cmd}, wit_{cmd}) = 1\}$. A (zero-knowledge) proof for \mathcal{L}_{cmd} allows a prover to convince a verifier that $stm_{cmd} \in \mathcal{L}_{cmd}$ for a common input stm_{cmd} (without revealing wit_{cmd}). In this paper, two main types of verification occur (1) verification of a single client's puzzle solution, in this case, cmd = clientPzl, and (2) verification of a linear combination, in this case, cmd = evalPzl. In this work, we assume parties interact through a secure channel. Moreover, we consider a malicious (or active) adversary that will corrupt the server.

3.2 Pseudorandom Function

Informally, a pseudorandom function is a deterministic function that takes a key of length λ and an input; and outputs a value. Informally, 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: PRF : $\{0, 1\}^* \times \{0, 1\}^{poly(\lambda)} \to \mathbb{F}_p$. In practice, a pseudorandom function can be obtained from an efficient block cipher [22]. 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 this setting, 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 enables the receiver to learn only $s = a \cdot c + b \in \mathbb{F}_p$, while the sender learns nothing. Ghosh *et al.* [20] proposed an efficient OLE that has O(1) overhead and mainly uses symmetric-key operations.²

Later, in [21] an enhanced OLE, denoted as OLE^+ , was introduced. OLE^+ ensures that the receiver cannot learn anything about the sender's inputs, even if it sets its input to OLE^+ is also accompanied by an efficient symmetric-key-based verification mechanism that enables a party to detect its counterpart's misbehavior during the protocol's execution. In this paper, we use OLE^+ to securely switch the blinding factors of secret messages (encoded in the form of puzzles) held by a server. We refer readers to Appendix A, for the construction of OLE^+ .

3.4 Polynomial Representation of a Message

Encoding a message m as a polynomial $\pi(x)$ allows imposing a certain structure on the message. Polynomial representation has previously been used in different contexts, for example in secret sharing [36], private set intersection [25], or error-correcting codes [32]. To encode a message m in a polynomial $\pi(x)$, we can use one following the full structure of $\pi(x)$ and $\pi(x)$ and $\pi(x)$ and $\pi(x)$ are the set of the full structure of $\pi(x)$ and $\pi(x)$ are the set of the full structure of $\pi(x)$ and $\pi(x)$ are the set of the set

of the following approaches, (1) setting m as the constant terms of $\pi(x)$, e.g., $m + \sum_{j=1}^{n} x^j \cdot a_j \mod p$ and (2)

setting *m* as the root of $\pi(x)$, e.g., $\pi(x) = (x-m) \cdot \tau(x) \mod p$. In this paper, we utilize both approaches. The former approach allows us to perform a linear combination of the constant terms of different polynomials. Furthermore, we use the latter approach to insert a secret random root (or a trap) into the polynomials encoding the messages. Hence, the resulting polynomial representing the linear combination maintains this root, allowing the verification of the correctness of the computation.

In general, polynomials can be represented in the point-value form, in the following way. A polynomial $\pi(x)$ 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 $\pi_i = \pi(x_i)$ for all $i, 1 \le i \le l$. A polynomial in this form can be converted into coefficient form via polynomial interpolation, e.g., via Lagrange interpolation [5].

² The scheme uses an Oblivious Transfer (OT) extension as a subroutine. However, the OT extension requires only a constant number of public-key-based OT invocations. The rest of the OT invocations are based on symmetric-key operations. The exchanged messages in the OT extension are defined over a small-sized field, e.g., a field of size 128-bit [6].

Arithmetic of polynomials in point-value representation can be done by adding or multiplying the corresponding y-coordinates of polynomials. Let a be a scalar and $\{(x_1, \pi_1), \ldots, (x_l, \pi_l)\}$ be (y, x)-coordinates of a polynomial $\pi(x)$. Then, the polynomial θ interpolated from $\{(x_1, a \cdot \pi_1), \ldots, (x_l, a \cdot \pi_l)\}$ is the product of a and polynomial $\pi(x)$, i.e., $\theta(x) = a \cdot \pi(x)$.

3.5 Unforgeable Encrypted Polynomial with a Hidden Root

The idea of the "unforgeable encrypted polynomial with a hidden root" was introduced in [2]. Informally, it can be explained as follows. Let us consider a polynomial $\pi(x) \in \mathbb{F}_p[x]$ with a random secret root β . We can represent $\pi(x)$ in the point-value form and then encrypt its *y*-coordinates. We give all the *x*-coordinates and encrypted *y*-coordinates to an adversary and we locally delete all the *y*-coordinates. The adversary may modify any subset of the encrypted *y*-coordinates and send back to us the encrypted *y*-coordinates. If we decrypt all the *y*-coordinates sent by the adversary and then interpolate a polynomial $\pi'(x)$, the probability that $\pi'(x)$ has β has a root is negligible in the security parameter $\lambda = \log_2(p)$. Below, it is formally stated.

Theorem 1 (Unforgeable Encrypted Polynomial with a Hidden Root). Let $\pi(x)$ be a polynomial of degree n with a random root β , and $\{(x_1, \pi_1), \ldots, (x_l, \pi_l)\}$ be point-value representation of $\pi(x)$, where l > n, p denotes a large prime number, $\log_2(p) = \lambda'$ is the security parameter, $\pi(x) \in \mathbb{F}_p[x]$, and $\beta \stackrel{*}{\leftarrow} \mathbb{F}_p$. Let $o_i = w_i \cdot (\pi_i + z_i) \mod p$ be the encryption of each y-coordinate π_i of $\pi(x)$, 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 $\vec{o} = [\vec{o}_1, \ldots, \vec{o}_l]$, such that: (i) $\exists \vec{o}_i \in \vec{o}, \vec{o}_i \neq o_i$, and (ii) the polynomial $\pi'(x)$ interpolated from unencrypted y-coordinates $\{(x_1, (w_1 \cdot \vec{o}_1) - z_l), \ldots, (x_l, (w_l \cdot \vec{o}_l) - z_l)\}$ will have root β is negligible in λ' , i.e., $Pr[\pi'(\beta) \mod p = 0] \leq \mu(\lambda')$.

We refer readers to [2] for the proof of Theorem 1. This paper uses the unforgeable encrypted polynomial with a hidden root concept to detect a server's misbehaviors, a technique also used in [1].

3.6 Commitment Scheme

A commitment scheme involves a sender and a receiver, and it includes two phases: commitment and opening. During the commitment phase, the sender commits to a message, using algorithm $\operatorname{Com}(m, r) = com$, where m is the message and r is a secret value randomly chosen from $\{0, 1\}^{\operatorname{poly}(\lambda)}$. Once the commitment phase concludes, the sender forwards the commitment, com, to the receiver. During the opening phase, the sender transmits the pair $\hat{m} := (m, r)$ to the receiver. The receiver verifies the correctness of the pair using the algorithm $\operatorname{Ver}(com, \hat{m})$. It accepts the pair if the output of the verification algorithm is 1.

A commitment scheme offers two main properties hiding and binding. Hiding ensures that it is computationally infeasible for an adversary, i.e., the receiver, to gain any knowledge about the committed message mbefore *com* is opened. Binding ensures that it is computationally infeasible for an adversary, i.e., the sender, to open *com* to different values $\hat{m}' := (m', r')$ than the ones originally used during the commit phase. Thus, it should be infeasible to find an alternative pair \hat{m}' such that $\operatorname{Ver}(com, \hat{m}) = \operatorname{Ver}(com, \hat{m}') = 1$, where $\hat{m} \neq \hat{m}'$.

There is a standard efficient hash-based commitment scheme that involves computing $\mathbb{Q}(m||r) = com$ during the commitment. The verification step requires checking $\mathbb{Q}(m||r) \stackrel{?}{=} com$. Note that $\mathbb{Q} : \{0,1\}^* \to \{0,1\}^{poly(\lambda)}$ is a collision-resistant hash function, where the probability of finding two distinct values m and m' such that $\mathbb{Q}(m) = \mathbb{Q}(m')$ is negligible with regard to the security parameter λ . In this paper, we use this hash-based commitment scheme to identify a server's misbehaviors.

3.7 Time-Lock Puzzle

In this section, we restate the TLP's formal definition.

Definition 1. A TLP consists of three algorithms: $(\mathsf{Setup}_{\mathsf{TLP}}, \mathsf{GenPuzzle}_{\mathsf{TLP}})$. It meets completeness and efficiency properties. TLP involves a client C and a server S.

- Algorithms:

- Setup_{TLP}(1^λ, Δ, max_{ss}) → (pk, sk). A probabilistic algorithm run by C. It takes as input a security parameter, 1^λ, time parameter Δ (in seconds) that specifies how long a message must remain hidden, and time parameter max_{ss} which is the maximum number of squaring that a solver (with the highest level of computation resources) can perform per second. It outputs pair (pk, sk) that contains public and private keys.
- GenPuzzle_{TLP} $(m, pk, sk) \rightarrow o$. A probabilistic algorithm executed by C. It takes as input a solution m and (pk, sk). It outputs a puzzle o.
- Solve_{TLP} $(pk, o) \rightarrow s$. A deterministic algorithm run by S. It takes as input pk and o. It outputs a solution s.
- Completeness. For any honest C and S, it always holds that $\mathsf{Solve}_{\mathsf{TLP}}(pk, o) = m$.
- Efficiency. The run-time of $\mathsf{Solve}_{\mathsf{TLP}}(pk, o)$ is upper-bounded by $poly(\Delta, \lambda)$, with a fixed polynomial poly.

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

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

$$\Pr \begin{bmatrix} \mathsf{Setup}_{\mathsf{TLP}}(1^{\lambda}, \Delta, max_{ss}) \to (pk, sk) \\ \mathcal{A}_1(1^{\lambda}, pk, \Delta) \to (m_0, m_1, state) \\ b \stackrel{\$}{\leftarrow} \{0, 1\} \\ \underbrace{\mathsf{GenPuzzle}_{\mathsf{TLP}}(m_b, pk, sk) \to o}_{\mathcal{A}_2(pk, o, state) \to b} \end{bmatrix} \leq \frac{1}{2} + \mu(\lambda)$$

where $\delta(\Delta) = (1 - \epsilon)\Delta$ and $\epsilon < 1$, according to [11].

Note that, in the literature, the TLP definition includes two adversaries \mathcal{A}_1 and \mathcal{A}_2 because their run-times are different. We refer readers to Appendix C for the description of the original RSA-based TLP, which is the core of the majority of TLPs. By definition, TLPs are sequential functions. Their construction requires that a sequential function, such as modular squaring, is invoked iteratively a fixed number of times. The sequential function and iterated sequential functions notions, in the presence of an adversary possessing a polynomial number of processors, are formally defined in [11]. We restate the definitions in Appendix D.

3.8 Tempora-Fusion: A TLP with Efficient Verifiable Homomorphic Linear Combination

Recently, Abadi [1] proposed a TLP, called Tempora-Fusion, that supports an efficient verifiable homomorphic linear combination of puzzles, where which belongs to a different client. Since our schemes are built on top of Tempora-Fusion, we briefly describe it in the remainder of this section. Tempora-Fusion mainly relies on (1) a polynomial representation of a message, (2) an unforgeable encrypted polynomial, (3) oblivious linear function evaluation, and (4) the original RSA-based TLP initially introduced by Rivest *et al.* [34].

They consider a malicious server and semi-honest clients. In their scheme, the malicious server is considered strong in the sense that it can act arbitrarily and access the secret keys and parameters of a subset of clients. The scheme designates at random a subset of clients as leaders. Let \mathcal{I} be this subset, containing \ddot{t} leaders. The scheme allows the malicious server to gain access to the secret keys and parameters of some of these leaders. Specifically, for a set $P = \{S, C_1, \ldots, C_n\}$ containing all the parties involved, the scheme allows the adversary adaptively corrupt a subset \mathcal{W} of P. It will fully corrupt S and act arbitrarily on its behalf. It can retrieve the secret keys of a subset of clients in P. They define a threshold t and require the number of non-corrupted leaders (i.e., the parties in \mathcal{I}) to be at least t. This setting allows a high number of clients to be corrupted by the adversary, under a constraint. For instance, when |P| = 500, and the total number of leaders is 10 (i.e., $\ddot{t} = 10$), and t = 2, then the adversary may corrupt 498 parties in P (i.e., $|\mathcal{W}| = 98$), as long as at most 8 parties from \mathcal{I} are in \mathcal{W} , i.e., $|\mathcal{W} \cap \mathcal{I}| \leq t$.

Outline of Tempora-Fusion. Initially, each client creates a puzzle that encodes a secret solution and sends it to a server S. Upon receiving each puzzle, S processes it to determine the solution within a specified timeframe. The time at which each client's solution is found can be distinct from others. Upon finding a solution, S generates a proof that asserts the solution's correctness. Later, possibly after a certain amount of time has passed since sending their puzzles to S, some clients whose puzzles remain undiscovered may collaborate and request S to homomorphically combine their puzzles. The combined puzzles will encode a linear combination of their solutions. The plaintext result of this computation will be determined by S after a specific timeframe, potentially before any of their puzzles are solved.

To enable S to impose a specific structure (that can be considered as a hidden "trap") on its outsourced puzzles and homomorphically combine them, each client interacts with S and also communicates a message to other clients. After a certain period, S can find the solution to the puzzle encoding the computational result. It generates proof asserting the correctness of the solution, which anyone can efficiently verify to ensure that the result maintains the intended structure. Eventually, S discovers the solution to each client's puzzle as well. It then publishes both the solution and a proof that enables anyone to verify the validity of the solution too. Appendix B presents a detailed description of Tempora-Fusion.

4 Definition of Multi-Instance Verifiable Partially Homomorphic TLP

4.1 Private Linear Combination

The basic functionality \mathcal{F}^{PLC} that any z-input Private Linear Combination (PLC) computes takes as input a pair of coefficient q_j and plaintext value m_j (for every $j, 1 \leq j \leq n$), and returns their linear combination $\sum_{j=1}^n q_j \cdot m_j$, as stated in [1]. More formally, \mathcal{F}^{PLC} is defined as:

$$\mathcal{F}^{\text{PLC}}((q_1, m_1), \dots, (q_z, m_z)) \to \sum_{j=1}^{z} q_j \cdot m_j \tag{1}$$

We proceed to present a formal definition of Muti-Instance Verifiable Homomorphic Linear Combination TLP (\mathcal{TLP}_{MH}) , by presenting the syntax followed by the security and correctness definitions.

4.2 Syntax of $\mathcal{TLP}_{\mathcal{MH}}$

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

• S.Setup $(1^{\lambda}) \to K_s$. It is an algorithm run by the server S. It takes a security parameter 1^{λ} . It generates $\overline{a \text{ pair } K_s := (sk_s, pk_s)}$, that includes a set of secret parameters sk_s and a set of public parameters pk_s . It returns K_s . Server S publishes pk_s .

- C.Setup $(1^{\lambda}) \to K$. It is a probabilistic algorithm run by a client C. It takes security parameter 1^{λ} as input. It returns a pair K := (sk, pk) of secret key sk and public key pk. Client C publishes pk.
- GenPuzzle $(\vec{m}, K, pk_s, \vec{\Delta}, max_{ss}) \rightarrow (\vec{o}, prm)$. It is an algorithm run by C. It takes as input a vector $\vec{m} = [m_1, \ldots, m_z]$ of plaintext solutions, key pair K, server's public parameters set pk_s , a vector $\vec{\Delta} = [\Delta_1, \ldots, \Delta_z]$ of time parameters, where each Δ_j determines the period in which m_j should remain private, and the maximum number max_{ss} of sequential steps (e.g., modular squaring) per second that the strongest solver can perform. It returns (i) a vector $\vec{o} = [\vec{o}_1, \ldots, \vec{o}_z]$, where each \vec{o}_j represents a single puzzle corresponding to m_j , and (ii) a pair prm := (sp, pp) of secret parameter sp and public parameter pp of the puzzles. Client C publishes (\vec{o}, pp).
- Evaluate $(\langle S(\vec{o}, \Delta, \max_{ss}, pp, pk, pk_s), C(\Delta, \max_{ss}, K, prm, q_1, pk_s), \dots, C(\Delta, \max_{ss}, K, prm, q_z, pk_s) \rangle) \rightarrow (\vec{g}, \vec{pp}^{(Evl)})$. It is an (interactive) algorithm run by S (and client C). When no interaction between S and \overline{C} is required, the client's inputs will be null \perp . Server S takes as input vector \vec{o} of z puzzles, time parameter Δ within which the evaluation result should remain private (where $\Delta < \min(\Delta_1, \dots, \Delta_z))$, max_{ss}, pp, pk, and pk_s. For each puzzle \vec{o}_j , client C takes as input Δ , max_{ss}, K, prm, coefficient q_j , and pk_s. The algorithm returns a puzzle vector \vec{g} , representing a single puzzle and a vector of public parameters $\vec{pp}^{(Evl)}$.
- Solve(õ, pp, g, pp^(Evt), pk, pk_s, cmd) → (m, ζ). It is a deterministic algorithm run by S. It takes as input client C's puzzle vector ö, the puzzle's public parameter pp, a vector g representing the puzzle that encodes evaluation of the client's puzzles, a vector of public parameter pp^(Evt), pk, pk_s, and a command cmd, where cmd ∈ {clientPzl, evalPzl}. When cmd = clientPzl, it solves each puzzle ö_j in õ (which is an output of GenPuzzle()), this yields a solution m_j. It appends each m_j to m. In this case, input g can be ⊥. When cmd = evalPzl, it solves puzzle g (which is an output of Evaluate()), which results in a solution m. It appends m to m. In this case, input õ can be ⊥. Depending on the value of cmd, it generates a proof vector ζ (asserting that m or m_j ∈ L_{cmd} for each solution). It outputs plaintext solution(s) m and proof ζ. Server S publishes (m, ζ).
- Verify $(m, \zeta, \vec{o_j}, pp, \vec{g}, \vec{pp}^{(Evl)}, pk_s, cmd) \rightarrow \ddot{v} \in \{0, 1\}$. It is a deterministic algorithm run by a verifier. It takes as input a plaintext solution m, proof ζ , puzzle $\vec{o_j}$, public parameters pp of $\vec{o_j}$, a puzzle \vec{g} for a linear combination of the puzzles, public parameters $\vec{pp}^{(Evl)}$ of \vec{g} , server's public key pk_s , and command cmd that determines whether the verification corresponds to C's single puzzle or linear combination of the puzzles. It returns 1 if it accepts the proof. It returns 0 otherwise.

In the above syntax, the prover is required to generate a witness/proof ζ for the language $\mathcal{L}_{cmd} = \{stm_{cmd} = (p\bar{p}, m) | R_{cmd}(stm_{cmd}, \zeta) = 1\}$, where if cmd = clientPzl, then $\bar{p}p = pp$ and if cmd = evalPzl, then $\bar{p}p = p\bar{p}^{(\text{Evl})}$.

4.3 Security Model of \mathcal{TLP}_{MH}

A $\mathcal{TLP}_{\mathcal{MH}}$ scheme must satisfy security (i.e., privacy and solution-validity), completeness, efficiency, and compactness properties. Each security property of a $\mathcal{TLP}_{\mathcal{MH}}$ 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 will provide formal definitions of the properties of $\mathcal{TLP}_{\mathcal{MH}}$.

Oracles. To allow \mathcal{A} to adaptively select plaintext solutions and corrupt parties, inspired by [1], we provide \mathcal{A} with access to oracles: puzzle generation GenPuzzle^{Oracle}() and evaluation Evaluate^{Oracle}(). Moreover, to enable \mathcal{A} to have access to the messages exchanged between the corrupt and honest parties during the execution of Evaluate(), we define an oracle called Reveal^{Oracle}(). Below, we define these oracles.

• GenPuzzle^{Oracle} $(st_{\varepsilon}, GeneratePuzzle, \vec{m}, \vec{\Delta}) \rightarrow (\vec{o}, pp)$. It is executed by the challenger \mathcal{E} . It receives a query $(GeneratePuzzle, \vec{m}, \vec{\Delta})$, where GeneratePuzzle is a string, $\vec{m} = [m_1, \ldots, m_z]$ is a vector of plaintext

solutions, and $\vec{\Delta} = [\Delta_1, \ldots, \Delta_z]$ is a vector of time parameters, where each Δ_j determines the period within which m_j should remain private. \mathcal{E} retrieves (K, pk, max_{ss}) from its state st_{ε} and then executes $\text{GenPuzzle}(\vec{m}, K, pk_s, \vec{\Delta}, max_{ss}) \rightarrow (\vec{o}, prm)$, where prm := (sp, pp). It returns (\vec{o}, pp) to \mathcal{A} .

- Evaluate^{Oracle}(st_ε, evaluate) → (g, pp^(Ev1)). It is executed interactively between the challenger *E* and the adversary to run Evaluate(⟨A(input_s), C(Δ, max_{ss}, K, prm, q₁, pk_s), ..., C(Δ, max_{ss}, K, prm, q_z, pk_s)⟩) → (g, pp^(Ev1)). The inputs of honest parties are extracted by *E* from st_ε. The adversary may provide arbitrary inputs input_s or the input of an honest S (in this case input_s is set to (o, Δ, max_{ss}, pp, pk, pk_s)) during the execution of Evaluate. *E* returns (g, pp^(Ev1)) to *A*.
- Reveal^{Oracle} $(st_{\varepsilon}, reveal^{(\text{Evl})}, \vec{g}, \vec{pp}^{(\text{Evl})}) \rightarrow transcript^{(\text{Evl})}$. It is run by \mathcal{E} which is provided with a corrupt party S and a state st_{ε} . It receives a query $(reveal^{(\text{Evl})}, \vec{g}, \vec{pp}^{(\text{Evl})})$, where $Reveal^{(\text{Evl})}$ is a string, and pair $(\vec{g}, \vec{pp}^{(\text{Evl})})$ is an output pair (previously) returned by an instance of Evaluate(). If the pair $(\vec{g}, \vec{pp}^{(\text{Evl})})$ was never generated, then the challenger sets $transcript^{(\text{Evl})}$ to \emptyset . Otherwise, the challenger retrieves from st_{ε} a set of messages that honest parties sent to the corrupt S while executing the specific instance of Evaluate(). It appends these messages to $transcript^{(\text{Evl})}$ and returns $transcript^{(\text{Evl})}$ to \mathcal{A} .

Properties. We proceed to formally define the main properties of a $\mathcal{TLP}_{\mathcal{MH}}$ scheme. Initially, we will focus on the *privacy* property. Informally, privacy states that the *j*-th solution m_j to the *j*-th puzzle \vec{o}_j must remain concealed for a predetermined period from any adversaries equipped with a polynomial number of processors.

Specifically, an adversary with a running time of $\delta(\sum_{i=1}^{j} \bar{\Delta}_{i}) < \sum_{i=1}^{j} \bar{\Delta}_{i}$ using at most $poly(Max(\bar{\Delta}_{1}, \dots, \bar{\Delta}_{j}))$

parallel processors is unable to discover a message significantly earlier than $\delta(\sum_{i=1}^{j} \bar{\Delta}_{i})$. This requirement is also applicable to the evaluation result. Specifically, an adversary with a running time of $\delta(\Delta) < \Delta$ using a polynomial number of parallel processors cannot learn the linear combination of messages significantly earlier than $\delta(\Delta)$.

To capture privacy, we define an experiment $\mathsf{Exp}_{\mathsf{prv}}^{\mathcal{A}}(1^{\lambda}, z)$, in Figure 1. This experiment involves a challenger \mathcal{E} which plays honest parties' roles and a pair of adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. This experiment considers an adversary that has access to the oracles: (i) puzzle generation $\mathsf{GenPuzzle}^{\mathsf{Oracle}}()$, (ii) evaluation $\mathsf{Evaluate}^{\mathsf{Oracle}}()$, and (iii) transcript revelation $\mathsf{Reveal}^{\mathsf{Oracle}}()$. The adversary initially outputs a key pair and a state (line 1). Next, \mathcal{E} generates a pair of secret and public keys (line 2). Given the \mathcal{E} 's public key, and having access to $\mathsf{GenPuzzle}^{\mathsf{Oracle}}()$ and $\mathsf{Evaluate}^{\mathsf{Oracle}}()$, \mathcal{A}_1 outputs z pairs of messages $[(m_{0,1}, m_{1,1}), \ldots, (m_{0,z}, m_{1,z})]$ (line 4).

After that, \mathcal{E} for each pair of messages, provided by \mathcal{A}_1 , selects a random bit b_j and generates a puzzle and related public parameter for the message with index b_j (lines 5–8). Given these puzzles and the corresponding public parameters, and having access to GenPuzzle^{Oracle}() and Evaluate^{Oracle}(), \mathcal{A}_1 outputs a state (line 9). Using this state as input, \mathcal{A}_2 guesses the value of bit b_j for its selected puzzle (line 10). The adversary wins the game (and accordingly, the experiment outputs 1) if its guess is correct (line 11).

Next, the adversary \mathcal{A}_1 and \mathcal{E} interactively execute Evaluate() (line 12). The experiment enables \mathcal{A}_1 to learn about the messages exchanged between the honest parties and the corrupt party, by providing \mathcal{A}_1 with access oracle Reveal^{Oracle} (). Having access to this transcript and the output of Evaluate(), \mathcal{A}_1 outputs a state (line 13). Given this state and the output of Evaluate(), adversary \mathcal{A}_2 guesses the value of bit b_j for its chosen puzzle (line 14). The adversary wins the game and the experiment returns 1 if its guess is correct (line 15).

Definition 4 (Privacy). A $\mathcal{TLP}_{\mathcal{MH}}$ scheme is privacy-preserving if for any security parameter λ , any plaintext input solution m_1, \ldots, m_z and coefficient q_1, \ldots, q_z (where each m_j and q_j belong to the plaintext universe U), any time interval $\overline{\Delta}_i$ between two consecutive points in time when two solutions are found, any difficulty parameter $T = \Delta_i \cdot \max_{ss}$ (where $\Delta_i \in \{D = \{\Delta_1, \ldots, \Delta_z\} \cup \{\Delta\}\}$ is the period, polynomial in λ , within which a message must remain hidden, $\Delta_j \in D$, $\Delta_j = \sum_{i=1}^j \overline{\Delta}_i$, and \max_{ss} is a constant in λ), and any polynomial-size adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$, where \mathcal{A}_1 runs in time $O(\operatorname{poly}(T, \lambda))$ and \mathcal{A}_2 runs in time

 $\delta(T) < T$ using at most poly(T) parallel processors, there exists a negligible function $\mu()$ such that for any experiment $\mathsf{Exp}_{prv}^{\mathcal{A}}(1^{\lambda}, z)$, presented in Figure 1, it holds that: $\Pr[\mathsf{Exp}_{prv}^{\mathcal{A}}(1^{\lambda}, z) \to 1] \leq \frac{1}{2} + \mu(\lambda)$.

$\mathsf{Exp}^{\mathcal{A}}_{\mathrm{prv}}(1^{\lambda}, z)$

 $\begin{array}{ll} &: \mathcal{A}_{1}(1^{\lambda}) \rightarrow (K_{S} := (sk_{S}, pk_{S}), state) \\ &: \vec{c}. \text{Setup}(1^{\lambda}) \rightarrow K := (sk, pk) \\ &: \vec{b} \leftarrow \mathbf{0} \\ &: \mathcal{A}_{1}(state, pk, \text{GenPuzzle}^{\text{Oracle}}(), \text{Evaluate}^{\text{Oracle}}()) \rightarrow \vec{m} = [(m_{0,1}, m_{1,1}), \dots, (m_{0,z}, m_{1,z})] \\ &: \text{For } j = 1, \dots, z \text{ do}: \\ &: b_{j} \stackrel{e}{\leftarrow} \{0, 1\} \\ &: \vec{b}[j] \leftarrow b_{j} \\ &: \text{GenPuzzle}(\vec{m}' = [m_{b_{1}}, \dots, m_{b_{z}}], K, pk_{S}, \vec{\Delta}, max_{ss}) \rightarrow (\vec{o}, prm) \\ &: \mathcal{A}_{1}(state, pk, \text{GenPuzzle}^{\text{Oracle}}(), \text{Evaluate}^{\text{Oracle}}(), \vec{o}, pp) \rightarrow state \\ &: \mathcal{A}_{2}(\vec{o}, pp, state) \rightarrow (b'_{j}, j) \\ &: \text{If } b'_{j} = \vec{b}[j], \text{then return 1} \\ &: \text{Evaluate}(\langle \mathcal{A}_{1}(\vec{o}, \Delta, max_{ss}, pp, pk, pk_{S}, state), C(\Delta, max_{ss}, K, prm, q_{1}, pk_{S}), \dots, C(\Delta, max_{ss}, K, prm, q_{z}, pk_{S}) \rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Ev1})}) \\ &: \mathcal{A}_{1}(state, \text{Reveal}^{\text{Oracle}}(), \vec{g}, \vec{pp}^{(\text{Ev1})}) \rightarrow state \\ &: \mathcal{A}_{2}(\vec{o}, pp, \vec{g}, \vec{pp}^{(\text{Ev1})}, state) \rightarrow (b'_{i}, i) \\ &: \text{If } b'_{i} = \vec{b}[i], \text{then return 1}, \text{else return 0} \\ \end{array}$

Fig. 1: The $\mathsf{Exp}_{prv}^{\mathcal{A}}$ experiment.

Informally, for a solution to be considered *valid*, it must be infeasible for a probabilistic polynomial time (PPT) adversary to generate an invalid solution that can still pass the verification process. This holds true whether the solution is for an individual puzzle posed by the client or for a puzzle encoding a linear combination of the client's messages. To capture solution validity, we define an experiment $\text{Exp}_{val}^{\mathcal{A}}(1^{\lambda}, z)$, presented in Figure 2. It involves \mathcal{E} which plays honest parties' roles and an adversary \mathcal{A} .

The adversary initially outputs a key pair and a state (line 1). Next, \mathcal{E} generates a pair of secret and public keys (line 2). Given the state and the public key, as well as having access to GenPuzzle^{Oracle}() and Evaluate^{Oracle}(), \mathcal{A} outputs a message vector $\vec{m} = [m_1, \ldots, m_z]$ (line 3). Next, \mathcal{E} generates a puzzle for each message that \mathcal{A} selected (line 4). The experiment allows \mathcal{A} and \mathcal{E} to interactively execute Evaluate() (line 5). Given the output of Evaluate() which is itself a puzzle, \mathcal{E} solves the puzzle and outputs the solution and associated proof (line 6).

The experiment lets \mathcal{A} learn about the messages exchanged between the corrupt party and \mathcal{E} (acting as honest parties) during the execution of Evaluate(), by providing \mathcal{A} with access to Reveal^{Oracle} (). Given this transcript, the output of Evaluate(), the plaintext solution, and the proof, \mathcal{A} outputs a solution and proof (line 7). After that, \mathcal{E} verifies the solution and proof provided by \mathcal{A} . The experiment outputs 1 and \mathcal{A} wins if \mathcal{A} convinces \mathcal{E} to accept an invalid evaluation result (line 8). \mathcal{E} solves every puzzle of the client (line 9). Given the puzzles and the solutions, and having access to oracles GenPuzzle^{Oracle}() and Evaluate^{Oracle}(), \mathcal{A} provides a solution and proof for its selected puzzle (line 10). Next, \mathcal{E} checks 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 puzzle of the client (line 11).

Definition 5 (Solution-Validity). A $\mathcal{TLP}_{\mathcal{MH}}$ scheme preserves a solution validity, if for any security parameter λ , any plaintext input solution m_1, \ldots, m_z and coefficient q_1, \ldots, q_z (where each m_j and q_j belong

to the plaintext universe U), any time interval $\bar{\Delta}_i$ between two consecutive points in time when two solutions are found, any difficulty parameter $T = \Delta_i \cdot \max_{ss}$ (where $\Delta_i \in \{D = \{\Delta_1, \ldots, \Delta_z\} \cup \{\Delta\}\}$) is the period, polynomial in λ , within which a message must remain hidden, $\Delta_j \in D$, $\Delta_j = \sum_{i=1}^j \bar{\Delta}_i$, and \max_{ss} is a constant in λ), and any polynomial-size adversary \mathcal{A} that runs in time $O(\operatorname{poly}(T, \lambda))$, there is a negligible function $\mu()$ such that for any experiment $\operatorname{Exp}_{val}^{\mathcal{A}}(1^{\lambda}, z)$ (presented in Figure 2), it holds that: $\Pr[\operatorname{Exp}_{val}^{\mathcal{A}}(1^{\lambda}, z) \to 1] \leq \mu(\lambda)$.

$$\begin{split} & \mathsf{Exp}_{\mathsf{val}}^{\mathcal{A}}(1^{\lambda}, z) \\ & 1: \mathcal{A}_{1}(1^{\lambda}) \to (K_{S} := (sk_{S}, pk_{S}), state) \\ & 2: \mathsf{C.Setup}(1^{\lambda}) \to K := (sk, pk) \\ & 3: \mathcal{A}(state, pk, \mathsf{GenPuzzle}^{Oracle}(), \mathsf{Evaluate}^{Oracle}()) \to \vec{m} = [m_{1}, \dots, m_{z}] \\ & 4: \mathsf{GenPuzzle}(\vec{m}, K, pk_{S}, \vec{\Delta}, max_{ss}) \to (\vec{o}, prm) \\ & 5: \mathsf{Evaluate}(\langle \mathcal{A}_{1}(\vec{o}, \Delta, max_{ss}, pp, pk, pk_{S}, state), C(\Delta, max_{ss}, K, prm, q_{1}, pk_{S}), \dots, C(\Delta, max_{ss}, K, prm, q_{z}, pk_{S}) \rangle) \to (\vec{g}, \vec{p} \vec{p}^{(\mathsf{Ev1})}) \\ & 6: \mathsf{Solve}(., ., \vec{g}, \vec{p} \vec{p}^{(\mathsf{Ev1})}, pk, pk_{S}, \mathsf{evalPzl}) \to (\vec{m}, \vec{\zeta}) \\ & 7: \mathcal{A}(state, pk_{S}, \mathsf{Reveal}^{\mathsf{Oracle}}(), \vec{m}, \vec{\zeta}, \vec{\sigma}, \vec{g}, pp, \vec{p} \vec{p}^{(\mathsf{Ev1})}) \to (m', \zeta') \\ & 8: \mathrm{If} \mathsf{Verify}(m', \zeta', .., g, p\vec{p}^{\mathsf{D}^{(\mathsf{Ev1})}}, pk_{S}, \mathsf{evalPzl}) \to 1, \mathrm{s.t.} m' \notin \mathcal{L}_{\mathsf{evalPzl}}, \mathrm{then \ return \ 1} \\ & 9: \mathsf{Solve}(\vec{o}, pp, .., .pk, pk_{S}, \mathrm{clientPzl}) \to (\vec{m}, \vec{\zeta}) \\ & 10: \mathcal{A}(state, pk_{S}, \mathsf{GenPuzzle}^{\mathsf{Oracle}}(), \mathsf{Evaluate}^{\mathsf{Oracle}}(), \vec{m}, \vec{\zeta}, \vec{m}, \vec{\zeta}, \vec{\sigma}, pp, p\vec{p}^{(\mathsf{Ev1})}) \to (m'_{j}, \zeta'_{j}, j) \\ & 11: \mathrm{If} \mathsf{Verify}(m'_{j}, \zeta'_{j}, \vec{o}_{j}, pp, .., .pk_{S}, \mathrm{clientPzl}) \to 1, \mathrm{s.t.} \ m'_{j} \notin \mathcal{L}_{\mathrm{clientPzl}}, \mathrm{then \ return \ 1}; \mathrm{otherwise}, \mathrm{return \ 0} \\ \end{split}$$

Fig. 2: The $\mathsf{Exp}_{\mathsf{val}}^{\mathcal{A}}(1^{\lambda}, z)$ experiment.

Informally, *completeness* examines the behavior of algorithms when honest parties are involved. It asserts that Solve() will always retrieve (1) a correct solution for a puzzle related to a linear combination and (2) a set of correct solutions for z puzzles. It also asserts that Verify() will always return 1 when given an honestly generated solution. Because algorithm Solve() is used for two cases: (a) to find a solution for a single puzzle generated by a client and (b) to discover a solution for a puzzle that encodes the linear evaluation of messages, we will state the correctness of this algorithm separately for each case. Similarly, we will state the correctness of Verify() for each scenario. In the following definitions, because the experiments' descriptions are relatively short, we integrate the experiment Exp into the probability notation. Thus, we use the notation $\Pr\left[\frac{\mathsf{Exp}}{\mathsf{Cond}}\right]$, where Exp is an experiment, and Cond represents the set of the conditions under which the property must hold.

Definition 6 (Completeness). A $\mathcal{TLP}_{\mathcal{MH}}$ is correct if for any security parameter λ , any plaintext input message m_1, \ldots, m_z and coefficient q_1, \ldots, q_z (where each m_j and q_j belong to the plaintext universe U), any difficulty parameter $T = \Delta_l \cdot \max_{ss}$ (where Δ_l is the period, polynomial in λ , within which a message must remain hidden and \max_{ss} is a constant in λ), the following conditions are satisfied.

1. Solve($\vec{o}, pp, ..., pk, pk_s, cmd$), that takes a vector \vec{o} of puzzles each encoding a plaintext solution m_j and its related parameters, always returns a vector $\vec{m} = [m_1, ..., m_z]$ of solutions:

$$Pr \begin{bmatrix} \mathsf{S}.\mathsf{Setup}(1^{\lambda}) \to K_{\scriptscriptstyle S} \\ \mathsf{C}.\mathsf{Setup}(1^{\lambda}) \to K \\ \underbrace{\mathsf{GenPuzzle}(\vec{m}, K, pk_{\scriptscriptstyle S}, \vec{\Delta}, max_{\scriptscriptstyle ss}) \to (\vec{o}, prm)}_{\mathsf{Solve}(\vec{o}, pp, .., pk, pk_{\scriptscriptstyle S}, cmd) \to (\vec{m}, .)} \end{bmatrix} = 1$$

where $pp \in prm$ and cmd = clientPzl.

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

$$Pr \begin{bmatrix} \mathsf{S}.\mathsf{Setup}(1^{\lambda}) \to K_{S} \\ \mathsf{C}.\mathsf{Setup}(1^{\lambda}) \to K \\ \mathsf{GenPuzzle}(\vec{m}, K, pk_{S}, \vec{\Delta}, max_{ss}) \to (\vec{o}, prm) \\ \mathsf{Evaluate}\left(\langle S(\vec{o}, \Delta, max_{ss}, pp, pk, pk_{S}), C(\Delta, max_{ss}, K, prm, q_{1}, pk_{S}), \dots, C(\Delta, max_{ss}, K, prm, q_{2}, pk_{S}) \rangle \right) \to (\vec{g}, \vec{p} \vec{p}^{(Evl)}) \\ \hline \mathsf{Solve}(.,., \vec{g}, \vec{p} \vec{p}^{(Evl)}, pk, pk_{S}, cmd) \to (\sum_{j=1}^{z} q_{j} \cdot m_{j}, .) \end{bmatrix} = 1$$

where $\vec{o} = [\vec{o}_1, \dots, \vec{o}_z]$, $pk \in K$, $pk_s \in K_s$, and cmd = evalPzl.

3. Verify $(m, \zeta, \vec{o_j}, pp, ..., pk_s, cmd)$, that takes a solution for a puzzle, related proof, and public parameters, always returns 1:

$$Pr \begin{bmatrix} \mathsf{S}.\mathsf{Setup}(1^{\lambda}) \to K_{s} \\ \mathsf{C}.\mathsf{Setup}(1^{\lambda}) \to K \\ \mathsf{GenPuzzle}(\vec{m}, K, pk_{s}, \vec{\Delta}, max_{ss}) \to (\vec{o}, prm) \\ \\ \underbrace{\mathsf{Solve}(\vec{o}, pp, ..., pk, pk_{s}, cmd) \to (\vec{m}, \vec{\zeta})}_{\mathsf{Verify}(m, \zeta, \vec{o_{j}}, pp, ..., pk_{s}, cmd) \to 1} \end{bmatrix} = 1$$

where cmd = clientPzl.

4. Verify(m, ζ, ., ., g, pp^(Evl), pk_s, cmd), that takes a solution for a puzzle that encodes a linear combination of z messages, related proof, and public parameters, always returns 1:

$$\Pr \begin{bmatrix} \mathsf{S}.\mathsf{Setup}(1^{\lambda}) \to K_{S} \\ \mathsf{C}.\mathsf{Setup}(1^{\lambda}) \to K \\ \mathsf{GenPuzzle}(\vec{m}, K, pk_{S}, \vec{\Delta}, max_{ss}) \to (\vec{o}, prm) \\ \mathsf{Evaluate}(\langle S(\vec{o}, \Delta, max_{ss}, pp, pk, pk_{S}), c(\Delta, max_{ss}, K, prm, q_{1}, pk_{S}), \dots, c(\Delta, max_{ss}, K, prm, q_{z}, pk_{S}) \rangle) \to (\vec{g}, \vec{p}\vec{p}^{(Evl)}) \\ \underbrace{\mathsf{Solve}(.,., \vec{g}, \vec{p}\vec{p}^{(Evl)}, pk, pk_{S}, cmd) \to (\vec{m}, \vec{\zeta}) \\ \overline{\mathsf{Verify}}(m, \zeta, .., ., \vec{g}, \vec{p}\vec{p}^{(Evl)}, pk_{S}, cmd) \to 1 \end{bmatrix} = 1$$

where cmd = evalPzl.

Now, we move on to the *efficiency* property, and along the way, we formally define the multiple instance notion for the first time. Intuitively, efficiency states that (1) the running time of dealing with multiple instances of a puzzle through multi-instance TLP is faster than via traditional TLPs, (2) Solve() returns a solution in polynomial time, i.e., polynomial in the time parameter T, (3) GenPuzzle() generates a puzzle faster than solving it, with a running time of at most logarithmic in T, and (4) the running time of Evaluate() is faster than solving any puzzle involved in the evaluation, that should be at most logarithmic in T [16].

Definition 7 (Efficiency). A $\mathcal{TLP}_{\mathcal{MH}}$ is efficient if for any security parameter λ , any plaintext input message m_1, \ldots, m_z and coefficient q_1, \ldots, q_z (where each m_j and q_j belong to the plaintext universe U), any time interval $\overline{\Delta}_i$ between two consecutive points in time when two solutions are found, any difficulty parameter $T = \Delta_i \cdot \max_{s_s}$ (where $\Delta_i \in \{D = \{\Delta_1, \ldots, \Delta_z\} \cup \{\Delta\}\}$ is the period, polynomial in λ , within which a message must remain hidden, $\Delta_j \in D$, $\Delta_j = \sum_{i=1}^{j} \overline{\Delta}_i$, and \max_{ss} is a constant in λ), the following conditions are satisfied.

- 1. <u>Multi-instance</u>: Let Ψ_{single} be the time complexity to solve a single traditional time-lock puzzle, e.g., [34]. It holds that the time complexity of a solving algorithm in traditional time-lock puzzles to deal with z puzzle is $\Psi_{trad}(z) = z \cdot \Psi_{single}$. Let $\Psi_{multi}(z)$ be the time complexity to solve z puzzles using a multiinstance algorithm. The solving algorithm of a multi-instance scheme is efficient if: $\Psi_{trad}(z) - \Psi_{multi}(z) \ge$ $poly(z, max_{ss}, \Delta_1, \dots, \Delta_z)$, for a fixed polynomial poly.
- 2. <u>Polynomial-time solving</u>: The running time of $\mathsf{Solve}(\vec{o}, pp, \vec{g}, \vec{pp}^{(Evl)}, pk, pk_s, cmd)$ is upper bounded by $\hat{poly}(z, T_{max}, \lambda)$, where $\hat{poly}()$ is a fixed polynomial and $T_{max} = Max(\Delta_1, \dots, \Delta_z, \Delta) \cdot max_{ss}$.
- 3. Faster puzzle generation: The running time of GenPuzzle($\vec{m}, K, pk_s, \vec{\Delta}, max_{ss}$) is upper bounded by $poly'(z, \log(T_{max}), \lambda)$, where poly'() is a fixed polynomial.
- 4. <u>Faster puzzles evaluation</u>: The running time of $\mathsf{Evaluate}(\langle S(\vec{o}, \Delta, max_{ss}, pp, pk, pk_s), c(\Delta, max_{ss}, K, prm, q_1, pk_s), \ldots, c(\Delta, max_{ss}, K, prm, q_z, pk_s)\rangle) \rightarrow (\vec{g}, \vec{p}^{(Evl)})$ is upper bounded by poly" $(\log(T), \lambda, \mathcal{F}^{PLC}((q_1, m_1), \ldots, (q_z, m_z)))$, where poly"() is a fixed polynomial and $\mathcal{F}^{PLC}()$ is the functionality that computes a linear combination of messages (as stated in Relation 1).

Next, we consider *compactness* which requires that the size of evaluated ciphertexts is independent of the complexity of the evaluation function \mathcal{F}^{PLC} .

Definition 8 (Compactness). A $\mathcal{TLP}_{\mathcal{MH}}$ is compact if for any security parameter λ , any difficulty parameter $T = \Delta_l \cdot \max_{ss}$, any plaintext input solution m_1, \ldots, m_z and coefficient q_1, \ldots, q_z (where each m_j and q_j belong to the plaintext universe U), any time interval $\overline{\Delta}_i$ between two consecutive points in time when two solutions are found, any difficulty parameter $T = \Delta_l \cdot \max_{ss}$ (where $\Delta_l \in \{D = \{\Delta_1, \ldots, \Delta_n\} \cup \{\Delta\}\}$ is the period, polynomial in λ , within which a message m must remain hidden, $\Delta_j \in D$, $\Delta_j = \sum_{i=1}^{j} \overline{\Delta}_i$, and \max_{ss} is a constant in λ), always Evaluate() outputs a puzzle (representation) whose bit-size is independent of \mathcal{F}^{PLC} 's complexity $O(\mathcal{F}^{PLC})$:

$$\Pr\left[\begin{array}{l} \text{S.Setup}(1^{\lambda}) \to K_{S} \\ \text{C.Setup}(1^{\lambda}) \to K \\ \text{GenPuzzle}(\vec{m}, K, pk, \vec{\Delta}, max_{ss}) \to (\vec{o}, prm) \\ \text{Evaluate}(\langle S(\vec{o}, \Delta, max_{ss}, pp, pk, pk_{S}), c(\Delta, max_{ss}, K, prm, q_{1}, pk_{S}), \dots, c(\Delta, max_{ss}, K, prm, q_{z}, pk_{S}) \rangle) \to (\vec{g}, \vec{pp}^{(Evl)}) \\ s.t. \\ ||\vec{g}|| = poly\Big(\lambda, ||\mathcal{F}^{PLC}\big((q_{1}, m_{1}), \dots, (q_{z}, m_{z})\big)||\Big) \\ \end{array} \right] = 1$$

Definition 9 (Security). A TLP_{MH} is secure if it satisfies privacy and solution validity as outlined in Definitions 4 and 5.

5 Construction of \mathcal{TLP}_{MH} : Multi-instance Verifiable Partially Homomorphic TLP (MH-TLP)

5.1 An Overview

We develop MH-TLP on top of Tempora-Fusion, discussed in Section 3.8. Inspired by previous multi-instance TLPs proposed in [3], we use the chaining technique, to chain different puzzles, such that when S solves

one puzzle it will obtain enough information to work on the next puzzle. Therefore, it can solve puzzles *sequentially* rather than dealing with all of them at once. However, we develop a new technique for chaining puzzles to support homomorphic linear combinations as well.

In the multi-instance TLPs introduced in [3], during the puzzle generation phase (and chaining process) the base r_{j+1} for the (j + 1)-th puzzle is concatenated and concealed with the *j*-th solution, m_j . Consequently, the *j*-th puzzle is created on solution $m_j ||r_{j+1}$. The issue with this approach is that it cannot support homomorphic operations on the puzzle's actual solution m_j , because the solution is now $m_j ||r_{j+1}$.

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. We briefly explain how it works. 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, (similar to Tempora-Fusion) client C picks a random base r_j , sets $a_j = 2^{T_j} \mod \phi(N)$, and then sets master key mk_j as $mk_j = r_j^{a_j} \mod N$, with a difference being T_j now determines how long a solution m_j must remain concealed after the previous solution m_{j-1} is discovered. Next, as in Tempora-Fusion, C derives some pseudorandom values from mk_j and encrypts the *y*-coordinates of a polynomial representing m_j .

However, when j > 1, client C derives a fresh base r_j from the previous master key as $r_j = \text{PRF}(j||0, mk_{j-1})$. As before, it sets $a_j = 2^{T_j} \mod \phi(N)$ and then sets the current master key mk_j as $mk_j = r_j^{a_j} \mod N$. 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.

Clearly, this new approach does not require C to modify each solution m_j . Thus, we can now use the techniques developed for Tempora-Fusion in [1] to allow S to perform homomorphic linear combinations on the puzzles. In this setting, S has to solve puzzles in ascending order, starting from the first puzzle, to find a solution and a base for the next puzzle. Given this base, it starts repeated modular squaring to find the next solution and base until it finds the last puzzle's solution.

We proceed to briefly explain how MH-TLP operates. We highlight that there are overlaps between the design of MH-TLP and Tempora-Fusion. However, there are major differences as well. We will shortly discuss these differences.

- 1. Server-Side Setup. Initially, S generates and publishes a set of public parameters, without requiring it to generate any secret parameter. The public parameters include a sufficiently large prime number p and a vector $\vec{x} = [x_1, x_2, x_3]$ of non-zero elements. The elements in \vec{x} can be considered as x-coordinates and will help each client to represent its message as a polynomial in point-value form.
- 2. Client-side Key Generation. A client C independently generates a set of public and private keys, which includes an RSA-public modulus N. The client publishes its public key.
- 3. Client-side Puzzle Generation. Client C uses its secret key and time parameter T_j that determines how long a solution m_j must remain concealed after the previous solution m_{j-1} is discovered, to generate a set of master keys mk_1, \ldots, mk_z , using the chaining technique described above. C uses each master key mk_j , that is associated with the *j*-th message, to derive pseudorandom values $(z_{i,j}, w_{i,j})$ for each element x_i of \vec{x} . The client represents its secret solution m_j as (a constant term of) a polynomial and then represents the polynomial in the point-value form. This results in a vector of y-coordinates: $[\pi_{1,j}, \pi_{2,j}, \pi_{3,j}]$. It encrypts each y-coordinate using the related pseudorandom values: $o_{i,j} = w_{i,j} \cdot (\pi_{i,j} + z_{i,j}) \mod p$. Each vector of encrypted y-coordinates $\vec{o}_j = [o_{1,j}, o_{2,j}, o_{3,j}]$ represents a puzzle for *j*-th solution m_j . Note that polynomial representation is used to allow a homomorphic linear combination of solutions and efficient verification of the computation. C publishes these puzzles along with a set of public parameters. Given these public parameters, by solving the related puzzle, anyone can sequentially find each master key mk_j and remove the blinding factors to extract the corresponding solution. To support the public verifiability of a solution that will be found by S, C commits $com_j = \text{Com}(m_j, mk_j)$ to each message m_j using the related mk_j as

the commitment's randomness. It publishes each com_j . Anyone who solves a puzzle encoding solution m_j can find mk_j and prove (m_j, mk_j) matches com_j .

4. Linear Combination. In this phase, similar to Tempora-Fusion, client C produces certain messages that allow server S to find a linear combination of its plaintext solutions after a certain time. C, 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 and public parameters $pp^{(Ev1)}$. Anyone who solves a puzzle for the computation will be able to find tk, after a certain time. C uses tk to derive new pseudorandom values (z'_i, w'_i) for each element x_i of \vec{x} . It also uses its secret key to regenerate the pseudorandom values $(z_{i,j}, w_{i,j})$ used to encrypt each y-coordinate related to its solution m_j . The client selects a single random root: root. It commits to this root, using tk as the randomness: com' = Com(root, tk). C represents root as a polynomial in point-value form. This yields a vector of y-coordinates: $[\gamma_1, \gamma_2, \gamma_3]$. The client will insert this random root into its outsourced puzzle, to give a certain structure to the result of the homomorphic linear combination, which will ultimately allow future verification.

For each solution m_2, \ldots, m_z it possesses, the client selects a fresh key f_i . This key is used by the client to generate zero-sum pseudorandom values for (each y-coordinate related to) each solution. These values are generated such that if are summed, they will cancel out each other. They are used to ensure that S learns only the linear combination of all the solutions that belong to C, and S cannot learn the combination of a subset of these solutions. C for each y-coordinate of each j-th puzzle participates in an instance of OLE^+ with S. At a high level, C's input includes the y-coordinate γ_i of the random root, the new pseudorandom values (z'_i, w'_i) , the inverse of the old pseudorandom values $(w_{i,j})^{-1}$, its coefficient q_j , and the pseudorandom values $y_{i,j}$ derived from f_j . The input of S is the client's j-th puzzle. Each instance of OLE^+ returns to S an encrypted y-coordinate. At the end of this process, C publishes its public parameters $pp^{(\text{Evl})}$.

Consequently, S sums the outputs of OLE^+ component-wise, resulting in a vector of encrypted y-coordinates, $\vec{g} = [g_1, g_2, g_3]$. Then, S publishes \vec{g} . Each g_i has a layer of blinding factor, inserted by C during the invocation of OLE^+ . Anyone who solves the related puzzle can regenerate the temporary master key tk and remove the blinding factor.

5. Solving a Puzzle. To find the solution encoding the result of the computation, S operates as follows. Given public parameters previously released by C, S solves the related puzzle to find the temporary master key tk. Using tk, S removes the blinding factors from each g_i , that results in three plaintext y-coordinates. S uses these y-coordinates and the x-coordinates in \vec{x} to interpolate a polynomial $\theta(x)$. It finds the roots of $\theta(x)$. It publishes the root and tk that match the published commitment com'. It also retrieves from $\theta(x)$ and publishes the linear combination of C's solutions.

To find a solution for a single puzzle of C, S operates as follows. Given public parameters and each puzzle, S after a certain time retrieves a master key mk_j . Using mk_j , S removes the blinding factors from each $o_{i,j}$, that yields a vector of y-coordinates. It uses them and x-coordinates in \vec{x} to interpolate a polynomial $\pi_j(x)$ and retrieves message m_j from polynomial $\pi_j(x)$. It publishes m_j and mk_j that match the published commitment com_j . Moreover, solving the j-th puzzle provides S with enough information to begin working on the next puzzle.

6. Verification. To verify a solution related to the linear combination of solutions, the verifier checks if the root root and the temporary master key tk provided by S match the commitment com'. It also removes the blinding factors from $[g_1, g_2, g_3]$, yielding three y-coordinates. It uses these y-coordinates to interpolate a polynomial. From this polynomial, the verifier retrives its constant term (which is the computation result). It checks whether root is a root of this polynomial and the computation result matches the one S published. If all the checks pass, it accepts the result. To verify a solution m_j related to a single puzzle of C, the verifier checks if m_j and the temporary master key mk_j , provided by S, match the commitment com_j .

Beyond offering the multi-instance feature, MH-TLP differs from Tempora-Fusion in two main ways. Firstly, since there is only one client, only one random root is selected and inserted into the outsourced polynomials (rather than inserting $|\mathcal{I}|$ roots in Tempora-Fusion). During the verification, a verifier checks whether the evaluation of the resulting polynomial at this root is zero. Secondly, in MH-TLP, using only three *x*-coordinates $\vec{x} = [x_1, x_2, x_3]$ will suffice, because the outsourced puzzle's degree is one, and when it is multiplied by a polynomial representing a random root (while computing the linear combination), the resulting polynomial's degree will become two. Thus, three (y, x)-coordinates are sufficient to interpolate a polynomial of degree two. Figure 3 outlines the MH-TLP workflow.



Fig. 3: MH-TLP Workflow Outline.

5.2 Detailed Construction

Next, we present a detailed description of MH-TLP.

1. Setup. S.Setup $(1^{\lambda}) \rightarrow (., pk_s)$

Server S only once takes the following steps:

- (a) <u>Setting a field's parameter</u>: generates a sufficiently large prime number p, where $\log_2(p)$ is the security parameter.
- (b) Generating public x-coordinates: sets $\vec{x} = [x_1, x_2, x_3]$, where $x_i \neq x_j, x_i \neq 0$, and $x_i \notin U$.
- (c) Publishing public parameters: publishes $pk_s = (p, \vec{x})$. Note that, C itself can generate (p, \vec{x}) , too.
- 2. Key Generation. C.Setup $(1^{\lambda}) \to K$

Client C takes the flowing steps.

- (a) <u>Generating RSA public and private keys</u>: computes $N = p_1 \cdot p_2$, where p_i is a large randomly chosen prime number, e.g., $\log_2(p_i) \ge 2048$. 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 $sk = \phi(N)$ and publishes public key pk = N.
- 3. **Puzzle Generation**. GenPuzzle $(\vec{m}, K, pk_s, \vec{\Delta}, max_{ss}) \rightarrow (\vec{o}, prm)$

Client *C* takes the following steps to generate *z* puzzles for messages $\vec{m} = [m_1, \ldots, m_z]$. It wants *S* to learn each message m_j at time $time_j \in time$, where $time = [time_1, \ldots, time_z]$, $\bar{\Delta}_j = time_j - time_{j-1}$, $\vec{\Delta} = [\bar{\Delta}_1, \ldots, \bar{\Delta}_z]$, and $1 \leq j \leq z$.

- (a) <u>Checking public parameters</u>: checks the bit-size of p and elements of \vec{x} in pk_s , to ensure $\log_2(p) \ge 128$, $x_i \ne x_j, x_i \ne 0$, and $x_i \notin U$. If it does not accept the parameters, it returns (\bot, \bot) and does not take further action.
- (b) <u>Generating secret keys</u>: generates a vector of master keys $\vec{mk} = [mk_1, \dots, mk_z]$ and two secret keys $\vec{k_i}$ and $\vec{s_i}$ for each $m\vec{k_i}$ in \vec{mk} as follows. It constructs an empty vector \vec{mk} . Then, it
 - i. sets each exponent a_i .

$$\forall j, 1 \le j \le z : \quad a_j = 2^{T_j} \mod \phi(N)$$

where $T_j = max_{ss} \cdot \bar{\Delta}_j$ is the total number of squaring needed to decrypt an encrypted solution m_i after previous solution m_{i-1} is revealed.

- ii. computes each master key mk_j as follows. For every j, where $1 \le j \le z$:
 - when j = 1:
 - A. selects a uniformly random base $r_j \stackrel{\hspace{0.1em}\mathsf{\scriptscriptstyle\$}}{\leftarrow} \mathbb{Z}_N$.
 - B. sets key mk_j as $mk_j = r_j^{a_j} \mod N$.
 - C. appends mk_i to \vec{mk} .
 - when j > 1:

A. derives a fresh base r_j from the previous master key as $r_j = \text{PRF}(j||0, mk_{j-1})$.

- B. sets key mk_j as $mk_j = r_j^{a_j} \mod N$.
- C. appends mk_i 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_i and s_j .

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

- (d) Encoding plaintext message:
 - i. represents each plaintext solution m_j as a polynomial, such that the polynomial's constant term is the message.

$$\forall j, 1 \leq j \leq z : \quad \boldsymbol{\pi}_j(x) = x + m_j \mod p$$

ii. computes three y-coordinates of each $\pi_j(x)$.

 $\forall j, 1 \leq j \leq z \quad ext{and} \quad \forall i, 1 \leq i \leq 3: \quad \pi_{i,j} = \boldsymbol{\pi}_j(x_i) ext{ mod } p$

where $x_i \in \vec{x}$.

(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:

$$\forall j, 1 \le j \le z: \quad com_j = \operatorname{Com}(m_j, mk_j)$$

Let $\vec{com} = [com_1, \dots, com_z].$

- (g) <u>Managing messages</u>: publishes $\vec{o} = \left[[o_{1,1}, o_{2,1}, o_{3,1}], \dots, [o_{1,z}, o_{2,z}, o_{3,z}] \right]$ and $pp = (c\vec{om}, r_1)$. It locally keeps secret parameters $sp = \vec{mk}$. It sets prm = (sp, pp). It deletes everything else, including $m_j, \pi_j(x), \pi_{1,j}, \pi_{2,j}$, and $\pi_{3,j}$ for every $j, 1 \leq j \leq z$.
- 4. Linear Combination. Evaluate $(\langle S(\vec{o}, \Delta, max_{ss}, pp, pk, pk_s), C(\Delta, max_{ss}, K, prm, q_1, pk_s), \dots, C(\Delta, max_{ss}, K, prm, q_2, pk_s))) \rightarrow (\vec{g}, pp^{(\text{Evl})})$

In this phase, client C produces certain messages that allow S to find a linear combination of its plaintext solutions after time Δ .

- (a) Granting the computation: client C takes the following steps.
 - i. <u>Generating temporary secret keys</u>: generates a temporary master key tk and two secret keys k' and s'. Moreover, it generates z 1 secret key $[f_j, \ldots, f_z]$. To do that, the following steps are taken: C computes the exponent:

$$b = 2^Y \mod \phi(N)$$

where $Y = \Delta \cdot max_{ss}$. It selects a base uniformly at random: $h \stackrel{\$}{\leftarrow} \mathbb{Z}_N$ and then sets a temporary master key tk:

$$tk = h^b \mod N$$

It derives two keys from tk:

$$k' = \Pr(1, tk), \quad s' = \Pr(2, tk)$$

It picks fresh z-1 random keys $\vec{f} = [f_2, \ldots, f_z]$, where $f_i \stackrel{\$}{\leftarrow} \{0, 1\}^{poly(\lambda)}$.

ii. <u>Generating blinding factors</u>: 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.

$$\forall j, 1 \le j \le z: \quad k_i = \mathsf{PRF}(1, mk_i), \quad s_i = \mathsf{PRF}(2, mk_i)$$

Then, it regenerates pseudorandom values, by using k_i and s_i .

 $\forall j, 1 \leq j \leq z \quad \text{and} \quad \forall i, 1 \leq i \leq 3: \quad z_{i,j} = \mathtt{PRF}(i,k_j), \quad w_{i,j} = \mathtt{PRF}(i,s_j)$

It also generates new pseudorandom values using keys (k', s'):

$$\forall i, 1 \le i \le 3: \quad z'_i = \mathsf{PRF}(i, k'), \quad w'_i = \mathsf{PRF}(i, s')$$

It computes new sets of (zero-sum) blinding factors, using the keys in f, as follows. $\forall j, 1 \leq j \leq z$:

• if j = 1:

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

• if j > 1:

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

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

iii. <u>Generating y-coordinates of a random root</u>: 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 $\gamma(x)$ as:

$$\gamma(x) = x - root \mod p$$

Then, it computes three *y*-coordinates of $\gamma(x)$:

$$\forall i, 1 \leq i \leq 3 : \quad \gamma_i = \gamma(x_i) \mod p$$

- iv. Committing to the root: computes com' = Com(root, tk).
- v. <u>Re-encoding outsourced puzzle</u>: participates in an instance of OLE^+ with S, for every j-th puzzle and every i, where $1 \le j \le z$ and $1 \le i \le 3$. The inputs of client C to the i-th instance of OLE^+ are:

$$e_{i,j} = \gamma_i \cdot q_j \cdot w'_i \cdot (w_{i,j})^{-1} \bmod p, \qquad e'_{i,j} = -(\gamma_i \cdot q_j \cdot w'_i \cdot z_{i,j}) + z'_i + y_{i,j} \bmod p$$

The input of S to the (i, j)-th instance of OLE^+ is corresponding encrypted y-coordinate: $e''_{i,j} = o_{i,j}$. Accordingly, the (i, j)-th instance of OLE^+ returns to 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 $pp^{(Evl)} = (h, com')$.
- (b) <u>Computing encrypted linear combination</u>: Server S sums all of the outputs of OLE^+ instances that it has invoked as follows. $\forall i, 1 \le i \le 3$:

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

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: server S publishes $\vec{g} = [g_1, g_2, g_3]$.
- 5. Solving a Puzzle. Solve $(\vec{o}, pp, \vec{g}, \vec{pp}^{(\text{Evl})}, pk, pk_s, cmd) \rightarrow (\vec{m}, \vec{\zeta})$

Server 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 $tk = h^{2^{Y}} \mod N$, via repeated squaring of $h \mod N$.

ii. derives two keys from tk:

$$k' = \Pr(1, tk), \quad s' = \Pr(2, tk)$$

(b) <u>Removing blinding factors</u>: removes the blinding factors from [g₁, g₂, g₃] ∈ g. ∀i, 1 ≤ i ≤ 3:

$$\theta_{i} = \underbrace{\left(\mathsf{PRF}(i,s')\right)^{-1}}_{(w_{i}')^{-1}} \cdot \left(g_{i} - \overbrace{\mathsf{PRF}(i,k')}^{z_{i}'}\right) \mod p$$
$$= \gamma_{i} \cdot \sum_{j=1}^{z} q_{j} \cdot \pi_{i,j} \mod p$$

(c) Extracting a polynomial: interpolates a polynomial $\boldsymbol{\theta}$, given pairs $(x_1, \theta_1), (x_2, \theta_2), (x_3, \theta_3)$. Note that $\boldsymbol{\theta}$ will have the form:

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

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

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

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

(d) <u>Extracting the linear combination</u>: 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:

$$m = cons \cdot (-root)^{-1} \mod p$$

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

- (e) <u>Extracting valid roots</u>: 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 $\operatorname{Ver}(com', (root, tk)) = 1$.
- (f) <u>Publishing the result</u>: initiates vectors \vec{m} and $\vec{\zeta}$. It appends m to \vec{m} and (root, tk) to $\vec{\zeta}$. It publishes \vec{m} and $\vec{\zeta}$.
- Case 2. when solving each *j*-th puzzle $\vec{o}_{j,u}$ of client C_u (i.e., when cmd = clientPzl), server S takes the following steps. $\forall j, 1 \leq j \leq z$:
 - (a) Finding secret bases and keys: sets base 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 $mk_1 = r_1^{2^{T_1}} \mod N$ through repeated squaring of r_1 modulo N. It also initiates vectors \vec{m} and $\vec{\zeta}$.

• if j > 1: computes base r_j as $r_j = \text{PRF}(j||0, mk_{j-1})$. Next, it finds mk_j where $mk_j = r_j^{2^{T_j}} \mod N$ through repeated squaring of r_j modulo N.

It derive two keys from mk_i :

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

(b) Removing blinding factors: re-generates six pseudorandom values using k_j and s_j :

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

Next, it uses the blinding factors to unblind $\vec{o}_j = [o_{1,j}, o_{2,j}, o_{3,j}]$:

$$\forall i, 1 \le i \le 3: \quad \pi_{i,j} = ((w_{i,j})^{-1} \cdot o_{i,j}) - z_{i,j} \mod p$$

- (c) <u>Extracting a polynomial</u>: interpolates a polynomial π_j , given pairs $(x_1, \pi_{1,j}), (x_2, \pi_{2,j}), (x_3, \pi_{3,j})$.
- (d) <u>Publishing the solution</u>: considers the constant term of π_j as the plaintext message, m_j . It appends (m_j, j) to \vec{m} and mk_j to $\vec{\zeta}$. If j = z, then it publishes \vec{m} and $\vec{\zeta}$.
- 6. Verification. 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) <u>Checking the commitment's opening</u>: verify the validity of $(root, tk) \in \zeta$, provided by S in step 5f of Case 1:

$$\operatorname{Ver}(com', (root, tk)) = 1$$

If the verification passes, it proceeds to the next step. Otherwise, it returns $\ddot{v} = 0$ and takes no further action.

- (b) <u>Checking the resulting polynomial's valid roots</u>: checks if the resulting polynomial contains the root root in ζ , by taking the following steps.
 - i. derives two keys from tk:

$$k' = \Pr(1, tk), \quad s' = \Pr(2, tk)$$

ii. removes the blinding factors from $\vec{g} = [g_1, g_2, g_3]$ that were provided by server S in step 4c. Specifically, for every $i, 1 \le i \le 3$:

$$\theta_{i} = \underbrace{\left(\mathsf{PRF}(i, s')\right)^{-1}}_{(w_{i}')^{-1}} \cdot \left(g_{i} - \overbrace{\mathsf{PRF}(i, k')}^{z_{i}'}\right) \mod p$$
$$= \gamma_{i} \cdot \sum_{j=1}^{z} q_{j} \cdot \pi_{i,j} \mod p$$

iii. interpolates a polynomial θ , given $(x_1, \theta_1), (x_2, \theta_2), (x_3, \theta_3)$. Note that θ will have the form:

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

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

- iv. checks if *root* is a root of $\theta(x)$, by evaluating $\theta(x)$ at *root* and checking if the result is 0, i.e., $\theta(root) \stackrel{?}{=} 0$. It proceeds to the next step if the check passes. It returns $\ddot{v} = 0$ and takes no further action, otherwise.
- (c) <u>Checking the final result</u>: retrieves the result (which is the linear combination of m_1, \ldots, m_z) from polynomial $\boldsymbol{\theta}(x)$'s constant term: $t = -root \cdot \sum_{j=1}^{z} q_j \cdot m_j$ as follows:

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

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

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

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

Case 2. when verifying the j-th solution of a single puzzle belonging to client C:

(a) <u>Checking the commitment' opening</u>: checks whether opening $m_j \in m$ and $mk_j \in \zeta$, given by S in step 5d of Case 2, matches the commitment:

$$\texttt{Ver}(com_i, (m_i, mk_i)) \stackrel{\scriptscriptstyle t}{=} 1$$

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

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.

Theorem 2. If the sequential modular squaring assumption holds, factoring N is a hard problem, PRF, OLE^+ , and the commitment schemes are secure, then MH-TLP presented above is secure, regarding Definition 9.

Theorem 3. The MH-TLP protocol presented above meets completeness, efficiency, and compactness, regarding Definitions 6, 7, and 8 respectively.

The remainder of this section provides proof for Thereoms 2 and 3.

5.3 Proof of Theorem 2

Proof (sketch). There will be a significant overlap between the proofs of Theorems 2 and Tempora-Fusion (i.e., Theorem 5 in Appendix B). The proof of Theorem 2 differs from that of Theorem 5 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 $1 \leq j \leq z$. 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 $r_{j'} = \text{PRF}(j'||0, mk_{j'-1})$, for any j', where $2 \leq j' \leq z$. 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 $mk_{j'-1}$.

5.4 Proof of Theorem 3

Proof. We begin with proving the completeness of MH-TLP.

Lemma 1. MH-TLP satisfies completeness, regarding Definition 6.

We start by addressing Case 1, which involves solving a single puzzle. Due to the correctness and deterministic nature of the original TLP [34], particularly through repeated squaring, a server can consistently derive the master key mk_j through a fixed number of repeated squaring. Furthermore, the correctness and deterministic property of **PRF** ensures that the server can extract r_j and subsequently compute keys k_j , s_j and blinding factors $z_{i,j}$ and $w_{i,j}$. Given these blinding factors, the server can remove them from each $o_{i,j}$, resulting in a set of *y*-coordinates. Due to the correctness (especially deterministic nature) of interpolation algorithms, such as Lagrange interpolation, the server can recover the identical polynomial that the client initially constructed, i.e., polynomial $\pi_j(x) = x + m_j \mod p$. Given, each polynomial $\pi_j(x)$, the server can easily recover its constant term m_j , which is the solution to the related puzzle.

Case 2 focuses on the correctness of solving a puzzle related to the linear combination. Similar to the above case, due to the correctness and deterministic nature of repeated squaring and PRF, the server can find the temporary key tk and accordingly discover keys k' and s'. These keys allow the server to remove the blinding factors from each masked y-coordinate g_i , yielding three y-coordinates θ_1, θ_2 , and θ_3 . Because the interpolation algorithm is deterministic, the server will recover a polynomial of the form $\boldsymbol{\theta}(x) = (x - root) \cdot \sum_{j=1}^{z} q_j \cdot (x + m_j) \mod p$. The main reasons that polynomial $\boldsymbol{\theta}(x)$ maintains this form are (1) the correctness of OLE^+ , (2) the properties of polynomial arithmetic, as explained in Section 3.4, and (3) the product of multiple polynomials preserves each individual polynomial's roots. Given $\boldsymbol{\theta}(x)$, one can easily retrieve its constant term and multiply it by $(-root)^{-1}$ which yields the linear combination of solutions: $\sum_{i=1}^{z} q_i \cdot m_j$.

We proceed to Case 3, which pertains to the correctness of verification for a single puzzle. Building on the argument presented in Case 1, the server can always retrieve the master key mk_j and the related solution m_j . Assuming the commitment verification algorithm is correct, an honest server's proof (mk_j, m_j) is always accepted by an honest verifier.

Case 4 considers the correctness of verification for the linear combination. As discussed in Case 2, an honest server can always find the temporary key tk and the keys derived from it k' and s'. Given these keys, it can find the root *root*. Due to the correctness of the commitment's verification algorithm, proof (*root*, tk) is always accepted by an honest verifier. Moreover, due to the correctness and deterministic nature of PRF, a verifier can derive from tk the same keys (k', s') as the client used to blind its *y*-coordinates. These keys allow the verifier to unblind the *y*-coordinates to obtain θ_1, θ_2 , and θ_3 . Because the interpolation algorithm is deterministic, the verifier will recover a polynomial of the form $\boldsymbol{\theta}(x) = (x - root) \cdot \sum_{j=1}^{z} q_j \cdot (x + m_j) \mod p$. Also, evaluating polynomial $\boldsymbol{\theta}(x)$ at *root* will always result in 0, because *root* is a root of $\boldsymbol{\theta}(x)$. Given *root* and the constant term *t* of $\boldsymbol{\theta}(x)$, the verifier can always extract the linear combination of messages $-t \cdot root^{-1} \mod p = \sum_{j=1}^{z} q_j \cdot m_j$, which will be equal to the result $\sum_{j=1}^{z} q_j \cdot m_j$ that the prover sends.

Lemma 2. MH-TLP satisfies efficiency, regarding Definition 7.

We will initially focus on Condition 1: multi-instance. The time complexity of (traditional and multi-instance) TLPs is dominated by the number of modular squaring. When a traditional TLP encounters z instances of a puzzle at once, it must deal with each puzzle individually. Therefore, in this setting, its time complexity is $\Psi_{trad}(z) = max_{ss} \cdot \sum_{j=1}^{z} \Delta_{j}$. However, in MH-TLP, the puzzles are solved sequentially, resulting in the time complexity of $\Psi_{multi}(z) = max_{ss} \cdot (\Delta_1 + \sum_{j=2}^{z} (\Delta_j - \Delta_{j-1}))$. Therefore, the difference between the time complexity of the TLPs in these two settings is:

$$\Psi_{trad}(z) - \Psi_{multi}(z) = max_{ss} \cdot \left(\left(\sum_{j=1}^{z} \Delta_j \right) - \left(\Delta_1 + \sum_{j=2}^{z} (\Delta_j - \Delta_{j-1}) \right) \right)$$
$$= max_{ss} \cdot \sum_{j=1}^{z-1} \Delta_j$$

Thus, $\Psi_{trad}(z) - \Psi_{multi}(z) = poly(z, max_{ss}, \Delta_1, \dots, \Delta_z)$, for a fixed polynomial poly, meeting the criteria set out in Condition 1.

We proceed to Condition 2: polynomial-time solving. The core primitive upon which MH-TLP and in particular the algorithm Solve() relies to solve a puzzle is the standard sequential squaring. The complexity of Solve() is $max_{ss} \cdot \left(\Delta_1 + \sum_{j=2}^{z} (\Delta_j - \Delta_{j-1})\right)$, which itself can be represented as $poly(z, T_{max}, \log(N))$, where polyis a fixed polynomial and $T_{max} = max_{ss} \cdot \Delta_z$.

We move on to Condition 3: faster puzzle generation property. The main operation in the algorithm GenPuzzle() that generates puzzles is generating each value $mk_j = r_j^{a_j} \mod N$, where $a_j = 2^{T_j} \mod \phi(N)$. The complexity of generating each a_j is $O(\log_2(T) \cdot \log_2^2(\phi(N))) \approx O(\log_2(T) \cdot \log_2^2(N))$, while the complexity of generating each mk_j is $O(\log_2(a_j) \cdot \log_2^2(N)) \approx O(\log_2(N) \cdot \log_2^2(N))$. Thus, the total complexity is $O((\log_2(T) + \log_2(N)) \cdot z \cdot \log_2^2(N))$, which can be represented as $poly'(z, \log(T), \log(N))$, where poly' is a fixed polynomial and T is the maximum time paramter.

Next, we turn our attention to Condition 4: faster puzzle evaluation. The Evaluate() algorithm involves generating a temporary key $tk = h^b \mod (N)$, where $b = 2^{\gamma} \mod \phi(N)$. Based on the above analysis, these operations' total complexity is $O((\log_2(T) + \log_2(N)) \cdot \log_2^2(N))$, where $T = \Delta \cdot max_{ss}$. The Evaluate() algorithm also involves operations to compute a linear combination of solutions m_1, \ldots, m_z using the coefficients q_1, \ldots, q_z to realize the functionality \mathcal{F}^{PLC} . The operation to complete the linear combination is linear with the total number of puzzles. Hence, the complexity of Evaluate() can be represented as $poly''(\log(T), \log(N), \mathcal{F}^{PLC}((q_1, m_1), \ldots, (q_z, m_z))$, for a fixed polynomial poly''.

Lemma 3. MH-TLP satisfies compactness, regarding Definition 8.

The algorithm Evaluate() outputs a vector of three elements as a puzzle $\vec{g} = [g_1, g_2, g_3]$ along with a small set of public parameters $\vec{pp}^{(\text{Evl})}$. The bit size of each element g_i of the puzzle vector \vec{g} is $\log_2(p)$. Thus, the bit size of \vec{g} can be represented as $||\vec{g}|| = poly(\log(p), ||\mathcal{F}^{\text{PLC}}((q_1, m_1), \dots, (q_z, m_z))||)$, for a fixed polynomial poly.

This concludes the proof of Theorem 3 as we have proved the completeness (Lemma 1), efficiency (Lemma 2), and compactness (Lemma 3). \Box

6 Multi-Instance Multi-Client Partially Homomorphic TLP

In this section, we present Multi-Instance Multi-Client Partially Homomorphic TLP (MMH-TLP), a protocol that can be considered as a generalization of MH-TLP, presented in Section 5. MMH-TLP is built upon

MH-TLP and the Tempora-Fusion introduced in [1]. It offers the features of both schemes within one unified protocol. MMH-TLP will (i) allow a client to generate multi-puzzles such that the server can solve them sequentially, (ii) enable the server to homomorphically compute a linear combination of puzzles of a single client, (iii) allow the server to homomorphically compute a linear combination of puzzles of different clients, and (iv) enable anyone to verify the correctness of each puzzle's solution and computations' outputs.

6.1 An Overview

At a high level, the protocol works as follows. Initially, S generates and publishes a set of public parameters, including vector \vec{x} and a sufficiently large prime number p. Each client independently generates its secret and public keys. It publishes the public key. In the puzzle generation phase, each client, possessing a set of solutions, creates puzzles for these solutions using the chaining technique described in Section 5. Each client then publishes the puzzles along with some public parameters.

To enable S to learn a homomorphic linear combination of messages (encoded into the published puzzles) belonging to a single client, the client engages with S through the interactive algorithm $\mathsf{Evaluate}_{sc}()$. Following the execution of this algorithm, the client publishes a set of public parameters, and S releases a puzzle encoding the computation result.

To facilitate S learning a homomorphic linear combination of messages, where each message (encoded into a published puzzle) originates from a different client, the clients interact with S using algorithm $\mathsf{Evaluate}_{mc}()$. Upon completing this algorithm, the clients publish a set of public parameters, and S publishes a puzzle encoding the computation result. After a certain period, S solves a puzzle, related to (i) the linear combination of a single client's solutions, (ii) the linear combination of multiple clients' solutions, or (iii) a single client's solution. S then publishes the solution and the corresponding proof. Given the public parameters and the solution, anyone can verify the proof. Figure 4 illustrates the MMH-TLP workflow.

Regarding system design, MMH-TLP differs from MH-TLP (and Tempora-Fusion) in several ways. We briefly outline the differences.

- 1. In MMH-TLP, the server needs to generate \bar{t} x-coordinates (instead of generating only 3 x-coordinates in MH-TLP) and each client should use these \bar{t} x-coordinates, for the following reason. In MMH-TLP, each client's outsourced polynomial (that represents its puzzle) is of degree 1. During Phase 5 (Linear Combination for Multiple Clients), this polynomial is multiplied by \ddot{t} polynomials each representing a random root and is of degree 1. The resulting polynomial will have degree $\ddot{t} + 1$. Therefore, at least $\bar{t} = \ddot{t} + 2 (y, x)$ -coordinate pairs are needed to interpolate a polynomial encoding a linear combination of the solutions.
- 2. There will be two different algorithms for the linear combination, one algorithm, called $\mathsf{Evaluate}_{sc}$, to perform a linear combination of a single client's puzzles, and another one, called $\mathsf{Evaluate}_{mc}$, to perform a linear combination of n different clients' puzzles.
- 3. During the linear combination of different clients' puzzles, $\mathsf{Evaluate}_{mc}()$ takes a new input id_u for each client C_u . This approach allows each C_u to specify which one of its outsourced puzzles must be used as an input to the computation.
- 4. Algorithms Solve() and Verify() take a new input string $cnd \in \{\text{SingleClient}, \text{MultiClient}\}\$ that specifies whether the encrypted linear combination \vec{g} is the result of a homomorphic linear combination of messages belonging to a single client or multiple clients. Moreover, now, these two algorithms consider three different cases (rather than two); namely, (i) when solving or verifying a puzzle related to the linear combination of messages belong to the same client, (ii) when solving or verifying a puzzle related to the linear combination of messages, where each message belong to a different client, and (iii) when solving or verifying each client's single puzzle.

MMH-TLP and Tempora-Fusion differ from aspects 2–4 as well.



Fig. 4: MMH-TLP Workflow Overview.

6.2 Detailed Construction

In this section, we present a detailed description of MMH-TLP.

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

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

- (a) <u>Setting a field's parameter</u>: generates a sufficiently large prime number p, where $\log_2(p)$ is a security parameter, e.g., $\log_2(p) \ge 128$.
- (b) <u>Generating public x-coordinates</u>: let \ddot{t} be the total number of leader clients. It sets $\bar{t} = \ddot{t} + 2$ and $\vec{x} = [x_1, \dots, x_{\bar{t}}]$, where $x_i \neq x_j$, $x_i \neq 0$, and $x_i \notin U$.
- (c) Publishing public parameters: publishes $pk_s = (p, \vec{x}, t)$.
- 2. Key Generation. C.Setup $(1^{\lambda}) \rightarrow K_u$

Each party C_u in $C = \{C_1, \ldots, C_n\}$ takes the following steps:

- (a) <u>Generating RSA public and private keys</u>: computes $N_u = p_1 \cdot p_2$, where p_i is a large randomly chosen prime number, where $\log_2(p_i)$ is a security parameter, e.g., $\log_2(p_i) \ge 2048$. Next, it computes Euler's totient function of N_u , as: $\phi(N_u) = (p_1 1) \cdot (p_2 1)$.
- (b) <u>Publishing public parameters</u>: locally keeps secret key $sk_u = \phi(N_u)$ and publishes public key $pk_u = \overline{N_u}$.
- 3. <u>Puzzle Generation</u>. GenPuzzle $(\vec{m}_u, K_u, pk_s, \vec{\Delta}_u, max_{ss}) \rightarrow (\vec{o}_u, prm_u)$

Each client C_u takes the following steps to generate z puzzles for messages $\vec{m}_u = [m_{1,u}, \ldots, m_{z,u}]$ and wants S to learn each message $m_{j,u}$ at time $time_{j,u} \in time_u$, where $time_u = [time_{1,u}, \ldots, time_{z,u}]$, $\bar{\Delta}_{j,u} = time_{j,u} - time_{j-1,u}, \vec{\Delta}_u = [\bar{\Delta}_{1,u}, \ldots, \bar{\Delta}_{z,u}], 1 \leq j \leq z$, and $1 \leq u \leq n$.

- (a) <u>Checking public parameters</u>: checks the bit-size of p and elements of \vec{x} in pk_s , to ensure $\log_2(p) \ge 128$, $\overline{x_i \neq x_j, x_i \neq 0}$, and $x_i \notin \overline{U}$. If it does not accept the parameters, it returns (\bot, \bot) and does not take further action.
- (b) <u>Generating secret keys</u>: generates a vector of master keys $\vec{mk_u} = [mk_{u,1}, \dots, mk_{u,z}]$ and two secret keys $k_{u,j}$ and $s_{u,j}$ for each master key $mk_{u,j}$ in $\vec{mk_u}$ as follows. It constructs an empty vector $\vec{mk_u}$. Then, it
 - i. sets each exponent $a_{j,u}$.

$$\forall j, 1 \le j \le z : \quad a_{j,u} = 2^{T_j} \mod \phi(N_u)$$

where $T_j = max_{ss} \cdot \bar{\Delta}_{j,u}$ is the total number of squaring needed to decrypt an encrypted solution $m_{j,u}$ after the previous solution $m_{j-1,u}$ is revealed.

- ii. computes each master key $mk_{j,u}$ as follows. For every j, where $1 \le j \le z$:
 - when j = 1:
 - A. picks a uniformly random base $r_j \stackrel{\hspace{0.1em}\mathsf{\scriptscriptstyle\$}}{\leftarrow} \mathbb{Z}_{N_u}$.
 - B. sets key $mk_{j,u}$ as $mk_{j,u} = r_j^{a_j} \mod N_u$.
 - C. appends $mk_{j,u}$ to $\vec{mk_u}$.
 - when j > 1:
 - A. derives a fresh base r_j from the previous master key as $r_j = \text{PRF}(j||0, mk_{j-1,u})$.
 - B. sets key $mk_{j,u}$ as $mk_{j,u} = r_j^{a_j} \mod N_u$.
 - C. appends $mk_{i,u}$ to $\vec{mk_u}$.
- iii. derives two secret keys $k_{j,u}$ and $s_{j,u}$ from each $mk_{j,u}$.

$$\forall j, 1 \leq j \leq z: \quad k_{j,u} = \mathsf{PRF}(1, mk_{j,u}), \quad s_{j,u} = \mathsf{PRF}(2, mk_{j,u})$$

(c) Generating blinding factors: generates $2 \cdot \overline{t}$ pseudorandom values for each j, by using $k_{j,u}$ and $s_{j,u}$.

 $\forall j, 1 \leq j \leq z \quad \text{and} \quad \forall i, 1 \leq i \leq \bar{t}: \quad z_{i,j,u} = \mathsf{PRF}(i, k_{j,u}), \quad w_{i,j,u} = \mathsf{PRF}(i, s_{j,u})$

- (d) Encoding plaintext messages:
 - i. represents each plaintext solution $m_{j,u}$ as a polynomial, such that the polynomial's constant term is the message.

$$\forall j, 1 \le j \le z : \quad \boldsymbol{\pi}_{j,u}(x) = x + m_{j,u} \mod p$$

ii. computes \bar{t} y-coordinates of each $\pi_{i,u}(x)$:

$$\forall i, 1 \leq j \leq z \quad \text{and} \quad \forall i, 1 \leq i \leq \overline{t} : \quad \pi_{i,j,u} = \pi_{j,u}(x_i) \mod p$$

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

$$\forall j, 1 \leq j \leq z \quad \text{and} \quad \forall i, 1 \leq i \leq \overline{t}: \quad o_{i,j,u} = w_{i,j,u} \cdot (\pi_{i,j,u} + z_{i,j,u}) \mod p$$

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

$$com_{j,u} = \operatorname{Com}(m_{j,u}, mk_{j,u})$$

Let $\vec{com}_u = [com_{1,u}, \dots, com_{z,u}].$

- (g) <u>Managing messages</u>: publishes $\vec{o}_u = \left[[o_{1,1,u}, \dots, o_{\bar{t},1,u}], \dots, [o_{1,z,u}, \dots, o_{\bar{t},z,u}] \right]$ and $pp_u = (c\vec{o}m_u, r_1)$. It locally keeps secret parameters $sp_u = \vec{m}k_u$. It sets $prm_u = (sp_u, pp_u)$. It deletes everything else, including each $m_{j,u}$ and $\pi_{j,u}(x)$.
- 4. Linear Combination for a Single Client. Evaluate_{sc}($\langle S(\vec{o}_u, \Delta_u, max_{ss}, pp_u, pk_u, pk_s), C_u(\Delta_u, max_{ss}, K_u, prm_u, q_{z,u}, pk_s)\rangle) \rightarrow (\vec{g}_u, pp_u^{(\text{Evl})})$

In this phase, a client C_u produces certain messages that allow S to find a linear combination of its plaintext solutions after time Δ_u .

- (a) Granting the computation: client C_u takes the following steps.
 - i. <u>Generating temporary secret keys</u>: generates a temporary master key tk and two secret keys k' and s'. It also computes z 1 secret key $[f_j, \ldots, f_z]$. To generate them, it takes the following steps. It computes an exponent:

$$b = 2^Y \mod \phi(N_u)$$

where $Y = \Delta_u \cdot max_{ss}$ and Δ_u is the period after which the solution must be discovered. It selects a base uniformly at random: $h \stackrel{\$}{\leftarrow} \mathbb{Z}_{N_u}$ and then sets a temporary master key tk:

$$tk = h^b \mod N_u$$

It derives two keys from tk:

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

It picks fresh z - 1 random keys $\vec{f} = [f_2, \dots, f_z]$, where $f_j \stackrel{\$}{\leftarrow} \{0, 1\}^{poly(\lambda)}$.

ii. Generating blinding factors: regenerates its original blinding factors, for each j-th puzzle. Specifically, for every j, derives two secret keys $k_{j,u}$ and $s_{j,u}$ from $mk_{j,u} \in K_u$ as follow.

$$\forall j,1 \leq j \leq z: \quad k_{j,u} = \mathtt{PRF}(1,mk_{j,u}), \quad s_{j,u} = \mathtt{PRF}(2,mk_{j,u})$$

It regenerates $z \cdot \overline{t}$ pseudorandom values, by using $k_{j,u}$ and $s_{j,u}$.

 $\forall j, 1 \leq j \leq z \quad \text{and} \quad \forall i, 1 \leq i \leq \bar{t}: \quad z_{i,j,u} = \mathtt{PRF}(i, k_{j,u}), \quad w_{i,j,u} = \mathtt{PRF}(i, s_{j,u})$

It also generates new $2 \cdot \bar{t}$ pseudorandom values using keys (k', s').

$$\forall i, 1 \le i \le \bar{t}: \quad z'_i = \mathsf{PRF}(i, k'), \quad w'_i = \mathsf{PRF}(i, s')$$

It computes new sets of (zero-sum) blinding factors, using each key $f_j \in \vec{f}$, as follows. $\forall j, 1 \leq j \leq z$:

• if j = 1:

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

• if j > 1:

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

iii. <u>Generating y-coordinates of a random root</u>: picks a random root, $root \stackrel{\$}{\leftarrow} \mathbb{F}_p$. It represents root as a polynomial $\gamma(x)$, where root is the polynomial's root, as setting $\gamma(x) = x - root \mod p$. It generates \bar{t} y-coordinates of $\gamma(x)$:

$$\forall i, 1 \leq i \leq \overline{t} : \quad \gamma_i = \gamma(x_i) \mod p$$

- iv. Committing to the root: computes com' = Com(root, tk).
- v. <u>Re-encoding outsourced puzzle</u>: participates in an instance of OLE^+ with S, for every j-th puzzle and every i, where $1 \le j \le z$ and $1 \le i \le \overline{t}$. The inputs of client C_u to the i-th instance of OLE^+ are:

$$e_{i,j} = \gamma_i \cdot q_{j,u} \cdot w'_i \cdot (w_{i,j,u})^{-1} \bmod p, \qquad e'_{i,j} = -(\gamma_i \cdot q_{j,u} \cdot w'_i \cdot z_{i,j,u}) + z'_i + y_{i,j} \bmod p$$

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

$$\begin{aligned} d_{i,j} &= e_{i,j} \cdot e_{i,j}'' + e_{i,j}' \\ &= \gamma_i \cdot q_{j,u} \cdot w_i' \cdot \pi_{i,j,u} + z_i' + y_{i,j} \bmod p \end{aligned}$$

where $q_{j,u}$ is a coefficient for *j*-th solution $m_{j,u}$. If client C_u detects misbehavior during the execution of OLE^+ , it outputs a special symbol \perp and halts.

- vi. Publishing public parameters: publishes $pp_u^{(\text{Evl})} = (h, com')$.
- (b) <u>Computing encrypted linear combination</u>: Server S sums all of the outputs of OLE^+ instances that it has invoked. $\forall i, 1 \le i \le \overline{t}$:

$$g_i = \sum_{j=1}^{z} d_{i,j} \mod p$$
$$= (w'_i \cdot \gamma_i \cdot \sum_{j=1}^{z} q_{j,u} \cdot \pi_{i,j,u}) + z'_i \mod p$$

- (c) Disseminating encrypted result: server S publishes $\vec{g} = [g_1, \ldots, g_{\bar{t}}]$.
- 5. Linear Combination for Multiple Clients. Evaluate_{mc}($\langle S(\vec{o}, \Delta, max_{ss}, p\vec{p}, p\vec{k}, pk_s), C_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s, id_1), \ldots, C_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s, id_n)\rangle) \rightarrow (\vec{g}, \vec{p}\vec{p}^{(\text{Evl})})$

In this phase, the clients interact with S to compute certain messages that enable S to find a linear combination of the clients' plaintext messages after time Δ .

- (a) <u>Randomly selecting leaders</u>: all parties in $C = \{C_1, \ldots, C_n\}$ agree on a random key \hat{r} , e.g., through a coin tossing protocol. Each C_u deterministically identifies indices of \ddot{t} leader clients: $\forall j, 1 \le j \le \ddot{t}$: $idx_j = \mathsf{G}(j||\hat{r})$. Let \mathcal{I} be a vector containing these \ddot{t} clients.
- (b) Granting the computation by each leader client: each leader client C_u in \mathcal{I} takes the following steps.
 - i. <u>Generating temporary secret keys</u>: generates a temporary master key tk_u and two secret keys k'_u and s'_u for itself. Moreover, it generates a secret key f_l for each client. To do that, it takes the following steps. It computes an exponent:

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

where $Y = \Delta \cdot max_{ss}$, Δ is the period after which the solution representing the linear combination of the messages must be discovered, and $N_u \in \vec{pk} = [N_1, \ldots, N_n]$. It selects a base uniformly at random: $h_u \stackrel{\$}{\leftarrow} \mathbb{Z}_{N_u}$ and then sets a temporary master key tk_u :

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

It derives two keys from tk_u :

$$k'_u = \Pr(1, tk_u), \quad s'_u = \Pr(2, tk_u)$$

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

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

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

iii. <u>Generating an encrypted random root</u>: picks a random root: $root_u \stackrel{\$}{\leftarrow} \mathbb{F}_p$. It represents $root_u$ as a polynomial $\gamma_u(x) = x - root_u \mod p$, such that the polynomial's root is $root_u$.

It computes \bar{t} y-coordinates of $\gamma_u(x)$:

$$\forall i, 1 \leq i \leq \overline{t}: \quad \gamma_{i,u} = \gamma_u(x_i) \mod p$$

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

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

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

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

Let $j = id_u$ be the index of one of its own outsourced puzzles, which it wants to use as an input for the linear combination. It regenerates its original blinding factors for its *j*-th solution:

$$k_{j,u} = \operatorname{PRF}(1, mk_{j,u}), \quad s_{j,u} = \operatorname{PRF}(2, mk_{j,u})$$

$$\forall i,1 \leq i \leq \bar{t}: \quad z_{\scriptscriptstyle i,j,u} = \mathtt{PRF}(i,k_{\scriptscriptstyle j,u}), \quad w_{\scriptscriptstyle i,j,u} = \mathtt{PRF}(i,s_{\scriptscriptstyle j,u})$$

where $mk_{j,u} \in prm_u$. It also generates new ones:

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

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

$$\begin{split} v_{i,u} &= \gamma'_{i,u} \cdot \prod_{\forall C_l \in \mathcal{I} \smallsetminus C_u} \gamma'_{i,l} \bmod p \\ y_{i,u} &= -\sum_{\forall C_l \in \mathcal{C} \setminus C_u} \mathtt{PRF}(i, f_l) + \sum_{\forall C_l \in \mathcal{I} \setminus C_u} \mathtt{PRF}(i, \bar{f_l}) \bmod \end{split}$$

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

v. <u>Re-encoding outsourced puzzle</u>: obliviously prepares the puzzle (held by S) for the computation. To do that, it participates in an instance of OLE^+ with S, for every i, where $1 \le i \le \overline{t}$. The inputs of C_u to the *i*-th instance of OLE^+ are:

$$\begin{aligned} e_{i,u} &= q_u \cdot v_{i,u} \cdot (w_{i,j,u})^{-1} \bmod p \\ e'_{i,u} &= -(q_u \cdot v_{i,u} \cdot z_{i,j,u}) + z'_{i,u} + y_{i,u} \bmod p \end{aligned}$$

The input of S to *i*-th instance of OLE^+ is C_u 's encrypted y-coordinate of its *j*-th puzzle: $e''_{i,u} = o_{i,j,u}$ (where $o_{i,j,u} \in \vec{o}$). Accordingly, the *i*-th instance of OLE^+ returns to S:

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

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

- vi. Committing to the root: computes $com'_u = \text{Com}(root_u, tk_u)$.
- vii. <u>Publishing public parameters</u>: publishes $pp_u^{(\text{Evl})} = (h_u, com'_u, Y)$. Note that every leader client $\overline{C_u \in \mathcal{I}}$ uses identical Y. Let $\vec{pp}^{(\text{Evl})}$ contain all the triples $pp_u^{(\text{Evl})}$ published by C_u , where $C_u \in \mathcal{I}$.
- (c) <u>Granting the computation by each non-leader client</u>: each non-leader client C_u takes the following steps.
 - i. Generating blinding factors: receives $(\bar{f}_l, \vec{\gamma}_l)$ from every other leader client in \mathcal{I} .

As before, let $j = id_u$ be the index of one of client C_u outsourced puzzles that it wants to use as an input for the linear combination. It regenerates its original blinding factors:

$$\begin{split} k_{j,u} &= \mathtt{PRF}(1, mk_{j,u}), \quad s_{j,u} &= \mathtt{PRF}(2, mk_{j,u}) \\ \forall i, 1 \leq i \leq \overline{t}: \quad z_{i,j,u} &= \mathtt{PRF}(i, k_{j,u}), \quad w_{i,j,u} &= \mathtt{PRF}(i, s_{j,u}) \end{split}$$

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

$$\begin{split} v_{i,u} &= \prod_{\forall C_l \in \mathcal{I}} \gamma'_{i,l} \bmod p \\ y_{i,u} &= \sum_{\forall C_l \in \mathcal{I}} \mathtt{PRF}(i,\bar{f}_l) \bmod p \end{split}$$

ii. <u>Re-encoding outsourced puzzle</u>: participates in an instance of OLE^+ with the server S, for every i, where $1 \le i \le \overline{t}$. The inputs of C_u to the *i*-th instance of OLE^+ are:

$$e_{i,u} = q_u \cdot v_{i,u} \cdot (w_{i,j,u})^{-1} \mod p$$

$$e'_{i,u} = -(q_u \cdot v_{i,u} \cdot z_{i,j,u}) + y_{i,u} \mod p$$

The input of S to the *i*-th instance of OLE^+ is C_u 's encrypted y-coordinate: $e''_i = o_{i,j,u}$. Accordingly, the *i*-th instance of OLE^+ returns to S:

$$\begin{aligned} d_{i,u} &= e_{i,u} \cdot e_{i,u}'' + e_{i,u}' \\ &= q_u \cdot v_{i,u} \cdot \pi_{i,j,u} + y_{i,u} \bmod p \\ &= q_u \cdot (\prod_{\forall C_l \in \mathcal{I} \backslash C_u} \gamma_{i,l} \cdot w_{i,l}') \cdot \pi_{i,j,u} + y_{i,u} \bmod p \end{aligned}$$

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

(d) <u>Computing encrypted linear combination</u>: server S sums all of the outputs of OLE^+ instances that it has invoked, $\forall i, 1 \le i \le \overline{t}$:

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

- (e) <u>Disseminating encrypted result</u>: server S publishes $\vec{g} = [g_1, \dots, g_{\bar{t}}]$.
- 6. Solving a Puzzle. Solve $(\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pp_u^{(\text{Evl})}, \vec{pk}, pk_s, cmd, cmd) \rightarrow (\vec{m}, \vec{\zeta})$

Server S takes the following steps.

- Case 1. when solving a puzzle corresponding to the linear combination of messages (i.e., when cmd = evalPzl), where all messages belong to the same client C_u , i.e., when cmd = SingleClient. Note that in this case, \vec{o}_u and $\vec{pp}^{(Evl)}$ can be null.
 - (a) Finding secret keys:
 - i. finds temporary key tk, where $tk = h^{2^{Y}} \mod N_{u}$, via repeated squaring of h modulo N_{u} , where $h \in pp_{u}^{(\text{Evl})}, N_{u} \in \vec{pk}$.
 - ii. derives two keys from tk:

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

(b) <u>Removing blinding factors</u>: removes the blinding factors from [g₁,...,g_t] ∈ g.
 ∀i, 1 ≤ i ≤ t̄:

$$\theta_i = \underbrace{\left(\mathsf{PRF}(i,s')\right)^{-1}}_{(w_i')^{-1}} \cdot \left(g_i - \overbrace{\mathsf{PRF}(i,k')}^{z_i'}\right) \bmod p$$
$$= \gamma_i \cdot \sum_{j=1}^{z} q_{j,u} \cdot \pi_{i,j,u} \bmod p$$

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

$$\boldsymbol{\theta}(x) = (x - root) \cdot \sum_{j=1}^{z} q_{j,u} \cdot (x + m_{j,u}) \bmod p$$

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

$$\boldsymbol{\theta}(x) = \boldsymbol{\psi}(x) - root \cdot \sum_{j=1}^{z} q_{j,u} \cdot m_{j,u} \mod p$$

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

(d) <u>Extracting the linear combination</u>: retrieves the result (i.e., the linear combination of $m_{1,u}, \ldots, m_{z,u}$) from polynomial $\boldsymbol{\theta}(x)$'s constant term: $cons = -root \cdot \sum_{j=1}^{z} q_{j,u} \cdot m_{j,u}$ as follows:

$$m = cons \cdot (-root)^{-1} \mod p$$
$$= \sum_{j=1}^{z} q_{j,u} \cdot m_{j,u}$$

- (e) <u>Extracting valid roots</u>: extracts the root(s) of polynomial $\theta(x)$. Let set R contain the extracted roots. It identifies the valid root, by finding a root root in R, where it can pass the verification of the commitment scheme, i.e., Ver(com', (root, tk)) = 1.
- (f) <u>Publishing the result</u>: initiates vectors \vec{m} and $\vec{\zeta}$. It appends m to \vec{m} and (root, tk) to $\vec{\zeta}$. It publishes \vec{m} and $\vec{\zeta}$.
- Case 2. when solving a puzzle related to the linear combination of messages (i.e., when cmd = evalPzl), where each message belongs to a different client, i.e., when $\hat{cmd} = \text{MultiClient}$. In this case, \vec{o}_u and $pp_u^{(\text{Evl})}$ can be null.
 - (a) Finding secret keys: for each leader client $C_u \in \mathcal{I}$:
 - i. finds tk_u (where $tk_u = h_u^{2^Y} \mod N_u$) through repeated squaring of h_u modulo N_u . Note that $(h_u, Y, N_u) \in \vec{pp}^{(E \vee l)}$.
 - ii. derives two keys from tk_u :

$$k'_{u} = \operatorname{PRF}(1, tk_{u}), \quad s'_{u} = \operatorname{PRF}(2, tk_{u})$$

(b) <u>Removing blinding factors</u>: removes the blinding factors from [g₁,...,g_i] ∈ g.
 ∀i, 1 ≤ i ≤ t̄:

$$\theta_{i} = \left(\prod_{\forall C_{u} \in \mathcal{I}} \underbrace{\operatorname{PRF}(i, s'_{u})}_{w'_{i,u}}\right)^{-1} \cdot \left(g_{i} - \sum_{\forall C_{u} \in \mathcal{I}} \underbrace{\operatorname{PRF}(i, k'_{u})}_{\operatorname{PRF}(i, k'_{u})}\right) \mod p$$
$$= \left(\prod_{\forall C_{u} \in \mathcal{I}} \gamma_{i,u}\right) \cdot \sum_{\forall C_{u} \in C} q_{u} \cdot \pi_{i,j,u} \mod p$$

(c) <u>Extracting a polynomial</u>: interpolates a polynomial $\theta(x)$, given pairs $(x_1, \theta_1), \ldots, (x_{\bar{\iota}}, \theta_{\bar{\iota}})$. Polynomial $\theta(x)$ will have the following form:

$$\boldsymbol{\theta}(x) = \prod_{\forall C_u \in \mathcal{I}} (x - root_u) \cdot \sum_{\forall C_u \in C} q_u \cdot (x + m_{j,u}) \bmod p$$

Note that $j = id_u$ and may have different value for different client C_u . It is possible to rewrite $\theta(x)$ as:

$$\boldsymbol{\theta}(x) = \boldsymbol{\psi}(x) + \prod_{\forall C_u \in \mathcal{I}} (-root_u) \cdot \sum_{\forall C_u \in C} q_u \cdot m_{j,u} \bmod p$$

with $\psi(x)$ being a polynomial of degree $\ddot{t} + 1$ that has constant term 0.

(d) <u>Extracting the linear combination</u>: retrieves the final result, that is the linear combination of the messages $m_{j,1}, \ldots, m_{j,n}$, from polynomial $\theta(x)$'s constant term: $cons = \prod (-root_u)$.

 $\sum_{\forall C_u \in C} q_u \cdot m_{j,u}$ as follows:

$$m = cons \cdot (\prod_{\forall C_u \in \mathcal{I}} (-root_u))^{-1} \mod p$$
$$= \sum_{\forall C_u \in C} q_u \cdot m_{j,u}$$

where each $j = id_u$.

- (e) <u>Extracting valid roots</u>: retreives the roots of polynomial $\theta(x)$. Let set R contain the extracted roots. It identifies the valid roots, by finding every $root_u$ in R, such that it passes the commitment's verification: $Ver(com'_u, (root_u, tk_u)) = 1$. This check is performed for every C_u in \mathcal{I} .
- (f) <u>Publishing the result</u>: initiates empty vectors \vec{m} and $\vec{\zeta}$. It appends m to \vec{m} . Also, for every C_u in \mathcal{I} , it appends $(root_u, tk_u)$ to $\vec{\zeta}$. It publishes \vec{m} and $\vec{\zeta}$.
- Case 3 when solving each *j*-th puzzle \vec{o}_j in \vec{o} of client C (i.e., when cmd = clientPzl), server S takes the following steps. Note that in this case, $\vec{pp}^{(\text{Evl})}$ and $pp_u^{(\text{Evl})}$ and can be null. $\forall j, 1 \leq j \leq z$:
 - (a) Finding secret bases and keys: sets base r_j and $mk_{j,u}$ as follows.
 - if j = 1: sets the base to r_1 , where $r_1 \in pp_u$. Then, it finds $mk_{1,u}$ where $mk_{1,u} = r_1^{2^{T_1}} \mod N_u$, through repeated squaring of $r_1 \mod N_u$. It initiates vectors \vec{m} and $\vec{\zeta}$.
 - if j > 1: computes base r_j as $r_j = \text{PRF}(j||0, mk_{j-1,u})$. Next, it finds $mk_{j,u}$ where $mk_{j,u} = r_j^{2^{T_j}} \mod N_u$, via repeated squaring of r_j modulo N_u .

It derive two keys from $mk_{j,u}$:

$$k_{j,u} = \mathsf{PRF}(1, mk_{j,u}), \quad s_{j,u} = \mathsf{PRF}(2, mk_{j,u})$$

(b) Removing blinding factors: re-generates $2 \cdot \overline{t}$ pseudorandom values using $k_{j,u}$ and $s_{j,u}$:

 $\forall i,1 \leq i \leq \bar{t}: \quad z_{i,j,u} = \mathtt{PRF}(i,k_{j,u}), \quad w_{i,j,u} = \mathtt{PRF}(i,s_{j,u})$

Next, it uses the blinding factors to unblind $\vec{o}_{j,u} = [o_{1,j,u}, \dots, o_{\bar{t},j,u}]$:

$$\forall i, 1 \le i \le \overline{t}: \quad \pi_{i,j,u} = ((w_{i,j,u})^{-1} \cdot o_{i,j,u}) - z_{i,j,u} \mod p$$

- (c) Extracting a polynomial: interpolates a polynomial $\pi_{j,u}(x)$, given pairs $(x_1, \pi_{1,j,u}), \ldots, (x_{\bar{t}}, \pi_{\bar{t},j,u})$.
- (d) <u>Publishing the solution</u>: considers the constant term of $\pi_{j,u}(x)$ as the plaintext message, $\overline{m_{j,u}}$. It appends $(\overline{m_{j,u}}, j)$ to \vec{m} and $mk_{j,u}$ to $\vec{\zeta}$. If j = z, then it publishes \vec{m} and $\vec{\zeta}$.
- 7. <u>Verification</u>. Verify $(\vec{m}, \vec{\zeta}, .., pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pp_u^{(\text{Evl})}, pk_s, cmd, cmd) \rightarrow \ddot{v} \in \{0, 1\}$

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

Case 1. when verifying a solution related to the linear combination of messages (i.e., when cmd = evalPzl), where all messages belong to the same client C_u , i.e., when cmd = SingleClient.

(a) <u>Checking the commitment's opening</u>: verify the validity of $(root, tk) \in \vec{\zeta}$ with the help of $com' \in pp_u^{(E \vee 1)}$.

$$\operatorname{Ver}(com', (root, tk)) \doteq 1$$

If the above check passes, it moves on to the next step. Otherwise, it returns $\ddot{v} = 0$ and takes no further action.

- (b) <u>Checking the resulting polynomial's valid roots</u>: checks if the resulting polynomial contains the root $root \in \vec{\zeta}$, by taking the following steps.
 - i. derives two keys from tk:

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

ii. removes the blinding factors from $\vec{g} = [g_1, \ldots, g_{\bar{t}}]$ that were provided by server S in step 4c. Specifically, for every $i, 1 \le i \le \bar{t}$:

$$\theta_i = \underbrace{\left(\operatorname{PRF}(i, s') \right)^{-1}}_{(w'_i)^{-1}} \cdot \left(g_i - \operatorname{PRF}(i, k') \right) \mod p$$
$$= \gamma_i \cdot \sum_{j=1}^{z} q_{j,u} \cdot \pi_{i,j,u} \mod p$$

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

$$\boldsymbol{\theta}(x) = (x - root) \cdot \sum_{j=1}^{z} q_{j,u} \cdot (x + m_{j,u}) \mod p$$
$$= \boldsymbol{\psi}(x) - root \cdot \sum_{j=1}^{z} q_{j,u} \cdot m_{j,u} \mod p$$

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

- iv. checks whether $root \in \vec{\zeta}$ is a root of $\theta(x)$, i.e., $\theta(root) \stackrel{?}{=} 0$. It proceeds to the next step if the check passes. It returns $\ddot{v} = 0$ and takes no further action, otherwise.
- (c) <u>Checking the final result</u>: retrieves the result (i.e., the linear combination of $m_{1,u}, \ldots, m_{z,u}$) from polynomial $\theta(x)$'s constant term: $t = -root \cdot \sum_{j=1}^{z} q_{j,u} \cdot m_{j,u}$ as follows:

$$res' = -t \cdot root^{-1} \mod p$$
$$= \sum_{j=1}^{z} q_{j,u} \cdot m_{j,u}$$

It checks $res' \stackrel{?}{=} m$, where $m \in \vec{m}$ is the result that S sent to it, in step 6f of Case 1.

- (d) <u>Accepting or rejecting the result</u>: if all the checks pass, it accepts \vec{m} and returns $\ddot{v} = 1$. Otherwise, it returns $\ddot{v} = 0$.
- Case 2. when solving a puzzle related to the linear combination of messages (i.e., when cmd = evalPzl), where each message belongs to a different client, i.e., when cmd = MultiClient.

(a) <u>Checking the commitments' openings</u>: verifies the validity of every $(root_u, tk_u) \in \vec{\zeta}$, provided by S in Case 2, step 6f:

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

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

- (b) <u>Checking the resulting polynomial's valid roots</u>: checks if the resulting polynomial contains all the roots in $\vec{\zeta}$, by taking the following steps.
 - i. derives two keys from tk_u :

$$k'_u = \Pr(1, tk_u), \quad s'_u = \Pr(2, tk_u)$$

ii. removes the blinding factors from $[g_1, \ldots, g_{\bar{t}}] \in \vec{g}$ that were provided by S in step 5e. $\forall i, 1 \leq i \leq \bar{t}$:

$$\theta_{i} = \left(\prod_{\forall C_{u} \in \mathcal{I}} \mathtt{PRF}(i, s'_{u})\right)^{-1} \cdot \left(g_{i} - \sum_{\forall C_{u} \in \mathcal{I}} \mathtt{PRF}(i, k'_{u})\right) \bmod p$$
$$= \prod_{\forall C_{u} \in \mathcal{I}} \gamma_{i, u} \cdot \sum_{\forall C_{u} \in C} q_{u} \cdot \pi_{i, u} \bmod p$$

iii. interpolates a polynomial $\theta(x)$, using pairs $(x_1, \theta_1), \ldots, (x_{\bar{t}}, \theta_{\bar{t}})$. This results in a polynomial $\theta(x)$ having the form:

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

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

- iv. if the following checks pass, it will proceed to the next step; it checks if every $root_u \in \vec{\zeta}$ is a root of $\theta(x)$, i.e., $\theta(root_u) \stackrel{?}{=} 0$. Otherwise, it returns $\ddot{v} = 0$ and takes no further action.
- (c) <u>Checking the final result</u>: retrieves the result (i.e., the linear combination of the messages $\overline{m_1, \ldots, m_n}$) from polynomial $\boldsymbol{\theta}(x)$'s constant term: $cons = \prod_{\forall C_u \in \mathcal{I}} (-root_u) \cdot \sum_{\forall C_u \in C} q_u \cdot m_u$ as follows:

$$res' = cons \cdot (\prod_{\forall C_u \in \mathcal{I}} (-root_u))^{-1} \mod p$$
$$= \sum_{\forall C_u \in C} q_u \cdot m_u$$

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

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

Case 3. when verifying a solution of a single puzzle belonging to C_u , i.e., when cmd = clientPzl:

(a) <u>Checking the commitment' opening</u>: checks whether opening $m_{j,u} \in \vec{m}$ and $mk_{j,u} \in \tilde{\zeta}$ matches the commitment:

$$\operatorname{Ver}(com_{i,u}, (m_{i,u}, mk_{i,u})) \stackrel{\scriptscriptstyle f}{=} 1$$

where $com_{j,u} \in pp_u$.

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

Theorem 4 (informal). If MH-TLP and Tempora-Fusion are secure, then MMH-TLP is secure.

Proof (sketch). From the security perspective, MMH-TLP does not introduce any new security mechanism and relies on those proposed in MH-TLP and Tempora-Fusion. Thus, its security (i.e., privacy and solution validity) boils down to the security of MH-TLP and Tempora-Fusion. \Box

7 Evaluation

In this section, we evaluate the costs and features of our schemes and compare them with those of existing TLPs that support homomorphic linear combinations, namely, with the TLPs proposed in [1,28,27,16]. We exclude the TLP in [37], from our analysis, as its authors acknowledge that it is far from practically efficient. Tables 1 and 2 summarize the results.

7.1 Asymptotic Cost

MH-TLP. We begin by analyzing the computation cost of a client.

<u>Client's Costs.</u> The computation costs of a client are as follows. In the Puzzle Generation phase, in each step 3(b)i and 3(b)ii, a client performs z modular exponentiation over $\phi(N)$ and N respectively. Furthermore, 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 addition. In step 3(d)ii, it evaluates a polynomial of degree one at three x-coordinates, which will involve $3 \cdot z$ modular additions. In step 3e, the client performs $3 \cdot z$ additions and multiplications to encrypt the y-coordinates. In step 3f, the client invokes the hash function z times to commit to each message.

In the Linear Combination Phase, in step 4(a)i, it performs two modular exponentiations, one over $\phi(N)$ and the other over N. In the same step, it invokes PRF twice. In step 4(a)ii, it invokes $11 \cdot z + 3$ instances of PRF. In the same step, it performs z - 1 modular addition. In step 4(a)iii, it performs 3 additions. In step 4(a)iv, the client invokes the hash function one. In step 4(a)v, the client performs $6 \cdot z$ additions and $12 \cdot z$ multiplications. In the same step, it invokes $3 \cdot z$ instances of OLE^+ . Therefore, the computation complexity of the client is O(z).

The communication costs of a client are as follows. In the Key Generation phase, step 2b, the client publishes a single public key of size about 2048 bits. In the Puzzle Generation phase, step 3g, the client publishes $4 \cdot z + 1$ values. In the Linear Combination phase, in step 4(a)v, it invokes $3 \cdot z$ instances of OLE^+ where each instance imposes O(1) communication cost. In step 4(a)v, the client publishes two elements. Therefore, the client's communication complexity is O(z).

<u>Verifier's Costs.</u> The computation costs of a verifier include the following operations. In the Verification phase, the computation cost of a verifier in Case 1 is as follows. In step 6a, it invokes an instance of the hash function. In step 6b, it invokes 6 instances of PRF. In step 6(b)ii, it performs 3 additions and 1 multiplication. In step 6(b)iii, it interpolates a polynomial of degree 2. In step 6(b)iv, it evaluates a polynomial of degree 2 at a single point, requiring 2 additions and multiplications. In step 6c, it performs a single multiplication. Thus, the verifier's computation complexity in Case 1 is O(1). In the Verification phase, 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, for each puzzle. Hence, the computation complexity of the verifier O(z).

<u>Server's Costs.</u> Now, we consider the computation cost of a server. In step 4(a)v, server S engages $3 \cdot z$ instances of OLE^+ with each client. In step 4b, S performs $3 \cdot z$ modular addition. During the Solving Puzzles phase, in Case 1 step 5a, S performs Y repeated modular squaring and invokes two instances of PRF. In step 5b, S performs 3 additions and 3 multiplications. In step 5c, it interpolates a polynomial of degree 2 that involves O(1) addition and multiplication operations (note that the complexity is constant with regard to the number of puzzles). In step 5d, it performs a single modular multiplication. In step 5e, it factorizes a polynomial of degree 2 to find its root, costing O(1). In the same step, it invokes the hash function once to identify the valid roots. Thus, the computation complexity of S in Case 1 is O(Y + z).

In Case 2, the cost of S for a client C_u involves the following operations. S performs $O(\max_{ss} \cdot \sum_{j=1}^{z} \bar{\Delta}_{j,u})$ modular squaring over N to find the master keys. It invokes $9 \cdot z - 1$ instances of PRF. It performs 3 addition and 3 multiplication to decrypt *y*-coordinates. It interpolates a polynomial of degree 2 using 3 coordinates, requiring O(1) addition and multiplication operations. Hence, the complexity of S in Case 2 is $O(\max_{ss} \cdot \sum_{j=1}^{z} \bar{\Delta}_{j,u})$.

Next, we analyze the communication costs of S. In the Setup phase, S publishes 4 messages. In the Linear Combination phase, step 4(a)v, it invokes $3 \cdot z$ instances of OLE^+ with the client, where each instance imposes O(1) communication cost. In step 4c, it publishes 3 messages. In the Solving a Puzzle phase, step 5f, it publishes 3 messages. In Case 2, step 5d, the server publishes two messages. Hence, the communication complexity of S is O(z).

MMH-TLP. As before, we begin by evaluating the computation cost of a client.

<u>Client's Costs.</u> The computation costs of a client are as follows. Recall that in MMH-TLP, the number of x-coordinates is linear with the number of leaders \ddot{t} , whereas in MH-TLP it is 3. However, the client in MMH-TLP takes the same types of steps as it takes in MH-TLP. Thus, the computation cost complexity of a client in MMH-TLP is $O(\ddot{t} \cdot z)$.

We proceed to analyze the communication costs of a client. In Phase 2, the client publishes a single public key of size about 2048 bits. In Phase 3, the client publishes $(\bar{t}+1) \cdot z + 1$ messages. In Phase 4, it invokes $\bar{t} \cdot z$ instances of OLE^+ where each instance imposes O(1) communication cost. In the same phase, it publishes two elements. In Phase 5, we will consider the communication cost of a leader client, as it transmits more messages than non-leader clients. The leader client transmits to each client a key for PRF. It also sends \bar{t} encrypted *y*-coordinates of a random root to the rest of the clients. It invokes \bar{t} instances of OLE^+ . The leader client also publishes three elements (h_u, com'_u, Y) . Therefore, the leader client's communication complexity is $O(\bar{t} \cdot n)$. Hence, the client's communication complexity is $O((\bar{t} + n) \cdot z)$.

<u>Verifer's Costs.</u> We will analyze only the computation costs of a verifier, as the protocol imposes no communication overhead on the verifier. In the Verification phase, in Case 1, a verifier performs the same type of computation it does in Case 1 of MH-TLP, however, in the former the number of x-coordinates is \bar{t} (instead of being 3 in MH-TLP). Hence, the verifier's computation complexity in Case 1 is $O(\bar{t})$. In Case 2, the computation cost of the verifier is as follows. In step 7a, it invokes \bar{t} instances of the hash function. In step 7b it invokes $2 \cdot (\bar{t} \cdot \bar{t} + 1)$ instances of PRF. In step 7(b)ii, it performs $\bar{t} \cdot \bar{t} + 1$ additions and $\bar{t} \cdot \bar{t}$ multiplication. In step 7(b)iii, it interpolates a polynomial of degree $\bar{t} + 1$. This involves $O(\bar{t})$ addition and $O(\bar{t})$ multiplication. In step 7(b)iv, it evaluates a polynomial of degree $\bar{t} + 1$ at \bar{t} points, resulting in $\bar{t}^2 + \bar{t}$ additions and $\bar{t}^2 + \bar{t}$ multiplication. Moreover, in step 7c, it performs $\bar{t} + 1$ multiplication. Therefore, its complexity in Case 2 is $\bar{t}^2 + \bar{t}$. The computation cost of the verifier in Case 3 involves a single invocation of the hash function to check the opening of a commitment for each puzzle. Thus, its complexity in this case is O(z). We conclude that when it verifies z puzzles of a client and a linear combination of n clients' messages, the total computation complexity of the verifier is $O(\ddot{t}^2 + \ddot{t} + z)$.

<u>Server's Costs.</u> Initially, we will focus on the computation costs of S. During Phase 4 (Linear Combination for a Single Client), in step 4(a)v, S engages $\bar{t} \cdot z$ instances of OLE^+ with a client. In step 4b, S performs $\bar{t} \cdot z$ modular addition. Thus, its computation complexity in this phase is $O(\bar{t} \cdot z)$. Within Phase 5 (Linear Combination for Multiple Clients), in step 5(b)v, S engages \bar{t} instances of OLE^+ with each client. In step 5d, S performs $\bar{t} \cdot n$ modular addition. Therefore, its complexity in this phase is $O(\bar{t} \cdot n)$.

During Phase 6, Case 1, step 6a, server S performs Y repeated modular squaring and invokes two instances of PRF. In step 6b, it performs \bar{t} addition and \bar{t} multiplication. In step 6c, it interpolates a polynomial using \bar{t} data points, which results in $O(\bar{t})$ computation complexity. In step 6d, it performs a single modular multiplication. In step 6e, it factorizes a polynomial of degree 2 to find its root, which will cost O(1). In step 6e, it invokes the hash function once. Therefore, the overall computation complexity of S in Phase 6, Case 1 is $O(Y + \bar{t} \cdot (z + n))$.

In Phase 6, Case 2, the cost of S is as follows. In step 6a, S performs Y modular squaring to find master key mk_u for each leader client. In the same step, it invokes 2 instances of PRF for each leader client. In step 6b, it invokes $2 \cdot (\bar{t}+1)$ instances of PRF. In the same step, it performs $\bar{t}+1$ addition and \bar{t} multiplication. In step 6c, it interpolates a polynomial using \bar{t} points, involving $O(\bar{t})$ addition and $O(\bar{t})$ multiplication operations. In step 6d, it performs $\bar{t}+1$ multiplication. In step 6e, it factorizes a polynomial of degree $\bar{t}+1$ with the computation complexity of $O(\bar{t}^2)$. Hence, the total computation complexity of S (for n clients) in Phase 6, Case 2 is $O(\bar{t} \cdot n + \bar{t}^2 + \bar{t} \cdot Y)$.

In Phase 6, Case 3, the costs of S for a client C_u are as follows. In step 6a, S performs $O(\max_{ss} \cdot \sum_{j=1}^{z} \bar{\Delta}_{j,u})$ modular squaring over N to find the master keys $mk_{1,u}, \ldots, mk_{z,u}$. In steps 6a and 6b, in total, it invokes $z \cdot (3+\bar{t}) - 1$ instances of PRF. In step 6b, it performs \bar{t} addition and \bar{t} multiplication. In step 6c, it interpolates a polynomial using \bar{t} coordinates, involving $O(\bar{t})$ addition and multiplication. Hence, the complexity of S in Case 3 is $O(\bar{t} + \max_{ss} \cdot \sum_{j=1}^{z} \bar{\Delta}_{j,u})$.

Next, we move on to the communication costs of S. In Phase 1, S publishes \bar{t} messages. In Phase 4, it invokes $\bar{t} \cdot z$ instances of OLE^+ , where each instance imposes O(1) communication cost. In the same phase, it publishes \bar{t} encrypted y-coordinates. In Phase 5, it invokes $\bar{t} \cdot n$ instances of OLE^+ and also publishes \bar{t} encrypted y-coordinates. In Phase 6, Case 1, it publishes 3 messages. In Phase 6, Case 2, S publishes $2 \cdot \ddot{t} + 1$ messages. In Phase Case 3, S publishes $3 \cdot z$ messages. Thus, the total communication complexity of S is $O(\ddot{t} \cdot n + z)$.

The Scheme Proposed in [1]. Initially, we consider a client's costs in this multi-client scheme.

<u>Client's Costs.</u> During the Puzzle Generation phase, a client performs two modular exponentiations, one over $\phi(N)$ and another over N. Within the same phase, it invokes PRF and performs modular addition and multiplication linear with the number of leaders \ddot{t} . During the Linear Combination phase, it performs modular arithmetics, invocations of PRF, and executions of OLE⁺ linearly with \ddot{t} . Thus, the client's overall computation complexity is $O(\ddot{t})$. The communication cost of the client is $O(\ddot{t} \cdot n)$ as it transmits to each client \ddot{t} encrypted y-coordinates of a random root.

<u>Verifier's Costs.</u> During the verification of the result of the linear combination it (a) invokes O(t) instances of the hash function, (b) invokes $O(t^2)$ instances of PRF, and (c) performs $O(t^2)$ addition and multiplication. Its cost during the verification of a solution related to a client's single puzzle is O(1) as it involves a single invocation of a hash function. Hence, when it verifies z puzzles of a client and a linear combination of n clients' messages, the verifier's computation complexity is $O(t^2 + z)$.

Server's Computation Cost. During computing the linear combination of clients' puzzles, S invokes O(t)instances of OLE^+ with each client. In the same phase, it performs $O(t \cdot n)$ modular addition. During the Solving Puzzles phase, when it needs to deal with puzzles related to the linear combination, S performs $O(t \cdot Y)$ repeated modular squaring and invokes O(t) instances of PRF. Within the same phase, it performs $O(t^2)$ addition and multiplication. It also factorizes a polynomial, with the cost of $O(t^2)$. Therefore, the computation complexity of S in this case is $O(t^2 + t \cdot n + t \cdot Y)$. During the Solving Puzzles phase, when it needs to deal with a single puzzle of a client C_u , server S performs $max_{ss} \cdot \Delta_u$ modular squaring to find master key mk_u . It invokes O(t) instances of PRF and performs O(t) addition and multiplication. Therefore, the complexity of S in this case is $O(t + max_{ss} \cdot \Delta_u)$. In the multi-instance case, where the client has zpuzzles where each puzzle j-th puzzle needs to be disclosed after period $\Delta_{j,u}$, S needs to deal with each of the puzzles separately, which leads to the total computation complexity of $O(t + max_{ss} \cdot \sum_{j=1}^{z} \Delta_{j,u})$. The communication cost of S is dominated by OLE^+ invocations, which is linear with the total number of leaders

communication cost of S is dominated by ULE^{+} invocations, which is linear with the total number of leaders and clients, i.e., $O(\vec{t} \cdot n)$.

TLP in [28]. The homomorphic linear combination TLP proposed in [28, p.634], requires a trusted setup involving a trusted party.

<u>Trusted Party's Costs.</u> In the Setup phase, it computes a set of private and public parameters and publishes the public ones. In this phase, the trusted party, only once, performs a modular squaring over $\phi(N)$. Thus, this party's computation cost is O(1). The trusted party's communication complexity is also O(1), as it only publishes 4 values.

<u>Client's Costs.</u> In the Puzzle Generation phase, a client performs 3 modular exponentiations, one over N and the other two over N^2 . Thus, the computation complexity of the client is O(1), with respect to n which is the total number of clients involved. The client's communication complexity is O(1).

<u>Server's Costs.</u> To solve a puzzle (related to a single client's puzzle or a puzzle encoding a linear combination of solutions), a server performs $max_{ss} \cdot \Delta$ repeated modular squaring, similar to conventional TLPs. To compute a homomorphic linear combination of puzzles, the server performs n modular multiplication over N and n modular multiplication over N^2 . Therefore, the computation complexity of the server is O(n). The server's communication complexity is O(1). In the multi-instance case, where a client C_u has z puzzles, where each j-th puzzle must be found after period $\Delta_{j,u}$, the server needs to deal with each puzzle independently, leading to the additional computation complexity of $O(max_{ss} \cdot \sum_{j=1}^{z} \Delta_{j,u})$.

TLP in [27]. The additive TLP proposed in [27] heavily relies on the above additive TLP of Malavolta and Thyagarajan [28]. As a result, the overall complexities of the client and server in this scheme are similar to that of the additive TLP in [28] with a main difference. Namely, the server in this TLP needs to perform $O(\frac{max_{ss} \cdot \Delta}{\log(max_{ss} \cdot \Delta)})$ group operations to generate a proof. To check z puzzles of a client, a verifier's complexity is O(z). This scheme also does not provide any mechanism to efficiently handle the multi-instance setting, imposing additional computation complexity of $O(max_{ss} \cdot \sum_{j=1}^{z} \Delta_{j,u})$ on the server, when each client has z puzzles. The communication complexity for the parties in this TLP is comparable to that described in [28].

TLP in [16]. This TLP is also built upon the additive TLP introduced by Malavolta and Thyagarajan [28]. Consequently, the computation complexity for a trusted party during the Setup phase and for each client during the Puzzle Generation phase is comparable to that in the TLP of Malavolta and Thyagarajan. However, this TLP requires the server to perform $O(n^2 + max_{ss} \cdot \Delta)$ operations to combine the puzzles and solve the combined puzzle. Since this scheme cannot efficiently handle the muti-instance setting, the server

must deal with each puzzle independently, yielding additional computation complexity of $O(\max_{ss} \cdot \sum_{j=1}^{z} \Delta_{j,u})$. The parties' communication complexity in this TLP is similar to that in [28].

7.2 Features

MH-TLP. This scheme can efficiently handle the multiple-instance setting and does not require a trusted setup. It enables anyone to efficiently check the correctness of a solution for a client's puzzle and a linear combination of puzzles. This scheme allows different clients to have different time parameters for their puzzles.

MMH-TLP. This TLP supports multi-client as well as efficiently handling the multiple-instance setting. This scheme also does not require a trusted setup. It also allows anyone to efficiently verify the correctness of a solution for a client's puzzle and a linear combination of puzzles. It allows different clients to have different time parameters for their puzzles.

TLP in [1]. This scheme supports multi-client and does not require a trusted setup. It also supports efficient verification of a solution for a client's puzzle and a linear combination of puzzles. Similar to the above two schemes, it is flexible regarding the time parameters of different puzzles.

TLP in [28]. The original homomorphic linear combination proposed in [28, p.634] requires all time parameters to be identical. This constraint limits its applicability, as different clients may prefer their solutions to be disclosed at different times. To address this, the authors suggested an extension that involves a trusted third party releasing a set of public parameters, each corresponding to a different time parameter during the setup phase. However, this solution also restricts clients' flexibility because they must choose from only the time parameters initially generated by the trusted third party. The scheme supports multiple clients, however, it does not support the multi-instance setting. It does not provide any verification mechanism to allow a verifier to check the solution that the server finds, which contributes to its overall lower cost compared to those that support verification.

TLP in [27]. One of the TLPs introduced in [27] supports multi-client and homomorphic linear combinations. It allows a server to prove the validity of a solution for a single client's puzzle, by relying on computationally expensive public-key-based primitives. However, this scheme does not support verifying the correctness of the linear combinations. This scheme also requires the involvement of a trusted party to generate a set of public and private parameters. This scheme does not support any efficient solution for the multi-instance setting. This scheme also lacks flexibility regarding the time parameter, as it assumes all clients use an identical time parameter.

TLP in [16]. This scheme supports multi-client and allows a server to check whether a puzzle has been created correctly. However, it does not offer any solution for the verification of a solution related to a single puzzle or homomorphic linear combinations. This scheme also requires a trusted party and lacks flexibility with respect to the time parameter, as it presumes that all clients use the same time parameter. It offers the batch-solving feature, that enables the server to combine n puzzles into a single combined puzzle, such that after solving this puzzle, the server can find the solution to each puzzle that was integrated into the combined puzzle.

7.3 Comparison

Cost. The overall computation and communication complexity of all schemes, except the one in [16], is linear with the number of puzzles z and the number of clients n. However, the computation complexity of

the TLP in [16] is quadradic regarding n. Note that the complexities of MH-TLP, MMH-TLP, and the TLP in [1] are quadratic regarding the total number of leader clients \ddot{t} . However, \ddot{t} can be set to a small value, e.g., between 3 and 10, depending on the setting and security assumption. Moreover, only MH-TLP, MMH-TLP, and the TLP in [1] can efficiently deal with the multi-instance setting.

Feature. Among the six schemes, only MMH-TLP provides both multi-client and multi-instance capabilities. It stands out as the scheme that offers the most features. Additionally, MH-TLP, MMH-TLP, and the TLP proposed in [1] (a) do not require a trusted setup, (b) support verification of both the solution to a client's puzzle and the solution to a linear combination of puzzles, and (c) allow flexible time parameters. Conversely, only the TLP in [16] supports batch verification.

7.4 An Overview of Concrete Cost

The three main operations that impose non-negligible costs to the participants of our schemes are polynomial factorization, invocations of PRF, and OLE⁺ execution. In our schemes, the computation complexity of a verifier is quadratic with the number of leaders \ddot{t} , which determines the degree of the polynomial to be factorized. The runtime of polynomial factorization is also influenced by the field size, $\log_2(p)$. As shown in [1], the total combined computation cost imposed due to factorization and PRF invocations is about 6 milliseconds when $\ddot{t} = 10$ and $\log_2(p) = 256$ -bit. The running time of OLE⁺ is low as well, for instance about 1 second for 2¹⁴ input elements, as shown in [35]. Thus, we estimate our schemes will impose an additional cost of about 10 seconds when the total number of clients is 20. This estimation excludes the standard cost of solving puzzles.

8 Conclusion and Future Work

Time-Lock Puzzles (TLPs) have been developed to securely transmit private information into the future without relying on a third party. They have applications in various domains, including transparent scheduled payments in private banking, e-voting, and secure aggregation in federated learning. To enhance the scalability of TLPs, multi-instance TLPs have been designed, enabling a server to efficiently handle multiple instances of a client's puzzles. Separately, homomorphic TLPs have been developed to allow (verifiable) computation on the puzzles of different clients.

In this work, we proposed two schemes MH-TLP and MMH-TLP to bridge these two research lines. Initially, we proposed Multi-instance verifiable partially Homomorphic TLP (MH-TLP), 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. In this setting, the server does not need to simultaneously deal with them; instead, it can solve them one after the other. MH-TLP enables the server to learn the linear combination of the puzzles' solutions after a certain time. It allows public verification of a single puzzle's solution and the computation's result.

Next, we introduced Multi-instance Multi-client verifiable partially Homomorphic TLP (MMH-TLP). This new variant combines the features of both (partially) homomorphic TLP and multi-instance TLP. It supports verifiable partially homomorphic operations on the puzzles belonging to single or multiple clients while maintaining the multi-instance feature. It enables single or multiple clients to ask the server to perform homomorphic linear combinations of their puzzles. This scheme allows anyone to verify whether the server has performed the computation correctly and provided a correct solution.

We have conducted a thorough analysis of these two schemes. Our analysis indicates that the overall overhead of our schemes is linear with respect to the total number of clients and the number of puzzles. By comparing our solutions to the state-of-the-art TLPs, we observed that MMH-TLP offers a set of appealing features not simultaneously provided by any existing TLP.

Batch solving is an intriguing feature that allows a server to combine multiple puzzles into a single composite puzzle. By solving this composite puzzle, one can determine the solution to each individual puzzle involved [16]. It will be interesting to explore how MMH-TLP can be enhanced to offer this property while maintaining its current features and efficiency.

References

- 1. Abadi, A.: Tempora-fusion: Time-lock puzzle with efficient verifiable homomorphic linear combination. Cryptology ePrint Archive, Paper 2024/1013 (2024)
- Abadi, A., Dong, C., Terzis, S.: VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In: FC (2016)
- 3. Abadi, A., Kiayias, A.: Multi-instance publicly verifiable time-lock puzzle and its applications. In: FC (2021)
- 4. Agrawal, S., Malavolta, G., Zhang, T.: Time-lock puzzles from lattices. Springer-Verlag (2024)
- Aho, A.V., Hopcroft, J.E.: The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1974)
- Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: CCS'13 (2013)
- Baum, C., David, B., Dowsley, R., Nielsen, J.B., Oechsner, S.: TARDIS: A foundation of time-lock puzzles in UC. In: EUROCRYPT (2021)
- Blum, M.: Coin flipping by telephone A protocol for solving impossible problems. In: COMPCON'82, Digest of Papers, Twenty-Fourth IEEE Computer Society International Conference, San Francisco, California, USA, February 22-25, 1982. pp. 133–137. IEEE Computer Society (1982)
- 9. Blum, M., Santis, A.D., Micali, S., Persiano, G.: Noninteractive zero-knowledge. SIAM J. Comput. 20(6) (1991)
- Bonawitz, K.A., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H.B., Patel, S., Ramage, D., Segal, A., Seth, K.: Practical secure aggregation for privacy-preserving machine learning. In: CCS (2017)
- Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO'18
- 12. Boneh, D., Naor, M.: Timed commitments. In: ACRYPTO. Springer (2000)
- 13. Chen, H., Deviani, R.: A secure e-voting system based on RSA time-lock puzzle mechanism. In: BWCCA'12 ,
- 14. Chvojka, P., Jager, T., Slamanig, D., Striecks, C.: Versatile and sustainable timed-release encryption and sequential time-lock puzzles (extended abstract). In: ESORICS 2021. Springer (2021)
- Department of Justice-U.S. Attorney's Office: Former jp morgan chase bank employee sentenced to four years in prison for selling customer account information (2018), https://www.justice.gov/usao-edny/pr/ former-jp-morgan-chase-bank-employee-sentenced-four-years-prison-selling-customer
- Dujmovic, J., Garg, R., Malavolta, G.: Time-lock puzzles with efficient batch solving. In: EUROCRYPT. Springer-Verlag (2024)
- 17. Dwork, C., Naor, M.: Zaps and their applications. In: FoCS (2000)
- Feige, U., Lapidot, D., Shamir, A.: Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In: 31st Annual Symposium on Foundations of Computer Science. IEEE Computer Society (1990)
- 19. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02
- Ghosh, S., Nielsen, J.B., Nilges, T.: Maliciously secure oblivious linear function evaluation with constant overhead. In: ASIACRYPT (2007)
- 21. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: EUROCRYPT (2019)
- 22. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
- 23. Katz, J., Loss, J., Xu, J.: On the security of time-lock puzzles and timed commitments. In: Theory of Cryptography
 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part III.
 Lecture Notes in Computer Science (2020)
- 24. Kavousi, A., Abadi, A., Jovanovic, P.: Timed secret sharing. Cryptology ePrint Archive (2023)
- Kissner, L., Song, D.X.: Privacy-preserving set operations. In: CRYPTO 2005, 25th International Cryptology Conference. pp. 241–257 (2005)
- Leigh, D., Ball, J., Garside, J., Pegg, D.: Hsbc files timeline: From swiss bank leak to fallout. The Guardian 12 (2015)
- Liu, Y., Wang, Q., Yiu, S.M.: Towards practical homomorphic time-lock puzzles: Applicability and verifiability. In: ESORICS (2022)
- 28. Malavolta, G., Thyagarajan, S.A.K.: Homomorphic time-lock puzzles and applications. In: CRYPTO'19
- 29. May, T.C.: Timed-release crypto (1993), https://cypherpunks.venona.com/date/1993/02/msg00129.html
- 30. McLean, R.: Ta hacker gained access to 100 million capital one credit card applications and accounts (2019), https://edition.cnn.com/2019/07/29/business/capital-one-data-breach/index.html
- McMahan, H.B., Moore, E., Ramage, D., y Arcas, B.A.: Federated learning of deep networks using model averaging. CoRR abs/1602.05629 (2016)

- 32. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the society for industrial and applied mathematics (1960)
- 33. Ristea, D., Abadi, A., Murdoch, S.J.: Delegated time-lock puzzle. arXiv preprint arXiv:2308.01280 (2023)
- 34. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep. (1996)
- Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-ole: Improved constructions and implementation. In: CCS (2019)
- 36. Shamir, A.: How to share a secret. Commun. ACM (1979)
- 37. Srinivasan, S., Loss, J., Malavolta, G., Nayak, K., Papamanthou, C., Thyagarajan, S.A.K.: Transparent batchable time-lock puzzles and applications to byzantine consensus. In: PKC (2023)
- 38. Thyagarajan, S.A.K., Bhat, A., Malavolta, G., Döttling, N., Kate, A., Schröder, D.: Verifiable timed signatures made practical. In: CCS (2020)
- Thyagarajan, S.A.K., Malavolta, G., Moreno-Sanchez, P.: Universal atomic swaps: Secure exchange of coins across all blockchains. In: IEEE Symposium on Security and Privacy, SP (2022)
- 40. Thyagarajan, S.A.K., Malavolta, G., Schmidt, F., Schröder, D.: Paymo: Payment channels for monero. IACR Cryptol. ePrint Arch. (2020)

A Enhanced OLE's Ideal Functionality and Protocol

The enhanced OLE ensures that the receiver cannot learn anything about the sender's inputs, when it sets its input to 0, i.e., c = 0. The enhanced OLE's protocol (denoted by OLE^+) is presented in Figure 5.

- 1. Receiver (input $c \in \mathbb{F}$): Pick a random value, $r \stackrel{\$}{\leftarrow} \mathbb{F}$, and send (inputS, (c^{-1}, r)) to the first \mathcal{F}_{OLE} .
- Sender (input a, b ∈ F): Pick a random value, u ^{*} F, and send (input R, u) to the first F_{0LE}, to learn t = c⁻¹ · u + r. Send (input S, (t + a, b u)) to the second F_{0LE}.
 Receiver: Send (input R, c) to the second F_{0LE} and obtain k = (t + a) · c + (b u) =
- $a \cdot c + b + r \cdot c$. Output $s = k r \cdot c = a \cdot c + b$.

Fig. 5: Enhanced Oblivious Linear function Evaluation (OLE^+) [21].

B The Tempora-Fusion Protocol

In this section, we present Tempora-Fusion, initially introduced in [1]

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

The server S only once takes the following steps:

- (a) generates a sufficiently large prime number p, where $\log_2(p)$ is a security parameter, e.g., $\log_2(p) \ge 128$.
- (b) let \ddot{t} be the total number of leader clients. It sets $\bar{t} = \ddot{t} + 2$ and $\vec{x} = [x_1, \dots, x_{\bar{t}}]$, where $x_i \neq x_j, x_i \neq 0$, and $x_i \notin U$.
- (c) publishes $pk_s = (p, \vec{x}, t)$.
- 2. Key Generation. C.Setup $(1^{\lambda}) \rightarrow K_u$

Each party C_u in $C = \{C_1, \ldots, C_n\}$ takes the following steps:

- (a) computes $N_u = p_1 \cdot p_2$, where p_i is a large randomly chosen prime number, where $\log_2(p_i)$ is a security parameter. It computes Euler's totient function of N_u , as: $\phi(N_u) = (p_1 1) \cdot (p_2 1)$.
- (b) stores secret key $sk_u = \phi(N_u)$ and publishes public key $pk_u = N_u$.

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

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

- (a) checks the bit-size of p and elements of \vec{x} in pk_s , to ensure $\log_2(p) \ge 128$, $x_i \ne x_j$, $x_i \ne 0$, and $x_i \notin U$. If it does not accept the parameters, it returns (\bot, \bot) and does not take further action.
- (b) generates a master key mk_{u} and two secret keys k_{u} and s_{u} as follows:
 - i. sets exponent a_u as: $a_u = 2^{T_u} \mod \phi(N_u)$.
 - where $T_u = \Delta_u \cdot max_{ss}$ and $\phi(N_u) \in K_u$.
 - ii. selects a base uniformly at random: $r_u \stackrel{s}{\leftarrow} \mathbb{Z}_{N_u}$ and then sets a master key mk_u as follows:

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

iii. derive two keys from mk_u as: $k_u = PRF(1, mk_u)$, $s_u = PRF(2, mk_u)$.

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

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

- (d) encodes plaintext message as follows:
 - i. represents plaintext message m_u as a polynomial, such that the polynomial's constant term is the message. Specifically, it computes polynomial $\pi_u(x)$ as: $\pi_u(x) = x + m_u \mod p$.
 - ii. computes \bar{t} y-coordinates of $\pi_u(x)$ as: $\forall i, 1 \leq i \leq \bar{t}$: $\pi_{i,u} = \pi_u(x_i) \mod p$, where $x_i \in \bar{x}$ and $p \in pk_s$.
- (e) encrypts the *y*-coordinates using the blinding factors as follows:

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

- (f) commits to the plaintext message: $com_u = Com(m_u, mk_u)$.
- (g) publishes $\vec{o}_u = [o_{1,u}, \dots, o_{\bar{t},u}]$ 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 $m_u, \pi_u(x), \pi_{1,u}, \dots, \pi_{\bar{t},u}$. It sets $prm_u = (sp_u, pp_u)$.
- 4. Linear Combination. Evaluate $(\langle S(\vec{o}, \Delta, max_{ss}, \vec{pp}, \vec{pk}, pk_s), C_1(\Delta, max_{ss}, K_1, prm_1, q_1, pk_s), \dots, C_n(\Delta, max_{ss}, K_n, prm_n, q_n, pk_s)\rangle) \rightarrow (\vec{g}, \vec{pp}^{(\text{Evl})})$

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

- (a) all parties in C agree on a random key \hat{r} , e.g., by participating in a coin tossing protocol [8]. Each 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) each leader client C_u in \mathcal{I} takes the following steps.
 - i. 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: $b_u = 2^{\gamma} \mod \phi(N_u)$.

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

It derives two keys from tk_u as: $k'_u = PRF(1, tk_u), \quad s'_u = PRF(2, tk_u).$

It picks a random key f_i for each client C_i excluding itself, i.e., $f_i \stackrel{\$}{\leftarrow} \{0,1\}^{poly(\lambda)}$, where $C_i \in C \setminus C_u$. It sends f_i to each C_i .

ii. derives \bar{t} pseudorandom values from s'_u :

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

iii. picks a random root: $root_u \stackrel{\$}{\leftarrow} \mathbb{F}_p$. It represents $root_u$ as a polynomial, such that the polynomial's root is $root_u$. Specifically, it computes polynomial $\gamma_u(x)$ as: $\gamma_u(x) = x - root_u \mod p$.

Then, it computes \bar{t} y-coordinates of $\gamma_u(x)$ as: $\forall i, 1 \leq i \leq \bar{t}$: $\gamma_{i,u} = \gamma_u(x_i) \mod p$.

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

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

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

iv. receives $(\bar{f}_i, \vec{\gamma}'_i)$ from every other client which are in \mathcal{I} . It regenerates its original blinding factors:

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

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

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

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

$$\begin{split} v_{i,u} &= \gamma'_{i,u} \cdot \prod_{\forall C_l \in \mathcal{I} \smallsetminus C_u} \gamma'_{i,l} \bmod p \\ y_{i,u} &= -\sum_{\forall C_l \in C \setminus C_u} \mathtt{PRF}(i, f_l) + \sum_{\forall C_l \in \mathcal{I} \setminus C_u} \mathtt{PRF}(i, \bar{f_l}) \bmod \end{split}$$

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

v. obliviously, without having to access a plaintext solution, prepares the puzzle (held by S) for the computation. To do that, it participates in an instance of OLE^+ with S, for every i, where $1 \le i \le \overline{t}$. The inputs of C_u to i-th instance of OLE^+ are:

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

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

$$d_{i,u} = e_i \cdot e_i'' + e_i'$$

= $q_u \cdot v_{i,u} \cdot \pi_{i,u} + z_{i,u}' + y_{i,u} \mod p$
= $q_u \cdot \gamma_{i,u} \cdot w_{i,u}' \cdot (\prod_{\forall C_l \in \mathcal{I} \setminus C_u} \gamma_{i,l} \cdot w_{i,l}') \cdot \pi_{i,u} + z_{i,u}' + y_{i,u} \mod p$

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

- vi. computes $com'_u = \text{Com}(root_u, tk_u)$.
- vii. publishes $pp_u^{(\text{Evl})} = (h_u, com'_u, Y)$. Note that all $C_u \in \mathcal{I}$ use identical Y. Let $\vec{pp}^{(\text{Evl})}$ contain all the triples $pp_u^{(\text{Evl})}$ published by C_u , where $C_u \in \mathcal{I}$.
- (c) each non-leader client C_u takes the following steps.
 - i. receives $(\bar{f}_i, \vec{\gamma}'_i)$ from every other client which is in \mathcal{I} . It regenerates its original blinding factors:

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

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

$$\begin{split} v_{i,u} &= \prod_{\forall C_l \in \mathcal{I}} \gamma'_{i,l} \bmod p \\ y_{i,u} &= \sum_{\forall C_l \in \mathcal{I}} \mathtt{PRF}(i,\bar{f}_l) \bmod p \end{split}$$

ii. participates in an instance of OLE^+ with the server S, for every i, where $1 \le i \le \overline{t}$. The inputs of C_u to *i*-th instance of OLE^+ are:

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

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

$$d_{i,u} = e_i \cdot e_i'' + e_i'$$

= $q_u \cdot v_{i,u} \cdot \pi_{i,u} + y_{i,u} \mod p$
= $q_u \cdot (\prod_{\forall C_l \in \mathcal{I} \setminus C_u} \gamma_{i,l} \cdot w_{i,l}') \cdot \pi_{i,u} + y_{i,u} \mod p$

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

(d) server S sums all of the outputs of OLE^+ instances that it has invoked, $\forall i, 1 \le i \le \overline{t}$:

$$g_i = \sum_{\forall C_u \in C} d_{i,u} \mod p$$
$$= (\prod_{\forall C_u \in \mathcal{I}} \gamma_{i,u} \cdot w'_{i,u} \cdot \sum_{\forall C_u \in C} q_u \cdot \pi_{i,u}) + \sum_{\forall C_u \in \mathcal{I}} z'_{i,u} \mod p$$

Note that in $g_{i,j}$ does not exist any $y_{i,j}$, because $y_{i,j}$ in different $d_{i,j}$ canceled out each other after they summed up.

- (e) server S publishes $\vec{g} = [g_1, \dots, g_{\bar{t}}].$
- 5. Solving a Puzzle. Solve ($\vec{o}_u, pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, \vec{pk}, pk_s, cmd$) $\rightarrow (m, \zeta)$

Server S takes the following steps.

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

(a) for each $C_u \in \mathcal{I}$:

- i. finds tk_u where $tk_u = h_u^{2^Y} \mod N_u$ through repeated squaring of h_u modulo N_u , where $(h_u, Y) \in \vec{pp}^{(\text{Evl})}$ and $N_u \in \vec{pk}$.
- ii. derives two keys from tk_u as: $k'_u = PRF(1, tk_u), \quad s'_u = PRF(2, tk_u).$
- (b) removes the blinding factors from $[g_1, \ldots, g_{\bar{t}}] \in \vec{g}$.
 - $\forall i, 1 \le i \le \bar{t}:$

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

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

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

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

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

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

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

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

- (e) 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 $Ver(com'_u, (root_u, tk_u)) = 1$. Note that S performs the check for every C_u in \mathcal{I} .
- (f) publishes the solution m = res and the proof $\zeta = \{(root_u, tk_u)\}_{\forall G_u \in \mathcal{T}}$.

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

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

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

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

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

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

$$\forall i, 1 \le i \le \overline{t}: \quad \pi_{i,u} = \left((w_{i,u})^{-1} \cdot o_{i,u} \right) - z_{i,u} \bmod p$$

- (c) interpolates a polynomial $\boldsymbol{\pi}_{u}$, given pairs $(x_1, \pi_{1,u}), \ldots, (x_{\bar{t}}, \pi_{\bar{t},u})$.
- (d) considers the constant term of π_u as the plaintext solution, m_u . It publishes the solution $m = m_u$ and the proof $\zeta = mk_u$.
- 6. Verification. Verify $(m, \zeta, .., pp_u, \vec{g}, \vec{pp}^{(\text{Evl})}, pk_s, cmd) \rightarrow \ddot{v} \in \{0, 1\}$

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

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

- (a) verifies the validity of every $(root_u, tk_u) \in \zeta$, provided by S in Case 1, step 5f: $\forall C_u \in \mathcal{I}$: $\operatorname{Ver}(com'_u, (root_u, tk_u)) \stackrel{?}{=} 1$, where $com'_u \in \vec{pp}^{(\text{Evl})}$. If all of the verifications pass, it proceeds to the next step. Otherwise, it returns $\ddot{v} = 0$ and takes no further action.
- (b) checks if the resulting polynomial contains all the roots in ζ , by taking the following steps.
 - i. derives two keys from tk_u as: $k'_u = \text{PRF}(1, tk_u), \quad s'_u = \text{PRF}(2, tk_u).$
 - ii. removes the blinding factors from $[g_1, \ldots, g_{\bar{t}}] \in \vec{g}$ that were provided by S in step 4e.

$$\forall i, 1 \le i \le t:$$

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

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

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

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

- iv. if the following checks pass, it will proceed to the next step. It checks if every $root_u$ is a root of $\boldsymbol{\theta}$, by evaluating $\boldsymbol{\theta}$ at $root_u$ and checking if the result is 0, i.e., $\boldsymbol{\theta}(root_u) \stackrel{?}{=} 0$. Otherwise, it returns $\ddot{v} = 0$ and takes no further action.
- (c) retrieves the linear combination of the messages m_1, \ldots, m_n from polynomial $\theta(x)$'s constant term: $cons = \prod_{\forall C_u \in \mathcal{I}} (-root_u) \cdot \sum_{\forall C_u \in C} q_u \cdot m_u$ as follows:

$$res' = cons \cdot (\prod_{\forall C_u \in \mathcal{I}} (-root_u))^{-1} \mod p$$
$$= \sum_{\forall C_u \in C} q_u \cdot m_u$$

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

(d) 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 C_u , i.e., when cmd = clientPzl:

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

$$\operatorname{Ver}(com_u, (m_u, mk_u)) \stackrel{\scriptscriptstyle f}{=} 1$$

where $com_u \in pp_u$.

(b) 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. 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 [1] for the proof of Theorem 5.

C The Original RSA-Based TLP

Below, we restate the original RSA-based time-lock puzzle proposed in [34].

1. <u>Setup</u>: Setup_{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., SKE.keyGen $(1^{\lambda}) \rightarrow k$.
- (d) choose a uniformly random value r, i.e., $r \stackrel{\hspace{0.1em}\hspace{0.1em}\hspace{0.1em}}{\leftarrow} \mathbb{Z}_{N}^{*}$.
- (e) set $a = 2^T \mod \phi(N)$.
- (f) set pk := (N, T, r) as the public key and $sk := (q_1, q_2, a, k)$ as the secret key.
- 2. <u>Generate Puzzle</u>: GenPuzzle_{TLP}(m, pk, sk).
 - (a) encrypt the message under key k using the symmetric-key encryption, as follows: $o_1 = SKE.Enc(k, m)$.
 - (b) encrypt the symmetric-key encryption key k, as follows: $o_2 = k + r^a \mod N$.
 - (c) set $o := (o_1, o_2)$ as puzzle and output the puzzle.
- 3. <u>Solve Puzzle</u>: Solve_{TLP}(pk, o).
 - (a) find b, where $b = r^{2^T} \mod N$, through repeated squaring of r modulo N.
 - (b) decrypt the key's ciphertext, i.e., $k = o_2 b \mod N$.
 - (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 [3] for the proof.

Theorem 6. 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.

D Sequential and Iterated Functions

Definition 10 $(\Delta, \delta(\Delta))$ -Sequential function). For a function: $\delta(\Delta)$, time parameter: Δ and security parameter: $\lambda = O(\log(|X|))$, $f: X \to Y$ is a $(\Delta, \delta(\Delta))$ -sequential function if the following conditions hold:

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

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

where $\delta(\Delta) = (1 - \epsilon)\Delta$ and $\epsilon < 1$, as stated in [11].

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

$$Pr\begin{bmatrix} \frac{\mathcal{A}(N,r,y) \to b}{r \stackrel{\diamond}{\leftarrow} \mathbb{Z}_N, b \stackrel{\diamond}{\leftarrow} \{0,1\}}\\ if \ b = 0, \ y \stackrel{\diamond}{\leftarrow} \mathbb{Z}_N\\ else \ y = r^{2^T} \end{bmatrix} \leq \frac{1}{2} + \mu(\lambda)$$