TEMP: Time-locked Encryption Made Practical

Leemon Baird
Swirlds

Pratyay Mukherjee
Visa Research

Rohit Sinha
Swirlds

Abstract

Time-locked encryption can encrypt a message to a future time such that it can only be decrypted after that time. Potential applications include sealed bid auctions, scheduled confidential transactions, and digital time capsules.

Prior practical schemes for time-locked encryption rely on a clock-equipped trusted server, who periodically publishes a time-specific decryption key based on a long-term secret. Their main idea is to model time periods as *identities* in an *identity-based encryption* scheme. While such schemes allow encryption to a future time periods, they offer limited support for decryption of past ciphertexts. In particular, they force a client to be online when the key is published, or interact with the server to re-generate the key.

This paper proposes a new notion of time-locked encryption where an *aggregated* decryption key can be used to decrypt any ciphertext locked to a prior time. Furthermore, we decentralize the trust among a number of servers, such that it can tolerate up to a threshold number of (malicious) corruptions. We call our notion *threshold aggregated time-locked encryption* (TATLE). We propose a practical construction that supports *compact decryption keys* as well as *compact ciphertexts* (both logarithmic in the total lifetime). Our construction is based on bilinear pairing and adapts ideas from Canetti et al.'s binary tree encryption [Eurocypt 2003] and Naor et al.'s distributed pseudorandom functions [Eurocrypt 1999].

1 Introduction

Time-locked encryption can encrypt a message to a future time such that a receiver can only decrypt the ciphertext after that time. It opens the door for several novel applications [26, 25]. Examples include: 1) sealed-bid auctions (to prevent malicious auctioneers from reading bids before deadline); 2) scheduled confidential transactions with non-repudiation (e.g. insiders issuing commitments to stock trades that cannot be discarded if later unfavorable); 3) digital equivalent of time capsules; 4) cryptocurrency wallet backups (e.g., escrow or a set of users assisting with key recovery after a deadline).

Prior schemes for time-locked encryption fall into two broad categories. Time-lock puzzles [26] requires the receiver to perform expensive sequential computations (a.k.a. puzzles), where the difficulty of the puzzle (e.g., repeated squaring modulo a product of two primes) determines a lower bound on the time that the receiver must execute (under some assumption on "best available" hardware). Although no trust is placed on third parties in the case of time-lock puzzles, they only let the sender specify a coarse-grained release time, dependent on factors including the receiver's CPU speed and when the receiver commenced the puzzle. Moreover, it requires a significant amount of computation and energy from the receiver; in an auction, it would require the auctioneer to solve a separate puzzle for each bidder, which is not scalable.

When the application demands a precise or absolute time for decryption (e.g. digital lockbox or voting), or requires efficiency, an approach based on one or more *trusted time servers* is far more practical, and that is our focus for this work. In fact, to avoid a single point of failure, we decentralize trust to a set of servers, who hold shares of a *long-term* secret key (under a threshold secret-sharing scheme, for instance), such that they collectively generate the *time-specific* decryption key after the deadline has passed, while keeping the long-term key distributed *at all times*. Prior schemes of this variant, by Blake and Chan [9] and Cheon et al. [12], use a trusted "timed-release" server that keeps a long-term secret to periodically compute and publish a decryption key for each time period or *epoch*. Both schemes are essentially adapted from identity-based encryption [4] (IBE), wherein each epoch is mapped to an identity; they borrow a useful property from IBE wherein encryption uses a short public key and can be done prior to generating the epoch's decryption key.

Now consider what happens when the user is offline at the time the server releases the decryption key, or receives a back-dated ciphertext, or simply gets around to decrypting the ciphertext at a later time (very likely in real-time settings with sub-second epochs). In all these situations, the user needs access to older keys.

To that end, the server can either make available the older

keys (by re-computing them on user's request or publishing them on a bulletin board), or alternatively, broadcast the entire set of keys to the client. Each approach has its drawbacks. Re-computing keys for user requests has prohibitive computational and deployment overheads, and possibly privacy concerns in some contexts as the user must reveal the ciphertext's epoch (or use a protocol for private information retrieval [6], which needs high bandwidth). On the other hand, publishing historical keys on a bulletin board or broadcasting them quickly exhausts storage and / or bandwidth. When using 1-second epochs, it would take over 1.8 GB of key material for each year worth of epochs (each epoch's key being a 64 byte group element). The underlying issue is that there is no correlation amongst the keys for different epochs.

We take a different view on time-locked encryption. A key released for epoch τ must be *aggregated*, in that it should be sufficient to decrypt for all prior epochs $\tau' \leq \tau$ as well. Not surprisingly, the epoch-specific keys[1] must also be sub-linear in size (in the lifetime of the system), and small in practice (ideally no more than a few KBs for practical lifetimes). The ability to decrypt for prior epochs has never been considered in prior works, and we believe it to be a natural notion for time-locked encryption. Combining with the decentralization of the key-generation, we propose a new notion called (threshold) *aggregated time-locked encryption* (TATLE).

We provide a simple and efficient construction that achieves chosen-plaintext security using an (asymmetric) bilinear pairing on a Gap Diffie-Hellman group – our security reduces to the decision Biliniear Diffie-Hellman (DBDH) assumption in the random oracle model. Our construction operates in a (t,n)-threshold setting for any $t \leq n$, where the lifetime secret (from which epoch keys are periodically derived) is secret-shared onto a set of n servers, such that compromising fewer than a threshold t of the servers reveals no information about the lifetime secret. We show how to obtain CCA-security using a standard transformation. Furthermore, using simple and efficient non-interactive zero-knowledge proofs, our construction also provides malicious security without much overhead, so a cheating server cannot force incorrect decryption. We emphasize that these augmentations are done independently such that it is possible to *combine* these properties (such as CCA and malicious security together) in any desired way.

Though we present a detailed empirical study later, consider a few metrics (for our maliciously secure scheme) for a sample data point: lifetime of 2^{30} epochs, or roughly 34 years with 1-second epochs. Our key size is logarithmic in the number of epochs; it ranges from 0.16 KB - 4.8 KB, depending on the specific epoch in a lifetime. Ciphertexts are also logarithmic in size (0.19-2 KB of ciphertext expansion) and decryption incurs logarithmic number of group operations (between 35-50 ms). Key derivation also incurs logarithmic number of group operations on the server (between 2-4 ms).

[1] We also refer to these keys by epoch keys or aggregated keys throughout the paper to distinguish them from the long-term keys.

Contributions In this work, we define, design and implement threshold aggregated time-locked encryption (TATLE). Our contribution can be summarized as follows:

- formalization of TATLE, and its CPA / CCA security properties in a threshold setting;
- constructions for semi-honest and malicious settings, achieving CPA and CCA security, with compact keys and ciphertexts (both logarithmic in the total lifetime), and practical running time;
- open-source implementation and empirical evaluation measuring the sizes of keys and ciphertexts, and the running time of the various algorithms in TATLE.

2 Related Work

We briefly summarize some closely related works here.

Computational Reference Clocks Instead of having an absolute decryption time, schemes based on time-lock puzzles require the recepient to perform an expensive sequential computation to recover the message, thus imposing a coarse-grained release time. Rivest et al. [26] provide a construction based on repeated squaring modulo a product of two primes. Mahmoody et al. [21] constructs time-lock puzzles in the random oracle model. A recent proposal by Liu et al. [20] constructs time-locked encryption using a *computational reference clock* (based on Bitcoin hashchains) and an extractable witness encryption scheme which is not practical.

Trusted Time Servers Blake and Chan [9] and Cheon et al. [12] provide schemes that are adaptations of the Boneh-Franklin IBE scheme [4]. Neither schemes enable decryption of prior ciphertexts, or alternatively, require aggregated keys of linear size in order to do so. The scheme of Rabin and Thorpe [25] requires the servers to compute a separate public key for each epoch, whose private component is released during that epoch. This requires the servers to apriori publish a long list of future public keys.

Additional Relevant Works Specter et al. [29] add deniability to emails by divulging private signing keys (so authenticity can no longer be proven) over time from a hierarchical identity-based signature scheme, adapted from the Gentry-Silverberg scheme [18]. Moreover, their hierarchy mimics that of a calendar, and they achieve succintness by allowing a child's key to be derivable from the parent's key. While our approaches have technical similarity, our scheme shows how a binary identity space can enable a more efficient tree-based encryption scheme with shorter keys. Ning et al. [24] design a time-release protocol that split a secret into several shares, and require the shareholders to release their shares on a future date or get penalized by a smart contract.

3 Technical Overview

We assume a finite *lifetime* for the scheme, which is discretized into a set of T time periods called *epochs*. Our work makes no assumption on the duration of each epoch in terms of wall-clock time, so our only system parameter is the total number of epochs T.

3.1 Deployment and Operation

We have a client-server setting. The server uses a long-term secret-key to compute per-epoch keys. Moreover, to avoid a single point of failure, we use a set of independent servers, and distribute the long-term secret material amongst them.

A setup phase establishes a *lifetime (long-term) secret key* lsk, and generates a corresponding public key lpk. Clients use lpk to encrypt messages. Instead of the whole lsk, the setup phase outputs shares of lsk computed using a t out of n threshold secret sharing scheme [28][2]; here, n denotes the number of servers and t is the corruption threshold, as in t shares are required to reconstruct lsk and any subset of $t-1$ shares reveals no information (in the information-theoretic sense) about lsk.

Each server S_i is given a share lsk_i of the whole secret lsk. At the start of epoch τ, server S_i publishes a *partial* aggregated/epoch key $K_{\tau,i}$. Given any t such partial keys, a client can combine them *locally* to attain the *whole* epoch key K_τ. The client then uses K_τ to decrypt any ciphertext locked to epoch τ or earlier.

We note that the whole lsk is never reconstructed. Moreover, the scheme is non-interactive, in that the keys output by the server do not depend on the message or the ciphertext.

3.2 Requirements

3.2.1 Efficiency Requirements

For the system to be practical for applications with real-time constraints (e.g. auctions) and large number of users (e.g. voting), it must have:

- **short keys**: the key size must be sub-linear in the lifetime T. For practical parameters (e.g. $T = 2^{32}$, or 136.2 years of 1-second epochs), we expect keys on the order of KBs.
- **efficient operations**: encryption and decryption operations should consume CPU time on the order of few milliseconds. We expect a similar requirement for key generation on the server.
- **short ciphertexts**: we expect ciphertexts to be short as well, with at most a few KBs of ciphertext expansion for practical parameters.

The above requirements ensure practical bandwidth overheads between the client and servers, and also practical running time for the clients.

3.2.2 Security Requirements

Message Privacy Any ciphertext encrypted for a future epoch cannot be decrypted using any subset of keys released in the past. Specifically, prior keys do not reveal any information about future keys (to a computationally-bounded adversary).

Malicious Security We assume that fewer than t out of n servers may behave maliciously, including producing incorrect shares of the epoch key. In order to prevent these attacks from causing incorrect decryptions, we require that the aggregated keys (both partial and whole) be publicly verifiable (using public parameters generated during the setup phase). Moreover, while corrupt servers may also cheat by releasing key material for future epochs, an attacker has negligible advantage in extracting the whole decryption key for a future epoch (again, assuming that fewer than t servers are corrupt).

Availability The system must be functional as long as at least t servers are operating correctly. In other words, we can have fewer than $n-t$ corrupt servers, who may go offline or publish incorrect values, but the client must still succeed in deriving the correct decryption key.

3.3 Overview of our Construction

Here we provide an elaborated technical overview of our main construction. For full details we refer to Section 6. First we assume a single centralized key-generation server – in this setting we describe the prior IBE based construction, which fails to meet our efficiency requirements. Then we describe our (centralized) construction. Later we show how to deploy it into the threshold setting. Finally we mention how to augment this into CCA-secure and malicious-secure setting.

3.3.1 Prior IBE-based Constructions

Let us briefly examine how prior works essentially adapt identity-based encryptions (IBE) to the purpose of time-locked encryption. The basic idea is to apply the pairing-based IBE scheme of Boneh and Franklin [4], where each identity is mapped to an epoch. The scheme makes use of source groups[3] $\mathbb{G}$ and $\mathbb{G}_T$ which are both cyclic groups of prime

[2] In the threshold setting, the setup phase can also be performed using a distributed key generation protocol [17], in lieu of assuming a trusted dealer, such that no single party learns the lifetime secret key, however we do not place any such demands on the setup phase in our definition and it does not matter that much because it is done only once in the beginning.

[3] For the ease of exposition here we assume the symmetric variant of pairing where both the source groups are same, as opposed to an asymmetric variant where different source groups are used – our implementation uses asymmetric pairing for better efficiency.

order q, a bilinear pairing $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, generator $g \in \mathbb{G}$, and hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{G}$. The scheme essentially works as follows:

- Setup : Sample random $\alpha \leftarrow_\$ Z_q$, and output the lifetime public key $lpk := g^\alpha$, and lifetime secret key $lsk := \alpha$.
- In each epoch, the server outputs epoch key $K_\tau := \mathcal{H}(\tau)^\alpha$.
- $\mathsf{Enc}(m, \tau)$ outputs $c := (g^r, m \cdot e(\mathcal{H}(\tau)^r, lpk)$ for $r \leftarrow_\$ Z_q$.
- $\mathsf{Dec}(K_\tau, c)$ outputs $m := v \cdot e(K_\tau, u)^{-1}$ where $c = (u, v)$.

Note that these operations correspond closely to the Boneh-Franklin IBE construction [4]. A debilitating property of this scheme is that the keys $\{\mathcal{H}(\tau)^\alpha\}_{\tau \in 1 \ldots T}$ are unstructured, in that they cannot be aggregated or compressed (without α). In particular, there is no way to compactly generate a key at epoch τ, which would be used to decrypt any ciphertext encrypted to an epoch $\tau' \leq \tau$.

3.3.2 HIBE-based approach

Our initial observation is that keys must mimic the inherent *hierarchy* amongst the epochs: key material for a later epoch must subsume that of earlier epochs, while not revealing any bits of information about the future epochs. This points us towards hierarchical identity-based encryption [18, 19] (HIBE).

In addition to the properties of an IBE scheme, HIBE assumes a partial ordering for the identity-space and derives keys hierarchically. That is, in addition to associating a key with each identity, given identities $id \preceq id'$, the key for id can be derived from the key for id', via a property called *delegation*. This property is beneficial for our use case, wherein we can define a partial order amongst the epochs, and thus avoid having to publish keys of lower-level epochs if a higher-level epoch key is already released. In particular, we focus on a particular HIBE construction for a restriced identity space of binary strings, known as Binary Tree Encryption (BTE) [7].

Consider arranging identities or epochs in a binary tree, as illustrated below for $T = 15$ epochs.

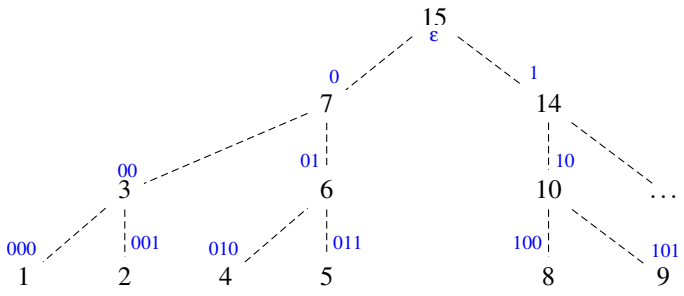

Figure 1: Double labeling of tree with epoch id (via post-order traversal) and path id (binary encoding of path from root node)

Each node is given two labels or identities: the epoch id τ ($0 \leq \tau \leq T$), and a path id ω containing a binary string that denotes the path to that node from the root (left child being 0 and right child being 1) – we find it convenient to present the construction using the path-based identity, but it is not strictly necessary. The path identities form a *prefix order relation*, where $\omega \preceq \omega'$ if ω' is a prefix of ω – we denote the root node (or the least upper bound) by the empty string ε. The epoch id is assigned via a post-order traversal on the binary tree, which gives us a very useful property.[4]

Consider a non-leaf node with path id ω and epoch id τ. Then, any of its descendant has an epoch id $\tau' < \tau$. Furthermore, ω is a prefix of ω', which is the path id corresponding to τ'. Clearly, a HIBE key (we refer to such keys as id keys) for a node labeled ω can be used to derive a key corresponding to node ω'. This satisfies our requirement because any such epoch τ' is smaller than τ. So, it is sufficient to include a single id key for τ to enable decryption corresponding to any τ' which is the epoch id of a descendant — this enables the desired aggregation. Note that, the set of descendants, however, do not exhaust all prior epochs, and therefore we need to include more id keys into an epoch key to enable the time hierarchy we want. However, the tree-structure guarantees that any epoch does not contain more than $\log(T)$ id keys.

For instance, we publish the HIBE key for node 000 in epoch 1, 000 and 001 in epoch 2, 00 in epoch 3, nodes 00 and 010 in epoch 4, and so on. In particular, for a node with path id ω, an epoch key consists of w id keys, where w denotes the hamming weight of bit-string ω. Therefore, in the worst case an epoch-key consists of $\log(T) + 1$ id keys.[5] Instantiating with a scheme such as BTE, that uses $O(\log(T))$ group elements for one id key, we obtain a scheme where an epoch key requires $O(\log^2(T))$ group elements in total.

A natural question is whether we can achieve a better space efficiency. While investigating this, we find that the delegation property – using an id key to derive keys for lower-level identities – of HIBE is not strictly required for our setting (also see Remark 3.1). In particular, in contrast to the HIBE requirements, our key-generation procedure always has access to the lifetime secret-key lsk (which actually makes it similar to IBE). Let us summarize the main properties that we seek in our encryption scheme:

- public-key encryption (similar to both IBE and HIBE): encryption should only require the lifetime public key lpk and the epoch id ω, and nothing else;
- key-generation (similar to IBE): key for id ω may be generated from the lifetime secret key lsk and an epoch id ω;

[4] It is evident later when we present the construction that binary tree structure provides best space efficiency for our scheme. Furthermore, we also explain why post-order traversal is chosen as opposed to pre-order (as chosen in [7]) or in-order ones.

[5] Note that this happens for nodes with path id $111 \ldots 1$.

– hierarchical decryption (similar to HIBE): epoch key for id ω (epoch id τ) is sufficient to decrypt ciphertexts "locked to" $\omega' \preceq \omega$ (or equivalently $\tau' \leq \tau$)

A key contribution of this work is to propose an encryption scheme satisfying these properties, while using $O(1)$ key material for each node, and hence, $O(\log(T))$ size epoch keys.

We focus on the base construction first, and then augment the encryption to achieve malicious-security and CCA-security.

3.3.3 Our TATLE scheme

We now present the core aspects of our scheme below (within the box), and give the full details in Sec. 6. We note that our construction is inspired by the BTE construction of Canetti et al. [7]. Sometimes we use path id and epoch id of a node alternatively — as discussed in Sec. 6, this is enabled by an efficient bijective mapping between these two labels. In the following description, we shall use $\omega|_i$ to denote the first i bits of ω, $|\omega|$ to denote the bit-length of ω, and S_ω to denote the id key for node with path id ω.

Lifetime Keys : $lsk = \alpha$, $lpk = g^\alpha$ where $\alpha \leftarrow_\$ Z_q$

Key for path id ω generated with lsk :

$$S_\omega = \mathcal{H}(\varepsilon)^\alpha \cdot \prod_{j=1}^{|\omega|} \mathcal{H}(\omega|_j)^\alpha$$

Encryption of M :

$C = (g^\gamma, \mathcal{H}(\omega|_1)^\gamma, \mathcal{H}(\omega|_2)^\gamma \ldots, \mathcal{H}(\omega)^\gamma, M \cdot d)$
where $d = e(lpk, \mathcal{H}(\varepsilon))^\gamma = e(g, \mathcal{H}(\varepsilon))^{\alpha\gamma}$

Decryption of C with S_ω :

Parse C as $(U_0, U_1, \ldots, U_t, V)$

Output $M = V \cdot d^{-1}$ where $d = \dfrac{e(U_0, S_\omega)}{\prod_{i=1}^{|\omega|} e(lpk, U_i)}$

Observe from the bilinearity property of pairings that:

$$d = \frac{e(U_0, S_\omega)}{\prod_{i=1}^{|\omega|} e(g^\alpha, U_i)}$$
$$= \frac{e(g^\gamma, \mathcal{H}(\varepsilon)^\alpha \cdot \prod_{j=1}^{|\omega|} \mathcal{H}(\omega|_j)^\alpha)}{\prod_{i=1}^{|\omega|} e(g^\alpha, \mathcal{H}(\omega|_i)^\gamma)}$$
$$= \frac{e(g, \mathcal{H}(\varepsilon))^{\alpha\gamma} \cdot \prod_{i=1}^{|\omega|} e(g, \mathcal{H}(\omega|_i))^{\alpha\gamma}}{\prod_{i=1}^{|\omega|} e(g, \mathcal{H}(\omega|_i))^{\alpha\gamma}}$$
$$= e(g, \mathcal{H}(\varepsilon))^{\alpha\gamma}$$

Let us consider an example assuming $T = 15$ as depicted below, where the id keys are highlighted in red.

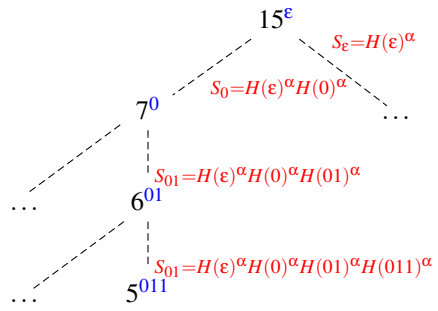

Figure 2: Id keys for our doubly-labeled tree

For instance, say that we wish to encrypt for $\omega = 01$, and decrypt first using S_{01} and later using S_0 (note that for 01, the epoch key consists of two id keys). Encryption of message M for $\omega = 01$ produces the ciphertext:

$$(U_0 = g^\gamma,\ U_1 = \mathcal{H}(0)^\gamma,\ U_2 = \mathcal{H}(01)^\gamma,\ V = M \cdot e(g^\alpha, \mathcal{H}(\varepsilon))^\gamma)$$

Observe that the $e(g^\alpha, \mathcal{H}(\varepsilon))^\gamma$ term acts as a random mask based on the client's randomness γ. Given id key $S_{01} = \mathcal{H}(\varepsilon)^\alpha \cdot \mathcal{H}(0)^\alpha \cdot \mathcal{H}(01)^\alpha$, we decrypt the above ciphertext as follows:

$$V \cdot d^{-1} = V \cdot \frac{e(g^\alpha, U_1) \cdot e(g^\alpha, U_2)}{e(U_0, S_{01})} = M$$

More importantly, due to hierarchical structure, we can also decrypt the same ciphertext using id key $S_0 = \mathcal{H}(\varepsilon)^\alpha \cdot \mathcal{H}(0)^\alpha$:

$$V \cdot d^{-1} = V \cdot \frac{e(g^\alpha, U_1)}{e(U_0, S_0)}$$
$$= M \cdot e(g^\alpha, \mathcal{H}(\varepsilon))^\gamma \cdot \frac{e(g^\alpha, \mathcal{H}(0)^\gamma)}{e(g^\gamma, \mathcal{H}(\varepsilon)^\alpha) \cdot e(g^\gamma, \mathcal{H}(0)^\alpha)} = M$$

We emphasize that $S_{\omega'}$ for any prefix ω' of ω is sufficient to decrypt a ciphertext locked to ω. The intuition behind this scheme is that the ciphertext for ω contains an element corresponding to each prefix of ω, and hence, can be thought of as encrypting to each prefix of ω (or an epoch corresponding to each node from the root to the node for ω). Therefore, when given a node key for any prefix $\omega' \succ \omega$, we can ignore the remaining elements of the ciphertext (i.e., beyond $U_{|\omega'|} = \mathcal{H}(\omega|_{|\omega'|})^\gamma$) and decrypt as if the ciphertext was instead locked to id ω'.

Remark 3.1 (Comparison with HIBE and IBE)**.** *Delegation means that* anyone *with a key for id* ω *can derive a key for id* ω'*. This is useful in the original motivation for HIBE, where a separate party can assume full ability to derive keys in an identity subspace (e.g. a team within a larger organization) with respect to the assigned hierarchical structure. However, in* TATLE*,* all epoch keys *are issued by the same server, and*

access to the lifetime secret lsk gives the server ability to compute the key for any id — this is rather similar to IBE. Therefore, in our case, it suffices to enforce the hierarchy in an efficient manner without providing the ability to delegate.

Given the above scheme, it is straightforward to construct a TATLE scheme (albeit with centralized key-generation, i.e., for $n = t = 1$): For any epoch τ, the epoch/aggregated key consists of $O(\log(T))$ S values, such that their subtrees cover *all epochs* between 1 and τ. For instance, in the above example with $T = 15$, the key for epoch 4 is $K_4 = \{S_{00}, S_{010}\}$, epoch 5 is $K_5 = \{S_{00}, S_{010}, S_{011}\}$, and so on. The final epoch 15 is $K_{15} = \{S_\varepsilon = \mathcal{H}(\varepsilon)^\alpha\}$, which simply allows decryption of all ciphertexts encrypted with the lpk. Since each S value is a single group element, our epoch keys have $O(\log(T))$ size (as opposed to $O(\log^2(T))$ in HIBE). We stress that hierarchical decryption is the key enabler here, as it allows us to prune S values of children once a parent node's id key can be emitted. We refer the reader to Sec. 6 for a full presentation.

3.3.4 Drawbacks of Unbalanced Trees

We have a hierarchical structure of a *balanced* binary tree similar to BTE [7]. As explained above, the standard IBE gives a linear structure which fails to achieve our efficiency requirement of compact aggregated/epoch keys. Therefore, a tree-like structure seems inevitable in order for an aggregated key to cover a large number of epochs. However, as we have seen, an id key for epoch τ does not cover all keys corresponding to epochs $< \tau$ — this leads to an epoch key size of $O(\log(T))$. One might wonder whether this blow up can be avoided by instead employing an unbalanced tree: each node in the tree would have a right child which is a leaf, and a left child which branches out further.

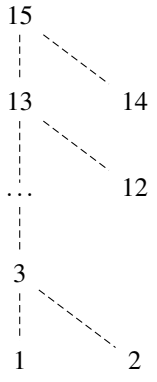

Figure 3: Unbalanced id tree

Such a tree (for $T = 15$) is illustrated to our left. It is easy to see that such a structure indeed supports epoch keys of $O(1)$ sizes, in that all non-leaf nodes with epoch id $\tau' < \tau$ are contained within the subtree rooted a τ. That would mean that an aggregated key could contain as few as two id-keys (two group elements). However, this leads to a blow up in the ciphertext size, rendering it to contain $O(T)$ many group elements — intuitively, this occurs because each ciphertext for a node with path id ω must contain $\Omega(|\omega|)$ group elements when using the above encryption technique with hierarchical decryption. In an unbalanced tree, a path id ω can have upto T bits. Thus, such alternatives fail to achieve our efficiency requirements.

3.3.5 Thresholdizing

Since our keys consist of a set of S values, each computed by exponentiating the lifetime secret key $lsk = \alpha$ on a known group element, it is simple to compute them when the lifetime secret key is distributed in a manner such that there are n servers holding a (t,n)-threshold secret sharing of lsk. Basically, instead of computing values like $\mathcal{H}(v)^{lsk}$ (for some value v) the i-th server now computes $\mathcal{H}(v)^{lsk_i}$. The client, on receiving any t of such values, can combine them using Lagrange reconstruction in the exponent. This step is similar to the threshold computation of PRF as proposed in [23, 2]. Having thresholdized the scheme in this manner, we decentralize the trust as it protects against upto $t-1$ server compromises.

3.3.6 Malicious and CCA security

Our base construction (c.f. Fig. 4) achieves CPA-security against semi-honest attackers. In Section 6.1, we show how to augment this to a verifiable construction, in that a client can verify the responses from each server and thus protect against malicious corruption. Furthermore, in Section 6.2, we outline how to use a variant of the Fujisaki-Okamoto [16] transformation (also used in BTE [7]) to obtain CCA-security. Importantly, these two modifications can be made independently of each other and hence one can easily combine them to obtain a CCA-secure construction (c.f. Corollary 1) which is verifiable and thus resilient against malicious attacks.

4 Formal Security Model

4.1 Definition of ATLE Scheme

Definition 1 (Threshold Aggregate Timed-Locked Encryption)**.** A threshold aggregate time-locked encryption (TATLE) scheme is a tuple of algorithms (Setup, PartAggKeyGen, KeyCombine, Enc, Dec) with the following syntax:

- $\mathsf{Setup}(1^\kappa, T, n, t) \to (pp, lpk, (lsk_1, \ldots, lsk_n),$: On input the security parameter 1^κ and the lifetime duration T (in the number of epochs), Setup generates public parameters pp (to be used by all algorithms that follow), a life-time public key lpk, n shares $\{lsk_i\}_{i\in[n]}$ of the life-time secret key lsk, and the initial aggregated key K_0.
- $\mathsf{PartAggKeyGen}(lsk_j, \tau) \to K_{\tau,j}$: On input a long-term key-share lsk_j and an epoch $\tau \in \{1, \ldots, T\}$, this algorithm outputs a partial aggregated key $k_{\tau,j}$ specific to the time period τ and the key-share used (i.e. j).
- $\mathsf{KeyCombine}(K_{\tau,1}, \ldots, K_{\tau,t}) \to K_\tau$ combines t partial aggregated keys into a whole aggregated key.

– $\mathsf{Enc}(lpk, m, \tau) \to c$: encrypts a message m "locked to" epoch τ, using the lifetime public key lpk, and outputs the ciphertext c.

– $\mathsf{Dec}(lpk, K, c) \to m/\perp$: decrypts the ciphertext c using an aggregated key K. If unsuccessful, Dec returns $\perp$.

Then, the following condition holds for any $n, t, \kappa, T \in \mathbb{N}$, (such that $t \leq n$). Let $(pp, lpk, (lsk_1, \ldots, lsk_n)) \leftarrow \mathsf{Setup}(1^\kappa, T, n, t)$; then, for any message m, any two epochs $\tau, \tau' \in [T]$ for which $\tau \leq \tau'$, it satisfies:

(i) *correctness*, that is there exists a negligible function $\mathsf{negl}(\cdot)$ for which the following probability is at least $1 - \mathsf{negl}(\kappa)$:

$$\Pr\Big[m \leftarrow \mathsf{Dec}(lpk, K_{\tau'}, c) \;\mid\; (pp, lpk, K_0, (lsk_1, \ldots, lsk_n)) \leftarrow \mathsf{Setup}(1^\kappa, T, n, t) \\ c \leftarrow \mathsf{Enc}(lpk, m, \tau) \\ \big\{(K_{\tau,j}) \leftarrow \mathsf{PartAggKeyGen}(lsk_j, \tau)\big\}_{j\in[t]} \\ K_\tau \leftarrow \mathsf{KeyCombine}(K_{\tau,1}, \ldots, K_{\tau,t}) \\ \big\{(K_{\tau',j}) \leftarrow \mathsf{PartAggKeyGen}(lsk_j, \tau')\big\}_{j\in[t]} \\ K_{\tau'} \leftarrow \mathsf{KeyCombine}(K_{\tau',1}, \ldots, K_{\tau',t})\Big]$$

where the probability is over the random coin tosses of the parties involved in Setup, PartAggKeyGen and Enc;

(ii) *efficiency*, that is both $|K_{\tau,i}|$ and $|c|$ are proportional to $O(\log(T))$.

4.2 Security

We define a security game (IND-TL-CCA) for achieving chosen ciphertext security against a "selective" TATLE attacker who commits to the epoch to be attacked in advance (before the setup phase). For that reason, we call this attack a *selective-epoch* attack. Our definition is inspired by the security definitions for binary tree encrypion [7].

First, the attacker $\mathcal{A}$ submits her target epoch $\tau^\star$. As is common in CCA games, we allow $\mathcal{A}$ to perform a set of queries both before and after sending the challenge plaintexts. To avoid trivial wins, the game checks whether $\mathcal{A}$ issued a key generation query for any epoch on or after $\tau^\star$, whose output can be used to derive the keys for epoch $\tau^\star$. The challenger $\mathcal{C}$ responds to a polynomial number of decryption and key generation queries by $\mathcal{A}$, after which $\mathcal{A}$ submits a challenge pair of equal-length messages m_0, m_1; $\mathcal{C}$ selects a random bit b and sends $\mathcal{A}$ the encryption of m_b locked to the epoch $\tau^\star$ (selected by $\mathcal{A}$ earlier). After receiving the challenge ciphertext, $\mathcal{A}$ submits another set of decryption and key generation queries, under the constraint that $\mathcal{A}$ is not requesting the decryption of the challenge ciphertext nor is requesting a key for any epoch $\geq \tau^\star$. Finally, $\mathcal{A}$ outputs the guess bit b' and wins if $b' = b$.

Definition 2 (IND-TL-CPA/CCA)**.** A TATLE := (Setup, PartAggKeyGen, KeyCombine, Enc, Dec) scheme satisfies *indistinguishability under chosen ciphertext attack* if for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl such that the *advantage* of $\mathcal{A}$ is given by

$$\Big|\Pr\big[\mathsf{CCA}_{\mathrm{TATLE},\mathcal{A}}(1^\kappa, 0) = 1\big] - \Pr\big[\mathsf{CCA}_{\mathrm{TATLE},\mathcal{A}}(1^\kappa, 1) = 1\big]\Big| \leq \mathsf{negl}(\kappa),$$

in a security game CCA which is defined below.

$\underline{\mathsf{CCA}_{\mathrm{TATLE},\mathcal{A}}(1^\kappa, b)}$:

– *Selection.* $\mathcal{A}(1^\kappa, T)$ outputs an epoch $0 \leq \tau^\star \leq T$.

– *Initialization.* Run $\mathsf{Setup}(1^\kappa, T)$ to get (lsk, pp). Give pp to $\mathcal{A}$. Initialize $\tau_{max} := 0$.

– *Corruption.* $\mathcal{A}$ outputs a set of corrupt party's identities $C \subseteq [n]$ such that $|C| < t$. Give lsk_i to $\mathcal{A}$ for all $i \in C$.

– *Phase 1.* $\mathcal{A}$ adaptively issues a polynomial number of queries, each of one of two types:

  – *Pre-challenge decryption.* In response to $\mathcal{A}$'s decryption query $(\mathtt{Decrypt}, \tau, c)$, $\mathcal{C}$ responds by generating the epoch key k_τ (by running PartAggKeyGen and KeyCombine many times in sequence as needed), and using it to decrypt c.

  – *Pre-challenge key derivation.* In response to $\mathcal{A}$'s key derivation query $(\mathtt{Derive}, \tau, j)$ where $j \in [n] \setminus C$, run PartAggKeyGen with lsk_j and τ and return the output to $\mathcal{A}$. Update $\tau_{max} := max(\tau_{max}, \tau)$ if $(\mathtt{Derive}, \tau, j)$ is asked for at least t different j's — this can be tracked by storing the queries into a list L_τ for each τ.

– *Challenge.* $\mathcal{A}$ outputs $(\mathtt{Challenge}, m_0, m_1)$ where $|m_0| = |m_1|$. Give $c^\star \leftarrow \mathsf{Enc}(pp, m_b, \tau^\star)$ to $\mathcal{A}$. Output 1 if $\tau_{max} \geq \tau^\star$.

– *Phase 2.* $\mathcal{A}$ adaptively issues a polynomial number of queries, each of one of two types:

  – *Post-challenge decryption.* Repeat phase 1 but with the following caveat. Only process $\mathcal{A}$'s decryption query $(\mathtt{Decrypt}, \tau, c)$ if $c \neq c^\star$, else return $\perp$ to $\mathcal{A}$.

  – *Post-challenge key derivation.* Repeat phase 1 but with the following caveat. Only respond to $\mathcal{A}$'s key derivation query $(\mathtt{Derive}, \tau, j)$ only if either $\tau < \tau^\star$ or L_τ has $< t$ distinct j in total, else return $\perp$ to $\mathcal{A}$.

– *Guess.* Finally, $\mathcal{A}$ returns a guess b'. Output b'.

When the attacker is prohibited from invoking the decryption oracle, the above definition achieves a weaker guarantee called indistinguishability under chosen plaintext attack or IND-TL-CPA However, even in IND-TL-CPA, the adversary is given access to the key-derivation oracle. The corresponding experiment is denoted by $\mathsf{CPA}_{\mathsf{TATLE},\mathcal{A}}$.

Remark 4.1 (Semi-honest vs malicious security)**.** *The adversary $\mathcal{A}$ in the above definition can either be semi-honest or malicious. Clearly a construction that is secure against the malicious adversary achieves a stronger security guarantee, albeit with additional overhead. Our base construction (c.f. Fig. 4) provides security against the semi-honest attacker. In Section 6.1 we show how to augment that to a construction providing security against a malicious attacker, requiring changes only to the* PartAggKeyGen *and* Combine *algorithms. In particular, a client who runs* Combine *verifies proofs produced by each server (independently, i.e., no interaction amongst the servers or between clieant and server) that it executed* PartAggKeyGen *correctly. Therefore, we call such a construction (publicly)* verifiable *TATLE.*

Remark 4.2 (Adaptive security)**.** *The definition achieves a stronger adaptive security if the "selection" phase takes place after 'corruption" but before the challenge phase. Our construction can be generically transformed to saisfy adaptive security by using complexity leveraging, that is by assuming sub-exponential security of the underlying assumption.*

5 Notations and Primitives

Notation The set of all binary strings of length ℓ is denoted as $\{0,1\}^\ell$. The output y of a probabilistic algorithm A on input x is denoted by $y \leftarrow A(x)$. For deterministic algorithms sometimes we use $y := A(x)$. Moreover, occasionally we need to explicitly specify the randomness r of a probabilistic algorithm, which is denoted by $y := A(x;r)$. For any bitstring w, we write $w|_i$ to denote the first i bits of w. We denote the empty string by ε.

5.1 Bilinear Pairings

Certain elliptic curves have an additional structure, called a *bilinear pairing*. We use the following definitions from [5].

Definition 3. *Let $\mathbb{G}_0$, $\mathbb{G}_1$, $\mathbb{G}_T$ be three cyclic groups of prime order q where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. A pairing is an efficiently computable function $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ satisfying the following properties:*

- *bilinear:*

$$\forall u,u' \in \mathbb{G}_0.\ \forall v \in \mathbb{G}_1.\ e(u\cdot u',v) = e(u,v)\cdot e(u',v)$$
$$\forall u \in \mathbb{G}_0.\ \forall v,v' \in \mathbb{G}_1.\ e(u,v\cdot v') = e(u,v)\cdot e(u,v')$$

- *non-degenerate: $g_T =: e(g_0,g_1)$ is a generator of $\mathbb{G}_T$.*

Bilinearity implies the following property:

$$e(g_0^\alpha, g_1^\beta) = e(g_0,g_1)^{\alpha\cdot\beta} = e(g_0^\beta, g_1^\alpha)$$

The decision-BDH assumption states that given random elements g_0^α, g_0^β, $g_0^\gamma \in \mathbb{G}_0$, the value $e(g_0,g_1)^{\alpha\cdot\beta\cdot\gamma} \in \mathbb{G}_T$ is indistinguishable from a random element in $\mathbb{G}_T$.

Definition 4. Attack Game for Decision bilinear Diffie - Hellman (DBDH) assumption*: let $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ be a bilinear pairing where $\mathbb{G}_0$, $\mathbb{G}_1$, $\mathbb{G}_T$ are cyclic groups of prime order q with generators $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$. For a given adversary $\mathcal{A}$, we define two experiments.*
Experiment $b \in \{0,1\}$*:*
The challenger computes

- $\alpha, \beta, \gamma, \delta \leftarrow \mathbb{Z}_q$.
- $u_0 \leftarrow g_0^\alpha$, $u_1 \leftarrow g_1^\alpha$, $v_0 \leftarrow g_0^\beta$, *and* $w_1 \leftarrow g_1^\gamma$
- $z^{(0)} \leftarrow e(g_0,g_1)^{\alpha\cdot\beta\cdot\gamma} \in \mathbb{G}_T$, $z^{(1)} \leftarrow e(g_0,g_1)^\delta \in \mathbb{G}_T$

The adversary is given $(u_0,u_1,v_0,w_1,z^{(b)})$ outputs a bit $\hat{b} \in \{0,1\}$. Let W_b be the event that $\mathcal{A}$ outputs 1 in experiment b. We define $\mathcal{A}$'s advantage in solving the DBDH problem as:

$$\mathsf{DBDHadv}[\mathcal{A},\mathsf{e}] = |\ Pr[W_0] - Pr[W_1]\ |$$

5.2 Secret Sharing

Definition 5 (Shamir's Secret Sharing)**.** *Let p be a prime. An (n,t,p,s)-Shamir's secret sharing scheme is a randomized algorithm* SSS *that on input four integers n,t,p,s, where $0 < t \le n < p$ and $s \in Z_p$, outputs n shares $s_1,\ldots,s_n \in Z_p$ such that the following two conditions hold for any set $\{i_1,\ldots,i_\ell\}$:*

- *if $\ell \ge t$, there exists fixed (i.e., independent of s) integers $\lambda_1,\ldots,\lambda_\ell \in Z_p$ (a.k.a. Lagrange coefficients) such that $\sum_{j=1}^{\ell} \lambda_j s_{i_j} = s \bmod p$;*
- *if $\ell < t$, the distribution of $(s_{i_1},\ldots,s_{i_\ell})$ is uniformly random.*

Concretely, Shamir's secret sharing works as follows. Pick $a_1, \ldots, a_{t-1} \leftarrow_\$ Z_p$. Let $f(x)$ be the polynomial $s + a_1\cdot x + a_2\cdot x^2 + \ldots + a_{t-1}\cdot x^{t-1}$. Then s_i is set to be $f(i)$ for all $i \in [n]$.

6 Our construction

In this section we put forward our main TATLE construction. Our construction, though inspired by the Binary-tree encryption [7], is much simpler and thereby more efficient. Our base construction (Fig. 4) satisfies CPA-security and is protected only against semi-honest attacker. In Sec. 6.1 we show how to augment our base construction to achieve malicious security. Finally, we show how to augment that to CCA-security in

Sec. 6.2 by a variant of Fujisaki-Okamoto transformation [16] analogous to Canetti et al. [7]. Since these two augmentations are orthogonal, it is possible to combine them easily to obtain construction satisfying CCA and malicious security simultaneously (c.f. Corollary 1).

Doubly-labeled tree. We use a binary-tree in our construction analogous to BTE [7]. Each node of the tree is labeled with a binary bit-string as follows: let the depth of the tree be d; then the root is labeled with the empty string ε, its left child is labeled 0 and the right child is labeled 1; then the entire tree is labeled recursively such that for each node with label $\omega \in \{0,1\}^*$, its left child is labeled by $\omega 0$ and right child by $\omega 1$. Clearly, any node at level $\delta \in \{1,\ldots,d\}$ is labeled with a binary string of length δ, which is equal to the length of the path from the root to this node. These labels of the nodes are called *primary labels*. We refer to a node by its primary label. Additionally, each node is labeled with an integer (referred to as secondary labels), which is assigned through a post-order traversal on the tree. Recall that a post-ordered traversal assigns integer labels in an increasing sequence (that is $1,2,\ldots$) in order left-right-root recursively. So, we can define a bijective mapping $M : \{0,1\}^* \to \mathbb{N}$ which maps the primary labels to the secondary labels. The inverse mapping from the secondary to primary labels is denoted by $M^{-1} : \mathbb{N} \to \{0,1\}^*$. An example is given in Fig. 1. For the lack of a better name we shall refer to this structure by *doubly-labeled tree*. For any node ω in the tree we define its left-extended (similarly right-extended family) family (denoted as $\mathsf{LEF}(\omega)$) as the set which contains all nodes that are left children of any node in the path from root to ω, but do not belong to the path themselves. For example, let $\omega = 0100$, then the path from root to ω is the ordered set $(\varepsilon, 0, 01, 010, 0100)$. Among them, only the node 0 has a left child, namely 00, which does not belong to the path. So $\mathsf{LEF}(0100)$ consists of only one node $\{00\}$. Similarly for 111, we have $\mathsf{LEF}(111) = \{0, 10, 110\}$, because every node of the path $(\varepsilon, 1, 11, 111)$ has a left child that does not belong to the path. It is worth noting that the size of $\mathsf{LEF}(\omega)$ is equal to the *hamming weight* of ω. Furthermore, no two nodes within the same LEF can have ancestor-descendant relation.

Our CPA-secure construction is provided in Fig. 4.[6] We show the following theorem, the proof is given in Appnedix B.

Theorem 6.1. *Under the decisional BDH assumption (DBDH, defined in Def. 4), there exists an TATLE scheme that satisfies* IND-TL-CPA *security against semi-honest adversary as per Def. 2 in the random oracle model.*

The construction follows the basic description from Sec. 3. We emphasize that while our construction works for both symmetric and asymmetric pairings, the latter provides smaller sized groups $\mathbb{G}_0$ (for the same level of security) (requiring fewer bits for encoding), and also more efficient group and pairing operations. Moreover, we designed our construction so that the elements of the aggregated key (i.e., the S values) are elements of the smaller group $\mathbb{G}_0$, while the public key is an element of $\mathbb{G}_1$. The ciphertext consists of $|\omega|$ elements of $\mathbb{G}_0$ (where $\omega = M^{-1}(\tau)$), 1 element from $\mathbb{G}_1$, and 1 element from target group $\mathbb{G}_T$; since a majority of elements of the ciphertext come from $\mathbb{G}_0$, we get a further reduction in our ciphertext size as well.

Thresholdizing Our threshold mechanism is a straightforward adaptation of distributed psuedo-random function [23] for multiplicative prime-order groups. We first recall their mechanism. The PRF functionality being computed collectively can be written as $f_\alpha(x) = \mathcal{H}(x)^\alpha$, where $\mathcal{H} : \{0,1\}^* \to G$ is a hash function (modeled as a random oracle) and the secret key is $\alpha \in \mathbb{Z}_p$. To distribute the evaluation of f, the secret key α must be secret shared between the parties. In the setup phase, a trusted party samples a master key $\alpha \leftarrow_\$ \mathbb{Z}_p$ and uses Shamir's secret sharing scheme [28] (see Def. 5) with a threshold t to create n shares $\alpha_1,\ldots,\alpha_n$ of α. Share α_i is given privately to the server i. We know that for any set of t parties $\{i_1,\ldots,i_t\} \subseteq [n]$, there exists integers (i.e. Lagrange coefficients) $\lambda_{i_1},\ldots,\lambda_{i_t} \in \mathbb{Z}_p$ such that $\sum_{j\in\{i_1,\ldots,i_t\}} \alpha_j\lambda_j = \alpha$. Therefore, it holds that

$$f_s(x) = \mathcal{H}(x)^\alpha = \mathcal{H}(x)^{\sum_{j\in\{i_1,\ldots,i_t\}}\lambda_j\alpha_j} = \prod_{j\in\{i_1,\ldots,i_t\}} (\mathcal{H}(x)^{\alpha_j})^{\lambda_j}$$

which can be computed in a distributed manner, by having each server i produce $H(x)^{\alpha_j}$.

Coming to our TATLE construction, we can write S_ω as a combination of values produced by the above DPRF f:

$$S_\omega = \mathcal{H}(\varepsilon)^\alpha \cdot \prod_{j=1}^{|\omega|} \mathcal{H}(\omega|_j)^\alpha = f_\alpha(\varepsilon) \cdot \prod_{j=1}^{|\omega|} f_\alpha(\omega|_j)$$

Reconstruction from partial keys leverages the natural homomorphism. Consider any set of t servers $\{i_1,\ldots,i_t\} \subseteq [n]$, who publish $\{S_{\omega,1},\ldots,S_{\omega,t}\}$ respectively. Then, we get:

$$\begin{aligned}
\prod_{j\in\{i_1,\ldots,i_t\}} S_{\omega,j}^{\lambda_j} &= \prod_{j\in\{i_1,\ldots,i_t\}} \left(\mathcal{H}(\varepsilon)^{\alpha_j} \cdot \prod_{k=1}^{|\omega|} \mathcal{H}(\omega|_k)^{\alpha_j} \right)^{\lambda_j} \\
&= \prod_{j\in\{i_1,\ldots,i_t\}} \left(\mathcal{H}(\varepsilon)^{\alpha_j\lambda_j} \cdot \prod_{k=1}^{|\omega|} \mathcal{H}(\omega|_k)^{\alpha_j\lambda_j} \right) \\
&= \mathcal{H}(\varepsilon)^\alpha \cdot \prod_{k=1}^{|\omega|} \left(\prod_{j\in\{i_1,\ldots,i_t\}} \mathcal{H}(\omega|_k)^{\alpha_j\lambda_j} \right) \\
&= \mathcal{H}(\varepsilon)^\alpha \cdot \prod_{k=1}^{|\omega|} \mathcal{H}(\omega|_k)^\alpha = S_\omega
\end{aligned}$$

[6] For notational convenience we assume that the KeyCombine algorithm works with responses from the first t servers, as opposed to any t servers. The generalization can be done in a straightforward manner.

Ingredients

- Let $\mathbb{G}_0$, $\mathbb{G}_1$, and $\mathbb{G}_T$ be multiplicative cyclic groups of prime order q such that there exists a bilinear pairing e: $\mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ that is efficiently computable and non-degenerate. Let $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ be generators of the respective groups.
- Hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{G}_0$ modeled as a random oracle
- $\mathsf{SSS}_{n,t,q}$ is a t out of n Shamir Secret Sharing scheme for Z_q
- A doubly-labeled tree Γ of depth d such that $T = 2^d - 1$

TATLE construction

- $\underline{\mathsf{Setup}(1^\kappa, T, n, t) \to (pp, lpk, (lsk_1, \ldots, lsk_n))}$:
  Choose uniform random $\alpha \leftarrow_\$ Z_q$. Then, set $pp := (\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T, e, q, \mathcal{H}, \Gamma)$; $lsk_{:=}\alpha$; $(lsk_1, \ldots, lsk_n) := \mathsf{SSS}_{n,t,q}(\alpha)$ and $lpk := g_1^\alpha$;
- $\underline{\mathsf{PartAggKeyGen}(lsk_j, \tau) \to K_{\tau,j}}$:
  Parse $\alpha_j := lsk_j$. Let $\omega := M^{-1}(\tau)$ and $(\omega_1, \ldots, \omega_\eta) := \mathsf{LEF}(\omega)$ where $\eta \in \{0, \ldots, d\}$ is the hamming weight of ω.
  Then letting $\omega_{\eta+1} := \omega$ set for $i = 1, \ldots, \eta+1$ compute:
  - $S_{i,j} := \left(\mathcal{H}(\varepsilon) \prod_{k=1}^{\ell_i} \mathcal{H}(\omega_i|_k)\right)^{\alpha_j}$, where $\ell_i = |\omega_i|$
  - $\tau_i := M(\omega_i)$.

  Output $K_{\tau,j} := ((\tau_1, S_{1,j}), \ldots, (\tau_{\eta+1}, S_{\eta+1,j}))$.
- $\underline{\mathsf{KeyCombine}(K_{\tau,1}, \ldots, K_{\tau,t}) =: K_\tau}$:
  Parse each $K_{\tau,j} := (\tau_1, S_{1,j}), \ldots, (\tau_{\eta+1}, S_{\eta+1,j})$ for some $\eta \in \{0, \ldots, d\}$. For each $i \in \{1, \ldots, \eta+1\}$ collect $(S_{i,1}, \ldots, S_{i,t})$ and then use Lagrange coefficients $\lambda_j \in Z_q$ to compute $S_i := \prod_{j \in [t]} S_{i,j}^{\lambda_j}$.
  Output $K_\tau := ((\tau_1, S_1), \ldots, (\tau_{\eta+1}, S_{\eta+1}))$.
- $\underline{\mathsf{Enc}(lpk, m, \tau) \to c}$:
  Let $\omega = M^{-1}(\tau)$. Sample uniform random $r \leftarrow_\$ Z_q$ and then compute:
  - $c_1 := (\tau, g_1^r, \mathcal{H}(\omega|_1)^r, \mathcal{H}(\omega|_2)^r, \ldots, \mathcal{H}(\omega)^r)$;
  - $c_2 := m \cdot e(\mathcal{H}(\varepsilon)^r, lpk)$;

  Output $c = (c_1, c_2)$
- $\underline{\mathsf{Dec}(lpk, K, c) =: m/\perp}$:
  Parse c as (c_1, c_2) and then:
  - parse $c_1 := (\tau', R, h_1, \ldots, h_\ell)$;
  - parse $((\tau_1, S_1), \ldots, (\tau_{\eta+1}, S_{\eta+1})) := K$.
  - if $\tau' > \tau_{\eta+1}$ then output $\perp$, else go to the next step;
  - identify the unique (τ_i, S_i) such that either $\tau_i = \tau'$ or $\omega_i := M^{-1}(\tau_i)$ is a prefix of $\omega' := M^{-1}(\tau')$;
  - set $d := e(S_i, R) \cdot (\prod_{i=1}^{\ell_i} e(h_i, lpk))^{-1}$ where $\ell_i := |\omega_i|$;

  Output $m := c_2 \cdot d^{-1}$

Figure 4: Our CPA-secure TATLE construction

6.1 Verifiable TATLE Construction

In this section we put forward an augmented TATLE construction which satisfies security against a malicious attacker and thereby yields a (publicly) verifiable TATLE construction. In addition to publishing elements of the aggregated key, the server must publish a NIZK proof (specifically, Schnorr's proof [27, 11] via the Fiat-Shamir transform [15]) to prove the key's validity.

For provable security, we use trapdoor commitments to commit to secret key shares of parties and generate NIZKs with respect to these commitments, in lieu of simply proving correctness with respect to the public key lpk – since our IND-TL-CCA adversary is allowed to corrupt parties after obtaining the public parameters output by Setup, we make use of trapdoor commitments to let the simulator open the commitments to a different values using a trapdoor. Correctness follows from the extractability property of the NIZK scheme and the binding property of the commitment scheme.

The changes from the base construction are highlighted in blue. Notably, the only changes take place in three algorithms: Setup, PartAggKeyGen, and KeyCombine. The Enc and Dec algorithms remain the same, so we omit mentioning them. We need some additional ingredients:

- A trapdoor commitment $(\mathsf{Setup}_{\mathsf{com}}, \mathsf{Commit})$ (Def. 7).
- Another hash function $\mathcal{H}' : \{0,1\}^* \to \{0,1\}^{\mathsf{poly}(\kappa)}$ modeled as a random oracle (within the NIZK).
- A SS-NIZK $:= (\mathsf{Prove}^{\mathcal{H}'}, \mathsf{Verify}^{\mathcal{H}'})$ (Def. 8).

The changed algorithms are described in Figure 5. We state the following theorem — the proof is provided in Appendix B.

Theorem 6.2. *Under the decisional BDH assumption (DBDH, defined in Def. 4), there exists an TATLE scheme that satisfies* IND-TL-CPA *security against malicious adversary as per Def. 2 in the random oracle model.*

Fig. 5 also defines our concrete instantiation of the trapdoor commitment and NIZK proofs. We use Pedersen commitments (using independent generators $g, h \in \mathbb{G}_0$, whose discrete log is the trapdoor), and Schnorr-style proofs (more generally, sigma protocols (see Sec. A.3)) made non-interactive using the Fiat-Shamir transformation in the random oracle model. The Setup phase outputs a commitment γ_i to each share α_i using randomness ρ_i.

Concretely, server i proves the following statement for ω:

$$\exists \alpha_i, \rho_i . \gamma_i := g^{\alpha_i} \cdot h^{\rho_i} \ \wedge \ S_{\omega,i} = \left(\mathcal{H}(\varepsilon) \prod_{j=1}^{|\omega|} \mathcal{H}(\omega|_j)\right)^{\alpha_i}$$

We emphasize that our proof contains 3 field elements of Z_q (where q is the order of group $\mathbb{G}_0$), and its size is independent of the bit-length of ω. The reason is that even though $S_{\omega,i}$ is a product of $|\omega|$ terms, it can be written as x^{α_i}, where $x =$

Verifiable TATLE construction

- $\mathsf{Setup}(1^\kappa, T, n, t) \to (pp, lpk, (lsk_1, \ldots, lsk_n))$: Choose uniform random $\alpha \leftarrow_\$ Z_q$. Let $lsk_{:=}\alpha$, $lpk := g^\alpha$ and run $(\alpha_1, \ldots, \alpha_n) := \mathsf{SSS}_{n,t,q}(\alpha)$. Run $\mathsf{Setup}_{\mathsf{com}}(1^\kappa)$ to get pp_{com}. Sample uniform random ρ_i and compute $\gamma_i := \mathsf{Commit}(pp_{\mathsf{com}}, \alpha_i; \rho_i)$. Then set: $pp := (\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T, e, q, \mathcal{H}, \Gamma, , pp_{\mathsf{com}}, \gamma_1, \ldots, \gamma_n)$ and $lsk_i := (\alpha_i, \rho_i)$.
- $\mathsf{PartAggKeyGen}(lsk_j, \tau) \to K_{\tau,j}$: Parse $\alpha_j := lsk_j$. Let $\omega := M^{-1}(\tau)$ and $(\omega_1, \ldots, \omega_\eta) := \mathsf{LEF}(\omega)$ where $\eta \in \{0, \ldots, d\}$ is the hamming weight of ω.

  Letting $\omega_{\eta+1} := \omega$ set for $i = 1, \ldots, \eta+1$ compute:
  - $w_0 := \mathcal{H}(\varepsilon)$, for $k \in \{1, \ldots, \ell_i\}$ $w_k := \mathcal{H}(\omega_i|_k)$.
  - $S_{i,j} := (\prod_{k=0}^{\ell_i} w_k)^{\alpha_j}$, where $\ell_i = |\omega_i|$
  - $\tau_i := M(\omega_i)$.
  - Run $\mathsf{Prove}^{\mathcal{H}'}$ for the language $\{\exists\ \alpha, \rho \text{ s.t. } S_{i,j} = (\prod_{k=0}^{\ell_i} w_k)^\alpha \wedge \gamma_i = \mathsf{Commit}(pp_{\mathsf{com}}, \alpha; \rho)\}$ with statement $(S_{i,j}, w_0, w_1, \ldots, w_{\ell_i}, \gamma_i)$ and witness (α_i, ρ_i) to obtain a proof $\pi_{i,j}$

  Then, set $K_{\tau,j} := ((\tau_1, S_{1,j}, \pi_{1,j}), \ldots, (\tau_{\eta+1}, S_{\eta+1,j}, \pi_{\eta+1,j}))$.
- $\mathsf{KeyCombine}(K_{\tau,1}, \ldots, K_{\tau,t}) =: K_\tau / \perp$: Parse each $K_{\tau,j} := (\tau_1, S_{1,j}, \pi_{i,j}), \ldots, (\tau_{\eta+1}, S_{\eta+1,j}, \pi_{\eta+1,j})$ for a $\eta \in \{0, \ldots, d\}$. For each $i \in \{1, \ldots, \eta+1\}$ collect $(S_{i,1}, \ldots, S_{i,t})$ and then let $\omega_i := M^{-1}(\tau_i)$ do as follows:
  - Compute $w_0 := \mathcal{H}(\varepsilon)$, for $k \in \{1, \ldots, \ell_i\}$ and $w_k := \mathcal{H}(\omega_i|_k)$.

  first check whether proof $\pi_{i,j}$ verifies with respect to the statements $(S_{i,j}, \tau_i, \gamma_j)$ ($w_0, w_1, \ldots,$ are calculated from τ_i), if not then output $\perp$, otherwise use Lagrange coefficients $\lambda_j \in Z_q$ to compute $S_i := \prod_{j \in [t]} S_{i,j}^{\lambda_j}$. Set $K_\tau := ((\tau_1, S_1), \ldots, (\eta+1, S_{\eta+1}))$.

Concrete instantiation

- $\mathsf{Setup}_{\mathsf{com}}(1^\kappa)$: Sample generator $h \leftarrow_\$ \mathbb{G}_0$, and output $pp_{\mathsf{com}} =: (g_0, h)$.
- $\mathsf{Commit}(pp_{\mathsf{com}} = (g, h), \alpha; \rho)$: output $g^\alpha \cdot h^\rho$
- $\mathsf{Prove}^{\mathcal{H}'}(\{\exists\ \alpha, \rho.\ S = (\prod_{k=0}^{\ell_i} w_k)^\alpha \wedge \gamma = g^\alpha \cdot h^\rho\}$ with statement $(S, w_0, w_1, \ldots, w_{\ell_i}, \gamma)$ and witness (α, ρ):
  Let $w = \prod_{k=0}^{\ell_i} w_k$. Sample $v, v' \leftarrow_\$ Z_p$ and set $t := w^v$, $t' := g^v \cdot h^{v'}$. Compute $c := \mathcal{H}'(g, h, S, w_0, w_1, \ldots, w_{\ell_i}, \gamma, t, t')$. Let $u := v - c \cdot s$ and $u' := v' - c \cdot r$. Output proof $\pi = (c, u, u')$.
- $\mathsf{Verify}(\pi = (c, u, u'))$ for statement $(S, w_0, w_1, \ldots, w_{\ell_i}, \gamma)$:
  Compute $t := w^u \cdot S^c$, $t_i' := g^u \cdot h^{u'} \cdot \gamma^c$ and output 1 iff $c = \mathcal{H}'(g, h, S, w_0, w_1, \ldots, w_{\ell_i}, \gamma, t, t')$.

Figure 5: Changes for the verifiable TATLE construction

$\mathcal{H}(\varepsilon) \prod_{j=1}^{|\omega|} \mathcal{H}(\omega|_j)$ is 1 group element. Therefore, aggregated keys are of size $\Theta(log(T))$ even in the verifiable construction.

6.2 CCA-security

Our base construction achieves IND-TL-CCA-security by using a variant of Fujisaki-Okamoto [16] transformation, analogous to the BTE construction of Canetti et al. [7]. Remarkably the only changes we need to make from our IND-TL-CPA secure base construction (c.f. Figure 4) are in the Enc and Dec algorithms. Therefore, one can easily deploy these changes together with the augmentation needed for malicious security as discussed in the previous subsection, thereby achieving a construction simultaneously satisfying verifiability and IND-TL-CCA security. We describe the changed Enc' and Dec' algorithms below. Those are generic extensions of the algorithms Enc and Dec respectively from the base construction. For that we will be needing a few more ingredients:

- A symmetric-key encryption scheme $(\mathsf{SE.Enc}, \mathsf{SE.Dec})$ that takes $\{0,1\}^\kappa$ bit key.
- Two hash functions $\mathcal{H}_1 : \mathbb{G}_T \to \{0,1\}^\kappa$ and $\mathcal{H}_2 : \mathbb{G}_T \times \{0,1\}^* \times \{0,1\}^* \to Z_q$ modeled as random oracles.
- $\mathsf{Enc}'(lpk, m, \tau) \to c$: Let $\omega := M^{-1}(\tau)$. Then:
  - Sample uniform random $s \leftarrow_\$ \mathbb{G}_T$.
  - Compute $c_1 \leftarrow \mathsf{Enc}(lpk, s, \tau; \mathcal{H}_2(s, \omega, m))$.
  - Compute $c_2 \leftarrow \mathsf{SE.Enc}(\mathcal{H}_1(s), m)$ where $\mathcal{H}_1(s)$ is used as the key.
  - Set $c := (\tau, c_1, c_2)$.
- $\mathsf{Dec}'(lpk, K, c) \to m/\perp$: Parse $(\tau, c_2, c_2) := c$ and let $\omega := M^{-1}(\tau)$ then:
  - Compute $s := \mathsf{Dec}(c_1)$.
  - Use $\mathcal{H}_1(s)$ as the key to decrypt $m := \mathsf{SE.Dec}(\mathcal{H}_1(s), c_2)$.
  - Then re-encrypt with the randomness $\mathcal{H}_2(s, \omega, m)$ to check whether $c_1 = \mathsf{Enc}(lpk, s, \tau; \mathcal{H}_2(s, \omega, m))$.
  - If the check succeeds then output m otherwise output $\perp$.

We state the following theorem, the proof for which is deferred to Appendix B.

Theorem 6.3. *Under the decisional BDH assumption (DBDH, defined in Def. 4), there exists an TATLE scheme that satisfies* IND-TL-CCA *security against semi-honest adversary as per Def. 2 in the random oracle model.*

Combining Theorem 6.2 with the above theorem we get the following corollary immediately.

Corollary 1. *Under the decisional BDH assumption (DBDH, defined in Def. 4), there exists an TATLE scheme that satisfies* IND-TL-CCA *security against malicious adversary as per Def. 2 in the random oracle model.*

7 Implementation and Evaluation

We measure several attributes of our TATLE scheme, including the size of aggregate keys, size of ciphertexts, and the running time of the individual algorithms.

We implement our TATLE scheme in Go, and make it available open source[7]. We use the 256-bit Barretto-Naehrig curves that support the Optimal Ate pairings as described in [22] (implementation from [1]). Benchmarks were run on a Macbook Pro with a 2.6 GHz 6-Core Intel Core i7 CPU and 16 GB DDR4 RAM.

7.1 Key Size

We measure the size of the aggregate key (produced by PartAggKeyGen) as a function of the (number of epochs in) lifetime T. This metric denotes the amount of key material output by each server in each epoch, and is indicative of the client-server bandwidth overhead, or alternatively, the space overhead of publishing the aggregate key on a bulletin board.

Recall that the size of the key K_τ for epoch $\tau < T$ depends on the number of tree nodes required to cover the range of epochs from 1 to τ. For that reason, we get a range of key sizes within a lifetime T. We expect $\Theta(log(T))$ number of nodes in the worst case, and each node has an associated S value in K_τ – each S value is an element of $\mathbb{G}_0$, of length 64 bytes when serialized in binary form. So, we collect key sizes for the entire range of epochs in a lifetime, and report the min, max, and the average (which unsurprisingly ends up being half of the largest key size). We also report the key size for both semi-honest and malicious settings, with the distinction being that the maliciously secure scheme has a NIZK proof (containing 3 field elements of 32 bytes each) alongside each S value. The results are given in Table 1.

epochs	stat	Semi-honest		Malicious	
		TATLE	IBE	TATLE	IBE
2^{10}	min	0.064	0.064	0.16	0.16
	max	0.640	65.47	1.6	163.7
	avg	0.320	32.74	0.8	81.84
2^{15}	min	0.064	0.064	0.16	0.16
	max	0.960	2.09×10^3	2.4	5.24×10^3
	avg	0.480	1.04×10^3	1.2	2.62×10^3
2^{20}	min	0.064	0.064	0.16	0.16
	max	1.280	6.71×10^4	3.2	1.68×10^5
	avg	0.640	3.35×10^4	1.6	8.39×10^4
2^{30}	min	0.064	0.064	0.16	0.16
	max	1.920	6.87×10^7	4.8	1.72×10^8
	avg	0.960	3.44×10^7	2.4	8.59×10^7

Table 1: Key size (in KBs): logarithmic growth for TATLE

For illustrative purposes, we compare with the size of key material when using an IBE-based scheme, such as the schemes from prior works [12, 9]. Since their keys grow linearly with the number of epochs T, it ends up being prohibitively large (in the order of terabytes) for even modest sized lifetimes. On the other hand, our keys grow logarithmically in T, and it is under 2 KB in the semi-honest and 5 KB in the malicious setting for a lifetime of 2^{30} epochs.

It is worth re-stating that when we operate in a t out of n threshold setting, the client needs to be given at least t partial keys of the size reported above.

7.2 Ciphertext Size

We report the ciphertext size when the message is encoded as a group element, which can also be thought of as the overhead from ciphertext expansion when encrypting a binary string in our CCA construction (see Sec. 6.2).

Similar to the case with key size, different epochs produce ciphertexts of varying size, depending on the position of the node in the tree – recall that for path-based identity ω of the node labelled τ, a ciphertext locked to epoch τ will have $|\omega|$ group elements from $\mathbb{G}_1$ (64 byte each), 1 element from $\mathbb{G}_0$ (128 bytes), and 1 element from $\mathbb{G}_T$ (384 bytes). Table 2 reports the min, max, and average statistics over the ciphertext sizes across the lifetime, for various values of T.

	Epochs			
	2^{10}	2^{15}	2^{20}	2^{30}
min	0.576	0.576	0.576	0.576
max	1.088	1.408	1.728	2.368
avg	1.025	1.344	1.664	2.304

Table 2: Ciphertext Expansion (in KBs)

Ciphertexts in TATLE grow logarithmically in the lifetime T, unlike IBE-based schemes which have constant-size ciphertexts containing 2 group elements. While this property of our scheme is far from ideal, we do not find it to be a practical hinderance for the proposed applications of time-locked encryption – it may present a challenge when encrypting large databases, but such use cases are out of scope.

7.3 Running Time

We report the running times of algorithms in our TATLE construction. The server runs PartAggKeyGen, while the client runs Enc and Dec using the whole aggregated key.

7.3.1 Key Generation

Caching Optimization We implement an obvious caching optimization when running PartAggKeyGen for consecutive epochs. For any node with id ω, S_ω is defined recursively as $S_{\omega'} \cdot \mathcal{H}(\omega)^\alpha$ (where ω' denotes the parent of ω). So, if we have cached the S value of any parent node, we can avoid

[7] https://github.com/gotatle/tatle.git

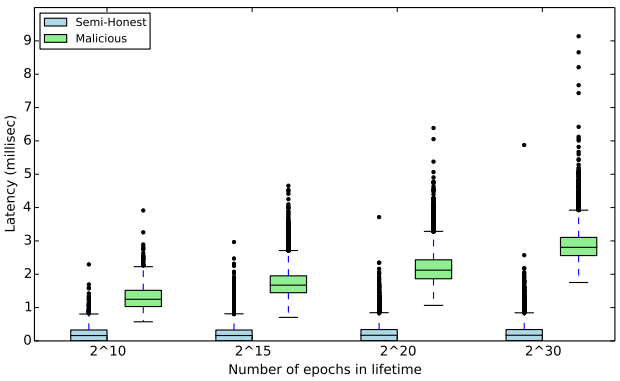

Figure 6: Running Time of PartAggKeyGen

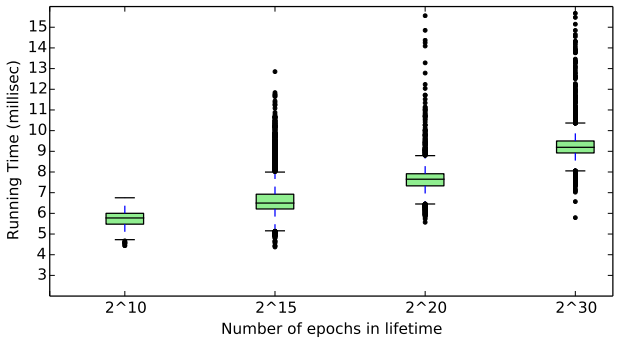

Figure 7: Running Time of Enc

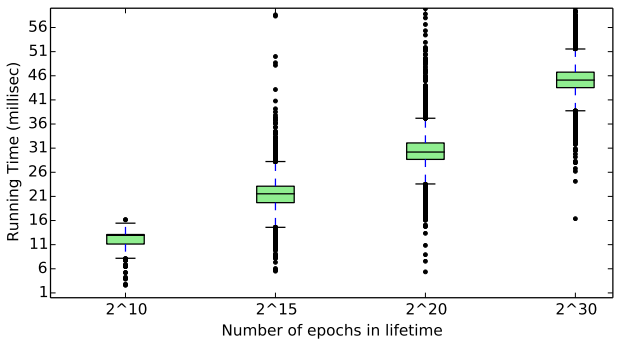

Figure 8: Running Time of Dec

recomputing several group operations. To that end, we maintain a cache comprising S values for each node along the path from the root node to ω, and remove S values of nodes that are no longer needed (because we will never output it in a key nor compute its child in future). Consider Fig. 1; in epoch 4 for instance, we remove S_{000} and S_{001} from the cache, and add S_{01} and S_{010}. We still compute $\Theta(log(T))$ new S values in the worst case (e.g. in epoch 8, where fresh S values must be computed along the entire path from the root). However, observe that intermediate nodes never compute fresh S values (since a leaf node would have already computed the necessary S values); more generally, we find a significant drop in the required computation across a large fraction of the nodes. We stress that the cache never exceeds the height of the tree, so it is at most 64 * $\lceil log(T) \rceil$ bytes (2 KB for 2^{30} epochs).

We illustrate the distribution of running times in Fig. 6. Due to caching, a large fraction of evaluations of PartAggKeyGen terminate under 1 ms, but we can observe the $\Theta(log(T))$ worst case running time in the outliers. Moreover, the quantiles grow with T in our malicious construction because the NIZK proof generation does not benefit from caching in the same manner as the S values.

Encryption and Decryption Fig. 7 and Fig. 8 report the running times for encryption and decryption, respectively. Similar to all other metrics, the running time for each Enc and Dec operation depends on the depth of the node – recall that a ciphertext includes a group element corresponding to each node along the path from the root node – so we plot the distribution from millions of trials, with each trial encrypting or decrypting from a random epoch.

References

[1] Advanced crypto library for the Go language. https://github.com/dedis/kyber.

[2] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. DiSE: Distributed symmetric-key encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1993–2010. ACM Press, October 2018.

[3] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. DiSE: Distributed symmetric-key encryption. Cryptology ePrint Archive, Report 2018/727, 2018. https://eprint.iacr.org/2018/727.

[4] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 213–229. Springer, Heidelberg, August 2001.

[5] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. Manuscript, 2020. https://toc.cryptobook.us/.

[6] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.

[7] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 255–271. Springer, Heidelberg, May 2003.

[8] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. Cryptology ePrint Archive, Report 2003/083, 2003. http://eprint.iacr.org/2003/083.

[9] Aldar C. F. Chan and Ian F. Blake. Scalable, server-passive, user-anonymous timed release cryptography. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, page 504–513, USA, 2005. IEEE Computer Society.

[10] Sanjit Chatterjee and Alfred Menezes. On cryptographic protocols employing asymmetric pairings – the role of Ψ revisited. Cryptology ePrint Archive, Report 2009/480, 2009. http://eprint.iacr.org/2009/480.

[11] David Chaum and Hans Van Antwerpen. Undeniable signatures. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 212–216. Springer, Heidelberg, August 1990.

[12] Jung Hee Cheon, Nicholas Hopper, Yongdae Kim, and Ivan Osipkov. Provably secure timed-release public key encryption. *ACM Trans. Inf. Syst. Secur.*, 11(2), May 2008.

[13] Mihai Christodorescu, Sivanarayana Gaddam, Pratyay Mukherjee, and Peter Rindal. Amortized threshold symmetric-key encryption. To appear in ACM CCS 2021, 2021. via personal communication.

[14] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the non-malleability of the Fiat-Shamir transform. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT 2012*, volume 7668 of *LNCS*, pages 60–79. Springer, Heidelberg, December 2012.

[15] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[16] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999.

[17] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, January 2007.

[18] Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 548–566. Springer, Heidelberg, December 2002.

[19] Jeremy Horwitz and Ben Lynn. Toward hierarchical identity-based encryption. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 466–481. Springer, Heidelberg, April / May 2002.

[20] Jia Liu, Tibor Jager, Saqib A. Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Des. Codes Cryptography*, 86(11):2549–2586, November 2018.

[21] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Time-lock puzzles in the random oracle model. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 39–50. Springer, Heidelberg, August 2011.

[22] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. Cryptology ePrint Archive, Report 2010/186, 2010. https://eprint.iacr.org/2010/186.

[23] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.

[24] Jianting Ning, Hung Dang, Ruomu Hou, and Ee-Chien Chang. Keeping time-release secrets through smart contracts. Cryptology ePrint Archive, Report 2018/1166, 2018. https://eprint.iacr.org/2018/1166.

[25] Michael O Rabin and Christopher Thorpe. Time-lapse cryptography. 2006.

[26] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, 1996.

[27] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[28] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[29] Michael Specter, Sunoo Park, and Matthew Green. Keyforge: Mitigating email breaches with forward-forgeable signatures, 2021.

A Additional Primitives

A.1 Commitment

Definition 6. *A (non-interactive) commitment scheme* Σ *consists of two PPT algorithms* $(\mathsf{Setup}_{\mathsf{com}}, \mathsf{Commit})$ *which satisfy hiding and binding properties:*

- $\mathsf{Setup}_{\mathsf{com}}(1^\kappa) \to pp_{\mathsf{com}}$: *It takes the security parameter as input, and outputs some public parameters.*
- $\mathsf{Commit}(m, pp_{\mathsf{com}}; r) =: \alpha$: *It takes a message m, public parameters* pp_{com} *and randomness r as inputs, and outputs a commitment* α.

Hiding. A commitment scheme $\Sigma = (\mathsf{Setup}_{\mathsf{com}}, \mathsf{Commit})$ is hiding if for all PPT adversaries $\mathcal{A}$, all messages m_0, m_1, there exists a negligible function negl such that for $pp_{\mathsf{com}} \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\kappa)$,

$$|\Pr[\mathcal{A}(pp_{\mathsf{com}}, \mathsf{Commit}(m_0, pp_{\mathsf{com}}; r_0)) = 1] - \Pr[\mathcal{A}(pp_{\mathsf{com}}, \mathsf{Commit}(m_1, pp_{\mathsf{com}}; r_1)) = 1]| \leq \mathsf{negl}(\kappa),$$

where the probability is over the randomness of $\mathsf{Setup}_{\mathsf{com}}$, random choice of r_0 and r_1, and the coin tosses of $\mathcal{A}$.

Binding. A commitment scheme $\Sigma = (\mathsf{Setup}_{\mathsf{com}}, \mathsf{Commit})$ is binding if for all PPT adversaries $\mathcal{A}$, if $\mathcal{A}$ outputs m_0, m_1, r_0 and r_1 $((m_0, r_0) \neq (m_1, r_1))$ given $pp_{\mathsf{com}} \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\kappa)$, then there exists a negligible function negl such that

$$\Pr[\mathsf{Commit}(m_0, pp_{\mathsf{com}}; r_0) = \mathsf{Commit}(m_1, pp_{\mathsf{com}}; r_1)] \leq \mathsf{negl}(\kappa),$$

where the probability is over the randomness of $\mathsf{Setup}_{\mathsf{com}}$ and the coin tosses of $\mathcal{A}$.

Definition 7 (Trapdoor (Non-interactive) Commitments.)**.** *Let* $\Sigma = (\mathsf{Setup}_{\mathsf{com}}, \mathsf{Commit})$ *be a (non-interactive) commitment scheme. A trapdoor commitment scheme has two more PPT algorithms* $\mathsf{SimSetup}$ *and* $\mathsf{SimOpen}$*:*

- $\mathsf{SimSetup}(1^\kappa) \to (pp_{\mathsf{com}}, \tau_{\mathsf{com}})$: *It takes the security parameter as input, and outputs public parameters* pp_{com} *and a trapdoor* τ_{com}.
- $\mathsf{SimOpen}(pp_{\mathsf{com}}, \tau_{\mathsf{com}}, m', (m, r)) =: r'$: *It takes the public parameters* pp_{com}*, the trapdoor* τ_{com}*, a message* m' *and a message-randomness pair* (m, r)*, and outputs a randomness* r'.

For every (m, r) *and* m'*, there exists a negligible function* negl *such that* $pp_{\mathsf{com}} \approx_{stat} pp'_{\mathsf{com}}$*, where* $pp_{\mathsf{com}} \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\kappa)$ *and* $(pp'_{\mathsf{com}}, \tau_{\mathsf{com}}) \leftarrow \mathsf{SimSetup}(1^\kappa)$*; and*

$$\Pr\left[\mathsf{Commit}(m, pp'_{\mathsf{com}}; r) = \mathsf{Commit}(m', pp'_{\mathsf{com}}; r')\right] \geq 1 - \mathsf{negl}(\kappa),$$

where $r' := \mathsf{SimOpen}(pp'_{\mathsf{com}}, \tau_{\mathsf{com}}, m', (m, r))$ *and* $(pp'_{\mathsf{com}}, \tau_{\mathsf{com}}) \leftarrow \mathsf{SimSetup}(1^\kappa)$.

Clearly, a trapdoor commitment can be binding against PPT adversaries only.

A.1.1 Concrete instantiations.

Practical commitment schemes can be instantiated under various settings:

Random oracle. In the random oracle model, a commitment to a message m is simply the hash of m together with a randomly chosen string of length r of an appropriate length.

DLOG assumption. A popular commitment scheme secure under DLOG is Pedersen commitment. Here, $\mathsf{Setup}_{\mathsf{com}}(1^\kappa)$ outputs the description of a (multiplicative) group G of prime order $p = \Theta(\kappa)$ (in which DLOG holds) and two randomly and independently chosen generators g, h. If $\mathcal{H} : \{0,1\}^* \to Z_p$ is a collision-resistant hash function, then a commitment to a message m is given by $g^{\mathcal{H}(m)} \cdot h^r$, where $r \leftarrow_\$ Z_p$. A trapdoor is simply the discrete log of h with respect to g. In other words, $\mathsf{SimSetup}$ picks a random generator g, a random integer a in $Z_p^\star$ and sets h to be g^a. Given (m, r), m' and a, $\mathsf{SimOpen}$ outputs $[(\mathcal{H}(m) - \mathcal{H}(m'))/a] + r$. It is easy to check that commitment to m with randomness r is equal to the commitment to m' with randomness r'.

A.2 Non-interactive Zero-knowledge

Let R be an efficiently computable binary relation. For pairs $(s, w) \in R$, we refer to s as the statement and w as the witness. Let L be the language of statements in R, i.e. $L = \{s : \exists w \text{ such that } R(s, w) = 1\}$. We define non-interactive zero-knowledge arguments of knowledge in the random oracle model based on the work of Faust et al. [14].

Definition 8 (Non-interactive Zero-knowledge Argument of Knowledge)**.** *Let* $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{\mathsf{poly}(\kappa)}$ *be a hash function modeled as a random oracle. A* NIZK *for a binary relation R consists of two PPT algorithms* Prove *and* Verify *with oracle access to* $\mathcal{H}$ *defined as follows:*

- $\mathsf{Prove}^{\mathcal{H}}(s, w)$ *takes as input a statement s and a witness w, and outputs a proof* π *if* $(s, w) \in R$ *and* $\perp$ *otherwise.*
- $\mathsf{Verify}^{\mathcal{H}}(s, \pi)$ *takes as input a statement s and a candidate proof* π*, and outputs a bit* $b \in \{0,1\}$ *denoting acceptance or rejection.*

These two algorithms must satisfy the following properties:

- ***Perfect completeness**: For any $(s,w) \in R$,*

$$\Pr\left[\mathsf{Verify}^{\mathcal{H}}(s,\pi) = 1 \mid \pi \leftarrow \mathsf{Prove}^{\mathcal{H}}(s,w)\right] = 1.$$

- ***Zero-knowledge**: There must exist a pair of PPT simulators $(\mathcal{S}_1,\mathcal{S}_2)$ such that for all PPT adversary $\mathcal{A}$,*

$$\left|\Pr[\mathcal{A}^{\mathcal{H},\mathsf{Prove}^{\mathcal{H}}}(1^\kappa) = 1] - \Pr[\mathcal{A}^{\mathcal{S}_1(\cdot),\mathcal{S}_2'(\cdot,\cdot)}(1^\kappa) = 1]\right| \leq \mathsf{negl}(\kappa)$$

  for some negligible function negl*, where*

  - *$\mathcal{S}_1$ simulates the random oracle $\mathcal{H}$;*
  - *$\mathcal{S}_2'$ returns a simulated proof $\pi \leftarrow \mathcal{S}_2(s)$ on input (s,w) if $(s,w) \in R$ and $\perp$ otherwise;*
  - *$\mathcal{S}_1$ and $\mathcal{S}_2$ share states.*

- ***Argument of knowledge**: There must exist a PPT simulator $\mathcal{S}_1$ such that for all PPT adversary $\mathcal{A}$, there exists a PPT extractor $\mathcal{E}^{\mathcal{A}}$ such that*

$$\Pr\Big[(s,w) \notin R \text{ and } \mathsf{Verify}^{\mathcal{H}}(s,\pi) = 1 \mid (s,\pi) \leftarrow \mathcal{A}^{\mathcal{S}_1(\cdot)}(1^\kappa); w \leftarrow \mathcal{E}^{\mathcal{A}}(s,\pi,Q)\Big] \leq \mathsf{negl}(\kappa)$$

  for some negligible function negl*, where*

  - *$\mathcal{S}_1$ is like above;*
  - *Q is the list of (query, response) pairs obtained from $\mathcal{S}_1$.*

Fiat-Shamir transform. Let $(\mathsf{Prove},\mathsf{Verify})$ be a three-round public-coin honest-verifier zero-knowledge interactive proof system (a sigma protocol) with unique responses. Let $\mathcal{H}$ be a function with range equal to the space of the verifier's coins. In the random oracle model, the proof system $(\mathsf{Prove}^{\mathcal{H}},\mathsf{Verify}^{\mathcal{H}})$ derived from $(\mathsf{Prove},\mathsf{Verify})$ by applying the Fiat-Shamir transform satisfies the zero-knowledge and argument of knowledge properties defined above. See Definition 1, 2 and Theorem 1, 3 in Faust et al. [14] for more details. (They actually show that these properties hold even when adversary can ask for proofs of false statements.)

A.3 Sigma Protocols

A sigma protocol allows a prover to convince the verifier that a witness satisfies a statement containing arbitrary linear relations (once we take discrete logarithms) in zero-knowledge, i.e., without revealing any other information about the witness. Let $\mathbb{G}$ be a cyclic group of prime order q generated by $g \in \mathbb{G}$. We consider statements of the following type:

$$\exists x_1,\ldots,x_n.\ u_1 = \prod_{j=1}^{n} g_{1j}^{x_j} \wedge \ldots \wedge u_m = \prod_{j=1}^{n} g_{mj}^{x_j}$$

Here, a witness is an assignment $(\alpha_1,\ldots,\alpha_n) \in Z_q^n$ to the variables $x_1,\ldots,x_n$ that makes the formula true, while g_{ij} and u_i values are group elements that are known to the verifier (e.g., public values or constants). The protocol between (P,V) for such a relation is as follows:

$$P \to V : u_1',\ldots,u_m' \in \mathbb{G} \text{ where } u_i' \leftarrow \prod_{j=1}^{n} g_{ij}^{\alpha_j'}, \alpha_j' \leftarrow_\$ Z_q$$

$$V \to P : c \leftarrow_\$ Z_q$$

$$P \to V : \tilde{\alpha}_1,\ldots,\tilde{\alpha}_n \in Z_q \text{ where } \tilde{\alpha}_j = \alpha_j' + \alpha_j c$$

$$V \text{outputs 1 iff } \left(\prod_{j=1}^{n} g_{ij}^{\tilde{\alpha}_j} \stackrel{?}{=} u_i' \cdot u_i^c\right) \text{ for } i = 1,\ldots,m$$

Fiat-Shamir for Sigma protocol Using the Fiat-Shamir transform, we can convert the Sigma protocol into a non-interactive zero-knowledge proof system as follows. Instead of obtaining the challenge c from the verifier V, the prover P uses $c = H(u_1,g_{1j},\ldots,u_m,g_{mj},u_1',\ldots,u_m')$, where H is modeled as a random oracle. In other words, we use a hash of the statement and the first message of P as the challenge.

B Security Proofs

B.1 Proof of Theorem 6.1

B.1.1 Correctness

First note that, the threshold secret-sharing scheme ensures that the algorithm KeyCombine correctly computes the aggregated key K_τ from t values $K_{\tau,1},\ldots,K_{\tau,t}$. A aggregated/epoch key K_τ consists of $\eta+1$ pairs $(\tau_1,S_1),\ldots,(\tau_{\eta+1},S_{\eta+1})$ where $\tau_{\eta+1} = \tau$. Note that, encryption of a message for some epoch τ' is given by

$$\left(\tau, g_1^r, \mathcal{H}(\omega|_1)^r, \mathcal{H}(\omega|_2)^r, \ldots, \mathcal{H}(\omega)^r, m \cdot d\right)$$

where $d = e(\mathcal{H}(\varepsilon)^r, g_1^\alpha)$. During decryption of such ciphertext using a key K_τ for which $\tau \geq \tau'$, the first task is to find the unique (τ_i,S_i) for which $\omega = M^{-1}(\tau)$ is a prefix of $\omega' = M^{-1}(\tau')$. By the doubly-labeled tree construction the uniqueness is easy to see — no two nodes can have a common descendant unless one of them is an ancestor of another. By our key-generation algorithm no epoch key can have two such node with ancestor-descendant relationships. Now, once such pair (τ_i,S_i) is found, then assuming $\ell_i := |\omega_i|$ the decryption algorithm computes

$$\begin{aligned} & e(S_i,R) \cdot \Big(\prod_{k=1}^{\ell_i} e(\mathcal{H}(\omega_i|_k)^r, g_1^\alpha)\Big)^{-1} \\ &= \frac{e(\mathcal{H}(\varepsilon)^\alpha, g_1^r) \cdot e(\mathcal{H}(\omega_i|_k)^\alpha, g_1^r) \ldots e(\mathcal{H}(\omega_i)^\alpha, g_1^r)}{e(\mathcal{H}(\omega|_1)^r, g_1^\alpha) \cdot \ldots \cdot e(\mathcal{H}(\omega)^r, g_1^\alpha)} \\ &= e(\mathcal{H}(\varepsilon)^r, g_1^\alpha) = d \end{aligned}$$

The final line follows by observing that ω_i is a prefix of ω, hence $\omega_i|_k = \omega|_k$ as long as $k \le \ell_i$ and using bilinear pairing. This conclude the proof of correctness.

B.1.2 CPA-security

Proof. For simplicity we assume that the adversary corrupts exactly $t-1$ parties. If it corrupts less than that, then some subtleties arise which can be resolved with techniques similar to the threshold symmetric encryption of Agrawal et al. [2]. Without loss of generality we assume that it corrupts parties with identities $1,\ldots,t-1$.

We assume a PPT adversary $\mathcal{A}$ that has a non-negligible advantage in the $\mathsf{CPA}_{\mathrm{TATLE},\mathcal{A}}$ game. We use $\mathcal{A}$ to construct a new adversary $\mathcal{B}$ that attacks the decisional BDH game with non-negligible success probability. That is, $\mathcal{B}$ acts as a Game challenger to $\mathcal{A}$ (and simulates the random oracle $\mathcal{H}_1$) and uses the output of $\mathcal{A}$ to solve the following DBDH problem[8]: when given description of groups $\mathbb{G}_0$ (with generator g_0), $\mathbb{G}_1$ (with generator g_1), $\mathbb{G}_T$, bilinear map e, and values ($A_0 = g_0^a$, $A_1 = g_1^a, B_1 = g_1^b, C_0 = g_0^c, C_1 = g_1^c$ and $D = e(g_0,g_1)^d$), $\mathcal{B}$ must determine whether $d = abc$ or not (where $a,b,c,d \leftarrow_\$ Z_q$).

First, $\mathcal{B}$ initiates the execution of $\mathcal{A}$, who must commit to the target epoch $\tau^\star$ that it wishes to attack. Let $\omega^\star := M^{-1}(\tau^\star)$ be the primary label corresponding to $\tau^\star$ in the doubly-labeled tree Γ and also let $\ell := |\omega^\star|$ bits.

Recall that for a bit string σ, $\sigma|_i$ denotes the i-bit prefix of σ, and we now let $\sigma|_{\bar{i}}$ denote the $i-1$-bit prefix followed by the negation of the ith bit of σ. Below we often sample a uniform random value, denoted as $\chi_\omega \leftarrow_\$ Z_q$ for each node ω.

Now $\mathcal{B}$, on query ω, chooses $\chi_\omega \leftarrow_\$ Z_q$ and programs the random oracle $\mathcal{H}$ as follows:

$$\mathcal{H}(\omega) := \begin{cases} B_0 & \omega = \varepsilon \\ g_0^{\chi_\omega}/B_0 & \omega \in \{\omega^\star|_{\bar{i}}, \omega^\star 0, \omega^\star 1\} \text{ for } i \in [\ell] \\ g_0^{\chi_\omega} & \text{otherwise} \end{cases}$$

Then it sets $lpk = A_0$, and give lpk to $\mathcal{A}$. On the corruption query, it sends $t-1$ uniform random values $lsk_1,\ldots,lsk_{t-1}$ and thereby implicitly sets $lsk_t := a_t$, which is obtained by Lagrange interpolation in the exponent from $A_0 = g_0^a$ and $g_0^{lsk_1},\ldots g_0^{lsk_{t-1}}$. Each aggregate key contains a set of S_ω values, which $\mathcal{B}$ computes as follows:

$$S_\omega = \prod_{j=1}^{|\omega|} A_1^{\chi_{\omega|_j}}$$

Note that the summation may comprise zero terms (if the root is the only common node between the path-based identities of ω and $\omega^\star$).

[8] we use the DBDH-3 assumption defined in [10] for Type 3 Pairings

Remark B.1 (Key Distribution)**.** *We verify that keys given to $\mathcal{A}$ have the correct distribution.*

$$\begin{aligned} S_\omega &= \mathcal{H}(\varepsilon)^a \cdot \prod_{j=1}^{|\omega|} \mathcal{H}(\omega|_j)^a \\ &= B_0^\alpha \cdot \left(\prod_{j=1, j\neq i}^{|\omega|} g_0^{\chi_{\omega|_j}\alpha} \right) \cdot (g_0^{\chi_{\omega|_i}}/B_0)^\alpha \\ &= \prod_{j=1}^{|\omega|} A_1^{\chi_{\omega|_j}} \end{aligned}$$

After some number of queries, $\mathcal{A}$ generates a challenge query with messages m_0 and m_1. $\mathcal{B}$ responds by sampling a random bit b and returning $(c_1, m_b \cdot D)$ where $c_1 := (C_1, C_0^{\chi_{\omega^\star|_1}}, \ldots, C_0^{\chi_{\omega^\star|_\ell}})$.

Remark B.2 (Ciphertext Distribution)**.** *We must verify that the ciphertext given to $\mathcal{A}$ follows the correct distribution.*

$$\begin{aligned} &(C_1, C_0^{\chi_{\omega^\star|1}}, \ldots, C_0^{\chi_{\omega^\star|\ell}}, m_b \cdot D) \\ &= (g_1^c, g_0^{\chi_{\omega^\star|_1}c}, \ldots, g_0^{\chi_{\omega^\star|_\ell}c}, m_b \cdot e(g_0,g_1)^d) \\ &= (g_1^c, \mathcal{H}(\omega^\star|_1)^c, \ldots, \mathcal{H}(\omega^\star|_\ell)^c, m_b \cdot e(g_0,g_1)^d) \end{aligned}$$

Finally, $\mathcal{A}$ responds with bit b', and $\mathcal{B}$ outputs 1 if $b = b'$ and 0 otherwise. Clearly, if the DBDH game sets $d = abc$, we get a valid encryption of m_b and then $\mathcal{B}$ has the same advantage at breaking DBDH as $\mathcal{A}$ at breaking CPA. On the other hand, if d is a random element of Z_p, then the last element is a random element of $\mathbb{G}_T$ and therefore the ciphertext is independent of b — probability that $\mathcal{B}$ outputs 1 is exactly $1/2$. Therefore the probability that $\mathcal{B}$ succeeds in breaking the DBDH game is at least $|\mathsf{CCAadv}[\mathcal{A}, \mathrm{TATLE}] - 1/2|$. This concludes the proof. □

B.2 Proof of Verifiability (Theorem 6.2)

This proof is very similar to the proofs provided in Agrawal et al. [2] and Christodorescu et al. [13] in the context of threshold symmetric-key encryptions. This is because the only changes made in the base TATLE construction (c.f. Figure 4) to obtain verifiability (c.f. Figure 5) is in the algorithms PartAggKeyGen and KeyCombine in a way that is very similar to the maliciously secure constructions provided in those works (see, for example, Figure 4 of [3]). In particular, similar to those constructions, here too we use a NIZK proof for exponent with respect to a statement involving trapdoor commitments. In the proof, the reduction $\mathcal{B}$, that attempts to break DBDH, additionally needs to produce dummy commitments as part of pp and simulated proofs when responding to the PartAggKeyGen queries on behalf of the honest parties. This

can be done by a few hybrids from the initial security game. We omit the details.

B.3 Proof of CCA-security (Theorem 6.3)

This proof is basically the same as the proof provided in the BTE scheme (see Theorem 2 of [8]). Basically the only difference from the standard Fujisaki-Okamoto transform is the inclusion of the node ω (equivalently the epoch information) with the hash function to derive the randomness. This is necessary in our context because, otherwise one may maul the ciphertext by keeping everything same and just change the epoch (namely to a lower value than the target) to make a legitimate decryption query and subsequently break indistinguishability. We omit the details.