Perpetual Encryption

Yevgeniy Dodis[1] and Daniel Jost[2]

[1] New York University, New York, USA
dodis@cs.nyu.edu
[2] Blanqet LLC, Chicago, USA
daniel@blanqet.net

Abstract. Traditional public-key encryption (PKE) schemes have a static secret key. This means that the secret key owner can decrypt any old ciphertext in perpetuity. Schemes with changing secret keys, supporting so-called epochs, have been considered for a variety of reasons such as post-compromise security or more simply supporting communication among a dynamically changing group. This complicates perpetuity, especially in settings where the epoch number itself is supposed to be confidential.

We address this concern by introducing perpetual encryption (PE). The scheme leverages a trusted group manager to distribute states for each epoch such that: (1) without such a secret state, ciphertexts look pseudorandom and, hence, hide the epoch number; and (2) parties in a later epoch can still decrypt encryptions for old epochs in sublinear time (in the number of epochs). More stringently, we require that whenever a party in an earlier epoch decrypts to a message m then any party in a later epoch must arrive at the same message, even for adversarially crafted ciphertexts.

We build an efficient, unbounded-epoch PE scheme based on anonymous PKE, hashing, and other standard symmetric primitives. We also extend our scheme to protect against malicious group managers, at the expense of limiting the maximum number of epochs. All our schemes have fixed-sized states and decryption times, independent of the number of epochs.

Keywords: PKE · post-compromise security · anonymity · multi-cast

Table of Contents

1 Introduction . . . 3
1.1 Perpetual Encryption: Definition . . . 4
1.2 Applications . . . 4
1.3 Related Primitives . . . 5
1.4 Overview of Our Scheme . . . 8
1.5 Cryptographic Assumptions . . . 9
2 Preliminaries . . . 9
2.1 Notation . . . 9
2.2 Building blocks . . . 9
2.3 Merkle-Tree and MT traversal . . . 11
3 Perpetual Encryption . . . 12
4 Perpetual Encryption Scheme . . . 15
4.1 Protocol Description . . . 15
4.2 Correctness and Security . . . 19
4.3 Strong Perpetuity and Interval Access . . . 23
5 Insider-Secure Perpetual Encryption . . . 24
5.1 Syntax and Security . . . 24
5.2 A Bounded Epoch Scheme . . . 25
5.3 Unbounded Epochs from Anonymous HIBE . . . 32

1 Introduction

Traditional public-key encryption (PKE) schemes have a static secret key sk. This means that the secret key owner can decrypt any old ciphertext c in perpetuity. Unfortunately, in some settings, one might be forced to change the secret key of the scheme. Two benign examples include: (a) various compliance regulations, demanding that one cannot use the same secret/public key for too long; or (b) *Post-Compromise Security* (PCS), where the user is trying to recover from suspected compromise of sk to ensure that future ciphertexts are secure. In this work, we will also be interested in another setting (c), where the secret key sk is shared by multiple users in a dynamically changing group G.

For instance, in the IETF Message Layer Security protocol [5], parties are arranged at the leaves of a binary tree with intermediate nodes having PKE key pairs assigned such that each party knows exactly the secret keys for nodes on the path from their leaf to the root. In this case, when a member U leaves the group, the secret key must change, as U should not be able to decrypt future ciphertexts. In the following, we assume the group is managed by some (semi-)trusted group manager M, which is basically the setting of *multicast encryption* [18,45,46,38,41] extensively studied in the literature (see [10] and references therein).

Perpetuity. A natural question to consider here is whether we still want the members of group G to be able to decrypt ciphertexts encrypted using an old public key. In the traditional multicast setting, the answer is negative: the new secret key is not required to support the decryption of old ciphertexts. As we observe in this work (and describe below), there are natural applications where this answer is insufficient, and one wants any "future" secret key to still decrypt any "past" ciphertexts. This leads to the notion of *perpetual encryption* we introduce in this work. As such, perpetual encryption can be seen as the dual of the widely studied notion of forward-secure encryption.[3]

One naive way to build perpetual encryption is to never erase any of the old secrets $(sk_1, \ldots, sk_n)$, and then simply use the appropriate sk_i when decrypting a ciphertext c encrypted using pk_i. The obvious drawback of this approach is that the storage of the user grows linearly with the number of *epochs* n. Thus, as the first question, we ask:

Question 1: *Can we make users' storage sublinear in the number of epochs* n?

A simple scheme addressing this can be built from a (prefix-)delegatable PRF (DPRF) as introduced by Kiayias et al. [35]. Instead of using independent PKE secrets $(sk_1, \ldots, sk_n)$, the group manager samples sk_i using $\mathrm{DPRF}_K(i)$, i.e., uses the output of the DPRF as randomness seed for the key generation. The prefix-delegation property then allows the manager to delegate a compact key $K_{[0,n]}$ that allows users to evaluate the PRF exactly on the range $[0, n]$. A well-known DPRF is the Goldreich-Goldwasser-Micali (GGM) construction [29], which further allows each individual evaluation to be done in time sublinear in n. Thus, if each ciphertext c additionally contains the epoch i for which it was encrypted, this ciphertext can be decrypted in sublinear time using the compact key $K_{[0,n]}$, for any $n \geq i$. However, in some of our applications we will have another concern, describe below.

Anonymity. While PKE hides the message content, hiding metadata is becoming an increasingly important as well. For instance, while traditional secure end-to-end encrypted messaging applications protect the content of messages but rely on metadata being transmitted in the clear, Signal recently attempted to address this shortcoming by introducing "sealed-sender" [42]. In this anonymity wrapper, metadata is encrypted under a fixed key of the recipient, with the use of anonymous PKE ensuring that for two ciphertexts one cannot tell whether they are for the same or different recipients.

One might thus ask whether we can achieve metadata hiding also in our setting. To this end, Perpetual Encryption must extend the anonymity notion of PKE to also hide epoch numbers, as otherwise for two recipients in different epochs the distinguishing is trivial. For simplicity, we formalize this anonymity goal by requiring *all ciphertexts encrypted at epoch* i *to appear pseudorandom* over a suitably defined ciphertext

[3] While the notions of perpetual encryption and forward-secure encryption are inherently incompatible, one could consider hybrid notions for which, for instance, metadata enjoys the perpetuity property, while content enjoys the forward security.

space, even against users knowing some secret key sk_j for an earlier epoch $j < i$. In particular, in the naive solution, where the user U stores all keys $(sk_1, \ldots, sk_n)$, U might not immediately know which secret key sk_i to use in order to decrypt c. Meaning that it might take U linear time $\Omega(n)$ to decrypt c, by doing a trial-and-error decryption until it succeeds. Thus, if we require that all ciphertexts c look random even given some sk_j for $j < i$, we ask:

Question 2: *Can we make users' decryption time sublinear in the number of epochs n, while hiding the epoch number of the ciphertext?*

Coming back to the simple DPRF solution, we see it does not hide the epoch number. Therefore, we need a novel scheme. We describe our idea in Section 1.4, but first let us formalize the resulting notion, and state some of its applications.

1.1 Perpetual Encryption: Definition

We now present a brief overview of our resulting notion of Perpetual Encryption (PE). In this notion, the group manager M commits to PE parameters pub, and is in charge of updating the current public key $pk = pk_i$ for epoch i. M is also in charge of helping users U obtain the current secret key $sk = sk_i$. Using pk_i (and pub), anybody can compute a ciphertext c encrypting some message m. Moreover, c should be decryptable using any secret key sk_n for any $n \geq i$, but pseudorandom otherwise; even against attackers having some secret keys sk_j for $j < i$, accounting for dynamic groups and/or PCS.

We call these conflicting properties *perpetuity* and *pseudorandomness*, respectively. In fact, as discussed in Section 3, both of these properties are stronger than stated above. In particular, pseudorandomness holds in light of chosen ciphertext attack. More interestingly, perpetuity holds even against a corrupt manager M, and even when trying to decrypt potentially *malformed ciphertexts*. Intuitively, even for such malformed ciphertexts (and potentially corrupt M), the commitment pub ensures that whenever a party in an older epoch decrypts any ciphertext c to some message $m \neq \perp$, then another party in the *same or a later* epoch must decrypt c to the same message m.

As stated in Questions 1 and 2 above, the major challenge is to design PE schemes where the user's state and decryption time do not scale linearly in the number of epochs n. As our main result, we show that such PE schemes can be constructed! In fact, in our schemes both user's storage and decryption time are logarithmic in n, which means they are always (fixed) polynomial in the security parameter λ, since $\lambda \geq \log n$ anyway.

1.2 Applications

We now motivate our notion of anonymous Perpetual Encryption. First, we note that achieving either anonymity or perpetuity in isolation is relatively straightforward: First, only perpetuity is required one can use independent key pairs per epoch and send epoch numbers in the clear, making perpetuity primarily a state-management problem. Second, if one If one does not require post-compromise security or long-term key management across epochs, standard anonymous PKE with rotating keys would suffice.

Needing to decrypt old ciphertexts, however, is quite natural. Simultaneously, there is also a strong utility of anonymity. Thus, resolving the non-trivial technical tension that arises when both properties are required simultaneously is not only of theoretical interest. To illustrate this, we highlight four key scenarios:

Out-of-Order Delivery. Consider an asynchronous network with out-of-order delivery. For any applications where PCS is relevant, this poses the issue of being able to decrypt delayed ciphertexts. Anonymity ensures that these delayed packets do not inadvertently reveal user patterns or metadata during the transition between epochs.

PCS for Sealed Sender. A further example might be an improved version of Signal's sealed-sender mechanism, where, to protect metadata (including the sender's identity), ciphertexts are additionally encrypted under the recipient's static public key, which does not have any PCS. Using PE, we can obtain

post-compromise security, without having to worry that a sender is still using an old key or messages being delivered out of order. This application is a clear example of Type 2 motivation: The specific goal of Sealed Sender is to hide metadata via an outermost encryption wrapper; without strict anonymity, the protocol's purpose is defeated.

Private Blockchain. Another application might be to design a *private* blockchain that reuses a *public* ledger — and thus avoids having its own consensus mechanism — by putting encrypted transactions onto a public blockchain. If one imagines such a private blockchain to be used by a company, then membership will however need to change with new employees being added and existing ones being removed when they leave the company. Therefore, to be practical, the encryption scheme must support PCS and, thus, rotate keys. Since the current state of a blockchain, such as the account balances, typically depends on the entire transaction history, any party newly joining such a private blockchain must be able to decrypt the entire prefix, requiring perpetuity. Finally, to maintain transaction privacy, it is essential to hide which "subchain" or sender is involved. Our construction's anonymity property is thus a necessary requirement to ensure these system-level privacy guarantees are met.

Mitigating travel exposure. Consider a setting where a party has a set of encrypted files. They might need access to any of them during their travel, for instance because they need to get some work done. However, this party is also concerned that their device gets compromised. For example, a border agent might coerce them to hand over all their passwords to search their devices upon entering a country. To mitigate the damage, the user might want to ensure that at least future files will not be compromised. To this end, they could use PE by having some secure device they leave at home. Instead, for their travel they use a disposable "burner device." The main device can then delegate decryption access to read all existing and current files, but not future files. Anonymity ensures that the ciphertexts do not reveal travel patterns by revealing how often this delegation process has taken place or when a particular file was encrypted.

1.3 Related Primitives

Recall that PE combines the PCS, anonymity, sublinearity and perpetuity properties. We organize related work by discussing related primitive for each of these properties (some primitives will be discussed multiple times). In particular, we focus on the two most closely related primitives key-insulated encryption [23] and interval encryption [9], and show how perpetual encryption combines (the best) aspects of those two notions. We conclude with summary why none of the primitives in the literature solves our problem.

Post-compromise security. A multitude of encryption primitives that support dynamic groups, or post-compromise security more generally, have been proposed in the past. For example, multicast encryption [18,45,46,38,41] has been extensively studied in the literature (see [10] and references therein) and assumes a semi-trusted group manager maintaining the current group membership. Continuous group key agreement (CGKA) [2] is a related primitive from the secure group messaging literature that does not rely on an explicit group manager. More broadly speaking, a line of group messaging schemes proposed schemes achieving forward secrecy and post-compromise security simultaneously (see e.g. the references in [3]).

However, most of these protocols internally rely on the naive approach of just sampling and distributing fresh key material to achieve PCS. In the following, we discuss two lines of work that bring PCS to the PKE setting and are particularly relevant to our work.

Interval key-encapsulation mechanism. In recent work, Bienstock et al. [9] introduced the notion of an interval key-encapsulation mechanism (IKEM) that generalizes forward-secure KEM to also include post-compromise security. Interval PKE can then be built using the standard KEM/DEM paradigm. Analogous to forward-secure PKE, the decryptor can refresh their secret key at regular intervals called epochs. As shown by the authors, PCS however requires the public key to be updated for each epoch (instead of the encryptor just learning the correct epoch number to encrypt for). Therefore, to account for parties that do not have the latest public key, IKEM allows to derive a decryption key that works

for a specific *interval* of epochs — hence the name interval KEM. A party could, for instance, derive a secret key that allows to decrypt all epochs t in an interval $[i, j]$, while later be able to update that key for epoch $j + 1$.

It is then shown that in this model the secret key must be of size linear in the interval size, i.e., linear in $j - i$. The authors provide a construction based on standard KEM which matches this lower bound while having constant sized ciphertexts and public keys, in the interval length. The construction moreover only uses a standard KEM, thus being optimal in terms of assumptions given that IKEM clearly implies standard KEM. To address the growing size of the secret key, the authors moreover propose a modified scheme that uses outsourcing to an untrusted server to store an encryption of this (still growing) key. In contrast, all of our states remain compact, at the cost of relying on a trusted group manager.

Notice, while our PE definition defaults to an interval $[1, j]$ (i.e., $i = 1$), we briefly sketch an extension of PE to the general $[i, j]$ interval setting (while maintaining anonymity and sublinearity) in Section 3.

Key-insulated encryption. To minimize the impact of secret-key exposures, Dodis et al. [23] introduced the notion of key-insulated PKE. In their model, the secret key stored on a insecure device is refreshed regularly via interaction with a physically-secre but computationally-limited device which stores a master key. Importantly, the public key used to encrypt remains unchanged and the insecure device can decrypt without interaction with the secure device. More concretely, each refresh establishes a fresh epoch such that compromising the insecure device in epoch t does not reveal ciphertexts encrypted for any epoch other than t, yielding both FS and PCS. (Of course, PCS depends on the secure device not helping the attacker to update the key obtained during the compromise, i.e., assumes proper authentication.) The authors further considered a stronger notion in which the secure device is potentially untrusted such that compromising the secure device alone, but not the insecure device, does not affect the security of any of the encryptions. This, for example, rules out simple solutions in which the secure device samples a master key from which all epochs keys are derived.

Key insulated encryption has spurred a number of follow up work. Most notably, Cheon et al. [21] demonstrated that a notion of time-released public key encryption is equivalent to key-insulated PKE with an untrusted secure device. In [7], on the other hand, Bellare and Palacio showed that for weak key-insulated PKE — where the secure device is trusted — the notion is equivalent to IBE when further requiring random-access key updates, i.e., when key updates do not have to be applied in order. In [30], Hanaoka, Hanaoka, and Imai considered the notion of parallel key-insulated public key encryption where two (or more) secure helper devices are used in an alternating fashion, such that exposing one of the device's state does not affect security for epochs initiated by the other device. Distributing the updating process among multiple devices can moreover enable more frequent updates for schemes where the process is bounded by the computational capabilities of the helper device. In [31], Hanaoka et al. considered a hybrid notion of identity-based and key-insulated encryption where decryption keys can be refreshed without having to make update to the public keys or the user's identity. Finally, key-insulated variants of various other primitives have been considered such as symmetric key encryption [25] or signatures [24].

Anonymity. Key anonymity for public-key encryption was first formalized as anonymous PKE in [6] and is also known under different names, such as source-indistinguishable PKE and PKE with key privacy. Anonymous PKE is also a fundamental building block for various anonymity systems, such as anonymous credential systems or anonymous broadcast encryption.

Anonymity is implied by pseudorandom ciphertexts [22,39,8,4,44], i.e., for any scheme where ciphertexts appear indistinguishable from pseudorandom elements given the public key. Unlike in the symmetric-key setting, where these notions are equivalent, PKE with pseudorandom ciphertexts does not follow black-box from PKE alone. Nonetheless, in the IND-CCA2 domain, many practical schemes are naturally pseudorandom over a suitable ciphertext space.

Unfortunately, anonymity is more troublesome in the PCS setting. Neither key-insulated encryption nor IKEM offer anonymity. More concretely, both schemes [23,9] generally leak the epoch number t as part of the ciphertext. Hence, if two public keys have been updated a different number of times, then the epoch number makes it trivial to distinguish which one has been used. Of course, the weak key-insulated scheme based on IBE (where the secure device is fully trusted) can be made anonymous using an anonymous

IBE scheme [11]; it is, however, an open problem to construct an anonymous key-insulated PKE scheme with strong security. Similarly, the IKEM [9] construction can be made anonymous by omitting the epoch numbers. However, this comes at the significant cost of making the decryption time linear in the interval size.

Sublinear overhead. Key-insulated encryption has constant overhead compared to a regular PKE scheme but does not naturally support decrypting ciphertexts from older epochs. To the best of our knowlege, for all existing schemes the only way of achieving this would be to keep the secret state of all the still relevant epochs stored. Similarly, IKEM has has linear overhead in terms of storage (unless outsourcing the storage). In particular, if we add anonymity both the solution based on maintaining multiple states for key-insulated encryption and the the IKEM scheme from [9] require trial-decryption for every epoch in the current interval $[i, j]$. In other words, with anonymity the decryption overhead of also grows linear in the interval as well.

Perpetuity. Being able to decrypt old ciphertexts shows up in various applications (cf. Section 1.2). Depending on the model, various solutions might be feasible. For example, updatable encryption (UE) [14,26,37,36,15,12] allows updating ciphertexts that were encrypted under a key at an epoch i to become decryptable by the key at an epoch $j > i$.

In our work, however, we consider settings where the original unmodified ciphertext remains decryptable. We observe that IKEM provides this for the given interval $[i, j]$. More concretely, the derived key for such an interval can decrypt ciphertexts for any epoch $i \leq t \leq j$ and especially a key for interval $[1, j]$ roughly matches our perpetuity notion (albeit with the drawbacks regarding anonymity and state size as discussed above).

Key-insulated encryption, on the other hand, does not provide any form of perpetuity. Instead, it provides forward secrecy, which can be seen as the dual property to perpetuity. Several additional lines of work on forward-secure encryption primitives exist. Notable examples are forward-secure PKE (FS-PKE) [19]. FS-PKE can generically be built from hierarchical identity-based encryption (HIBE) [28,32]. Importantly, FS-PKE only has a single public key that is input to the encryption algorithm alongside the epoch number t, eliminating the communication overhead.

In the context of secure messaging, more recent notions emerged, such as key-evolving encryption schemes [33,40] or updatable public key encryption (UPKE) [34]. These notions trade some increased communication and coordination overhead for otherwise more efficient instantiations. For instance, in UPKE, a sender initiates an update of the keys, where the updated public key must be distributed to all other senders, and the receiver must process the updates to their secret key in order. Importantly, in the random oracle model, UPKE can be instantiated from various regular PKE schemes with minimal overhead.

Novelty of primitive. Summarizing the discussion above, in this work, we explore the tension between three dimensions: (1) post-compromise security (PCS), (2) hiding metadata/epoch numbers, and (3) perpetuity. We abstract away the issue of (4) roster management in dynamic groups, as a likely (but possibly not only) reason for (1). To the best of our knowledge, no existing solution addresses all three aspects simultaneously. More concretely,

1. *Multicast Encryption:* addresses (1),(4), and optionally (2), but not (3). PE could likely be combined with multicast to also achieve (4), although other methods might be possible (e.g., two-factor authentication, email, physical registration).
2. *CGKA:* same as multicast, but without group manager and, hence, less efficient.
3. *Key-Insulated Encryption:* addresses (1) and optionally (2), but not (3). In fact, it cannot efficiently support (3) as it has forward secrecy.
4. *Interval-based KEM:* addresses (1),(3), but not (2).
5. *Anything Forward-Secure:* clearly contradicts (3).

We stress that even ignoring perpetuity, our formalization of metadata hiding under PCS (i.e., compromise of prior state) is stronger than most of the primitives mentioned above. For instance, multicast and CGKA completely ignore metadata, while fixed-key schemes cannot provide any level of PCS. In particular, the only existing solutions that provide metadata hiding would either involve a fixed key (so no PCS) or do not care about perpetuity (only decrypt for the latest epoch).

1.4 Overview of Our Scheme

In this section, we provide an outline of our scheme as well as discuss some of the challenges encountered. Our base scheme is built on PKE with pseudorandom ciphertexts, non-interactive commitments, a collision-resistant hash function, as well as a PRG and a PRF. We remark that PKE with pseudorandom ciphertext is clearly a minimal assumption as it is trivially implied by PE with pseudorandom ciphertexts. All operations run in time logarithmic in the maximum epoch number, and all parties' states are of logarithmic size too.

Basic idea. As seen before, the naive solution of using the GGM tree as a DPRF does not work. Neverthless, we still use binary tree of key pairs, but structure encryption differently to solve anonymity without exhaustive search in the decryption algorithm. More concretely, the group manager sets up a complete binary tree, of a given height, where epochs are consecutively assigned to leaf nodes. For each node of the tree v, the manager samples a PKE key pair $(v.sk, v.pk)$. (To have a compact representation, they use a PRF applied to the node index to derive the respective key generation randomness.)

The PE public key for epoch i then consists of the vector of PKE public keys along the path from epoch i's leaf to the root. Encrypting a message m under said public key corresponds to encrypting it under each of the PKE public keys. A decryptor in epoch j will know the secret keys for the minimal set of nodes such that exactly all the leaves from epoch 0 to j can be reached when traversing down the tree. A moment of reflection reveals that this set is the left-sibling path of the leaf of epoch $(j+1)$, i.e., all the direct left children of nodes encountered when traversing up from that leaf. Hence, the decryptor's state is of logarithmic size in the number of epochs while one can observe that as long as $i \leq j$ then the decryptor knows the secret key of exactly one node the message is encrypted to. Hence, by trial-decrypting under each of the logarithmic many secret keys they know (for each secret key trying to decrypt the component at the position in the vector corresponding to the node's height) they can recover the message.

Handling malicious ciphertexts. So far this obviously only handles honestly generated ciphertexts, as the different PKE encryptions are not guaranteed to be consistent. Since our notion allows for false positives, we can observe however that we only need upward consistency: if one of the component ciphertexts decrypts to m then all further up the tree must do so as well, but not necessarily the ones below. To see that this is sufficient, observe that for any two decryptors in epochs $j_1 \leq j_2$ the one in epoch j_2 will decrypt the one further up the tree.

As a result, we can use the following neat trick, instead of relying on NIZKs, to ensure consistency: The encryption derives the randomness for the various encryptions using a PRG from leaf to root while including the next seed as part of the encrypted message. That is, when encrypting to the ℓ-th component of the PE public key, the party will encrypt the message (s_ℓ, m) using randomness r_ℓ and then derive $(s_{\ell+1}, r_{\ell+1}) = \mathsf{PRG}(s_\ell)$. This does not introduce any circularity. Moreover, it now allows the decryptor to verify encryptions further up the tree by re-encrypting to obtained message under the recovered randomness.

For perpetuity, there is one more subtlety: The ciphertext must only successfully decrypt with respect to one epoch, despite being generated by somebody potentially knowing all secret keys. This is closely related to the notion of *strong robustness* introduced by Abdalla et al. [1], which is observed to not hold for most common schemes. (For our insider secure scheme discussed below, we would, however, need the robustness property to extend further to maliciously generated key pairs, called complete robustness [27].) To circumvent this, we have the PE ciphertext include a non-interactive commitment to the epoch number, with the opening included as part of the PKE encryptions. Parties then validate that the epoch number is consistent with the path they are trying to decrypt for, i.e., that the path under which they try to decrypt

is a suffix of the path from the epoch's corresponding leaf to the root. As a consequence, two parties in epochs j_1 and j_2 will now only both accept if one of their paths is a suffix of the other's path.

Chosen-ciphertext security. The scheme so far is IND-CPA secure; we want to achieve IND-CCA security. Assuming an IND-CCA secure PKE scheme does the job, except that we still need to tie the individual components together. We observe that any component already uniquely determines the consecutive components using our randomness trick. Hence it turns out to be sufficient to additionally form a hash chain over the components and the commitment, such that each component furthermore is bound to the prior components.

Insider security. The basic scheme achieves perpetuity provided the manager is honest. To achieve perpetuity against a malicious manager, we leverage a Merkle-Tree to ensure that parties agree on the tree of public keys. This, however, still does not necessarily ensure perpetuity, as it requires that whenever one of the parties successfully re-derives a certain ciphertext the other one who knows the secret key can actually decrypt, even if the key pair has been sampled under maliciously chosen randomness. This is addressed by requiring stronger properties on the PKE scheme to rule out such attacks. Since the manager now needs to compute the Merkle-Tree root as part of the initial setup, we have to limit the number of epochs to be polynomial in the security parameter.

1.5 Cryptographic Assumptions

Our constructions are based on standard public key encryption (PKE) and a number of symmetric primitives. Thus, while anonymous perpetual encryption is a stronger primitive than plain PKE in terms of security, we essentially establish an equivalence between PKE and PE in terms of cryptographi assumptions.

In the full version of this work, we introduce an alternative unbounded-epoch insider-secure scheme based on anonymous hierachical identity-based encryption (HIBE). This HIBE-based construction can be viewed as a more "direct" path to PE, at the cost of using stronger building blocks. It remains an open problem whether unbounded-epoch insider-secure PE can be built from plain PKE.

2 Preliminaries

2.1 Notation

We denote the security parameter as λ. For a probabilistic algorithm A, we write $\mathsf{A}(\cdot; r)$ to denote that A is run with explicit randomness r.

In pseudocode, we write $x \leftarrow a$ to assign the value a to the variable x. Moreover, for a set $\mathcal{S}$ we write $x \leftarrow\!\!\$\ \mathcal{S}$ to denote sampling an element from $\mathcal{S}$ uniformly at random. Finally, we make use of the following special keywords for both protocol descriptions and games: **req** *cond* evaluates a boolean condition *cond* and aborts the execution if *cond* evaluated to false, reverting changes to the state. That is, if in a game such a condition is violated in an oracle call, then said oracle call is ended with the game's state reset to the state before the invocation, while in a protocol the algorithm returns with a special error symbol $\perp$ and an unmodified state. When invoking a fallible algorithm on input x, we use **try** $y \leftarrow \mathsf{A}(x)$ to denote that in case A fails the error is propagated, i.e., the calling routine itself aborts with the error symbol $\perp$. We use **parse** $(x, y) \leftarrow z$ to denote the attempt of parsing z as a tuple (x, y) where the expected domains of x and y are assumed to be clear from the context.

We use binary trees. For a leaf of binary tree, we denote its root path as the path the leaf to the root of the tree. The leaf's sibling path is defined as the sequence of nodes obtained by taking the sibling for each node on the root path except for the root (ordered from leaf to root). Finally, the left-sibling path is obtained by restricting the sibling path to nodes that are the left child of their respective parent, and the right-sibling path is defined analogously.

2.2 Building blocks

Our schemes make use of a number of cryptographic building blocks.

Public Key Encryption. A public key encryption scheme $\mathsf{PKE} = (\mathsf{PKE.Keygen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$. We require the scheme to be IND-CCA2 secure with pseudo-random ciphertexts, over a suitably defined ciphertext space that is independent of the public key, even when given access to a decryption oracle. This, in particular, implies receiver anonymity. We remark that PKE with pseudorandom ciphertexts is somewhat of a non-standard assumption that does not follow black-box from PKE alone. (Unlike the symmetric-key setting, where these notions are equivalent.) Moreover, our schemes will require strong correctness properties. The exact properties vary by scheme, and we will detail them in the respective sections.

Non-Interactive Commitment. We assume a computationally binding and hiding non-interactive commitment scheme $\mathsf{COM} = (\mathsf{COM.Setup}, \mathsf{COM.Commit})$, for which a commitment to a value x $c_{\mathsf{COM}} \leftarrow \mathsf{COM.Commit}(crs_{\mathsf{COM}}, x; r_{\mathsf{COM}})$ looks indistinguishable from a uniformly random string. Here, $crs_{\mathsf{COM}} \leftarrow \mathsf{COM.Setup}(1^\lambda)$ denotes the CRS while $r_{\mathsf{COM}} \leftarrow\!\!\$\ \{0,1\}$ denotes the commitment randomness that simultaneously also serves as opening.

More concretely, for binding we require for every PPT adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{c|l} c_{\mathsf{COM}} = c'_{\mathsf{COM}} & crs_{\mathsf{COM}} \leftarrow \mathsf{COM.Setup}(1^\lambda); \\ \wedge\, x \neq x' & (x, x', r_{\mathsf{COM}}, r'_{\mathsf{COM}}) \leftarrow \mathcal{A}(1^\lambda, crs_{\mathsf{COM}}); \\ & c_{\mathsf{COM}} \leftarrow \mathsf{COM.Commit}(crs_{\mathsf{COM}}, x; r_{\mathsf{COM}}); \\ & c'_{\mathsf{COM}} \leftarrow \mathsf{COM.Commit}(crs_{\mathsf{COM}}, x'; r'_{\mathsf{COM}}) \end{array}\right]$$

to be negligible in λ. For hiding we require that

$$\begin{aligned} &\left\{\left(\mathsf{COM.Commit}(crs_{\mathsf{COM}}, x; U_\lambda), crs_{\mathsf{COM}}\right) \mid crs_{\mathsf{COM}} \leftarrow \mathsf{COM.Setup}(1^\lambda)\right\}_{\lambda\in\mathbb{N}} \\ &\quad\approx_c \left\{\left(U_{\ell(\lambda)}, \mathsf{COM.Setup}(1^\lambda)\right)\right\}_{\lambda\in\mathbb{N}} \end{aligned}$$

where U_n denotes the uniform distribution over $\{0,1\}^n$ and $\ell\colon \mathbb{N} \to \mathbb{N}$ denotes a suitable output-length function depending on the scheme's message space.

Hash functions. A family of hash functions $\mathsf{H}_k(\cdot)\colon \{0,1\}^* \to \{0,1\}^\lambda$, indexed by a public hash-key $k \in \{0,1\}^\lambda$. We require the function family to be collision-resistant, namely, that given k chosen uniformly at random any PPT adversary $\mathcal{A}$ has negligible probability in finding $x \neq x'$ such that $\mathsf{H}_k(x) = \mathsf{H}_k(x')$.

Pseudo-Random Function and Pseudo-Random Generators. We assume a pseudo-random function $\mathsf{PRF}\colon \{0,1\}^\lambda \times \{0,\ldots,N-1\} \to \{0,1\}^\lambda$. We require the standard security notion, i.e., that for a uniformly random chosen key $k \in \{0,1\}^\lambda$ any PPT adversary $\mathcal{A}$, having oracle access, cannot distinguish $\mathsf{PRF}(k,\cdot)$ from a uniformly random chosen function $F\colon \{0,\ldots,N-1\} \to \{0,1\}^\lambda$ with non-negligible advantage.

A length-doubling pseudo-random generator $\mathsf{PRG}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. We require that

$$\left\{\mathsf{PRG}(U_\lambda)\right\}_{\lambda\in\mathbb{N}} \approx_c \left\{U_{2\lambda}\right\}_{\lambda\in\mathbb{N}},$$

where U_n denotes the uniform distribution over $\{0,1\}^n$.

(Anonymous) Hierarchical Identity-Based Encryption. An identity-based encryption (IBE) scheme is an encryption scheme, which allows the owner of the master secret key to derive secret keys for identities $\mathrm{id} \in \{0,1\}^*$. A message encrypted for id can be decrypted only using the secret key for id. Hierarchical identity-based encryption (HIBE) [32,28] is IBE, which additionally allows delegation — every secret key corresponds to a vector of identities $\mathbf{id}$ and can be used to derive keys for vectors $\mathbf{id} \,\|\, \mathbf{id'}$ for arbitrary suffixes $\mathbf{id'}$. The HIBE secret keys can be seen as a tree, where the master secret key, corresponding to the empty $\mathbf{id}$, is the root, and keys at depth k correspond to $\mathbf{id}$'s of length k. A secret key allows decrypting ciphertexts encrypted for keys in its subtree, and only those.

Formally, an HIBE scheme is a tuple of algorithms $\mathsf{HIBE} = (\mathsf{HIBE.setup}, \mathsf{HIBE.kg}, \mathsf{HIBE.enc}, \mathsf{HIBE.dec})$ with the following syntax.

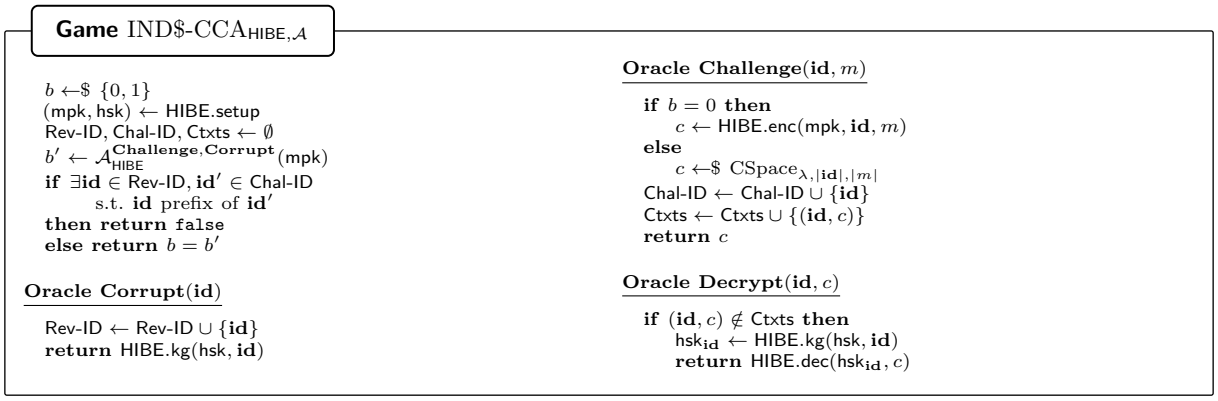

Fig. 1: The indistinguishability from randomness under chosen-ciphertext attacks game for HIBE.

- **Setup:** $(\mathsf{mpk}, \mathsf{hsk}_{()}) \leftarrow \mathsf{HIBE.setup}$ samples a fresh master public key and the corresponding root secret key.
- **Key Generation:** $\mathsf{hsk}_{\mathbf{id}\|\mathrm{id}} \leftarrow \mathsf{HIBE.kg}(\mathsf{hsk}_{\mathbf{id}}, \mathrm{id})$ outputs a lower-level secret key. For compactness, we write $\mathsf{hsk}_{\mathbf{id}\|\mathbf{id}'} \leftarrow \mathsf{HIBE.kg}(\mathsf{hsk}_{\mathbf{id}}, \mathbf{id}')$ to denote the iterative derivation of $\mathsf{hsk}_{\mathbf{id}\|\mathbf{id}'}$ using each $\mathrm{id} \in \mathbf{id}'$.
- **Encryption:** $c \leftarrow \mathsf{HIBE.enc}(\mathsf{mpk}, \mathbf{id}, m)$ encrypts m for identity vector $\mathbf{id}$.
- **Decryption:** $m \leftarrow \mathsf{HIBE.dec}(\mathsf{hsk}_{\mathbf{id}}, c)$ decrypts c.

We require that HIBE schemes satisfy IND$-CCA security, as captured by the game depicted in Fig. 1. The game formalizes a strong notion of recipient anonymity, by requiring each challenge ciphertext to be indistinguishable from a random ciphertext over a suitable ciphertext space that only depends on the security parameter λ and the depth of the identity $|\mathbf{id}|$ (and the message length). The game additionally offers corruption oracle, which outputs secret keys for identity vectors chosen by the adversary. In order to disable "trivial wins," the game stores the set Chal-ID of identity vectors used in challenge queries and the set Rev-ID of identity vectors used in corrupt queries, and checks that a corrupted identity is not a prefix of a challenged identity.

Definition 1. *Let* $\mathrm{Adv}^{\text{IND\$-CCA}}_{\mathsf{HIBE},\mathcal{A}} := 2\Pr[\text{IND\$-CCA}_{\mathsf{HIBE},\mathcal{A}} = \mathtt{true}] - 1$ *denote the advantage of* $\mathcal{A}$ *against the game defined in Fig. 1. A scheme* HIBE *with ciphertext space* $\mathrm{CSpace}_{\lambda,|\mathbf{id}|,|m|}$ *is* IND$-CCA *secure, if for all PPT adversaries* $\mathcal{A}$, $\mathrm{Adv}^{\text{IND\$-CCA}}_{\mathsf{HIBE},\mathcal{A}}$ *is negligible in* λ.

2.3 Merkle-Tree and MT traversal

One of our protocols furthermore uses a Merkle-Tree. More specifically, it uses a Merkle-Tree over a binary tree τ for which all nodes, even internal ones, have values to be authenticated by the Merkle-Tree root. In particular, our protocol will need to hand out the values along a root path, i.e., the path from a given leaf node to the root, with parties authenticating them against the Merkle-Tree root. To this end, the validating parties need the hashes of the sibling path, i.e., the hashes of all nodes that are siblings to nodes on the path. We call the sequence of values on the root path accompanied by the hashes of the sibling path a Merkle-Tree *opening*.

One peculiarity of our protocol will be that the group manager will represent this tree compactly, i.e., only using a logarithmically sized state. The manager keeps a different, logarithmically sized part, of the state expanded, re-deriving the missing pieces from a compact representation as needed. We can abstract this as a function ComputeNodeHash which computes the hash $v.h$ of a given node v based on the tree's compact representation as well as the hashes of its direct child nodes. As such, ComputeNodeHash abstracts away both the concrete hash function used and the representation of the data.

Note, however, that the tree's compact representation complicates Merkle-Tree openings since each node's hash can depend on linearly many values (preventing naive recomputation), and storing all the

hashes would require a linear-sized state. For our protocol it, however, suffices generate Merkle-proofs for paths from a leaf to the root *sequentially*, i.e., starting from the left-most leaf and ending at the right-most one. We observe that this problem exactly corresponds to the problem of *Merkle-Tree traversal* studied in e.g. [43,17].

We, therefore, for the rest of this work abstract this problem using the following interface that the protocol will use black-box.

Definition 2. *A* Merkle-Tree iterator*, for a binary tree τ, is a pair of PPT algorithms* (MtIterator.Init, MtIterator.Next) *defined as follows:*

- $(\mathrm{mtSt}, \tau) \leftarrow \mathrm{MtIterator.Init}(\tau, \mathrm{ComputeNodeHash})$: Initializes a new iterator for traversal and computes the Merkle-Tree root. It takes as input a binary tree τ (for which some of the nodes may have their data explicitly assigned) and a handle to the function ComputeNodeHash. The MtIterator.Init procedure outputs an iterator state mtSt as well as the modified tree τ in which the root has its Merkle-Tree hash assigned.
- $(\mathrm{mtSt}, \tau) \leftarrow \mathrm{MtIterator.Next}(\mathrm{mtSt}, \tau)$: Advances the iterator by one. After the n-th invocation, the updated tree τ is guaranteed to have the hashes on the sibling path of the n-th leaf filled in. In addition, we assume that it fills in the hashes of the sibling path's direct child nodes.[4] Note that MtIterator.Next does not assume any hashes to be filled in τ at the time of invocation; instead, it solely relies on its own state mtSt.

We remark that this interface can be implemented with a logarithmically sized iterator state mtSt and logarithmic time MtIterator.Next, see for instance [43,17]. For a complete binary tree with n leaves, MtIterator.Init takes $O(n)$ time in order to compute the Merkle-Tree root. It will be up to the protocol to ensure that τ is only of logarithmic size by pruning the parts of the state no longer needed.

3 Perpetual Encryption

We now give a precise definition of (public key) Perpetual Encryption. A PE scheme is a generalization of a public-key encryption (PKE) scheme and involves three types of parties: the manager, decryptors, and encryptors. In addition, we allow for a scheme to rely on a public setup, generated by a respective algorithm.

Definition 3 (Perpetual Encryption). *A public-key* perpetual encryption (PE) *scheme consists of the following PPT algorithms:*

- ▷ $crs \leftarrow \mathrm{GenCRS}(1^\lambda)$
  Samples the public setup based on the security parameter.
- ▷ $(mst_0, pub, pk_0, sk_0) \leftarrow \mathrm{Setup}(1^\lambda, crs, N)$
  Takes the security parameter, CRS, and the maximum number of supported epochs N. Initiates the group manager's state mst_0 and generates additional public parameters pub (chosen for the specific instance of the scheme). It further outputs an initial public-key/secret-key pair (pk_0, sk_0).
- ▷ $(mst_{i+1}, pk_{i+1}, sk_{i+1}) \leftarrow \mathrm{Update}(mst_i)$
  Initiates a new epoch by updating the manager's state and outputting a new public-key / secret-key pair.
- ▷ $c \leftarrow \mathrm{Encrypt}(crs, pub, pk_i, m)$
  Encrypts the message m under pk_i, the public key of the i-th epoch.

[4] Any Merkle-Tree traversal scheme can be adapted accordingly without loss of generality: either the additional values can be stored alongside the sibling-path node in mtSt, or they are needed to compute the sibling-path node's hash anyway.

▷ $(m, j) \leftarrow \text{Decrypt}(crs, pub, sk_i, c)$
Decrypts the ciphertext c using sk_i, the secret state of the i-th epoch. Outputs the message m and epoch $j \leq i$ under which the message was encrypted. The output can be $\perp$ to indicate a decryption error.

The group manager uses the Setup algorithm to initialize the scheme. Afterward, they can advance to the next epoch using Update. The group is maintained implicitly by distributing new secret keys to the respective parties — choosing an appropriate mechanism is left to the application. Encryption and decryption work analogously to the algorithm of a regular PKE scheme, except for also taking the public setup *crs* and public parameters *pub*, as well as the ability to decrypt under a later secret key as detailed below.

We remark that the above syntax is quite general. For instance, in case a scheme does not need to update its public key — analogous to, e.g., the formalization of forward secure PKE [19] — then one can simply set *pub* to be the static public key and have *pk* just denote the epoch number i.

Correctness and Security. We define the correctness and security of a PE scheme. For security, we consider two properties, formalized as separate games.

Correctness. We say that a PE scheme is correct, if any key pair (pk_i, sk_i) behaves like a PKE scheme. More concretely, if a message encrypted under the public key pk_i of epoch i can be successfully decrypted under the secret key sk_i for the same epoch. (The ability to decrypt under the secret key of a later epoch will be formalized as part of security.) Moreover, we require that honestly generated public keys and secret keys successfully validate by the respective algorithms.

Definition 4. *A PE scheme is called computationally* correct *if, for any number of maximal epochs N, any PPT adversary $\mathcal{A}$ has negligible winning probability in the game depicted in the left of Fig. 2. It is information-theoretically correct if this holds for any $\mathcal{A}$ that makes at most polynomially many oracle queries.*

Perpetuity. Perpetuity guarantees that members in a later epoch can still decrypt. Here we consider an honest-but-curious group manager. More concretely, the game depicted on the right side of Fig. 2 generates the public parameters *pub* as well as all public keys and secret keys honestly. The adversary, however, is given the manager's state *mst* as well as all secret keys. They then try to produce a ciphertext c that successfully decrypts in an epoch i, but not in a later one $j > i$.

Observe that perpetuity does allow for false negatives: it is possible for the party j in the later epoch to decrypt while the party i in an earlier epoch rejects by outputting $\perp$. As an alternative, we also introduce *strong perpetuity* which rules out such false negatives. Whether such spurious decryptions pose a problem is application dependent — for example, in our private blockchain, one is able to tolerate false negatives at the cost of some superfluous complaints.

Definition 5. *A PE scheme satisfies* perpetuity *if any PPT adversary $\mathcal{A}$ has at most negligible probability of having the game depicted on the right in Fig. 2 returning* 1*. It is said to satisfy* strong perpetuity *if this holds with respect to the game omitting the $m_i \neq \perp$ check.*

Pseudorandomness. First, we consider *pseudorandomness* which prescribes both the privacy of messages towards members that are not part — or no longer part — of the group, and receiver anonymity. For this property, the manager is assumed to be honest. In a nutshell, the property corresponds to IND-CCA security with pseudorandom-looking ciphertexts: The adversary chooses a challenge message m^* and epoch i^*. It is then given either a real encryption (if $b = 0$) or a uniform random ciphertext (if $b = 1$) to distinguish. We assume the ciphertext space $\text{CSpace}_{\lambda,N}$ to be a function of the security parameter and the maximal number of epochs only.[5] The adversary is given secret keys $sk_1, \ldots, sk_{i^*-1}$, representing the state

[5] For instance, for a scheme based on a cyclic group $\mathbb{G}$, the choice of $\mathbb{G}$ must not depend on *crs*. Of course, *crs* might still choose a specific generator $\langle g \rangle = \mathbb{G}$ of the group.

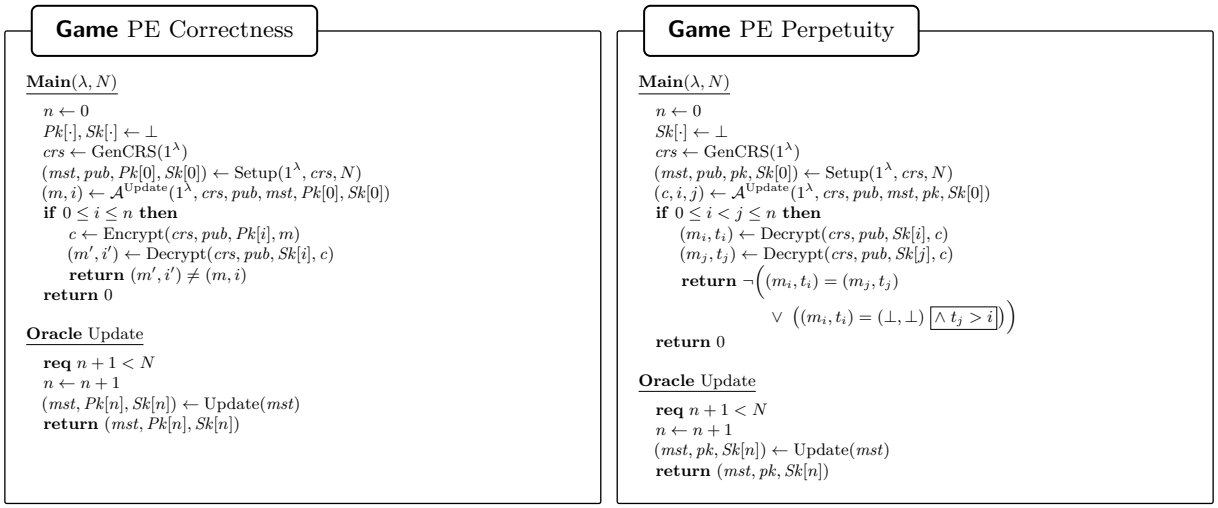

Fig. 2: The correctness and perpetuity games for a PE scheme. For the perpetuity game, the variant of strong perpetuity is obtained by adding the boxed condition.

of former members of the group, as well as a decryption oracle under keys $sk_{i^*}, sk_{i^*+1}, \ldots$ for ciphertexts other than the challenge c^*. Note that this formalizes PE for some fixed-length message space. This could be relaxed by making the ciphertext space $\mathrm{CSpace}_{\lambda,N}$ depend on the message length. The game is depicted in Fig. 3. Observe that the formal game allows the adversary to choose the challenge message m^* and epoch i^* adaptively based on public and secret keys observed — subject to a non-triviality condition that $i^* > e_{\mathsf{corr}}$ where e_{corr} denotes the maximum epoch for which the adversary learned the secret key.

Definition 6. *A PE scheme with ciphertext space* $\mathrm{CSpace}_{\lambda,N}$ *is* pseudorandom *if for any PPT adversary* $\mathcal{A}$ *the game depicted in Fig. 3 returns* 1 *with at most one-half plus negligible probability.*

Unbounded Perpetual Encryption. So far, we formalized a notion of PE with a bounded number of epochs N, which is fixed during Setup. One can also consider Perpetual Encryption with no a priori bound on the number of epochs. Syntax-wise, Setup then only takes the security parameter 1^λ and the *crs* as inputs. In terms of correctness and security, the games apply when setting the bound to infinity, thereby allowing the Update oracle to be called an arbitrary (polynomial) number of times.

Interval Access. Finally, we briefly sketch an extension inspired by interval KEM [9]. Namely, while in plain PE a secret key sk_i works to decrypt ciphertexts for epochs $1 \le t \le i$, we can also ask for the existence of weaker keys that can only decrypt within a specific interval $[i_0, i_1]$. More specifically, following the convention of [9], we require that a party can derive a key for any suffix of their current key without the help of the group manager. Using the convention that $sk_i = sk_{[}0, i]$, we thus augment PE as follows:

Definition 7. *An* interval perpetual encryption *scheme is a PE scheme with the following additional algorithm:*

▷ $sk_{i'_0,i_1} \leftarrow \mathrm{Delegate}(sk_{i_0,i_1}, i'_0)$
Derives a decrption key for epochs in the interval $[i'_0, i_1]$, where $i_0 < i'_0 < i_1$.

We only sketch the modifications to the corresponding games. For correctness, we require that sk_{i_0,i_1} can decrypt honestly generated messages for any epoch in the interval $[i_0, i_1]$. Similarly, for (strong) perpetuity we require that sk_{i_0,i_1} decrypts consistent with sk_j for epochs $i_0 \le j \le i_1$ (where the weak notion allows false positives if $j < i_1$. Finally, for pseudorandomness we require that a ciphertext for epoch $j < i_0$ remains pseudorandom even when given sk_{i_0,i_1}.

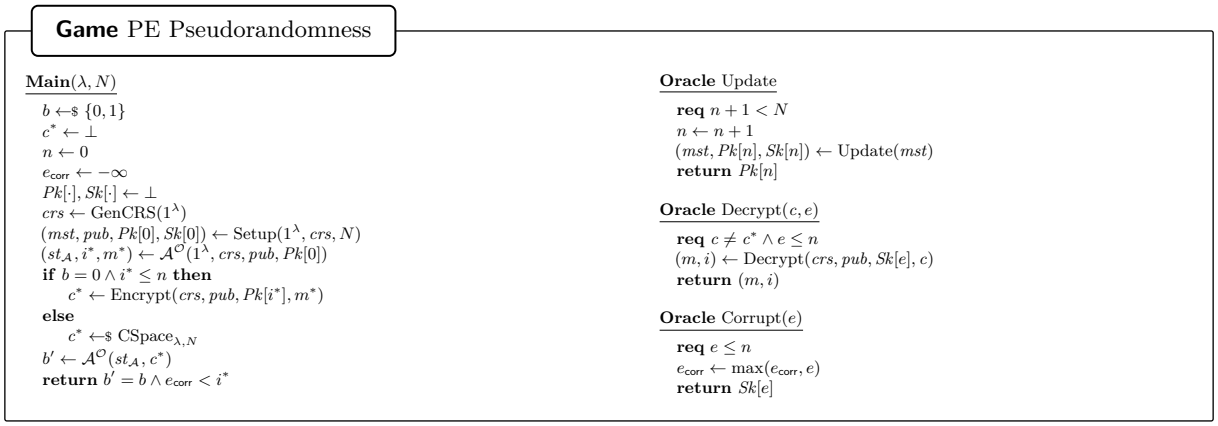

Fig. 3: The pseudorandomness game for a PE scheme. $\mathcal{O}$ denotes the set of defined oracles. The advantage of an adversary $\mathcal{A}$ is defined as the probability of the interaction between $\mathcal{A}$ and the game returning 1, minus $\frac{1}{2}$.

4 Perpetual Encryption Scheme

In this section, we first present a PE scheme for a bounded number of $N = 2^d$ epochs, input to Setup. We will, however, see that we can choose $N = 2^\lambda$ with the scheme still polynomially efficient, yielding an unbounded-epoch scheme. Of course this will come at a loss of concrete efficiency, as the key sizes as well as the efficiency of most algorithms is proportional to $\log(N)$. Therefore, there will be a trade-off between the number of epochs supported and the scheme's efficiency, making it advisable to choose N only as large as needed for the concrete use-case.

4.1 Protocol Description

We now present our (bounded-epoch) PE scheme. The scheme is primarily based on public key encryption (PKE). A pseudocode description of the scheme from Section 4 is presented in Figs. 4 to 6. For simplicity, the algorithms make use of a binary tree τ with N leaves as intermediate state. For example, the manager algorithms Setup and Update create such a tree and then expand the key pairs of the relevant nodes. Nodes $v \in \tau$ can have its PKE public key $v.pk$ and its secret key $v.sk$ assigned.

Based on this tree the algorithms then construct the PE public and secret keys. Analogously, the encryption and decryption algorithms first build the tree based on the PE public and secret keys, respectively. To this end, we use the following notation:

τ.root	Returns the root of the tree τ.
τ.Leaf(i)	Returns the i-th leaf of the tree.
v.idx	Returns the node index of v.
v.height	Returns the height of node v, i.e., the number of edges on the longest path from v to any leaf.
v.parent	Returns the parent of v (or $\perp$ if v is the root).
v.left	Returns the left child of v (or $\perp$ if v.IsLeaf).
v.right	Returns the right child of v (or $\perp$ if v.IsLeaf).
IsLeaf(v)	Returns true iff v has no children.
IsLeftChild(v)	Returns true iff v is its parent's left child.
IsRightChild(v)	Returns true iff v is its parent's right child.

Sibling(v)	Returns the unique node $u \neq v$ s.t. they have the same parent node (or $\perp$ if v is the root).
$\tau.\text{RootPath}(i)$	Returns the root path of the i-th leaf, i.e., the path from $\tau.\text{Leaf}(i)$ to $\tau.\text{root}$.
$\tau.\text{SiblingPath}(i)$	Returns the sibling path of the i-th leaf, i.e., siblings of nodes on $\tau.\text{RootPath}(i)$.
$\tau.\text{LeftSiblingPath}(i)$	Returns the left-sibling path of the i-th leaf, i.e., nodes on $\tau.\text{SiblingPath}(i)$ for which IsLeftChild(v) returns true.

Protocol overview. We now present a high-level overview of our bounded-epoch PE scheme.

Common reference string. The CRS consists of a hash key k for a collision-resistant hash function $\mathsf{H}_k(\cdot)$, as well as the public parameters crs_{COM} of the non-interactive commitment scheme COM.

Manager state. The manager algorithm Setup creates a binary tree with N leaves, one per epoch. For convenience, we assume nodes to be indexed by the binary string idx representing the path from the root (with an empty index) to the respective node. Each node idx has a public-key/secret-key pair (pk_{idx}, sk_{idx}) associated. To sample those keys, the manager first chooses a PRF seed s uniformly at random as part of the Setup algorithm. See Fig. 13 for a depiction of the key tree. Whenever the group manager needs any of those PKE keys for a node with index idx, it lazily generates them using the PRF: First it derives randomness $r_{idx} = \mathsf{PRF}(s, idx)$, under a suitable encoding from the node index idx to the domain of the PRF, and then uses this to derive the key pair $(pk_{idx}, sk_{idx}) = \mathsf{PKE.Keygen}(1^\lambda; r_{idx})$. This allows to keep only the PRF seed s as state, which remains unchanged throughout the rest of the protocol execution.

The public parameter pub simply stores the maximum number of epochs N. The initial public key pk_0 and pk_1 are computed analogously to the ones for later epochs — as described in the next section.

Public and secret keys. For each epoch i, the PE public key pk_i consists of:
- The epoch number i.
- For each node v on the path from the i-th leaf to the root, it contains the corresponding public key $v.pk$.

The corresponding secret key sk_i consists of the following components:
- The epoch number i.
- The PKE secret keys of the smallest set of nodes such that exactly the leaves from $idx_0 = \mathsf{encode}(0, d)$ to $idx_i = \mathsf{encode}(i, d)$ are reachable. It is easy to verify that this so-called cover consists exactly of the left-sibling path of node $idx_{i+1} = \mathsf{encode}(i+1, d)$ — i.e., the subset of the sibling path of node idx_{i+1} which are left children of their respective parents. Let V_i denote this set of node indices for epoch i. Figure 7 illustrates the sets V_2, V_3, and V_4.
- The PKE public keys on the root path of the $(i+1)$-th leaf.

Encryption and decryption. Finally, let us consider encryption and decryption. In a nutshell, when encrypting a message m in epoch i one considers the path from the corresponding leaf to the root and encrypts it to all those public keys. A party knowing secret key sk_j, for $j \geq i$ knows the PKE secret keys of nodes in V_j, which contains exactly one of the nodes on the path from the leaf to the root, ensuring that each such party can decrypt. Such a scheme, however, would not achieve perpetuity as a malicious encryptor could include inconsistent, or even invalid, encryptions as part of the PE ciphertext.

We make the following observation: for two parties in epochs j_1 and j_2 with $j_1 < j_2$, respectively, the party in epoch j_1 decrypts a ciphertext further down the tree (i.e., closer to the leaf). Furthermore, recall that perpetuity requires that if the party in epoch j_1 decrypts to a valid message m so does the one in j_2, not necessarily the other way round. We leverage this as follows. In the encryption process, the encryptor generates the encryption randomness using a length-doubling PRG, by deriving the randomness from leaf to root. That is, they choose an initial seed s_{-1}, which they then expand to (s_0, r_0) and use r_0 as randomness to encrypt to the leaf's public key. The encrypted message must now not only contain m

Protocol PE Manager

Setup($1^\lambda, crs, N$)

parse $(k, crs_{\mathsf{COM}}) \leftarrow crs$
$n \leftarrow 0$
// Sample PRF seed
$s \leftarrow\!\!\$\ \{0,1\}^\lambda$
// Initial keys
$\tau \leftarrow$ new tree with N leaves
$pk \leftarrow \mathrm{GenPk}(\tau, 0)$
$sk \leftarrow \mathrm{GenSk}(\tau, 0)$
// Only keep compact state
$mst \leftarrow s$
$pub \leftarrow N$
return (mst, pub, pk, sk)

Update(mst)

$s \leftarrow mst$
$n \leftarrow n + 1$
$\tau \leftarrow$ new tree with N leaves
$pk \leftarrow \mathrm{GenPk}(\tau, n)$
$sk \leftarrow \mathrm{GenSk}(\tau, n)$
return (mst, pk, sk)

Helper GenPk(τ, n)

pkKeys$[\cdot] \leftarrow \perp$
for $v \in \tau.\mathrm{RootPath}(n)$ **do**
  $r \leftarrow \mathsf{PRF}(s, v.\mathrm{idx})$
  $(v.sk, v.pk) \leftarrow \mathsf{PKE.Keygen}(1^\lambda; r)$
  pkKeys$[v.\mathrm{idx}] \leftarrow v.pk$
return (n, pkKeys)

Helper GenSk(τ, n)

for $v \in \tau.\mathrm{LeftSiblingPath}(n+1) \setminus \tau.\mathrm{RootPath}(n)$ **do**
  $r \leftarrow \mathsf{PRF}(s, v.\mathrm{idx})$
  $(v.sk, v.pk) \leftarrow \mathsf{PKE.Keygen}(1^\lambda; r)$
for $v \in \tau.\mathrm{RootPath}(n+1) \setminus \tau.\mathrm{RootPath}(n)$ **do**
  $r \leftarrow \mathsf{PRF}(s, v.\mathrm{idx})$
  $(v.sk, v.pk) \leftarrow \mathsf{PKE.Keygen}(1^\lambda; r)$
// Private state
skPriv$[\cdot] \leftarrow \perp$
for $v \in \tau.\mathrm{LeftSiblingPath}(n+1)$ **do**
  skPriv$[v.\mathrm{idx}] \leftarrow v.r$
// Public state
skPub$[\cdot] \leftarrow \perp$
for $v \in \tau.\mathrm{RootPath}(n+1)$ **do**
  skPub$[v.\mathrm{idx}] \leftarrow v.pk$
return $(n, \mathrm{skPriv}, \mathrm{skPub})$

Fig. 4: The group manager algorithms Setup and Update.

Protocol PE Encryptor

Encrypt(crs, pub, pk, m)

$(k, crs_{\mathsf{COM}}, n, \tau) \leftarrow \mathrm{ParsePk}(crs, pub, pk)$
// Commitment
$r_{\mathsf{COM}} \leftarrow\!\!\$\ \{0,1\}^\lambda$
$c_{\mathsf{COM}} \leftarrow \mathsf{COM.Commit}(crs_{\mathsf{COM}}, n; r_{\mathsf{COM}})$
// Encrypt to root path
$C[\cdot] \leftarrow \perp$
$\ell \leftarrow -1$
$s_\ell \leftarrow\!\!\$\ \{0,1\}^\lambda$
$h_{\ell+1} \leftarrow \mathsf{H}_k(c_{\mathsf{COM}})$
for $v \in \tau.\mathrm{RootPath}(n)$ **do**
  $\ell \leftarrow \ell + 1$
  $(s_\ell, r_\ell) \leftarrow \mathsf{PRG}(s_{\ell-1})$
  $C[\ell] \leftarrow \mathsf{PKE.Enc}(v.pk, (n, r_{\mathsf{COM}}, h_\ell, s_\ell, m); r_\ell)$
  $h_{\ell+1} \leftarrow \mathsf{H}_k(h_\ell, C[\ell])$
$c \leftarrow (c_{\mathsf{COM}}, C)$
return c

Helper ParsePk(crs, pub, pk)

parse $(k, crs_{\mathsf{COM}}) \leftarrow crs$
$N \leftarrow pub$
parse $(n, \mathrm{pkKeys}) \leftarrow pk$
$\tau \leftarrow$ new tree with N leaves
for $v \in \tau.\mathrm{RootPath}(n)$ **do**
  $v.pk \leftarrow \mathrm{pkKeys}[v.\mathrm{idx}]$
return $(k, crs_{\mathsf{COM}}, n, \tau)$

Fig. 5: The encryption and the public-key verification algorithms.

Protocol PE Decryptor

Decrypt(crs, pub, sk, c)

$(k, crs_{\mathsf{COM}}, h_{()}, n, \tau) \leftarrow \mathrm{ParseSk}(crs, pub, sk)$
parse $(c_{\mathsf{COM}}, C) \leftarrow c$
$m \leftarrow \bot$
// Trial decryption on left-sibling path
$(n', r_{\mathsf{COM}}, h, s, m) \leftarrow \bot$
$v^* \leftarrow \bot$
for $v \in \tau.\mathrm{LeftSiblingPath}(n+1)$ **do**
  $(n', r_{\mathsf{COM}}, h, s, m) \leftarrow \mathrm{TrialDec}(crs_{\mathsf{COM}}, c_{\mathsf{COM}}, n, C, v)$
  if $(n', r_{\mathsf{COM}}, h, s, m) \neq \bot$ **then**
    $v^* \leftarrow v$
    break
req $v^* \neq \bot$
// Check hash chain
$h' \leftarrow \mathsf{H}_k(c_{\mathsf{COM}})$
for $\ell = 0, \ldots, (v^*.\mathrm{height} - 1)$ **do**
  $h' \leftarrow \mathsf{H}_k(h', C[\ell])$
req $h = h'$
// Check consistency up the tree
while $v^*.\mathrm{parent} \neq \bot$ **do**
  $v^* \leftarrow v^*.\mathrm{parent}$
  $(s, r) \leftarrow \mathsf{PRG}(s)$
  $c' \leftarrow \mathsf{PKE.Enc}(v^*.pk, (n', r_{\mathsf{COM}}, h, s, m); r)$
  req $C[v^*.\mathrm{height}] = c'$
  $h \leftarrow \mathsf{H}_k(h, c')$
return m

Helper TrialDec$(crs_{\mathsf{COM}}, c_{\mathsf{COM}}, n, C, v)$

$\ell \leftarrow v.\mathrm{height}$
try $m' \leftarrow \mathsf{PKE.Dec}(v.sk, C[\ell])$
parse $(n', r_{\mathsf{COM}}, h, s, m) \leftarrow m'$
req $n' \leq n$
$c'_{\mathsf{COM}} \leftarrow \mathsf{COM.Commit}(crs_{\mathsf{COM}}, n'; r_{\mathsf{COM}})$
req $c'_{\mathsf{COM}} = c_{\mathsf{COM}}$
req $v \in \tau.\mathrm{RootPath}(n')$
return $(n', r_{\mathsf{COM}}, h, s, m)$

Helper ParseSk(crs, pub, sk)

parse $(k, crs_{\mathsf{COM}}) \leftarrow crs$
$N \leftarrow pub$
parse $(n, \mathrm{skPriv}, \mathrm{skPub}, \mathrm{skHashes}) \leftarrow sk$
$\tau \leftarrow$ new tree with N leaves
// Store secrets
for $v \in \tau.\mathrm{LeftSiblingPath}(n+1)$ **do**
  $v.sk \leftarrow \mathrm{skPriv}[v.\mathrm{idx}]$
// Store public keys
for $v \in \tau.\mathrm{RootPath}(n+1)$ **do**
  $v.pk \leftarrow \mathrm{skPub}[v.\mathrm{idx}]$
return $(k, crs_{\mathsf{COM}}, n, \tau)$

Fig. 6: The decryption and the secret-key validation algorithms. The ValidateMTPath helper method is the same as in Fig. 5.

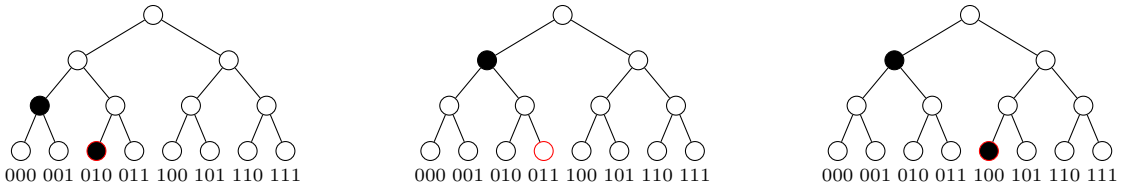

Fig. 7: An illustration of the node a party stores in epoch 2 (left), epoch 3 (middle), and epoch 4 (right). For nodes marked with black, the party's state *ust* contains the secret key sk, public key pk, as well as hash h.

but s_0 as well. This process is then repeated up the tree, by deriving $(s_1, r_1) = \mathsf{PRG}(s_0)$ and using r_1 to encrypt (s_1, m) under the next key and so forth. See Fig. 8 for a graphical overview of this process.

During the decryption, a party that knows the secret key of the node at height ℓ then gets (s_ℓ, m). It then verifies the encryptions further up the tree by re-encryption of the expected message. That is, it will derive $(s_{\ell+1}, r_{\ell+1}) = \mathsf{PRG}(s_\ell)$ and check that the next ciphertext corresponds to the encryption of $(s_{\ell+1}, m)$ under $r_{\ell+1}$ and so on.

Intuitively, the decryptor should be able to perform this trial decryption using each of the secret keys they know, and output m if it succeeds. We need, however, to ensure that at most one of the trial decryptions succeeds, even if a ciphertext has been crafted by an attacker knowing all secret keys. To this end, the ciphertext must uniquely bind to the epoch number while looking pseudorandom: The PE ciphertext additionally contains a commitment $c_{\mathsf{COM}} = \mathsf{COM.Commit}(crs_{\mathsf{COM}}, i; r_{\mathsf{COM}})$ to the epoch number i using randomness r_{COM}. All the PKE ciphertexts in addition then encrypt the opening r_{COM}.

Finally, to achieve IND-CCA security we need to ensure that each ciphertext component binds the prior ones (further down the tree) as well as the commitment. Let h_ℓ denote the hash over the commitment and ciphertexts 0 to $\ell - 1$. In summary, the party then encrypts $(i, r_{\mathsf{COM}}, h_\ell, s_\ell, m)$ at every height ℓ.

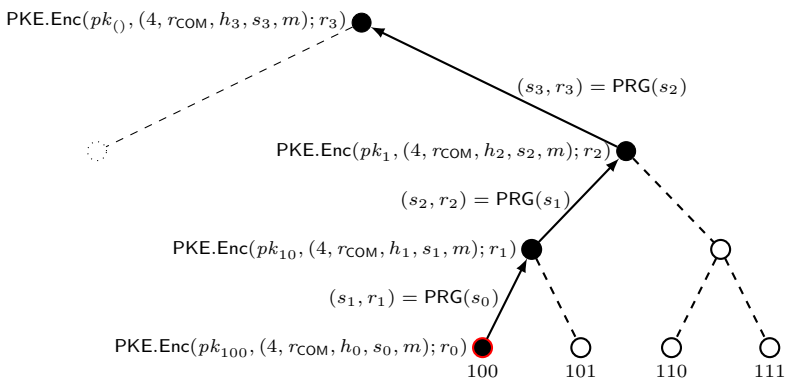

Fig. 8: An illustration of the encryption of a message m for epoch 4. The encryption randomness is generated "upwards" using a PRG, as indicated by the arrows. The term r_{COM} denotes a commitment to the epoch number 4 and h_ℓ a hash chain over the ciphertexts, and the commitment, that is computed bottom-up.

Efficiency. Let us briefly comment on the (asymptotic) efficiency of the scheme. Let N denote the maximal number of supported epochs and $d = \log_2(N)$. We then observe the following.

- *Manager:* The initial setup just consists of choosing a PRF seed and producing public- and secret-keys analogous to subsequent epochs. For each epoch i, the public- and secret-keys (pk_i, sk_i) consists of $\mathcal{O}(d)$ many underlying PKE key-pairs — each of which can be efficiently derived from the PRF seed. Hence, the manager's state is of constant size and Setup and Update both take time $\mathcal{O}(d)$.
- *Encryption:* A PE public key pk_i consists of $d + 1$ PKE public keys. Encryption performs one commitment, one PKE encryption per PKE public key, and $d + 1$ hash operations. Overall, Encrypt thus performs a logarithmic number of cryptographic operations.
- *Decryption:* The secret key sk_i is of size $\mathcal{O}(d)$, more concretely it consists of $d + 1$ PKE public keys and at most d PKE secret keys. Decryption performs at most d trial decryptions before verifying the output. The verification consists of first verifying the commitment and then verifying the remaining PKE ciphertexts by re-encryption. This takes at most as long as the corresponding encryption. Overall, the decryptor thus runs in time $\mathcal{O}(d)$ as well.

In summary, our scheme thus runs in logarithmic time (in the number of supported epochs). Hence, we can set, for instance, $d = \lambda$ and therefore support a virtually unbounded number of epochs.

4.2 Correctness and Security

We now state correctness, pseudorandomness, and perpetuity of our scheme. We refer to Section 2 for an overview of the (standard) security assumptions on the building blocks.

Theorem 1. *The PE scheme depicted in Figs. 4 to 6 is correct, if the underlying public-key encryption scheme is correct and the commitment scheme is binding.*

Proof. We start by observing that due to the deterministic nature of a PRF, the manager always re-derives the same keys as during the initial setup. Hence, in the following, we ignore the intermittent compaction of the manager's state. (It follows by inspection that whenever the manager needs some state, this is first properly re-derived.)

Next, we observe that for epoch i, the PE public key contains all the PKE public keys on the path from epoch's leaf to the root. The encryption algorithm, when input the public parameters, pk_i, and m, therefore has all the necessary PKE public keys available and will output a ciphertext $c = (c_{\mathsf{COM}}, C)$ where C is of the length of the root path.

Let us now consider decryption under the secret key for epoch i. Recall that the PE secret key contains the following elements:

- for all $v \in \text{LeftSiblingPath}(i+1)$, it contains the PKE secret key $v.sk$;
- for all $v \in \text{RootPath}(i+1)$, it contains the PKE public key $v.pk$;

We now argue that the trial decryption for epoch i works. The following observation reflects a basic property on left-sibling paths, valid in any binary tree.

Observation 1. For any i, the encryption path $\text{RootPath}(i)$ and the set of nodes $\text{LeftSiblingPath}(i+1)$, for which sk_i includes secrets, intersect in exactly one node, i.e.,

$$|\text{RootPath}(i) \cap \text{LeftSiblingPath}(i+1)| = 1.$$

Let v^* denote this unique node.

Moreover, since any node on $\text{LeftSiblingPath}(i+1)$ is a left child of a node on $\text{RootPath}(i+1)$, the following holds as well.

Observation 2. Let v^* denote the unique node defined above. Any ancestor node v of v^* (i.e., any node on the path starting from the parent of v^* to the root) is contained in $\text{RootPath}(i+1)$.

Using those two observations we now show that sk_i is sufficient to successfully decrypt the ciphertext $c = \text{Encrypt}(crs, pub, Pk[i], m)$.

Claim. Let $\ell = v^*.\text{height}$ denote the height of the vertex v^* in the tree and let s_ℓ denote the ℓ-th seed the encryptor used while generating the ciphertext. Then, the trial decryption under node v^* succeeds, i.e., $\text{TrialDec}(c_{\mathsf{COM}}, C, v^*)$ returns $(i, r_{\mathsf{COM}}, s_\ell, m)$.

Proof: Observe that $C[\ell] = \mathsf{PKE.Enc}(v^*.pk, (i, r_{\mathsf{COM}}, s_\ell, m); r_\ell)$. By correctness of the PKE scheme, the decryptor will thus get $(i, r_{\mathsf{COM}}, s_\ell, m)$. Moreover, by definition $c_{\mathsf{COM}} = \mathsf{COM.Commit}(crs_{\mathsf{COM}}, i; r_{\mathsf{COM}})$ and hence the decryptor will compute the same commitment value. Therefore, $\text{TrialDec}(c_{\mathsf{COM}}, C, v^*)$ will return $(i, r_{\mathsf{COM}}, s_\ell, m)$.

Claim. For any $v \in \text{LeftSiblingPath}(i+1)$ such that $v \neq v^*$, the trial decryption will fail.

Proof: Assume the initial decryption still succeeds and produces an output that can be parsed as $(i', r'_{\mathsf{COM}}, s', m')$. If the check on the commitment succeeds, i.e., $c_{\mathsf{COM}} = \mathsf{COM.Commit}(crs_{\mathsf{COM}}, i'; r'_{\mathsf{COM}})$ then by the binding property we can deduce that $i' = i$. Since v^* is however the only node on $\text{LeftSiblingPath}(i+1)$ for which $v \in \text{RootPath}(i)$, we can deduce that the respective check must fail. Using the second observation, we can conclude that the decryptor will successfully re-compute $C[t]$ for $t > \ell$ and, hence, output the correct message m. □

Pseudo-randomness requires both the commitment scheme and the PKE scheme to have pseudorandom-looking encryptions. We detail those assumptions in Section 2.

Theorem 2. *The PE scheme from Figs. 4 to 6 satisfies pseudorandomness (cf. Fig. 3), if the underlying public-key encryption scheme is correct and IND-CCA secure with pseudorandom ciphertexts, the commitment scheme has pseudorandomness, the hash function is collision-resistant, and the PRF and PRG satisfy their usual security notions.*

Proof. In the following, assume the challenge epoch i^* to be fixed. (A full reduction could simply guess it, given that there are only polynomially many epochs.) Let $d = \log_2(N)$ denote the height of the scheme's binary tree. We now consider a sequence of hybrids, $H_0, H_1, \ldots, H_{d+1}$, such that H_0 is indistinguishable from the pseudorandomness experiment with $b = 0$ while H_{d+1} is indistinguishable to the experiment with $b = 1$. Let H_v be defined as follows:

- The system chooses a binary tree of PKE key pairs just like the scheme, except that they are sampled completely independently rather than deriving the randomness using a PRF. Public and secret keys are all computed exactly like in the scheme.
- When encrypting the challenge message m^*, the hybrid proceeds as follows to compute the ciphertext c^*:
  - The commitment c_{COM} is computed analogously to the protocol.
  - For the PKE ciphertext $C[0,\ldots,d]$ the hybrid
    - derives the randomness (s_ℓ, r_ℓ) identical to the scheme
    - for $\ell < v$, it sets $C[\ell]$ as a uniform random value;
    - for $\ell \geq v$, it computes the ciphertext identical to the scheme (but using the random elements for the hash chain computation).
  - It then outputs $c = (c_{\mathsf{COM}}, C)$ as the scheme.
- For the decryption oracle, it returns $\perp$ if $c = c^*$ (or the decrypting party is not in a state at least i^*) and runs the scheme otherwise.

Let us first show that H_0 corresponds to the real-world case of the game (i.e., when $b = 0$) while H_{d+1} corresponds to the ideal-world variant (i.e., when $b = 1$).

Claim. The hybrid H_0 is computationally indistinguishable from the pseudorandom experiment with $b = 0$.

Proof: By inspection, we see that the only difference is the sampling of the public keys using independent uniform randomness rather than deriving the randomness using a PRF. Since the group manager's state, and especially the seed s is never revealed, this is indistinguishable.

Claim. The hybrid H_{d+1} is computationally indistinguishable from the pseudorandom experiment with $b = 1$.

Proof: We observe that for $v = d + 1$ all the PKE components of the challenge ciphertext have been replaced with uniformly random PKE ciphertexts. Thus, in particular C contains no information about the commitment's opening. Therefore, by our security assumption on the commitment scheme, it holds that c_{COM} is indistinguishable from a random string, rendering the overall ciphertext $c = (c_{\mathsf{COM}}, C)$ indistinguishable from random.

It remains to show that any two consecutive hybrids are computationally indistinguishable. We build a reduction to the PKE scheme's IND-CCA security game by considering a "wrapper" that emulates either H_v or H_{v+1} based on the embedded game's bit b.

- The reduction places the instance at height v (from the bottom) on the root path corresponding to epoch i^*, i.e., on the path under which the challenge message is encrypted, and uses the instance's public key as PKE public key for that node, and submits the respective message H_v would encrypt under that node when computing c^* by submitting it as the instance's challenge.
- Observe that the sole difference between H_v and H_{v+1} is the computation of that component of the challenge ciphertext c^*.
- The reduction emulates the rest of either H_v or H_{v+1} as usual. In particular, observe that the adversary must not corrupt any epoch that would reveal the challenge instance's secret key for $e_{\mathsf{corr}} < i^*$ to hold. Hence, the reduction can simply emulate all (relevant) corruptions.
- When answering decryption queries, for $c \neq c^*$, the reduction works as follows: It emulates H_v, which in turn just runs the protocol, except for whenever the protocol would decrypt $C[v]$ as part of TrialDec under the embedded instance's secret key. In that case,
  - If $C[v] \neq C^*[v]$, then the reduction simply uses the underlying game's decryption oracle.
  - Otherwise, the reduction considers that trial decryption to be unsuccessful.

Claim. The reduction behaves either like H_v or H_{v+1} depending on the underlying instance's challenge bit.

Proof: First, it is easy to verify that except for the decryption oracle, the statement is indeed true. In particular, if the bit is 1, then the computation of the challenge ciphertext exactly matches the one of H_{v+1}. If the bit is 0, then the only difference is that the challenge instance uses independent randomness while H_v uses r_v derived using the PRG. Since however none of the seeds $s, s_0, s_1, \ldots, s_{v-1}$ is revealed in H_v, by PRG security this is indistinguishable.

Now consider the decryption oracle. Clearly, if $C[v] \neq C^*[v]$ then the reduction directly matches the decryption oracle of H_v and H_{v+1}. Since $c \neq c^*$ we can distinguish two cases:

Case 1: Either $c_{\mathsf{COM}} \neq c^*_{\mathsf{COM}}$ or $C[\ell] \neq C^*[\ell]$ for some $\ell < v$. In this case observe that in H_v the component $C[v]$ is an encryption that contains a hash over $(c_{\mathsf{COM}}, C[0], \ldots, C[\ell-1])$. By correctness of the PKE scheme, this value will be recovered in H_v. Therefore, by collision resistance that hash would therefore no longer match and the protocol rejects. Similarly, in H_{v+1} when decrypting a uniformly random chosen $C[v]$ the probability of this decrypting to something containing the correct commitment opening r_{COM}, which has been chosen independently of the decryption key, is negligible.

Case 2: There exists $\ell > v$ such that $C[\ell] \neq C^*[\ell]$. Here, observe that $C^*[v]$ encrypts over the seed s_v from which all further components $C[\ell]$ are deterministically derived. Hence in H_v those checks will fail, whereas in H_{v+1} the same argument as in the prior case applies.

This concludes the overall proof. □

Perpetuity relies on the binding property of the commitment scheme, ensuring each ciphertext has a unique epoch associated, as well as slightly non-standard properties of the PKE scheme ensuring that decryption is deterministic and that the scheme is correct even when using adversarially chosen randomness. (Note that this is implied by perfect or information-theoretic correctness. As such, plenty of PKE schemes exist that satisfy the property.)

Theorem 3. *The PE scheme depicted in Figs. 4 to 6 is perpetual, if the commitment scheme is binding, the PKE scheme has a deterministic decryption algorithm* $\mathsf{PKE.Dec}$*, and is correct under maliciously chosen encryption randomness. More concretely, for any PPT adversary,* $\mathcal{A}$ *the following must be negligible*

$$\Pr\left[m \neq m' \;\middle|\; \begin{array}{l} (sk, pk) \leftarrow \mathsf{PKE.Keygen}(1^\lambda) \\ (m, r) \leftarrow \mathcal{A}(1^\lambda, sk, pk) \\ c \leftarrow \mathsf{PKE.Enc}(pk, m; r) \\ m' \leftarrow \mathsf{PKE.Dec}(sk, c) \end{array} \right]$$

when the probability is taken of the randomness of $\mathsf{PKE.Keygen}$ *and* $\mathcal{A}$*.*

Proof. Consider decryption under honestly-generated secret keys for epochs i and j, when given the same CRS crs, which contains the hash key k and the commitment CRS crs_{COM}. In the case of $i = j$, the property holds due to Decrypt being deterministic in our scheme. In the following assume that $j > i$.

Both decryptions only accept the ciphertext c if it can be parsed as $c = (c_{\mathsf{COM}}, C)$ where c_{COM} denotes a commitment string while C an array of PKE ciphertexts of the correct length. c_{COM} uniquely defines the epoch e to which the parties might decrypt (TrialDec checks that if decryption succeeds then it must be for a node on the path from e's leaf to the root). On the other hand, if $j < e$ then both parties reject with $\perp$, and if $i < e \leq j$ then the earlier party rejects, which are both valid outcomes. Hence, the interesting case is when $e \leq i$.

We now make the following observation:

Observation 3. Let $e \leq i \leq j$ and let v_i denote the unique node in the intersection of RootPath(e) and LeftSiblingPath$(i+1)$ and v_j the unique node in RootPath$(e) \cap$ LeftSiblingPath$(j+1)$, respectively. (Cf. observation 1 from the correctness proof.) Then we have that either $v_i = v_j$, or that v_j is closer to the root.

Claim. If $v_i \neq v_j$, then either the decryption using sk_i rejects, or both output the same message.

Proof: Assume the decryption under sk_i outputs a message m, after having decrypted the PKE ciphertext for node v_i as (r_{COM}, s, m). In this process, they re-encrypted the message under $v_j.pk$ and checked that the ciphertexts matched. Given our correctness assumption for the PKE scheme, however, this implies that the decryption under sk_j will decrypt that particular PKE ciphertext to (r_{COM}, s, m), as they were given randomness r such that $\mathsf{PKE.Keygen}(1^\lambda; r)$ produces the correct public key $v_j.pk$ and the ciphertext has been obtained as part of the honest encryption (with potentially maliciously chosen randomness). Further up the tree, the two parties then perform identical checks.

Claim. If $v_i = v_j$, then both produce the same message.

Proof: This follows trivially, in the case of an honest-but-curious group manager, as both PE secret keys will contain the same PKE secret key $v_i.sk$, and the same PKE public keys for nodes on the path from the leaf to the root. □

4.3 Strong Perpetuity and Interval Access

The scheme, as introduced so far, allows for false negatives in which a ciphertext is rejected when attempting to decrypt under a secret key sk_i, but accepted under a later secret key sk_j, for $j > i$. Let us sketch a modification of the scheme that satisfies strong perpetuity. Recall that, in the base scheme, the group manager computes a PKE key-pair for every node v as $(v.pk, v.sk) = \mathsf{PKE.Keygen}(1^\lambda; v.r)$, where the randomness $v.r = \mathsf{PRF}(s, v.idx)$ is derived using a PRF from the node index $v.idx$. We make three modifications:

1. Instead of including PKE secret keys $v.sk$ in the PE secret key, the manager includes the randomness v. (Looking ahead, we will use the same modification for the scheme in Section 5.2.) This in itself does not affect security, as the randomness for different nodes is independent by PRF security.
2. Next, we replace the generic PRF with the Goldreich-Goldwasser-Micali (GGM) construction. Note that we *do not* apply a prefix-free coding to the node indices. As such, $v.r$ and $v.\text{parent}.r$ are no longer independent! Instead, one can derive $v.r$ from $v.\text{parent}.r$. Note however, that the randomness for any two nodes where neither one is a descendant of the other still appears independent; similarly, conditioned on $v.r$, $v.\text{parent}.r$ is still indistinguishable from uniformly at random.
3. Finally, the decryption under sk_i is modified as follows: Once the correct epoch number j has been determined via trial decryption under one of the nodes $v \in V_i$, the algorithm does not just check the path from v to the root. Instead, the algorithm derives the secret key for the j-th leaf, which is a descendant of v, and the public keys along the path from that leaf to v. (This can now be done due to using the GGM construction.) Afterward, the decryption checks the entire path from the j-th path to the root, analogous to how decryption under sk_j would do. It only outputs m if doing so yields the same message as the initial trial decryption under $v.sk$.

Those modifications only incur a constant overhead when measured in worst-time complexity; instead of decrypting a path suffix, the modified scheme now always decrypts the entire path. Furthermore, the scheme needs $\mathcal{O}(d)$ additional PRG expansions (for the GGM construction) and key derivations. It follows by inspection that the modified scheme still satisfies correctness, and now additionally satisfies strong perpetuity. (Put simply, $\text{Decrypt}(crs, pub, sk_i, c)$ now only outputs a message m for epoch j after doing all the same checks as $\text{Decrypt}(crs, pub, sk_j, c)$ does.)

Theorem 4. *The modified PE scheme satisfies pseudorandomness, if the underlying public-key encryption scheme is correct and IND-CCA secure with pseudorandom ciphertexts, the commitment scheme has pseudorandomness, the hash function is collision-resistant, and the PRG satisfies the usual security notion.*

Proof (Sketch). Consider the encryption of the challenge message m^* under public key pk_{i^*} of epoch i^*. Assume the adversary has secret keys $sk_1, \ldots, sk_{i^*-1}$. Each secret key sk_j contains the GGM values for nodes $v \in V_j$. By properties of the GGM construction, intuitively, we can think of the secret key sk_j now not just containing secret keys for $v \in V_j$ but also all the descendants of those nodes. However, we note that none of those additional nodes is on the path from the i-th leaf to the root. Indeed, this follows

by construction: any PE ciphertext that those additional PKE secret-keys can decrypt could already be decrypted using the one for the node in V_j. IND-CCA security directly carries over from the proof of the basic scheme. □

Finally, we observe that this modification naturally allows users to derive a secret key sk_{i_0,i_1} limited to a given interval $[i_0, i_1]$ — as discussed in Section 3. Basically, we can observe that for every such interval, there exists a minimal cover set of nodes that only allows us to reach the respective leaves. The secret keys of these nodes can then be derived to yield sk_{i_0,i_1}. Moreover, the cover set still shares exactly one node with the path from leaf $j \in [i_0, i_1]$. Thus, decryption remains essentially unchanged.

5 Insider-Secure Perpetual Encryption

In this section, we extend Perpetual Encryption to the setting of malicious group managers choosing the keys. We call this setting *insider-security* and sometimes refer to a regular PE scheme as *outsider-secure* to avoid confusion.

5.1 Syntax and Security

To achieve any kind of security against an adversarial group manager, parties must have some notion of "correct" or at least consistent keys. As a consequence, we extend our syntax to allow for validations of both public- and secret keys, against the public parameters *pub* and the intended epoch number i.[6]

Definition 8 (Insider-Secure Public Key PE). *An* insider-secure perpetual encryption (PE) *scheme is a PE scheme that additionally has the following two algorithms:*

- ▷ $\{0,1\} \leftarrow \text{ValidatePk}(crs, pub, i, pk_i)$
  Validates the public key of the i-th epoch against the public setup and the public parameters.
- ▷ $\{0,1\} \leftarrow \text{ValidateSk}(crs, pub, i, sk_i)$
  Validates the secret key of the i-th epoch against the public setup and the public parameters.

We extend our correctness definition analogously to ensure ValidatePk and ValidateSk to work as intended for valid keys. This rules out trivializing insider-perpetuity, which will be conditioned on parties having keys that validate.

Definition 9. *An insider-secure PE scheme is correct if it satisfies correctness of a regular PE scheme and, additionally, for any key-pair (sk_i, pk_i) for epoch i (output by* Setup *for epoch* 0 *and* Update *for later epochs) both keys validate, i.e.,* $\text{ValidatePk}(crs, pub, i, pk_i)$ *and* $\text{ValidatePk}(crs, pub, i, pk_i)$ *both return* 1*.*

While the above definition still considers an honest group manager, we also ask for the correctness of encryption and decryption under adversarially chosen keys as long as they validate. In other words, it must be infeasible for an adversary to come up with public parameters *pub* and a key pair (sk, pk), presumably for some epoch i, such that the keys validate but not constitute a correct PKE scheme. We call the respective property *robustness*.

Definition 10. *An insider-secure PE scheme is* robust *if any PPT adversary $\mathcal{A}$ has at most negligible probability of having the game depicted on the left in Fig. 9 returning* 1*.*

Next, we consider perpetuity against an adversarial group manager choosing the keys. We call this property *insider perpetuity* or just perpetuity if clear from the context that we consider insider security. Again, we additionally consider the variant of *strong insider perpetuity*, ruling out false negatives. The respective security game is depicted on the right side of Fig. 9. When contrasted to the (outsider-)perpetuity game from Section 3, the group manager is replaced by the adversary, with only the public setup *crs* being honestly generated. In particular, the adversary freely chooses the secret keys. We say that a secret key

[6] A slightly weaker model in which parties are assumed to have a consistent view on the public keys, e.g. by the leader posting them on a blockchain, is conceivable. We opt here for the strongest definition where parties only need to agree on *crs* and *pub*.

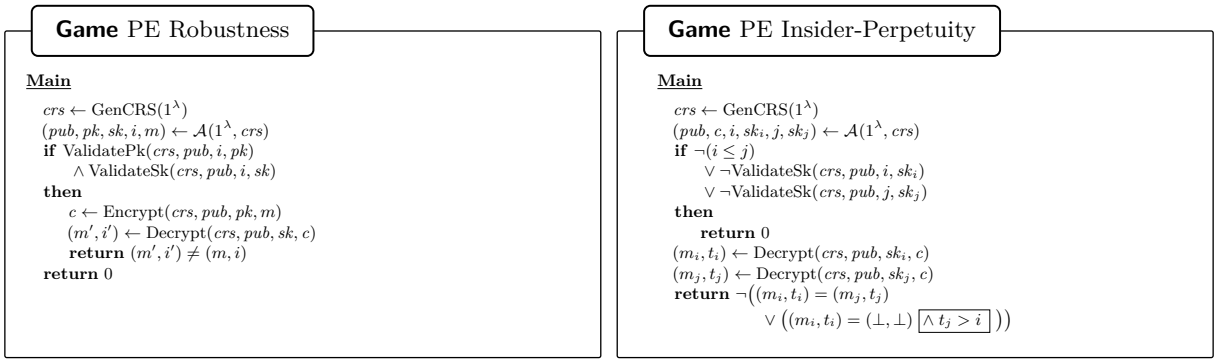

Fig. 9: Insider-security of a PE scheme. The robustness game (left) ensures validating key pairs function like a PKE scheme, while the perpetuity game (right) ensures future keys decrypt consistently. The variant of strong insider perpetuity adds the boxed condition.

sk_i is for epoch i if $\text{ValidateSk}(crs, pub, i, sk_i)$ returns true. In other words, perpetuity must hold if parties are given keys that validate with respect to the public parameters pub the (malicious) group manager initially committed to.

Definition 11. *An insider-secure PE scheme satisfies (insider) perpetuity, if any PPT adversary $\mathcal{A}$ has at most negligible probability of having either of the games depicted on the right in Fig. 9 returning* 1*. It is said to satisfy* strong insider perpetuity *if this holds with respect to the game omitting the $m_i \neq \bot$ check.*

Finally, we note that pseudorandomness against an adversarial group manager choosing all keys is futile. We, however, still require an insider-secure PE scheme to adhere to the pseudorandomness security of an outsider-secure PE scheme. In other words, the scheme must provide pseudorandom ciphertext assuming the manager to be honest.

5.2 A Bounded Epoch Scheme

In this section, we present an insider-secure PE scheme for a bounded number of epochs. The scheme is closely based on the outsider-secure one from the previous section — however, Setup will run in time $\mathcal{O}(N)$, meaning that the scheme only works for a bounded number of epochs.

Shortcomings of the outsider-secure scheme. First, let us gain some intuition on the shortcomings of the scheme from Section 4 with respect to an actively malicious group manager. Most obviously, the group manager could simply produce two completely unrelated secret keys sk_i and sk'_i, for presumably the same epoch. Similarly, they could produce unrelated key material across epochs, such as sk_i and sk_j for epochs $i < j$. A ciphertext that decrypts to some message m with respect to sk_i would then, with overwhelming probability, not decrypt to the same message under the other key.

A bit more subtly, malicious group members could tamper with individual PKE key pairs of the tree structure. For instance, for a node v, they could use unrelated public key pk_v and secret key sk'_v. A party that checks the ciphertext component for v by re-encrypting under pk_v and one that decrypts it under sk'_v would then yield different results. Finally, an adversary might try to reuse key material, or use related keys, for two different nodes v and v'. The last one will turn out not to be a security issue, of which we will have to convince ourselves.

Protocol description. The bounded-epoch insider-secure scheme extends the scheme from the previous section with the Merkle-Tree. To this end, recall the Merkle-Tree iterator from Section 2.3. The group manager now maintains non-trivial state. Among other, it maintains a binary tree as part of its state. (An

efficient representation of such a binary tree is left to a concrete implementation.) Nodes $v \in \tau$ can have the following properties assigned:

$v.r$	The randomness seed from which the public and secret key are derived.
$v.pk$	The node's public key.
$v.sk$	The node's secret key.
$v.h$	The node's hash for the Merkle-Tree.

We now enhance our outsider-secure protocol to thwart those attacks. At the high level, we will force the group manager to commit to the tree of PKE key pairs as part of the public parameters *pub*. The common reference string *crs* remains unchanged, and so do the encryption Encrypt and decryption Decrypt algorithms, respectively, except for the public keys and secret keys being slightly differently formatted. A pseudocode description of the scheme is presented in Figs. 10 to 12.

Committing to the public keys. Recall that in the outsider-secure scheme, the Setup algorithm created a binary tree with N leaves, one per epoch, where each node *idx* has an associated PKE key-pair (pk_{idx}, sk_{idx}). Those key-pairs are sampled using a PRF so that the manager's state can be compact.

For the insider-secure scheme we enhance this by computing a Merkle-Tree over all the PKE public keys and outputs the root hash $h_{()}$ as its public parameter *pub*. See Fig. 13 for a depiction of the (expanded) state the manager builds. Whenever the group manager hands out a PKE public key, whether as part of a PE public or secret key, they will include the respective Merkle-opening — this will then be verified by ValidatePk or ValidateSk, respectively.

Committing to the secret keys. The same strategy cannot be just applied to the secret keys. While one could, in principle, have the manager commit to the tree of PKE secret keys, this would not guarantee consistency between the public and secret keys of a node *idx*. To see why consistency is crucial, consider secret keys sk_i and sk_j, for $i < j$, and let *idx* be some node for which sk_i contains the PKE public key pk_{idx} whilst sk_j contains the PKE secret key sk_{idx}. Decryption using sk_i checks a ciphertext component for node *idx* by re-encrypting under pk_{idx}, while decrypting using sk_j will actually decrypt the component using sk_{idx}. If the former but not the latter succeeds, then this violates perpetuity. Therefore, ValidateSk(crs, pub, j, sk_j) must reject sk_j in case the key-pair is inconsistent.

To efficiently address this issue, we do not commit to the secret keys. Instead, the protocol only commits to the public keys, as described above, and requires stronger properties of the underlying PKE scheme. It validates PKE secret keys as follows:

- Instead of handing out sk_{idx}, the secret key sk_j will contain the randomness r_{idx} used to generate the key pair $(pk_{idx}, sk_{idx}) = \mathsf{PKE.Keygen}(1^\lambda; r_{idx})$;
- ValidateSk then derives (pk_{idx}, sk_{idx}) and checks pk_{idx} against the Merkle-Tree commitment.

Public- and secret-keys. Let us summarize the public key pk_i and secret key sk_i, respectively, for epoch i. The PE public key pk_i consists of the following components:

- The epoch number i.
- For each node v on the path from the epochs leaf idx_i to the root, it contains the corresponding public key $v.pk$.
- For each node on the sibling path, it contains the nodes' hashes, to verify the Merkle-Tree opening.

The corresponding secret key sk_i consists of the following components:

- The epoch number i.
- The *randomness* r_{idx} used to generate sk_{idx}, for each $idx \in V_i$. Recall that V_i denotes the smallest set of indices such that exactly the leaves from $idx_0 = \mathsf{encode}(0, d)$ to $idx_i = \mathsf{encode}(i, d)$ are reachable, and corresponds to the left-sibling path of the $(i+1)$-th leaf.
- The PKE public keys on the root path of the $(i+1)$-th leaf.

Protocol PE Manager

Setup$(1^\lambda, crs, N)$

parse $(k, crs_{\mathsf{COM}}) \leftarrow crs$
$n \leftarrow 0$
// Sample PRF seed
$s \leftarrow\$ \{0,1\}^\lambda$
// Initialize Merkle-Tree Iterators
$\tau_1 \leftarrow$ new tree with N leaves
$(\mathrm{mtSt}_1, \tau_1)$
  $\leftarrow \mathrm{MtIterator.Init}(\tau_1, \mathrm{ComputeNodeHash})$
$(\mathrm{mtSt}_2, \tau_1)$
  $\leftarrow \mathrm{MtIterator.Init}(\tau_1, \mathrm{ComputeNodeHash})$
$(\mathrm{mtSt}_2, \tau_1) \leftarrow \mathrm{MtIterator.Next}(\mathrm{mtSt}_2, \tau_1)$
// Initial public state
$pub \leftarrow (N, \tau_1.\mathrm{root}.h)$
$pk \leftarrow \mathrm{GenPk}()$
// Initial secret state
$\tau_2 \leftarrow$ new tree with N leaves
$v \leftarrow \tau_2.\mathrm{Leaf}(0)$
$v.r \leftarrow \mathsf{PRF}(s, v.\mathrm{idx})$
$sk \leftarrow \mathrm{GenSk}()$
// Only keep compact state
$\tau_1 \leftarrow \bot$
return (pub, pk, wel)

Update

$n \leftarrow n + 1$
// Update public state
$\tau_1 \leftarrow$ new tree with N leaves
$(\mathrm{mtSt}_1, \tau_1) \leftarrow \mathrm{MtIterator.Next}(\mathrm{mtSt}_1, \tau_1)$ // n
$(\mathrm{mtSt}_2, \tau_1) \leftarrow \mathrm{MtIterator.Next}(\mathrm{mtSt}_2, \tau_1)$ // $n+1$
$pk \leftarrow \mathrm{GenPk}()$
// Update secret state
for $v \in \tau_2.\mathrm{LeftSiblingPath}(n+1)$
    $\setminus \tau_2.\mathrm{LeftSiblingPath}(n)$ **do**
  $v.r \leftarrow \mathsf{PRF}(s, v.\mathrm{idx})$
  if $\neg\mathrm{IsLeaf}(v)$ **then**
    $v.\mathrm{left}.h \leftarrow \tau_1[v.\mathrm{left.idx}].h$
    $v.\mathrm{right}.h \leftarrow \tau_1[v.\mathrm{right.idx}].h$
for $v \in \tau_2.\mathrm{LeftSiblingPath}(n)$
    $\setminus \tau_2.\mathrm{LeftSiblingPath}(n+1)$ **do**
  $v.r \leftarrow \bot$
  if $\neg\mathrm{IsLeaf}(v)$ **then**
    $(v.\mathrm{left}.h, v.\mathrm{right}.h) \leftarrow (\bot, \bot)$
$sk \leftarrow \mathrm{GenSk}()$
$\tau_1 \leftarrow \bot$
return (pk, sk)

Helper ComputeNodeHash(v)

$r \leftarrow \mathsf{PRF}(s, v.\mathrm{idx})$
$(\cdot, v.pk) \leftarrow \mathsf{PKE.Keygen}(1^\lambda; r)$
if IsLeaf(v) **then**
  $v.h \leftarrow \mathsf{H}_k(v.pk, \bot, \bot)$
else
  $v.h \leftarrow \mathsf{H}_k(v.pk, v.\mathrm{left}.h, v.\mathrm{right}.h)$

Helper GenPk

$\mathrm{pkKeys}[\cdot] \leftarrow \bot$
$\mathrm{pkHashes}[\cdot] \leftarrow \bot$
for $v \in \tau_1.\mathrm{RootPath}(n)$ **do**
  $\mathrm{pkKeys}[v.\mathrm{idx}] \leftarrow v.pk$
for $v \in \tau_1.\mathrm{SiblingPath}(n)$ **do**
  $\mathrm{pkHashes}[v.\mathrm{idx}] \leftarrow v.h$
return $(n, \mathrm{pkKeys}, \mathrm{pkHashes})$

Helper GenSk

// Private state
$\mathrm{skPriv}[\cdot] \leftarrow \bot$
$\mathrm{skHashes}[\cdot] \leftarrow \bot$
for $v \in \tau_2.\mathrm{LeftSiblingPath}(n+1)$ **do**
  $\mathrm{skPriv}[v.\mathrm{idx}] \leftarrow v.r$
  if $\neg\mathrm{IsLeaf}(v)$ **then**
    $\mathrm{skHashes}[v.\mathrm{left.idx}] \leftarrow v.\mathrm{left}.h$
    $\mathrm{skHashes}[v.\mathrm{right.idx}] \leftarrow v.\mathrm{right}.h$
// Public state
$\mathrm{skPub}[\cdot] \leftarrow \bot$
for $v \in \tau_1.\mathrm{RootPath}(n+1)$ **do**
  $\mathrm{skPub}[v.\mathrm{idx}] \leftarrow v.pk$
  if $\neg\mathrm{IsLeaf}(v) \wedge \mathrm{IsLeftChild}(v)$ **then**
    $v' \leftarrow \mathrm{Sibling}(v)$
    $\mathrm{skHashes}[v'.\mathrm{idx}] \leftarrow v'.h$
return $(n, \mathrm{skPriv}, \mathrm{skPub}, \mathrm{skHashes})$

Fig. 10: The group manager algorithms. The algorithms implicitly maintain a shared state that is initialized by Setup.

Protocol PE Encryptor

Encrypt(crs, pub, pk, m)

$(k, crs_{\mathsf{COM}}, h_{()}, n, \tau) \leftarrow \text{ParsePk}(crs, pub, pk)$
// Commitment
$r_{\mathsf{COM}} \leftarrow\$ \{0,1\}^{\lambda}$
$c_{\mathsf{COM}} \leftarrow \mathsf{COM.Commit}(crs_{\mathsf{COM}}, n; r_{\mathsf{COM}})$
// Encrypt to root path
$C[\cdot] \leftarrow \bot$
$\ell \leftarrow -1$
$s_\ell \leftarrow\$ \{0,1\}^{\lambda}$
$h_{\ell+1} \leftarrow \mathsf{H}_k(c_{\mathsf{COM}})$
for $v \in \tau.\text{RootPath}(n)$ **do**
  $\ell \leftarrow \ell + 1$
  $(s_\ell, r_\ell) \leftarrow \mathsf{PRG}(s_{\ell-1})$
  $C[\ell] \leftarrow \mathsf{PKE.Enc}(v.pk, (n, r_{\mathsf{COM}}, h_\ell, s_\ell, m); r_\ell)$
  $h_{\ell+1} \leftarrow \mathsf{H}_k(h_\ell, C[\ell])$
$c \leftarrow (c_{\mathsf{COM}}, C)$
return c

ValidatePk(crs, pub, n, pk)

$(k, crs_{\mathsf{COM}}, h_{()}, n', \tau) \leftarrow \text{ParsePk}(crs, pub, pk)$
if $n' \neq n$ **then return** 0
return $\text{ValidateMTPath}(k, h_{()}, \tau, n)$

Helper ParsePk(crs, pub, pk)

parse $(k, crs_{\mathsf{COM}}) \leftarrow crs$
parse $(N, h_{()}) \leftarrow pub$
parse $(n, \text{pkKeys}, \text{pkHashes}) \leftarrow pk$
$\tau \leftarrow$ new tree with N leaves
for $v \in \tau.\text{RootPath}(n)$ **do**
  $v.pk \leftarrow \text{pkKeys}[v.\text{idx}]$
  if $v \neq \tau'.\text{root}$ **then**
    $v' \leftarrow \text{Sibling}(v)$
    $v'.h \leftarrow \text{pkHashes}[v'.\text{idx}]$
return $(k, crs_{\mathsf{COM}}, h_{()}, n, \tau)$

Helper ValidateMTPath($k, h_{()}, \tau, n$)

for $v \in \tau.\text{RootPath}(n)$ **do**
  if $\text{IsLeaf}(v)$ **then**
    $v.h \leftarrow \mathsf{H}_k(v.pk, \bot, \bot)$
  else
    $h_\ell \leftarrow v.\text{left}.h$
    $h_r \leftarrow v.\text{right}.h$
    $v.h \leftarrow \mathsf{H}_k(v.pk, h_\ell, h_r)$
return $h_{()} = \tau.\text{root}.h$

Fig. 11: The encryption and the public-key verification algorithms.

Protocol PE Decryptor

Decrypt(crs, pub, sk, c)

$(k, crs_{\mathsf{COM}}, h_{()}, n, \tau) \leftarrow \text{ParseSk}(crs, pub, sk)$
parse $(c_{\mathsf{COM}}, C) \leftarrow c$
$m \leftarrow \bot$
// Trial decryption on left-sibling path
$(n', r_{\mathsf{COM}}, h, s, m) \leftarrow \bot$
$v^* \leftarrow \bot$
for $v \in \tau.\text{LeftSiblingPath}(n+1)$ **do**
  $(n', r_{\mathsf{COM}}, h, s, m) \leftarrow \text{TrialDec}(crs_{\mathsf{COM}}, c_{\mathsf{COM}}, n, C, v)$
  if $(n', r_{\mathsf{COM}}, h, s, m) \neq \bot$ **then**
    $v^* \leftarrow v$
    break
req $v^* \neq \bot$
// Check hash chain
$h' \leftarrow \mathsf{H}_k(c_{\mathsf{COM}})$
for $\ell = 0, \ldots, (v^*.\text{height} - 1)$ **do**
  $h' \leftarrow \mathsf{H}_k(h', C[\ell])$
req $h = h'$
// Check consistency up the tree
while $v^*.\text{parent} \neq \bot$ **do**
  $v^* \leftarrow v^*.\text{parent}$
  $(s, r) \leftarrow \mathsf{PRG}(s)$
  $c' \leftarrow \mathsf{PKE.Enc}(v^*.pk, (n', r_{\mathsf{COM}}, h, s, m); r)$
  req $C[v^*.\text{height}] = c'$
  $h \leftarrow \mathsf{H}_k(h, c')$
return m

ValidateSk(crs, pub, n, sk)

$(k, crs_{\mathsf{COM}}, h_{()}, n', \tau) \leftarrow \text{ParseSk}(crs, pub, sk)$
if $n' \neq n$ **then**
  return 0
return $\text{ValidateMTPath}(k, h_{()}, \tau, n+1)$

Helper TrialDec($crs_{\mathsf{COM}}, c_{\mathsf{COM}}, n, C, v$)

$\ell \leftarrow v.\text{height}$
try $m' \leftarrow \mathsf{PKE.Dec}(v.sk, C[\ell])$
parse $(n', r_{\mathsf{COM}}, h, s, m) \leftarrow m'$
req $n' \leq n$
$c'_{\mathsf{COM}} \leftarrow \mathsf{COM.Commit}(crs_{\mathsf{COM}}, n'; r_{\mathsf{COM}})$
req $c'_{\mathsf{COM}} = c_{\mathsf{COM}}$
req $v \in \tau.\text{RootPath}(n')$
return $(n', r_{\mathsf{COM}}, h, s, m)$

Helper ParseSk(crs, pub, sk)

parse $(k, crs_{\mathsf{COM}}) \leftarrow crs$
$(N, h_{()}) \leftarrow pub$
parse $(n, \text{skPriv}, \text{skPub}, \text{skHashes}) \leftarrow sk$
$\tau \leftarrow$ new tree with N leaves
// Store secrets
for $v \in \tau.\text{LeftSiblingPath}(n+1)$ **do**
  $r \leftarrow \text{skPriv}[v.\text{idx}]$
  $(v.sk, v.pk) \leftarrow \mathsf{PKE.Keygen}(1^{\lambda}; r)$
  if $\text{IsLeaf}(v)$ **then**
    $h_\ell \leftarrow \bot$; $h_r \leftarrow \bot$
  else
    $h_\ell \leftarrow \text{skHashes}[v.\text{left.idx}]$
    $h_r \leftarrow \text{skHashes}[v.\text{left.idx}]$
  $v.h \leftarrow \mathsf{H}_k(v.pk, h_\ell, h_r)$
// Store public keys
for $v \in \tau.\text{RootPath}(n+1)$ **do**
  $v.pk \leftarrow \text{skPub}[v.\text{idx}]$
  if $\text{IsLeftChild}(v)$ **then**
    $v' \leftarrow \text{Sibling}(v)$
    $v'.h \leftarrow \text{skHashes}[v'.\text{idx}]$
return $(k, crs_{\mathsf{COM}}, h_{()}, n, \tau)$

Fig. 12: The decryption and the secret-key validation algorithms. The ValidateMTPath helper method is the same as in Fig. 5.

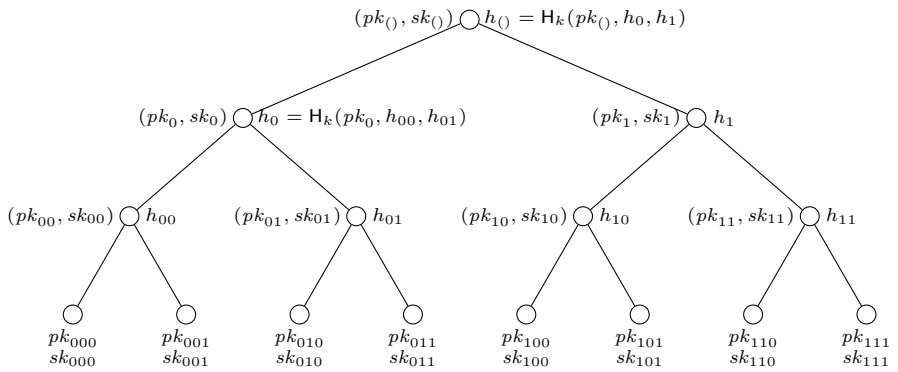

Fig. 13: The state set up by the manager for the insider-secure scheme, adding a Merkle-Tree over the public keys. The outsider-secure scheme uses the same tree of key pairs without the hashes.

– The necessary hashes to verify the PKE public keys in V_i and on the root path of the $(i+1)$-th leaf. In particular, the following values are included:
  - The hash for each node on the *right-sibling path* of the $(i+1)$-th leaf;
  - The hashes of the (immediate) *children* of each node on the *left-sibling path* of the $(i+1)$-th leaf.

  Note that from the latter hashes, together with the PKE public keys, the hashes on the left-sibling path itself can be derived. Therefore, the protocol can authenticate the entire root path of the $(i+1)$-th leaf. This in turn also authenticates its left sibling path.

Key validation. The ValidatePk algorithm then first checks that pk_i contains the correct epoch number and all the expected PKE public keys. It then verifies those PKE public keys with respect to the Merkle-Tree root contained in *pub*. The ValidateSk algorithm works analogously. For nodes on the left-sibling path of the $(i+1)$-th leaf, it first derives the key pair and computes its hash. Then it validates the Merkle-Tree commitments of the $(i+1)$-th root path.

Maintaining the manager state. Finally, let us discuss how to maintain the manager's state. In principle one could just store the PRF seed s as in the outsider-secure scheme. This would, however, require that essentially the whole Merkle-Tree is recomputed as part of each Update operation, taking $\mathcal{O}(2^d)$ time despite only needing $\mathcal{O}(d)$ hashes for each epoch. Alternatively, one could store all the hashes as part of the manager's state, making Update efficient at the cost of not having a compact state.

Fortunately, Merkle-Tree traversal [43,17] solves this trade-off. More specifically, for epoch i, the group manager needs to be able to open the Merkle-Tree commitments for the i-th leaf (for the public key) and the $(i+1)$-th leaf (for the secret key). A Merkle-Tree traversal exactly allows a party to open Merkle-Tree commitments for consecutive leaves in time $\mathcal{O}(d)$ while also keeping only $\mathcal{O}(d)$ state. One small caveat is that for the secret key, the manager does not just need the hashes of the left-sibling path but actually the hashes of the direct children of those nodes. Any Merkle-Tree traversal can be adjusted accordingly: Whenever the original traversal stores a hash of a node as part of its state, we can simply also keep the hashes of its two children, at a constant storage overhead. Since the only way to compute the hash of a node v is to know the hashes of its two children, this is clearly without loss of generality.

See Fig. 14 for a depiction of the state the manager expands when moving from epoch 3 to 4, to send as part of the secret key, and then keeps stored as its updated state.

Efficiency. Let us briefly comment on the (asymptotic) efficiency of the scheme. Let N denote the maximal number of supported epochs and $d = \log_2(N)$. We then observe the following.

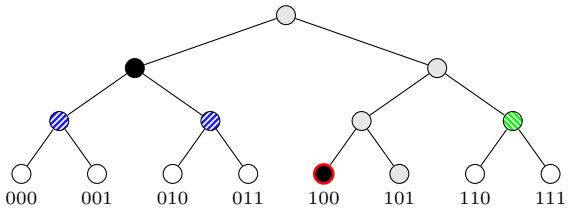

Fig. 14: An illustration of the manager's state in epoch 4. Consider the root path of the subsequent epoch 5 (nodes marked ○). The set V_4 then consists of its left-sibling path (nodes marked ●). For those nodes, the manager keeps around the randomness r_{idx}, the public key pk_{idx}, and the hash h_{idx}. For internal nodes, it additionally keeps the hash of its children as well as its sibling nodes (nodes marked ▨). Finally, it needs to be able to open the Merkle-Tree paths from the fourth and fifth leaf by having the remaining hashes for the sibling path ready (nodes marked ▨).

- Setup initially generates $O(N)$ PKE key-pairs and computes Merkle-Tree root over the PKE public keys. This takes $\mathcal{O}(N)$ overall time.
- A PE public key consists of $d+1$ PKE public keys and d hashes, while secret keys consists of $d+1$ PKE public keys, up to $d+1$ randomness seeds, and up to $2d$ hashes.
- Update re-computes all the PKE public keys needed for the next epoch's public and secret key, taking $\mathcal{O}(d)$ work. In addition, it generates the Merkle-Tree openings using Merkle-Tree traversal, which is known to be possible to implement in logarithmic space and time [43,17]. Hence, *mst* is of logarithmic size and Update runs in logarithmic time.
- ValidatePk and ValidateSk both run in time $\mathcal{O}(d)$, with the former simply verifying the Merkle-Tree openings and the latter doing some extra work to first derive some of the public keys.
- Encrypt and Decrypt have the same asymptotic running time as in the outsider-secure scheme.

In summary, our scheme thus runs in logarithmic time (in the number of supported epochs) except for the initial setup that runs in linear time.[7]

Cryptographic assumptions. The scheme uses the same cryptographic building blocks as the outsider-secure one. The cryptographic assumptions also remain the same except for the Public Key Encryption (PKE) scheme, of which we need stronger properties as keys may now be adversarially generated.

In the following, we assume decryption $\mathsf{PKE.Dec}$ to be deterministic. In addition, we require that

$$\Pr\left[m \neq m' \;\middle|\; \begin{array}{l} (m, r_1, r_2) \leftarrow \mathcal{A}(1^\lambda) \\ (sk, pk) \leftarrow \mathsf{PKE.Keygen}(1^\lambda; r_1) \\ c \leftarrow \mathsf{PKE.Enc}(pk, m; r_2) \\ m' \leftarrow \mathsf{PKE.Dec}(sk, c) \end{array} \right] \tag{1}$$

is negligible in λ, where the randomness is taken over the adversary. Intuitively, this property will ensure that the re-encryption checks done by a party agrees with the attempted decryption performed under a later epoch's secret key. We remark that this property is implied, for example, by perfect correctness.

Finally, we require that the following probability is negligible in the security parameter:

$$\Pr\left[\begin{array}{l} m_1 \neq m_2 \\ \quad \wedge\, pk_1 = pk_2 \end{array} \;\middle|\; \begin{array}{ll} (c, r_1, r_2) \leftarrow \mathcal{A}(1^\lambda) & \\ \forall i \in \{1,2\} : & (sk_i, pk_i) \leftarrow \mathsf{PKE.Keygen}(1^\lambda; r_i) \\ & m_i \leftarrow \mathsf{PKE.Dec}(sk_i, c) \end{array} \right] \tag{2}$$

[7] In a slightly weaker model where parties are just assumed to have a consistent view on the public keys, e.g. by having them posted on a blockchain, the scheme could thus be modified to still support an unbounded number of epochs.

when the randomness is taken over the adversary. Intuitively this will ensure that there cannot be two secret keys, for the same epoch, that decrypt the same PKE ciphertext inconsistently. The property, for instance, is implied by schemes for which the public key uniquely determines the secret key when taken in conjunction with the previous property.

Note that the two properties are independent — one can craft a (contrived) PKE scheme that satisfies one but not the other. The former considers proper ciphertext in the range of $\mathsf{PKE.Enc}$ only, implying that a scheme can be made to violate the second property by adding extraneous information to each ciphertext influencing decryption depending on the secret key. (This can be done while maintaining IND-CCA security.) The former can be violated by breaking correctness for certain specific choices of randomness, that only get chosen with negligible probability in an honest execution, which can also be done for schemes with a unique secret-key to public-key mapping.

Correctness and security. We now argue the correctness and security of our enhanced scheme. The proofs mostly extend the ones from the outsider-secure scheme, making use of the Merkle-proofs (assuming collision resistance of the hash function) and the stronger correctness properties of the PKE scheme (to thwart adversarially generated keys).

Theorem 5. *The PE scheme depicted in Figs. 10 to 12 is correct, if the underlying public-key encryption scheme is correct and the commitment scheme is binding.*

Proof. Correctness essentially follows from the correctness of the outsider-secure scheme. Whether we include PKE secret keys sk_{idx} or the randomness r_{idx} from which Decrypt re-computes sk_{idx} is irrelevant. Hence, it remains to argue that ValidatePk and ValidateSk accept honestly generated keys. This follows by inspection by observing that pk_i and sk_i, respectively, include all the hash values to verify the Merkle-Tree openings against the honestly computed root included in pub. □

Theorem 6. *The PE scheme from Figs. 10 to 12 satisfies pseudo-randomness (cf. Fig. 3), if the underlying public-key encryption scheme is correct and IND-CCA secure with pseudo-random ciphertexts, the commitment scheme has pseudo-randomness, the hash function is collision-resistant, and the PRF and PRG satisfy their usual security notions.*

Proof. Again, the proof from the outsider-secure scheme directly carries over. The reduction can compute the Merkle-root, to be included in pub, by sampling the (bounded number of) key pairs upfront. The reduction can at this point further store all the intermediate hashes to produce the Merkle-openings needed to be included in the public- and secret-keys. Note that upon corruption the reduction can include randomness seeds instead of PKE secret keys as part of sk_i. Since the challenge instance cannot be corrupted, this is without loss of generality. □

Theorem 7. *The PE scheme depicted in Figs. 10 to 12 is insider-secure, if the commitment scheme is binding, the hash function collision resistant, and the PKE scheme satisfies the strong correctness properties* (1) *and* (2) *introduced before.*

Proof. We adapt the proof from the outsider-secure scheme. Consider decryption under adversarially generated secret keys for epochs i and j, assuming that both secret keys sk_i and sk_j validate under the same (honestly generated) CRS crs, which contains the hash key k and the commitment CRS crs_{COM}, and (maliciously generated) public parameter pub, which contains the Merkle-Tree root. The verification checks the structure of the secret keys as well as the Merkle-Tree openings. By collision resistance of $\mathsf{H}_k(\cdot)$, we can thus observe that the two secret keys must agree on PKE public keys for common nodes in the tree, i.e., for any PKE public key that both PE secret keys contain.

First, let us consider the case of $i = j$. Note that in contrast to outsider-security sk_i and sk_j are not necessarily the same key; they agree on the PKE public keys (as per the Merkle commitments) but might contain different PKE secret keys. However, for any node idx, even if $sk_{idx,i} \neq sk_{idx,j}$, then the respective ciphertext component still decrypts to the same message, by Property (2). Since the re-encryptions are guaranteed to happen under the same PKE public keys, the overall decryption process will therefore still behave the same.

In the following assume that $j > i$ and assume that $e \leq i$. (The other cases are analogous to the outsider-secure scheme.) Recall the following observation from the proof of the outsider-secure scheme:

Observation 4. Let $e \leq i \leq j$ and let v_i denote the unique node in the intersection of RootPath(e) and LeftSiblingPath($i+1$) and v_j the unique node in RootPath(e) $\cap$ LeftSiblingPath($j+1$), respectively. (Cf. observation 1 from the correctness proof.) Then we have that either $v_i = v_j$, or that v_j is closer to the root.

We can now adjust the following claims.

Claim. If $v_i \neq v_j$, then either the decryption using sk_i rejects, or both output the same message.

Proof: The proof is mostly analogous, except having to account for maliciously generated key-pairs. Assume that the PKE ciphertext for node v_i decrypts to (r_{COM}, s, m). The ciphertext associated to node v_j equal to the one obtained by re-encrypts that message under $v_j.pk$, with the appropriate randomness. Therefore, Property (2) implies that decrypting under sk_j will also produce (r_{COM}, s, m), despite adversarially chosen randomness for the PKE key-generation and the encryption.

Claim. If $v_i = v_j$, then both produce the same message.

Proof: Again, the only subtlety is that $v_i.sk$ (the secret key used by party i) is no longer guaranteed to be the same as $v_j.sk$ (the secret key used by party j). However, by Property (1) of the PKE scheme, they will still agree on the decryption. □

Theorem 8. *The PE scheme depicted in Figs. 10 to 12 is robust, if the PKE scheme satisfies the strong correctness property* (1).

Proof. By inspection, we can see that robustness follows from Property (1) of the PKE scheme. The ciphertext contains valid PKE ciphertexts on the root path, each of them guaranteed to decrypt under a PKE secret-key that passes the checks performed in ValidateSk. □

Strong Perpetuity and Interval Access. The scheme can be enhanced to satisfy strong perpetuity analogous to the outsider-secure scheme (where the first modification becomes redundant). We refer to Section 4.3 for a description thereof. Finally, we observe once more that the corresponding scheme naturally supports delegation of interval keys, analogous to Section 4.

5.3 Unbounded Epochs from Anonymous HIBE

In this section, we sketch an unbounded-epoch insider-secure PE scheme from hierarchical identity-based encryption (HIBE) with pseudorandom ciphertexts. On a high level, the scheme is once more based on the outsider-secure scheme from Section 4 when swapping out the PKE for HIBE. Observe that many anonymous HIBE schemes, such as the one by Boyen and Waters [16], do feature pseudorandom ciphertexts over a suitable ciphertext space.[8] Elligator-Squared [44] is moreover compatible with (some) elliptic curves used for pairings, in case ciphertexts are required to look like pseudorandom bit strings. Finally, for CCA security, while the Canetti-Halevi-Katz transform [20] does not preserve pseudorandomness, the Fujisaki-Okamoto or the Boneh-Katz transform [13] do.

Protocol modifications. In a nutshell, we replace the binary tree of PKE key pairs with a HIBE tree, where each node index *idx* corresponds to a (binary) identity. The common reference string *crs* remains unchanged.

Encryption and decryption. Encryption works analogous to the outsider-secure scheme, except for encrypting to the identity *idx* instead of encrypting under the PKE public-key $v_{idx}.pk$. In particular, the encryption randomness is derived in the same manner, the individual ciphertext components are bound by the same hash chain, and the commitment to the epoch number is included. Decryption is even more directly analogous, using the HIBE secret keys for the respective node instead of its PKE secret key.

[8] While Boyen and Waters only state recipient anonymity, their proof is via pseudorandomness of the ciphertexts.

Committing to the public keys. We observe that HIBE uses a master public key instead of per-identity public keys. Therefore, the master public key inherently serves as a "commitment" to all public keys. This renders the Merkle-Tree unnecessary and allows us to avoid the $\mathcal{O}(N)$ runtime of the Setup algorithm in the bounded-epoch scheme. Instead, Setup simply initializes the HIBE scheme and outputs the master public key as part of the public parameters pub — the PE public key pk_i is then just to epoch number i.

This furthermore implies that the public key validation becomes trivial, as $\mathrm{ValidatePk}(crs, pub, i, pk_i)$ simply checks that the epoch number in the public key is equal to i.

Committing to the secret keys. For secret keys, however, we cannot rely on the same technique as in Section 5.2: Since the keys for nodes are no longer independent of each other, the group manager cannot send the randomness used for the key generation as proof of consistency. Nevertheless, ValidateSk needs to establish the validity of the individual HIBE secret keys, for the nodes the PE secret key sk_i contains.

For pairing-based schemes, this however can be done by checking that a fresh encryption of the neutral elements decrypts correctly. That is, ValidateSk just checks that

$$\mathsf{HIBE.dec}\big(\mathsf{hsk}_{\mathbf{id}}, \mathsf{HIBE.enc}(\mathsf{mpk}, \mathbf{id}, m)\big) \stackrel{?}{=} m$$

for some fixed message m, where $\mathsf{hsk}_{\mathbf{id}}$ is the presumed secret key for $\mathbf{id}$. The particular details of these pairing-based HIBE schemes are not important, so we only sketch the property we need to make things work for our setting using the Boyen and Waters HIBE as an example. We refer to [16] for details on the scheme.

To this end, observe that the Boyen-Waters HIBE encrypts a message $M \in \mathbb{G}_1$ by sampling $(r, r_0, \ldots, r_{1+d}) \in \mathbb{Z}_p^{3+d}$, where d denotes the maximal identity depth. Then it blinds M as $E = M \cdot \Omega^{-r}$, where $\Omega \in \mathbb{G}_1$ is part of the public key. It additionally includes g^r and similar information about $(r_0, \ldots, r_{1+d})$ as part of the ciphertext. Decryption then works by computing $M = E \cdot F$, where $F = E^{-1}$ when using the correct secret key. Crucially, we observe that E depends on the public key, the identity, and the encryption randomness, while F is computed using the secret key and information about $(r, r_0, \ldots, r_{1+d})$ included as part of the ciphertext (e.g. g^r). This is a general template followed by essentially all pairing-based schemes. More abstractly, for a pairing-based HIBE with encryption randomness $(r_0, \ldots, r_\ell)$ we observe the following:

1. as $E \cdot F$ has to be computable using the pairing operation during decryption, it is of the form
$$e(g,g)^{\sum_{i,j=1}^{\ell} a_{i,j} \cdot r_i \cdot r_j},$$
where the $a_{i,j}$ depend on the public key, the identity, and (alleged) secret key;
2. the schemes have perfect correctness, i.e., $\sum_{i,j=1}^{\ell} a_{i,j} r_i r_j \equiv 0$ when F is computed based on a valid secret key.

Therefore, we can apply the Schwartz–Zippel lemma to conclude that the above sketched ValidateSk indeed only accepts valid secret keys, except for negligible probability. The PE secret key sk_i can therefore be defined analogous to the outsider-secure scheme to contain:

- The epoch number i.
- The HIBE secret keys hsk_{idx} for each $idx \in V_i$, when idx is understood as an identity vector over $\mathbb{Z}_2$ of appropriate length.

The ValidateSk algorithm then validates sk_i as outlined above with respect to mpk contained in pub.

Efficiency. Let N denote the maximal number of supported epochs and let $d = \log_2(N)$ denote the maximum identity depth. For any unbounded HIBE scheme with pseudorandom ciphertexts, we then observe the following.

- Setup samples a single HIBE setup and derives the secret-key for the leftmost leaf node. This takes $\mathsf{poly}(d)$ time.

– The public parameters pub and the public keys pk_i are of polynomial size. ValidatePk is trivial.
– The secret key sk_i is of size $\mathsf{poly}(d)$. The ValidateSk algorithm can be implemented in time $\mathsf{poly}(d)$ as well.
– Encrypt and Decrypt can be implemented in time $\mathsf{poly}(d)$.

This allows us to choose $d = \lambda$, making the scheme unbounded. Note that for (weak) insider-perpetuity the decryption algorithm does not need to delegate secret keys further down the tree. For many schemes HIBE schemes, this allows to strip parts of the secret key handed to group members. For instance, for the Boyen-Water scheme, a secret key can be decomposed as $\mathsf{hsk}_{\mathbf{id}} = \mathsf{hsk}_{\mathbf{id}}^{\mathsf{decrypt}} || \mathsf{hsk}_{\mathbf{id}}^{\mathsf{rerand}} || \mathsf{hsk}_{\mathbf{id}}^{\mathsf{deleg}}$, with only the first component necessary for decryption. This reduced the size of a secret key from $\mathcal{O}(d^2)$ to $\mathcal{O}(d)$.

Correctness and Security. Correctness and security is, for the most part, a straightforward adaptation of the proofs from the outsider scheme. In the following, we state the main results and briefly point out the relevant properties of the underlying schemes.

Theorem 9. *The HIBE-based PE scheme is correct if the HIBE scheme is correct and the commitment scheme is binding.*

Theorem 10. *The HIBE-based PE scheme satisfies pseudorandomness, if the HIBE scheme is correct and IND-CCA secure with pseudorandom ciphertexts, the commitment scheme has pseudorandomness, the hash function is collision-resistant, and the PRF and PRG satisfy their usual security notions.*

Proof (Sketch). The proof follows analogous to the proof of the outsider-secure scheme. Observe that the delegation property of a HIBE scheme does no harm. More concretely, while in the original scheme, the secret key contained secrets for $v_{idx} \in V_i$ only, the HIBE key can also be thought of as containing keys for nodes that are descendants of v_{idx}. Since encryption always encrypts along a root path, no additional ciphertexts can be decrypted with those keys, however.

Theorem 11. *The HIBE-based PE scheme is insider-secure, if the commitment scheme is binding, the hash function is collision-resistant, and the HIBE scheme is perfectly correct and allows verifying a secret key for a given identity. Given those conditions on the HIBE scheme, it is moreover robust.*

Proof (Sketch). Outsider-security follows analogous to the proof of the outsider-secure scheme. To see that the scheme is, moreover, secure against maliciously generated key pairs, we observe the following. First, the master-public key does not have any structure, by perfect correctness the encryption under any $\mathsf{mpk} \in \mathbb{G}^{d+4}$ can be decrypted given a HIBE secret key that is valid with respect to mpk. This, in particular, implies robustness. In addition, however, it implies that if one party (using sk_i) checks a ciphertext component by re-encryption and another party (using sk_j) by decrypting it, then they will arrive at the same conclusion.

Strong Perpetuity and Interval Access. The HIBE-based scheme naturally supports strong perpetuity and interval delegation. By definition of a HIBE scheme, $\mathsf{hsk}_{\mathbf{id}}$ allows deriving the secret keys for identities further down the tree. Therefore, once the trial-decryption under sk_i determined the correct epoch $j \leq i$, the decryptor can derive the private key of the corresponding epoch's leaf and decrypt its ciphertext component equivalently to how decryption under sk_j would have done, then validating the entire path up. The decryption process is roughly of the same efficiency as in the scheme with weak perpetuity. The main downside is the increased secret key size — in the case of Boyen-Waters $\mathcal{O}(d^2)$ instead of $\mathcal{O}(d)$ — and the respective increased complexity for ValidateSk.

References

1. Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 480–497. Springer, Berlin, Heidelberg, February 2010.
2. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Cham, August 2020.
3. Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 69–82. ACM Press, November 2022.
4. Diego F. Aranha, Pierre-Alain Fouque, Chen Qian, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. Binary elligator squared. In Antoine Joux and Amr M. Youssef, editors, *SAC 2014*, volume 8781 of *LNCS*, pages 20–37. Springer, Cham, August 2014.
5. Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. Message layer security (mls) wg. https://datatracker.ietf.org/wg/mls/about/.
6. Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, Berlin, Heidelberg, December 2001.
7. Mihir Bellare and Adriana Palacio. Protecting against key exposure: Strongly key-insulated encryption with optimal threshold. *Applicable Algebra in Engineering, Communication and Computing*, 16, 06 2002.
8. Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013.
9. Alexander Bienstock, Yevgeniy Dodis, Paul Rösler, and Daniel Wichs. Interval key-encapsulation mechanism. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part II*, volume 15485 of *LNCS*, pages 361–393. Springer, Singapore, December 2024.
10. Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. Multicast key agreement, revisited. In Steven D. Galbraith, editor, *CT-RSA 2022*, volume 13161 of *LNCS*, pages 1–25. Springer, Cham, March 2022.
11. Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 506–522. Springer, Berlin, Heidelberg, May 2004.
12. Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. Improving speed and security in updatable encryption schemes. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 559–589. Springer, Cham, December 2020.
13. Dan Boneh and Jonathan Katz. Improved efficiency for CCA-secure cryptosystems built using identity-based encryption. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 87–103. Springer, Berlin, Heidelberg, February 2005.
14. Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Berlin, Heidelberg, August 2013.
15. Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. Fast and secure updatable encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 464–493. Springer, Cham, August 2020.
16. Xavier Boyen and Brent Waters. Anonymous hierarchical identity-based encryption (without random oracles). In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 290–307. Springer, Berlin, Heidelberg, August 2006.
17. Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, pages 63–78, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
18. Ran Canetti, Juan A. Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM'99*, pages 708–716, New York, NY, USA, March 21–25, 1999.
19. 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, Berlin, Heidelberg, May 2003.
20. Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-ciphertext security from identity-based encryption. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 207–222. Springer, Berlin, Heidelberg, May 2004.

21. Jung Hee Cheon, Nicholas Hopper, Yongdae Kim, and Ivan Osipkov. Timed-release and key-insulated public key encryption. In Giovanni Di Crescenzo and Avi Rubin, editors, *FC 2006*, volume 4107 of *LNCS*, pages 191–205. Springer, Berlin, Heidelberg, February / March 2006.
22. Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. A formal treatment of backdoored pseudorandom generators. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 101–126. Springer, Berlin, Heidelberg, April 2015.
23. Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Key-insulated public key cryptosystems. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 65–82. Springer, Berlin, Heidelberg, April / May 2002.
24. Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Strong key-insulated signature schemes. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 130–144. Springer, Berlin, Heidelberg, January 2003.
25. Yevgeniy Dodis, Weiliang Luo, Shouhuai Xu, and Moti Yung. Key-insulated symmetric key cryptography and mitigating attacks against cryptographic cloud software. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 57–58. ACM Press, May 2012.
26. Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 98–129. Springer, Cham, August 2017.
27. Pooya Farshim, Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. Robust encryption, revisited. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 352–368. Springer, Berlin, Heidelberg, February / March 2013.
28. Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 548–566. Springer, Berlin, Heidelberg, December 2002.
29. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 276–288. Springer, Berlin, Heidelberg, August 1984.
30. Goichiro Hanaoka, Yumiko Hanaoka, and Hideki Imai. Parallel key-insulated public key encryption. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 105–122. Springer, Berlin, Heidelberg, April 2006.
31. Yumiko Hanaoka, Goichiro Hanaoka, Junji Shikata, and Hideki Imai. Identity-based hierarchical strongly key-insulated encryption and its application. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 495–514. Springer, Berlin, Heidelberg, December 2005.
32. 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, Berlin, Heidelberg, April / May 2002.
33. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Cham, August 2018.
34. Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Cham, May 2019.
35. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.
36. Michael Klooß, Anja Lehmann, and Andy Rupp. (R)CCA secure updatable encryption with integrity protection. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 68–99. Springer, Cham, May 2019.
37. Anja Lehmann and Björn Tackmann. Updatable encryption with post-compromise security. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 685–716. Springer, Cham, April / May 2018.
38. Suvo Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of ACM SIGCOMM*, pages 277–288, Cannes, France, September 14–18, 1997.
39. Bodo Möller. A public-key encryption scheme with pseudo-random ciphertexts. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *ESORICS 2004*, volume 3193 of *LNCS*, pages 335–351. Springer, Berlin, Heidelberg, September 2004.
40. Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Cham, August 2018.

41. A.T. Sherman and D.A. McGrew. Key establishment in large dynamic groups using one-way function trees. *IEEE Transactions on Software Engineering*, 29(5):444–458, 2003.
42. Signal. Technology preview: Sealed sender for Signal, 29 October 2018. `https://signal.org/blog/sealed-sender/`.
43. Michael Szydlo. Merkle tree traversal in log space and time. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 541–554, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
44. Mehdi Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 139–156. Springer, Berlin, Heidelberg, March 2014.
45. D. M. Wallner, E. J. Harder, and R. C. Agee. Key management for multicast: Issues and architectures. Internet Draft, September 1998. `http://www.ietf.org/ID.html`.
46. Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM*, pages 68–79, Vancouver, BC, Canada, August 31 – September 4, 1998.