ELEKTRA: Efficient Lightweight multi-dEvice Key TRAnsparency*

Julia Len†
Cornell Tech
New York, USA
jlen@cs.cornell.edu

Melissa Chase
Microsoft Research
Redmond, USA
melissac@microsoft.com

Esha Ghosh
Microsoft Research
Redmond, USA
esha.ghosh@microsoft.com

Daniel Jost
New York University
New York, USA
daniel.jost@cs.nyu.edu

Balachandar Kesavan
Zoom Video Communications
New York, USA
surya.heronhaye@zoom.us

Antonio Marcedone
Zoom Video Communications
New York, USA
antonio.marcedone@zoom.us

ABSTRACT

Key Transparency (KT) systems enable service providers of end-to-end encrypted communication (E2EE) platforms to maintain a *Verifiable Key Directory* (VKD) that maps each user's identifier, such as a username or email address, to their identity public key(s). Users periodically monitor the directory to ensure their own identifier maps to the correct keys, thus detecting any attempt to register a fake key on their behalf to Meddler-in-the-Middle (MitM) their communications.

We introduce and formalize a new primitive called *Multi-Device Verifiable Key Directory* (MVKD), which strengthens both the security, privacy, and usability guarantees of VKD by leveraging the multi-device setting. We formalize three properties for a MVKD (*completeness*, *extraction-based soundness*, and *privacy*), striking a non-trivial balance between strong guarantees and the limitations imposed by a truly practical system. We then present a new MVKD system called ELEKTRA. This system combines the core of the Keybase KT system (running in production since 2014) with ideas from SEEMless (Chase et. al., 2019) and RZKS (Chen et. al., 2022). Our construction is the first to achieve the above multi-device guarantees while having formal security and privacy proofs. Finally, we implement ELEKTRA and present benchmarks demonstrating its practicality.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; **Key management**; *Public key encryption*; *Privacy-preserving protocols*.

KEYWORDS

key transparency; verifiable key directory; multi-device; PK; rotatable zero-knowledge set; post-compromise security

1 INTRODUCTION

End-to-end encrypted (E2EE) communication systems (such as Signal [23], Keybase [12], Zoom [6], Teams [21], WhatsApp [29], iMessage [3], and Webex [28]) are becoming increasingly ubiquitous. E2EE systems typically require each user to generate a public key pair and use their own on-device secret key, along with their communication partner's public key, to establish a secure communication channel between them. Notably, this process hinges on both parties receiving the correct key for their communication partner. Existing E2EE systems have the service providers maintain a directory that maps the user identifiers (such as a username, email address, phone number, etc.) to their most recent public keys, distributing users' keys on their behalf. This is great for usability, but consequently creates the need for verifying the authenticity of these keys. A malicious service provider (or one who is hacked or compelled to act maliciously) can, for instance, replace an honest user's identity public key with another public key whose secret key is known to the provider. The service provider could thus implement a Meddler-in-the-Middle (MitM) attack without the communicating users ever noticing. Most applications mitigate this risk by suggesting that users compare key fingerprints (rendered as QR codes or "security codes") with their conversation partners out-of-band, but this introduces high friction (see e.g. [9]) and can often be infeasible.

Key Transparency. An automated solution to this problem, and one that is, therefore, more likely to provide guarantees for the average user, is a Key Transparency (KT) system, which enables service providers to maintain an *auditable* directory that maps each user's username[1] to their identity public keys. Service providers compute and advertise a short "commitment" com to the whole directory, and update it (creating a new epoch) whenever users join the directory or update their keys. When users query a particular username, they get the corresponding public key and a proof that the reply is consistent with com. Users periodically monitor the directory to make sure their own user identifier maps to the correct keys, thus detecting any attempt to MitM their communications by associating a fake key with their identity.

However, as the landscape of E2EE messaging has evolved, prior work on KT systems [2, 5, 7, 11, 15, 17, 19, 24–27] has not considered or modeled important new features arising in practice. For instance, prior KT systems have not explicitly accounted for cases where a user can have multiple (trusted) devices (such as their laptop, mobile phone, etc.), which more recently has become a standard feature of E2EE messaging. One proposal for extending KT systems to support multiple devices is to simply have the auditable key directory map the user's identifier to a list of their identity public keys (one per device). In particular, this is the approach suggested by Parakeet [18], which WhatsApp recently announced it plans to deploy [16]. When a client queries for a username, a list of keys is returned along with a proof attesting that the query response is consistent with the posted commitment. However, this approach

*This is the full version of the article with the same title published in the Proceedings of ACM CCS '23.
†Author contributed in part while an intern at Zoom Video Communications.
[1]We will use username as the user identifier in the rest of the paper.

fails to capture the stronger security guarantee that a multi-device setting can provide in practice. For example, if a user has multiple trusted devices, they can sign a key update (e.g. adding a new device's public key or revoking an existing one's) with one of the existing device keys. This means that even a malicious service provider (or one who is coerced to act maliciously) cannot put a fake key on a user's account easily – they would need to forge a signature or compromise one of the user's devices (assuming the KT system is audited for consistency and users remember their conversation partners' keys).

This is the approach taken by Keybase, the only deployed KT system to the best of our knowledge. Keybase offers an auditable multi-device KT system that allows users to cryptographically link their identity keys to their social media accounts and leverages these keys for several applications, including E2EE chats.

Another important aspect to consider in a multi-device system is what happens if a user loses access to all of their devices. Typically, E2EE applications will issue an "account reset": both Signal [22] and Keybase [14], for instance, enable a user to register a new device with their account that replaces their old device(s) and notifies contacts about the change. Indeed, resets allow for the system to be more user-friendly, enabling a user to keep their username. While some may argue that allowing key resets might void the additional guarantees of a multi-device KT system (e.g. an attacker who compromises the service provider could simply reset a user's key and begin to impersonate them), the prevalence of reset features in today's E2EE messaging ecosystem points to the importance of this functionality for end users. Furthermore, in practice, key resets can be disabled in particularly sensitive deployments, and are often executed only with extra security measures. For example, Keybase only allows this under very stringent conditions: users need to prove ownership of the account through the associated password, email, or phone number and are forced to wait a week while Keybase tries to notify the owner through those channels. Moreover, since resets are rarer than a user just getting a new phone or re-installing the app, all the user's conversation partners are forced to actively acknowledge an urgent warning that invites them to check in with the user out-of-band. The server cannot suppress this notification as clients remember their partners' sets of keys and can thus detect these changes. Key resets are therefore an integral and practical feature of E2EE messaging that should be taken into consideration when constructing and analyzing KT systems.

Importantly, the necessary functionalities we have described thus far have not been considered in the formal literature on KT systems, pointing to a gap between the theory of these systems and the practice of what E2EE messaging requires. Furthermore, while Keybase's protocol does capture these practical requirements, it lacks formal security guarantees and another important feature that KT systems should target: privacy.

Privacy. Recent academic and industry proposals and the IETF draft charter for KT have shed light on the importance of achieving privacy [1, 6–8, 10, 18, 19]. In particular, the data stored by KT systems can be sensitive: users might want to hide the fact that they are using a specific KT system or applications from unauthorized parties, such as auditors and other users who would not otherwise be able to query for those usernames. These systems can also leak metadata, such as when a particular user first registered or when and how often their keys change. In fact, Google, Zoom, and WhatsApp explicitly advocate for adding both content and metadata privacy to KT systems [6, 10, 16]. Keybase, however, was originally designed as an alternative solution for PGP key distribution and therefore does not aim to achieve any user key privacy.

Moreover, while recent proposals for KT systems have considered privacy, none have considered the more difficult goal of recovering privacy even after a server state compromise, a notion similar to post-compromise security for E2EE messaging. Given the sensitivity of data stored in a KT system and the importance given to post-compromise security in practice, a practical KT system should have that any keys added to the system remain private even if the server state had been leaked *before* those additions were made. This property, first formalized in [8], is non-trivial because KT systems typically achieve privacy through asymmetric cryptographic primitives such as verifiable random functions (VRFs) [20]. If the server were temporarily compromised, the secret keys for these primitives would be leaked, and therefore the protocol would need a way to restore these privacy guarantees without compromising auditability or security. We therefore argue, as in [8], that a practical KT system must not only target security but also meet the more complex goal of privacy with post-compromise security.

New Primitive: Multi-Device Verifiable Key Directory. To support the multi-device setting discussed above, we define a new cryptographic primitive called the *Multi-Device Verifiable Key Directory* (MVKD), which extends the Verifiable Key Directory (VKD) primitive defined in [7]. Our primitive assumes three types of parties: the server, the auditors, and the users. The server maintains a key directory of users' public keys. It updates the directory in batches, periodically publishes a commitment to the current state of the directory, and distributes public keys on a user's behalf. Auditors, which can be dedicated third parties or even users themselves, then verify the consistency of successive versions of the directory based on the posted commitments.

Each user has a *username* that uniquely identifies them within the system. In our model, users can have several public key pairs corresponding to multiple devices and can dynamically add new keys and revoke old ones. Each change to a user's set of keys must be authenticated by an existing key[2], the obvious exceptions being the user's very first key and a full account reset. To simplify notation, we henceforth assume there to be a one-to-one correspondence between key pairs and devices, i.e., adding a new key corresponds to adding a new device and revoking a key to revoking a device.[3] Each device stores a secret key that only it knows, while the server stores and distributes the associated public key on the user's behalf. Users can thus query for a specific username to get the history of device updates (and key changes) on that user's account.

MVKD provides two major improvements over VKD: (1) stronger security guarantees for users with multiple devices, and (2) the

[2]Note that this means that our directory will store signing keys. While we will not explicitly model encryption keys or formalize applications like E2EE, we provide interfaces to allow the user to add additional externally generated keys, which makes these extensions straightforward. We provide a discussion of how to support E2EE in Section 2.

[3]This notation is not limiting any functionality since a device rotating their key can be seen as the addition of a new device followed by the revocation of the old one.

ability to recover privacy guarantees after a server state leak. To understand improvement (1), we highlight that VKD made a simplifying assumption that each user has only one key pair. In their model, clients could potentially post a new ephemeral one-time public key each time they use the system and then immediately delete the corresponding secret key without losing any security guarantees. VKD thus treats clients as stateless. However, our setting of multiple devices per user is significantly more complex. This functionality fundamentally means we need to support a notion of clients with persistent device state that contains secrets, which must be used in the MVKD protocol for authorizing updates. This forces us to model client corruptions as well as capture leakage related to a client's secret state. Furthermore, we require the addition of a new device to be authorized not only by the adding device but also by the device being added itself. This implies that parties cannot register other people's keys under their own name — which is a security property many real-world applications enforce, and a slightly weaker variant of the key-registration with knowledge [4] notion. Our model therefore not only allows for stronger security but also better captures the E2EE setting, where users typically have multiple devices, and those devices can be corrupted.

For improvement (2), while VKD targeted privacy as a security goal, it did not consider post-compromise security (PCS), e.g. how to recover privacy after a server state compromise. Our MVKD primitive, however, offers two ways of updating the directory: a more efficient Update algorithm to be used during regular intervals, and a separate PCSUpdate algorithm that leverages PCS functionality from Rotatable Zero-Knowledge Sets [8] (RZKS) to periodically "rotate" the server key and thus restore privacy in case of a compromise. This adds further complexity to our model, as now we must capture server-state corruptions.

Rigorous Security Definitions. We formally define the security properties of MVKD through *completeness*, *soundness*, and *privacy*. We stress that, in recent years, there has been a lot of progress in building KT systems, but none of these systems simultaneously support multiple devices (with the strong guarantees we discuss) and formally prove security. For instance, SEEMless [7] and Parakeet [18] have formally defined and proven security, but lack the multi-device aspect as described above. Keybase and other more recent academic works [11, 27], on the other hand, let client devices sign new updates, but do not provide any formal security or privacy guarantees for public keys.

To the best of our knowledge, ours is the first rigorous formalization of a multi-device KT system, and given the interest in deploying them [1, 6], we believe this to be of great significance. We have seen time and again that attacks are discovered on deployed systems, and not having rigorous security guarantees will only aggravate this problem. It is incredibly difficult to fix or introduce new features (such as privacy or multi-device support) to systems that are already in production without breaking backward compatibility. This is even harder for transparency systems, where such a change would have to occur in a fully transparent way.

ELEKTRA: a MVKD system. We design ELEKTRA: a practical multi-device key transparency system with PCS. Our construction takes the core protocol of Keybase and uses ideas from SEEMless [7] and RZKS [8] to add privacy. This results in a MVKD construction based on verifiable random functions (VRFs) [20], hash functions, and standard digital signature schemes. We show the protocol to satisfy our formal completeness, soundness, and privacy definitions.

Our starting point is the core protocol of Keybase, where each user has an associated hashchain of signed statements (referred to as "sigchains"). Keybase maintains a sparse Merkle tree, which maps each username to that user's most recent sigchain tail: the username corresponds to a leaf position in the sparse Merkle tree, and the leaf contains the sigchain tail. When a user updates their sigchain, the old tail is replaced with the new one at the same leaf position, which thus remains the same through all updates for the same user. The tree itself is publicly available (the server makes it available through an unauthenticated channel), and the server signs its root and posts it on the Stellar blockchain. Our protocol replaces the sparse Merkle tree of Keybase with a RZKS [8], an authenticated data structure that strengthens Merkle trees with stronger privacy properties and enforces that updates are append-only (once added, existing entries cannot be modified or deleted). We leverage the RZKS using the same approach as SEEMless: we store every update to a user's keychain (an abstraction of Keybase's sigchains) in a "new location" in the tree (determined using a VRF) as opposed to a fixed location. The RZKS allows an external auditor or third party to verify that the append-only property is respected while limiting leakage about which users updated their chains. The main novelty of our construction is in combining these somewhat orthogonal ideas into a cohesive and provably-secure protocol that retains the advantages of each of them: the strong multi-device security of Keybase with the extra privacy guarantees of SEEMless and the PCS of RZKS.

Implementation and experiments. To show the practicality of our system, we implemented ELEKTRA and measured its performance by running the server on an AWS EC2 `c5.4xlarge` instance (16 cores, 32 GB memory) and the client on a Google Pixel 6 (8 cores, 8 GB memory), interacting over the internet. Given a MVKD containing 4M keychain updates (for example, 1M users who each updated their set of keys/devices 4 times), a client can query for the keychains of 10 of those users in less than 300ms. The server can Update an MVKD of the same size with 10 new keychain updates in less than 100ms (including verifying the signed updates, but not including receiving them from clients). Performing a PCSUpdate operation instead takes 37 minutes (an operation that we expect to happen rarely, and that can almost completely happen in parallel with other queries and updates). Auditing the two update operations would take strictly less time than performing them.

We stress that our implementation is an academic prototype, and we expect significant performance improvements from applying standard systems engineering techniques such as more sharding, parallelization, and map-reduce-type workflows.

2 MULTI-DEVICE VERIFIABLE KEY DIRECTORY (MVKD)

In this section, we define and formalize the Multi-Device Verifiable Key Directory (MVKD), a new cryptographic primitive that extends the VKD primitive from [7]. At a high level, MVKD implements an auditable key directory, which allows each user to store a set of

multiple keys — including the history of how the set evolved — that can then be queried by other users.

2.1 Keychains

Before formalizing our MVKD primitive, we first introduce the data structure that MVKDs store, which we call a *keychain*. It captures the evolution of a user's public keys and is inspired by what Keybase calls *sigchains* [13][4]. It is composed of an ordered sequence of individual statements, called *keychain statements* or *links*.

Definition 1. *A **keychain statement** is a tuple of one of the following types:*

- (AddFirst, pk_a, pk_a, t)
- (Revoke, pk_a, pk, t)
- (Add, pk_a, pk, t)
- (Extra, pk_a, d, t)

The first element is a constant representing the action to be performed, the second element pk_a is the public key of the device authorizing the statement, the third element is the public key or data on which the action is performed, and the fourth element t is either an integer (the "epoch number" at which such statement is inserted in the keychain) or the special symbol $\perp$ (when this epoch has not yet been determined).

The AddFirst type represents when a user's very first public key is added or when a user's account is reset (see Section 1 for why we model account resets). Add is used to add additional keys, and Revoke denotes public key revocation. Extra enables the addition of user-specified data to a keychain. For example, this could be used to add a signed link to a social media profile, as in Keybase, or to include other externally generated keys to a user's profile, such as a cryptocurrency address.

We next define keychains and their validity, i.e., the sequence of operations a user can issue that are considered valid.

Definition 2. *A **keychain** consists of a string u (representing the username) and an ordered sequence of keychain statements that describe the keys associated with u. We say that $\mathbb{S}$ is valid, denoted* ValidKeychain($\mathbb{S}$)*, iff the following requirements are met:*

- *The epochs of keychain statements must be strictly increasing, except for the last statement which can have the epoch set to $\perp$.*
- *Each public key can only be added once.*
- *A public key can only be revoked if it has been previously added but not yet revoked (either via an explicit revoke or an* AddFirst *statement acting as the revocation for all prior keys).*
- *For each statement except for* AddFirst *(which is self-certifying) the authorizing key must be an unrevoked key (as defined above).*

Below we show an example of keychains and how they can capture data about public keys over time. We will use this as a running example throughout this work.

Example: Bob, Charlie, and Diana chat with each other via an E2EE communication platform, which stores users' public keys in its key directory using keychains. Bob registered his phone which has key pk_1^B at epoch 10 and then used his phone to register his laptop which has key pk_2^B at epoch 15. Charlie registered his phone with key pk_1^C at epoch 7, and Diana registered her phone with key pk_1^D at epoch 3.
Key Directory:
(Bob; (AddFirst, $pk_1^B, pk_1^B, 10$), (Add, $pk_1^B, pk_2^B, 15$))
(Charlie; (AddFirst, $pk_1^C, pk_1^C, 7$))
(Diana; (AddFirst, $pk_1^D, pk_1^D, 3$))

[4]For flexibility, we diverge from the notion of sigchains and do not include signatures in our formalization of keychains. We will later see in Section 3 how our construction adds signatures over updates in a way that captures Keybase's protocol.

2.2 MVKD Definition

Our MVKD primitive offers algorithms for users to amend their keychain. These algorithms produce an output that we call a *keychain statement authenticator* (auth) and represents the user device's authentication over the proposed change. These authenticators are sent to the server alongside the description of each intended change, where they are applied in batches to form the next epoch. For each new epoch, the server publishes a commitment com′ to the directory. Users then employ VerExtension to update their commitment and Query to request data from the directory. Finally, auditors verify the validity of each transition between two sequential commitments com_t and com_{t+1}. Such auditing could be performed by third parties or other users. We envision that MVKD will be deployed as part of a higher-level application that coordinates inputs to the various parties (e.g., it takes care of transmitting the statement authenticators from the clients to the server), and might perform additional access control (i.e., restrict who is allowed to query for a specific user's keychain).

To reduce the need for synchronization, a keychain statement's epoch is not fixed by the time a user submits a change request but rather filled in by the server upon processing it. Furthermore, clients can make queries with respect to a prior commitment. This enables a client to, e.g., only update their commitment once it has been verified by an auditor of their trust, yielding upfront guarantees rather than the after-the-fact detection offered by VerExtension.

Definition 3. *A* Multi-Device Verifiable Key Directory (MVKD) *scheme is a tuple of PPT algorithms* MVKD.{GenPP, ServerInit, DeviceSetup, AddFirstKey, AddKey, RevokeKey, AddExtra, Update, PCSUpdate, Query, VerExtension, Audit, Sign, Verify} *defined as follows:*

▷ $\mathsf{pp} \leftarrow \mathsf{MVKD.GenPP}(1^\lambda)$: This algorithm takes the security parameter as input and generates the public parameters for the scheme. Its output is implicitly assumed to be given as input to all other MVKD algorithms, even when not explicitly specified.
▷ $(st_0^s, \mathsf{com}_0) \leftarrow \mathsf{MVKD.ServerInit}(\mathsf{pp})$: Initializes the server state and initial commitment to the (empty) key directory for epoch 0.
▷ $(st, pk) \leftarrow \mathsf{MVKD.DeviceSetup}(\mathsf{pp}, u)$: Generates a new client device state st and public key pk to be used for username u. We assume w.l.o.g. that st stores a commitment com_t defining the last server epoch t the client has seen.
▷ $\mathsf{auth} \leftarrow \mathsf{MVKD.AddFirstKey}(st)$: Initiates a request to add the public key associated with this device (i.e. as output by DeviceSetup) to the user's keychain. It takes as input the device's state st and outputs the statement authenticator auth to be sent to the server.

Example: Alice joins the system. On her phone, she generates a new key pair (sk_1^A, pk_1^A) and calls AddFirstKey to start

her keychain. The first statement will be $(\mathsf{AddFirst}, pk_1^A, pk_1^A, \perp)$, where the epoch is $\perp$ as the statement has not been added by the server yet. AddFirstKey outputs $\mathsf{auth}_{\mathrm{Alice}}$ which Alice sends to the server.
Charlie loses his phone, his only device. He uses his new phone to generate a new key pair (sk_2^C, pk_2^C) and calls AddFirstKey to do a reset and register his new device using the statement $(\mathsf{AddFirst}, pk_2^C, pk_2^C, \perp)$. Once the server updates Charlie's keychain, his first key pk_1^C will be considered revoked.

▷ $(\mathsf{auth}\ ;\ b) \leftarrow \mathsf{MVKD.AddKey}(st_0, pk_1\ ;\ st_1, pk_0)$: AddKey is an interactive protocol for user u to use their existing device d_0 to authorize a new device d_1 and initiate a request to add the new device's public key to u's keychain $\mathbb{S}$. Each device d_i takes as input its own state st_i and the other device's public key pk_{1-i} for $i \in \{0, 1\}$. We assume that the public keys are exchanged out-of-band or by the higher layer application. The adding device d_0 outputs the authenticator auth to be sent to the server, while the added device outputs a bit b indicating whether it accepted the operation.

Example: Diana uses her phone to add her laptop as a second device. Her laptop has generated pk_2^D. Both devices jointly authenticate the keychain statement $(\mathsf{Add}, pk_1^D, pk_2^D, \perp)$. Diana's phone then sends $\mathsf{auth}_{\mathrm{Diana}}$ to the server.

▷ $\mathsf{auth} \leftarrow \mathsf{MVKD.RevokeKey}(st_0, pk_1)$: RevokeKey takes as input the device's state st_0, and the public key pk_1 of the device to revoke. It outputs the authenticator auth to be sent to the server. We note that devices are permitted to revoke themselves.
▷ $\mathsf{auth} \leftarrow \mathsf{MVKD.AddExtra}(st, d)$: AddExtra initiates a request to add an Extra keychain statement to the user's keychain. It takes as input the device's state st and some arbitrary data d which will be included in the statement. This algorithm outputs the statement authenticator auth to be sent to the server.

Example: Bob wants to include his bitcoin address `0xf3e...` in his keychain so that other users can send him money. He uses his phone to call algorithm AddExtra and generate $\mathsf{auth}_{\mathrm{Bob}}$ on the keychain statement $(\mathsf{Extra}, pk_1^B, \text{“}\texttt{BTC:0xf3e...}\text{”}, \perp)$.

▷ $(st_t^s, \mathsf{com}_t) \leftarrow \mathsf{MVKD.Update}(st_{t-1}^s, \mathsf{M})$: The server uses this algorithm to update its internal state. It takes as input the current state st_{t-1}^s for epoch $t-1$ and a set of updates $\mathsf{M} = \{(u_i, \mathsf{s}_i, \mathsf{auth}_i)\}_{i=1}^n$. In each triple, u_i represents a username, s_i represents the new keychain statement to be added to u_i's keychain, and auth_i is the keychain statement authenticator that u_i's device had output when authorizing this change. For simplicity, we assume that each user only submits one change request per epoch, ensuring that the ordering of keychain statements at the time of Update is obvious to the server. The server updates to the new epoch t and outputs its updated state st_t^s and the new commitment com_t.

Example: Alice, Bob, Charlie, and Diana have all submitted updates of their keychains to the server. It is the end of epoch 25 and the server is about to transition to epoch 26 with the respective update set M. The server updates its own internal state and computes a commitment over the new epoch's updated directory. The directory now looks as follows.

Key Directory:
(Alice; $(\mathsf{AddFirst}, pk_1^A, pk_1^A, 26)$)
(Bob; $(\mathsf{AddFirst}, pk_1^B, pk_1^B, 10)$, $(\mathsf{Add}, pk_1^B, pk_2^B, 15)$,
$(\mathsf{Extra}, pk_1^B, \text{“}\texttt{BTC:0xf3e...}\text{”}, 26)$)
(Charlie; $(\mathsf{AddFirst}, pk_1^C, pk_1^C, 7)$, $(\mathsf{AddFirst}, pk_2^C, pk_2^C, 26)$)
(Diana; $(\mathsf{AddFirst}, pk_1^D, pk_1^D, 3)$, $(\mathsf{Add}, pk_1^D, pk_2^D, 26)$)

▷ $(st_t^s, \mathsf{com}_t) \leftarrow \mathsf{MVKD.PCSUpdate}(st_{t-1}^s, \mathsf{M})$: This works analogously to MVKD.Update, except that it trades off efficiency for enhanced privacy guarantees (post-compromise security).
▷ $(st', \mathbb{S}\ ;\ \perp) \leftarrow \mathsf{MVKD.Query}(st, u\ ;\ st_t^s, u, t')$: Query is an interactive protocol between a user's device and the server. During a query, the device takes as input its state st and the username u for which they want to learn the keychain. The server takes as input its state st_t^s, the username u to look up, as well as the device's current epoch t'. The client outputs an updated keychain $\mathbb{S}$ for u (including all statements in the directory up to the epoch the client is in) and an updated state st' (which now includes the above $\mathbb{S}$). The server has no output. We assume that, before running Query, the higher level application on the server performs some sort of access control to ensure the client is allowed to learn this information.
▷ $(st', b\ ;\ \perp) \leftarrow \mathsf{MVKD.VerExtension}(st, \mathsf{com}_{t^{**}}\ ;\ st_t^s, t', t'')$: This algorithm is an interactive protocol between a user's device and the server. The device takes as input its state and a commitment $\mathsf{com}_{t^{**}}$ at epoch t^{**}, which it wants to verify is consistent with the commitment from its state for epoch t^*. That is, if $t^* < t^{**}$, then it verifies that the directory committed to by $\mathsf{com}_{t^{**}}$ extends the one in the device's current state, and vice versa if $t^* > t^{**}$. The client returns a bit b which indicates whether the verification succeeded. If $t^* < t^{**}$, then the client also updates the stored commitment in its state. The server takes as input its state st_t^s, which is at epoch $t \geq t', t''$, and the two epochs t', t'' that should match t^* and t^{**}, respectively. (The protocol may fail otherwise.)

Example: Alice's phone has stored the commitment com_{10} for epoch 10, which means it is out-of-date. Alice has recently learned the latest commitment com_{26} (we assume this happens through some out-of-band mechanism) and now wants to update her device to this commitment for epoch 26. Before Alice can do this, she must check that com_{26} is consistent with and extends com_{10}. Alice's phone thus initiates the interactive protocol VerExtension with the server, at epoch 26.
Alice's phone takes as input its own state st and com_{26}, while the server takes as input its state st_{26}^s and epochs 10 and 26. The server proves to Alice's device that her state is consistent with com_{26}; her device verifies this proof and outputs the result of this verification (e.g. $b = 1$). Since the proof verifies, Alice's device also updates its state to store com_{26}.

▷ $(b\ ;\ \perp) \leftarrow \mathsf{MVKD.Audit}(\mathsf{com}_{t^*}, \mathsf{com}_{t^*+1}\ ;\ st_t^s, t')$: This is an interactive protocol between the auditor and the server. The (stateless) auditor takes as input the two published commitments to the directory for consecutive epochs t^* and $t^* + 1$. It outputs a bit b indicating whether the audit is successful. The server takes as input its state st_t^s at epoch t and the epoch t'. The server has no output. We expect that $t^* = t'$ and $t' + 1 \leq t$ for the protocol to work.

▷ $\sigma \leftarrow \mathsf{MVKD.Sign}(st, m)$: A client device can produce a signature on any message m.
▷ $b \leftarrow \mathsf{MVKD.Verify}(pk, m, \sigma)$: Anyone can verify a signature on a message m with respect to pk.

The algorithms AddFirstKey, RevokeKey, AddExtra, Update, and PCSUpdate can return a special error symbol ERROR, instead of the above-described outputs, in case of erroneous inputs. For AddKey, the first device can output ERROR, while for Query only the device can.

Associated to a MVKD scheme is a set of functions. These functions assume w.l.o.g. that the honest participants' states have a representation that lets us identify certain values.

Definition 4. *For a* MVKD *scheme, we define the following functions:*

- $\mathbf{u}(st)$, $\mathbf{pk}(st)$, *and* $\mathbf{sk}(st)$ *map a device state to the associated username, public key, and secret key, respectively, as determined by the respective inputs and outputs to* DeviceSetup. *None must change over the run of the protocol.*
- $\mathbf{com}(st)$ *and* $\mathbf{com}(st^s)$ *map the device and server states, respectively, to the most recent commitment stored in the state.*
- $\mathbf{t}(\mathsf{com})$ *maps the commitment to its associated epoch number (or* ERROR *if the commitment is malformed). We further define* $\mathbf{t}(st)$ *and* $\mathbf{t}(st^s)$ *to be shorthands for* $\mathbf{t}(\mathbf{com}(st))$ *and* $\mathbf{t}(\mathbf{com}(st^s))$, *respectively.*
- $\mathsf{S}(st, u)$ *maps a device state and username to a pair* $(t^{\mathsf{last}}, \mathbb{S})$ *representing the epoch* t^{last} *the device last queried* u*'s keychain and the obtained keychain* $\mathbb{S}$. *For any users that have not been queried, the function returns* $t^{\mathsf{last}} = 0$, *with an empty keychain* $\mathbb{S} = (u; ())$. *If* u *had no keychain, it returns an empty keychain as well (along the proper* t^{last}*).*

We refer to Appendix C for some additional definitions with regard to keychains and MVKD schemes, which are used for the formal security notions.

Design choices. Our formalization makes some implicit design choices, both out of simplicity and in an attempt to capture real-world systems like Keybase. For one thing, our model prohibits re-adding revoked keys – we assume a setting where keys are not expensive to regenerate or disseminate (which is true in most E2EE systems). We also avoid the complications of reasoning about the trustworthiness of signatures made by keys that were revoked in the past but are now re-instated.

Furthermore, our design reveals which device authorized an action (such as adding or revoking a device), which might be sensitive to leak in some cases. While an alternative design that hides this information (such as employing group signatures) would allow for increased privacy, it would require a less standard primitive (compared to digital signatures); it would also make it harder to detect that a device that has since been revoked had made some specific change to a keychain, thereby enabling other users to learn the change might be less trustworthy.

Our model also enables account resets through our AddFirst algorithm, which enhances the functionality of MVKD in cases when users lose all their devices. When a user's account is reset, their contacts' clients will notice additional AddFirstKey statements in the user's keychain. As in Keybase, we recommend that any practical system deploying account resets only allow them after verifying the user's identity through additional access control mechanisms (outside the scope of this paper), such as checking passwords or verifying ownership of a phone number or email address. Moreover, KT client applications should notify users when they notice a reset (e.g. through an urgent warning), explaining that this might indicate an account compromise and suggesting that they might seek additional assurances of the user's identity. For example, the application could check with the user out-of-band that they intended to reset their account. If the possibility of malicious resets is a concern, however, users or platforms could simply disable resets. In particular, clients could reject keychains that have more than one AddFirstKey statement. Assuming the consistency of the key directory, this would disable resets in a way that could not be circumvented by the server.

Integrating with E2EE. While we do not explicitly model end-to-end encryption on top of MVKD, our formalization supports three ways of bootstrapping secure channels (or similar applications requiring authentication). One option is to use an AddExtra keychain statement to add either an externally generated encryption public key or an identity signing key to the user's keychain (in case the encrypted application requires specific key types or dedicated keys). The second is to use the Sign algorithm as described above to sign an encryption or identity signing key, analogous to a TLS certificate (which is helpful if the user does not want to publicly reveal their external key or that they are using a specific encryption app). The third option is to use the Sign algorithm directly for identity signing during the key exchange, which requires that the key exchange protocol uses the same signing algorithm as MVKD, but avoids having to introduce additional public keys. The best choice will be application-specific.

3 MVKD CONSTRUCTION

In this section, we describe the ELEKTRA protocol. The protocol is defined using a signature scheme SIG, a Rotatable Zero Knowledge Set RZKS, and a hash function hash.

Rotatable Zero Knowledge Set. Our protocol makes use of the RZKS primitive, introduced in [8], which implements a privacy-preserving ordered append-only dictionary storing (label, val) pairs. The server maintains the dictionary and publishes a commitment com_t after every update (epoch), producing a privacy-preserving proof of the append-only property for auditors to verify. There is a regular update and a PCS variant thereof. For a given label, the server can then produce a proof that either the respective value val is in the dictionary or that no pair for that label exists, at a certain epoch, which can be verified by a user having com_t. These proofs are privacy-preserving and only reveal a well-specified leakage about other labels or values to the user. Finally, RZKS offers extension proofs analogous to MVKD.VerExtension.

We refer to Appendix B for further details.

Construction Overview. At a high level, the protocol works using signed hash-chains which are analogous to Keybase sigchains. That is, whenever a device appends a statement $\mathsf{s} = (\mathsf{Type}, pk_a, pk, t)$

to the respective user u's keychain, they issue a signature authenticating the entire keychain. More formally, the statement authenticator $\mathsf{auth}_n = (h_n, z_n)$ for the n-th keychain statement consists of a pair of values: the n-th element of a hash-chain h_n that binds to the whole keychain prefix, and a signature z_n to authenticate this hash (0 is prepended for domain separation). This is computed as:

$$h_n \leftarrow \mathsf{hash}(\mathsf{auth}_{n-1}, t_{n-1}, u, \mathsf{Type}, pk_a, pk)$$
$$z_n \leftarrow \mathsf{SIG.Sign}(sk_a, 0 \| h_n).$$

For AddKey statements (where an existing device adds a new one) both of the involved devices sign the hash h_n, with z_n consisting of the pair of signatures. For AddExtra statements, d is used in lieu of pk. Note that t_{n-1} refers to the epoch of the prior keychain statement, which is not authenticated by z_{n-1}, given that it was set by the server at update time after the signature was generated. Finally, $\mathsf{auth}_0 \leftarrow \varepsilon$ and $t_0 \leftarrow 0$ are used to base the induction.

We refer to a *sigchain* as the augmented keychain that also includes auth for each entry. Whenever either a device or the server obtains new statements to add to a keychain — as part of MVKD.Update or MVKD.PCSUpdate in case of the server or MVKD.Query in case of the client, respectively — they also obtain the corresponding authenticators, and verify both that the resulting keychain is valid and that the authenticators match (by recomputing the hash-chain and verifying the signatures).

To prevent the server from handing out stale or inconsistent keychains to clients, the protocol directly leverages the RZKS: when the server updates u's keychain $\mathbb{S}$ with a new statement, it adds an entry with the respective authenticator (received from the client) to the RZKS under label = $(u, |\mathbb{S}| + 1)$. This can either be done using the regular RZKS.Update or, for healing from a compromise, the RZKS.PCSUpdate algorithm. The MVKD commitments just correspond to the RZKS commitments, allowing clients to check the inclusion proofs when querying for keychains.

Protocol algorithms. We provide a high-level description of the ELEKTRA protocol algorithms here. Please see Appendix D for the detailed pseudo-code.

Each client's state consists of the username of the intended user, a pair of signature keys, the latest commitment to the RZKS the client learned about, a dictionary Users that maps each username to a keychain for that user, the authenticator of its last statement, and the last epoch the client queried about that user's chain. The server state includes the server's RZKS state, a dictionary mapping each epoch to the corresponding commitment and update proof, and a dictionary Users^S mapping each user to their keychain.

- $\mathsf{pp} \leftarrow \mathsf{MVKD.GenPP}(1^\lambda)$: Generates public parameters for the underlying RZKS and outputs them as parameters for the MVKD.
- $(st_0^s, \mathsf{com}_0) \leftarrow \mathsf{MVKD.ServerInit}(\mathsf{pp})$: The server initializes a new RZKS and empty directory Users^S, and stores the RZKS initial commitment in its state.
- $(st, pk) \leftarrow \mathsf{MVKD.DeviceSetup}(\mathsf{pp}, u)$: When a new device is initialized, it first generates a new signing key pair. The device stores its username and key pair in its (fresh) state. Since the device has not yet communicated with the server, it leaves the commitment, epoch, and Users dictionary empty.
- $\mathsf{auth} \leftarrow \mathsf{MVKD.AddFirstKey}(st)$: Let (sk, pk) be the signing key pair, $\mathbb{S}$ be the keychain for this client's username, and auth_i and t_i be the authenticator and the epoch for the last statement in $\mathbb{S}$, as stored in the client's state st. The algorithm checks that adding a new statement $\mathsf{s} = (\mathsf{AddFirst}, pk, pk, \perp)$ to $\mathbb{S}$ yields a valid keychain, i.e., that the key has not been added to the user's keychain yet; otherwise it aborts. It then computes and outputs the authenticator auth_{i+1} for s, by computing the hash as described above, and then signing over this hash.
- MVKD.RevokeKey(st, pk) and MVKD.AddExtra(st, d) both work analogously to the MVKD.AddFirstKey, but using a Revoke or Extra statement respectively.
- $(\mathsf{auth}\ ;\ b) \leftarrow \mathsf{MVKD.AddKey}(st_0, pk_1\ ;\ st_1, pk_0)$: This interactive protocol also works similarly to the previous ones. But, instead of outputting auth_{i+1}, the added device sends it to the adding one, and outputs a bit $b = 1$ (if adding pk_1 would yield an invalid keychain the party aborts with $b = 0$). The adding party receives the authenticator (h', z') from the added party, verifies the signature z' received from the added party, and that the hash h' matches the h they computed from their own view of the keychain. Then, the adding party computes their own signature z over h and outputs $(h, (z, z'))$ as the combined authenticator.
- $(st_t^s, \mathsf{com}_t) \leftarrow \mathsf{MVKD.Update}(st_{t-1}^s, \mathsf{M})$: The update set M contains tuples of the form $(u, \mathsf{s}, \mathsf{auth})$, where auth is expected to be an authenticator over statement s to be added to user u's keychain. The server aborts if either M contains two updates for the same user, if for any tuple adding s to u's chain (as stored by the server) would lead to an invalid chain, or if the authenticator does not verify (hash mismatch or signature verification failure).[5] Otherwise, for each tuple, the server creates a new label-value pair to be added to the RZKS. The label is label = $(u, i{+}1)$ and the value is val = auth, where i is the length of u's current keychain. The server then calls RZKS.Update to add all the pairs, getting back a new RZKS state, commitment com, and update proof π, all of which the server stores in its state. It further updates its own keychain directory, appending the keychain statements but filled in with the new epoch. It outputs its updated state and com.
- $(st_t^s, \mathsf{com}_t) \leftarrow \mathsf{MVKD.PCSUpdate}(st_{t-1}^s, \mathsf{M})$: This works analogous to MVKD.Update, except for using RZKS.PCSUpdate instead of RZKS.Update to add the authenticators to the RZKS.
- $(st', b\ ;\ \perp) \leftarrow \mathsf{MVKD.VerExtension}(st, \mathsf{com}'\ ;\ st_{t_n}^s, t', t'')$: In this interactive protocol, the client holds commitment com in its state and wants to check whether the commitment com' given as input is consistent with it. The server gets as input two epochs t', t'', calls the RZKS algorithm ProveExt with these epochs (ordered increasingly) to generate the extension proof and sends this alongside $\mathsf{com}_{t''}$ to the client. In the special case of $t' = \perp$ (i.e., if the client does not have any commitment yet), it sends an empty proof. The client ensures that the received commitment equals its input com′,[6] and that RZKS.VerExt on input com, com′, and the received proof succeeds. If the checks succeed and the commitment in the client's state is less recent, the client updates it

[5] We let the Update algorithm abort to simplify the definitions. In practice, the server could accept just the valid updates, and only the first update request per chain. The client submitting the simultaneous update could then be informed to wait until the other update is completed, update their copy of the keychain, and retry the update.

[6] When interacting with an honest server, an honest client shall accept iff the commitments match. RZKS completeness guarantees that the client will accept if commitments match, but RZKS soundness does not imply the other direction.

with com′. The client outputs its state, and whether the checks have been successful.

- $(st', \mathbb{S}\ ;\ \perp) \leftarrow \mathsf{MVKD.Query}(st, u\ ;\ st^s_t, u, t')$: To query the keychain $\mathbb{S}_u$ of user u at epoch t', the client sends t^{last} to the server, where t^{last} denotes the epoch it last queried for the user u. The server verifies that $t^{\mathsf{last}} < t'$ and t' is at most equal to the current epoch and returns an error otherwise. Then, for each keychain statement added to u's keychain between t^{last} and t', the server sends a triple $(\mathsf{s}_i, \mathsf{auth}_i, \pi_i)$ consisting of the statement, its authenticator, and an RZKS proof that the pair label = (u, i) and val = auth_i is/has been in the RZKS for epoch t'. Here, i denotes the index of the keychain statement. Finally, the server sends an RZKS non-inclusion proof for label = $(u, |\mathbb{S}_u| + 1)$, showing that no further statement belongs to the keychain $\mathbb{S}_u$ at t'. The client verifies that the keychain obtained by appending those statements to the one in its own state is valid, that the authenticators are valid, and that all the received RZKS proofs verify with respect to the client's current commitment. If all checks succeed, the client returns the amended keychain and stores the keychain, authenticator, and current epoch in Users. Otherwise, the client returns an error.
- $(b\ ;\ \perp) \leftarrow \mathsf{MVKD.Audit}(\mathsf{com}_t, \mathsf{com}_{t+1}\ ;\ st^s_{t_n}, t')$: The server simply sends to the auditor the RZKS update proof it stored in its state when updating from epoch t' to $t' + 1$. The auditor checks the received proof against the two input commitments using RZKS.VerifyUpd and outputs the result.
- $\sigma \leftarrow \mathsf{MVKD.Sign}(st, m)$: The client calls $\mathsf{SIG.Sign}(\mathbf{sk}(st), 1\|m)$, where it prepends 1 to m before signing for domain separation from signing sigchain statements.
- $b \leftarrow \mathsf{MVKD.Verify}(pk, m, \sigma)$: The client calls $\mathsf{SIG.Ver}(pk, 1\|m, \sigma)$ and returns the resulting bit.

3.1 Complexity

The complexity of our protocol is overall dominated by its RZKS operations. Below, we detail the concrete complexity obtained when leveraging the RZKS construction from [8], ignoring the linear dependency on the security parameter.

The RZKS from [8] is constructed using a Rotatable VRF (RVRF), a hash-based commitment, an ordered accumulator (OA), and an append-only vector commitment (AVC). Roughly speaking, the OA and AVC are two authenticated data structures that implement a dictionary and a vector respectively, and can both be instantiated based on Merkle trees. To store a label/value pair in the RZKS, one first "obfuscates" the label using the RVRF (for privacy reasons), then stores the obfuscated label and a commitment to the value in the OA dictionary. The AVC maintains an ordered list of the OA commitments for all epochs. In particular, the PCSUpdate operation involves picking a new RVRF key and recomputing a new OA where the commitments to values remain the same, but obfuscated labels are computed using the new VRF. In addition, the server computes a RVRF rotation proof which guarantees that this operation was performed honestly (without revealing any labels), and finally adds any additional pairs to the new tree.

ELEKTRA operations related to initialization and clients modifying their keychain take time $O(1)$, i.e. up to linear in the security parameter, while Sign and Verify take time linear in the size of the message. The complexity of the other operations is described below.

Query. In a directory with n keychain links and e epochs, a Query for a keychain of length m takes $O(m\log(n) + \log(e))$ for both the client and the server. Note that a single MVKD query results in $m + 1$ RZKS.Query and RZKS.Verify operations, with each such operation taking $O(\log(n) + \log(e))$, as reported in [8]. Therefore, when performing this operation naively, its complexity would be $O(m\log(n) + m\log(e))$. However, looking at the specific RZKS instantiation from [8], it is easy to see how to optimize this operation to replace the $m\log(e)$ term with $\log(e)$. In particular, an RZKS query proof consists of three components: an $O(1)$ RVRF proof, an $O(\log(n))$ OA query proof, and an $O(\log(e))$ AVC query proof. To understand why the OA and AVC both have logarithmic proof costs, recall that both are constructed using Merkle trees. However, when querying multiple labels with respect to the same RZKS root, the AVC proof is the same and therefore only needs to be transmitted and checked once.

Updates and auditing. An MVKD.Update adding s new statements takes $O(s\log(n+s)+\log(e))$, while MVKD.PCSUpdate takes $O(n+s+\log(e))$, again due to the calls to the corresponding RZKS algorithms. MVKD.Audit has the same complexity as the operation that generated the relevant proof, either Update or PCSUpdate. Finally, the complexity of MVKD.VerExtension is $O(\log(e))$ for a directory with e epochs.

4 MVKD SECURITY

We now describe the security and completeness properties we require from a MVKD.

4.1 Completeness

Our completeness notion, at a high level, prescribes the desired functionality for honest parties (user devices and auditors) interacting with an honest server, keeping track of com_t for each epoch t as well as the current keychain $\mathsf{Dir}[u]$ for each user u. It also allows other dishonest parties to interact with the server, to ensure this cannot affect the interaction and execution between honest parties. We provide the formal experiment for completeness in Appendix E.

The adversary $\mathcal{A}$ is provided access to oracles creating honest users and having them modify their keychains as well as oracles to instruct the server to apply those updates either using Update or PCSUpdate. The adversary wins the game by violating one of the properties described below. We say that a scheme MVKD satisfies completeness if every PPT adversary $\mathcal{A}$ has negligible probability of winning the experiment.

Extending the directory. When the adversary instructs the server to extend the directory, the adversary gets to choose the set of updates and, thus, can include maliciously generated updates (to account for malicious devices). Correctness ensures that the respective algorithm only rejects if either (a) there is at least one update that has not been honestly generated by a device that was up-to-date with respect to the corresponding user's own keychain, or (b) there exists a collision in updates for a user, meaning there is more than one update for a user's keychain in the update set.

Querying the directory. When a device queries another user's keychain, the game ensures that the keychain output by the device matches the respective prefix of the keychain stored by the server (up to the epoch in which the querying device is currently in). The operation must only abort if there has been a mismatch between the device's and the server's views, i.e., either the input usernames do not match, the epoch input to the server does not correspond to the client's epoch, or the device uses a commitment that has never been output by the server.

Updating the commitment. When a device updates to a later commitment using VerExtension, the game verifies that the commitment stored in the device's state is updated as expected, and that the algorithm only rejects if the device's and server's views or inputs do not match.

Audits. The game verifies that external auditors can successfully audit two consecutive commitments produced by the honest server.

Signatures. The game ensures that the produced signatures are correct. That is, it verifies that any signature produced by Sign actually verifies under the device's public key.

4.2 Soundness

A MVKD protects against an active and fully compromised server who can lie about users' keychains and can also dictate which commitment users will accept. It enforces the unforgeability of keychain updates, that users holding the same commitment have the same view of the directory, and provides additional assurances when directory updates are audited. Auditing could be performed by dedicated third parties, or even by users themselves (although we do not expect this to happen in practice).

We formalize soundness as a real-ideal world indistinguishability game. The ideal experiment involves a stateful extractor expected to provide different functionality depending on its first input:

- $\mathsf{pp}, \mathsf{st} \leftarrow \mathsf{Extract}(\texttt{Init}, 1^\lambda)$: Samples public parameters (meant to be indistinguishable from honestly generated ones), and initializes its state.
- $out, \mathsf{st} \leftarrow \mathsf{Extract}(\texttt{Ideal}, \mathsf{st}, in)$: Implements any ideal functionalities (such as Random Oracles) that the scheme depends on, and can also update its own state when answering these queries.
- $\mathsf{D}_{\mathsf{com}}, \mathsf{C}_{\mathsf{com}} \leftarrow \mathsf{Extract}(\texttt{Extr}, \mathsf{st}, \mathsf{com})$: On input a commitment com and its own state st, the extractor outputs a map $\mathsf{D}_{\mathsf{com}}$ and a list $\mathsf{C}_{\mathsf{com}}$. $\mathsf{D}_{\mathsf{com}}$ maps a username-integer pair (u, e) to a keychain for u with $e > 0$ statements. Intuitively, this map restricts what an adversary can force an honest client who holds com, for epoch t, to output when querying for u, as we detail later. $\mathsf{C}_{\mathsf{com}}$ is a list of t commitments, meant to correspond to previous epochs of the data structure that com commits to. Note that, when answering these queries, the extractor cannot update its state. If com is malformed, the extractor can output ERROR.

The soundness game provides oracles that allow $\mathcal{A}$ to interact with honest clients, either in the role of the server or another client, or to instruct them to run MVKD algorithms and obtain their outputs. However, before the adversary can ask any client to update its commitment to com, the adversary has to "announce" it by invoking the Commit oracle. This ensures that the extractor is executed before clients can perform any queries and therefore that we can test whether a client output matches the extractor output. When the game itself executes one of the MVKD algorithms, that algorithm can also (implicitly) make Ideal oracle queries (say, to evaluate a Random Oracle hash). Guarantees are then phrased as assertions in the ideal world experiment. For example, the query oracle enforces that the output of a query from a client who holds commitment com must be consistent with the output of the extractor on that same commitment.

The real experiment is defined analogously to the ideal one, except that all assertions are removed (hence, triggering one in the ideal world makes the two worlds immediately distinguishable) and there is no extractor. Instead, the public parameters pp are sampled honestly at the beginning, and the Ideal oracle is implemented according to the specification of each idealized functionality. Note that formalizing the soundness game using indistinguishability ensures that the extractor must sample the public parameters and implement idealized functionalities in a way that is computationally indistinguishable from the real world.

We now proceed to describe the soundness guarantees. We refer to Appendix G for a formal definition of the soundness experiment.

Resiliency. Intuitively, interacting with an adversary (either posing as another client or the server) should not alter an honest client's state in unexpected ways. For example, the adversary should not be able to eradicate a client's ability to sign messages, confuse them about their own username, or induce them to output an invalid/malformed keychain.

Unforgeability. An adversary should not be able to forge keychains or sign messages on behalf of uncorrupted devices. We capture unforgeability via (1) a Forgery oracle, which specifically enforces the unforgeability of signatures, and (2) the HonestKeychain assertions run upon the adversary announcing a new commitment com. The latter enforces that $\mathsf{D}_{\mathsf{com}}$ output by the extractor does not contain forgeries and formalizes the following intuitive property. Assume $\mathcal{A}$ publishes a commitment com today, expecting to corrupt a device pk tomorrow. Even after the corruption, $\mathcal{A}$ cannot convince an honest device holding com to output a "forged" keychain statement allegedly from pk but that was not actually authorized before its corruption. To this end, the game keeps track of all keychains whose last statement has been authorized by the appropriate honest device as well as all corrupted devices.

Consistency. At a high level, consistency ensures that when two clients share the same MVKD commitment com and query for the same user, they should output the same keychain. Ideally, this would be enforced by requiring that each client outputs $\mathsf{D}_{\mathsf{com}}[(u, i)]$ as the keychain $\mathbb{S}'$ for u where i denotes the maximal number of statements for which $\mathsf{D}_{\mathsf{com}}[(u, \cdot)]$ is defined. However, our formalization allows for constructions to leverage an efficiency optimization that has clients cache keychains and only query for additional links. As a result, we achieve a slightly weaker version. Upon a client outputting $\mathbb{S}'$ as the keychain for u and having prior output $\mathbb{S}$ from an earlier commitment, the game enforces the following properties. First, for each new statement added to $\mathbb{S}$ to obtain $\mathbb{S}'$, the prefix of the keychain until that statement must match the output of the extractor for that length, i.e., that for $i = |\mathbb{S}| + 1, \ldots, |\mathbb{S}'|$ we have

$\mathbb{S}'[1 \ldots i] = \mathsf{D}_{\mathsf{com}}[(u, i)]$. Second, the client not receiving a statement $|\mathbb{S}'| + 1$ must be consistent with the output of the extractor, i.e., $\mathsf{D}_{\mathsf{com}}[(u, |\mathbb{S}'| + 1)]$ is not defined.

This relaxation intentionally allows for the following attack on our protocol (when no auditing is performed): The adversary publishes a keychain for user u with 2 statements at epoch t, then extends it with a 3rd statement at epoch $t + 1$, while also replacing the first statement with something malicious and erasing the 2nd statement from the RZKS. Consequently, a client who queries for u at epoch t and then at epoch $t + 1$ would output the chain with all 3 honest statements. However, a different client who queries for u at epoch $t + 1$ for the first time (with the same commitment) would output a keychain with only the first statement, since, in our protocol, the server can provide a non-membership proof for the second statement, indicating in our protocol that the keychain only has a single statement.

We remark that a trivial modification of our protocol without caching (where all clients always query for all statements) would satisfy the stronger notion and prevent the above attack. Moreover, the two definitions offer the same guarantees if we assume all commitments have been audited — more precisely, if we assume that for each commitment com held by a client, there is a chain of successful audits starting at a commitment for epoch 0 and ending at com. For example, the above attack requires violating the append-only property of RZKS, which would be caught by auditing. This relaxation allows us to capture the weaker consistency exhibited by more efficient protocols in situations when auditing is not performed, but also enforces stronger consistency when auditing occurs.

As in any KT protocol, our notion requires users and auditors to agree on the server's published commitments to achieve the strongest guarantees. Several approaches have been proposed for achieving this, including running a gossip protocol between clients and auditors, leveraging fully trusted auditors who will host commitments, or posting the commitments on a blockchain. As a result, our formalization is agnostic to the specific consensus mechanism — with the consistency guarantees only meaningful for parties that do have consensus. To make the aforementioned consensus requirements more practical, users run the (lightweight) VerExtension algorithm whenever they learn of a new commitment com′ from the server. Our formalization ensures consistency on the history of commitments upon VerExtension. This also ensures that at any point it is sufficient for parties to establish consensus, with the relevant users and auditors, on the most recent commitment only. In particular, as long as a user is satisfied with a-posteriori detection of inconsistencies, they only need to establish consensus sporadically.

Finally, note that — unlike many other KT systems that do not leverage keychains — our protocol provides some baseline security properties (a stronger form of what is commonly referred to as *Trust-On-First-Use*) even when there is no consensus. The game ensures that if a device queries for user u's keychain, the output keychain $\mathbb{S}'$ extends the keychain $\mathbb{S}$ previously stored as part of the client's state. Combined with the aforementioned unforgeability guarantees, this implies that except if the keychain is reset by an AddFirst statement (something that can be communicated to the user with a very prominent warning, since it is rare), changes to the set of keys in $\mathbb{S}'$ must be authorized by the devices whose keys are trusted in $\mathbb{S}$, and the adversary cannot forge those authorizations. In other words, assuming that users are distrustful of reset accounts, compromising the server (but not the user's devices) does not allow an adversary to tamper with keychains of users who have already interacted with each other at least once.

Persistency. If MVKD.Audit between two commitments com_a and com_b succeeds, then persistency ensures that all links in com_a are also part of com_b, and all newly added links in com_b contain the correct epoch. This, in particular, also implies strong consistency: in the consistency attack above, we would have that $\mathsf{D}_{\mathsf{com}_t}[(u, i)] = \mathsf{D}_{\mathsf{com}_{t+1}}[(u, i)]$ for $i = 1, 2$, and therefore the two devices would have to output the same keychain. Note that auditing itself only ensures the persistency of changes, i.e., the append-only property of the directory. While other users will detect blatantly unauthorized changes to a user's keychain upon query, users are still required to monitor their own keychain for account resets or unauthorized changes enabled by one of their devices being compromised.

4.3 Privacy

We capture the privacy of MVKD again using the real-ideal world indistinguishability paradigm. Ideally, in MVKD, the commitments and proofs from the server and interaction with the server should leak no extra information about the server's state (which includes the key directory). In other words, the proofs for the queries and the transcripts should be simulatable given the responses to the queries. However, an *efficient instantiation* of MVKD may leak some minimal information. To capture such schemes, our privacy definition is parameterized with leakage functions. Overall, the high-level goal of this definition is to express what information gets leaked from the server to malicious clients when they interact with the server, also accounting for a full server state leak.

More formally, we say that a MVKD scheme is *Zero Knowledge* with respect to a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{AddKey0}}, \mathcal{L}_{\mathsf{AddKey1}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}}, \mathcal{L}_{\mathsf{PCSUpdate}}, \mathcal{L}_{\mathsf{VerExt}}, \mathcal{L}_{\mathsf{Audit}}, \mathcal{L}_{\mathsf{Corr}}, \mathcal{L}_{\mathsf{LeakState}})$ if there exists a simulator $\mathcal{S}$ such that any PPT $\mathcal{A}$ has negligible advantage in distinguishing between a real-world experiment $\mathsf{ZK\text{-}REAL}_{\mathsf{MVKD}}$, in which $\mathcal{A}$ receives the outputs and transcripts of the actual MVKD algorithms, from an ideal-world experiment $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$, in which $\mathcal{A}$ receives the outputs and transcripts from $\mathcal{S}$. The simulator $\mathcal{S}$ is given the output of the leakage function $\mathcal{L}$ and the output of the queries. We remark that $\mathcal{L}$ is stateful with all functions operating on a shared state.

This definition is similar in spirit to the VKD privacy definition in [7]. However, there are some significant new challenges here, due to clients being stateful and some of the algorithms in MVKD being interactive. Recall that clients are not trusted for privacy. Thus, we need to be able to capture what a client state can leak about other users' keychains, especially after they get compromised. Further, we consider how privacy recovers after a compromise of the server's state. Our definition captures these subtleties as follows.

Handling interaction with corrupt clients. We want to provide privacy for honest client states even in the presence of malicious clients. Since malicious clients can interact with the (honest) server as well as with other honest clients through MVKD algorithms (VerExtension, Audit, Query, and AddKey), we need to model any possible leakage that can happen through these interactions. So, in

the privacy definition, the distinguishing adversary $\mathcal{A}$ can instruct the game to create honest devices belonging to arbitrary users, modify honest device keychains, or run queries. $\mathcal{A}$ can act as one of the two devices of the AddKey protocol. It can also directly interact with the server using VerExtension, Audit, and Query.

However, in MVKD, the public keys of the honest devices should not be visible to $\mathcal{A}$ (even if they instructed the game to create them) unless they specifically queried for the keychain of the corresponding user, or interacted with that honest device using the AddKey oracle. Hence, whenever $\mathcal{A}$ requests an honest device to be created for a user u, a handle h is drawn from the handle set $\mathcal{H}$, with the device being addressed as (u, h).

Yet another issue arises whenever $\mathcal{A}$ instructs an honest device to modify its keychain: in the real-world experiment, this action succeeds or fails depending on whether the action is valid given the device's view on its keychain. The ideal world needs to make the analogous decision without involving the simulator (since this is an interaction between the honest server and the honest client). This, however, poses a challenge. To see why, let the (symbolic) keychain of user u be $(u; (\mathsf{AddFirstKey}, h, h, t), (\mathsf{AddKey}, h, h', t'))$. We then may ask: is instructing device (u, h) to revoke public key pk a valid action? A moment of reflection reveals this is valid if and only if pk is the public key of either (u, h) or (u, h'). (Observe that the adversary could have learned a device's public key via Query from the server.) To allow the ideal-world experiment to make this decision, we thus have the simulator $\mathcal{S}$ maintain a mapping PK from username-handle pairs to public keys, for the devices it is aware of. This correctly models real-world behavior since any handles unknown to the simulator will have public keys that are also unpredictable to the adversary.

Adaptive corruption of client devices. In our definition, we also capture adaptive corruptions of client devices. This means a device can start out as an honest device, and later $\mathcal{A}$ can corrupt it and get its state using the CorrDev oracle. When a device is corrupted, the simulator $\mathcal{S}$ will need to simulate its correct state, which in practice has information on the queries the device had made while it was honest and their results. Likewise, for VerExtension. To capture that, we give $\mathcal{A}$ oracles HonQuery and HonVerExt. This essentially lets $\mathcal{A}$ instruct honest devices to run Query and VerExtension.

Updates produced by corrupt clients. Additionally, we guarantee that any server update does not leak information on the honest device states, but no privacy is guaranteed for malicious device states and their respective updates. In our game, we model this as follows. $\mathcal{A}$ provides two sets of keychain statements to be appended: M_{HonDev} contains honestly generated statements using the various oracles (for which the adversary does not know auth, and refers to these updates using the corresponding device handles), while M_{CorrDev} contains keychain statements alongside their authenticators auth generated by the adversary.

Post-compromise security. Finally, a major improvement of our privacy definition over that of VKD is that we consider leakage of the server's state. While this is expected to leak the entire directory, we do guarantee that PCSUpdate restores privacy for newly made changes to the directory from that point on. In our game, we model this via the shared state of the various leakage functions. Upon state corruption, in the ideal world, $\mathcal{L}_{\mathtt{LeakState}}$ not only can leak the entire directory, but can modify the shared state, thus adversely affecting the leakage provided by other leakage functions for their respective operations. Conversely, upon $\mathcal{A}$ invoking PCSUpdate in the real world, in the ideal world, $\mathcal{L}_{\mathtt{PCSUpdate}}$ can modify the leakage functions' shared state to restore the privacy guarantees.

We refer to Appendix I for a formal description of the zero-knowledge with leakage experiment.

4.4 Security Analysis of Our Construction

In the following theorems, we demonstrate that our construction satisfies completeness and soundness. The proofs are deferred to Appendices F and H, respectively.

Theorem 1. *The protocol from Section 3 satisfies completeness, if the underlying* RZKS *scheme satisfies completeness and the signature scheme satisfies correctness.*

Theorem 2. *Let* RZKS *be a Rotatable Zero Knowledge Set satisfying soundness,* SIG *be a strongly unforgeable signature scheme, and* hash *be modeled as a Random Oracle. Then the* MVKD *construction of Section 3 satisfies soundness, w.r.t.* RZKS*'s ideal objects and* hash.

We prove privacy assuming the RZKS is instantiated using the specific construction from [8], which allows us to prove our result with respect to the following concrete leakage function $\mathcal{L}_{\mathsf{MVKD}}$:

- AddKey between an honest and a malicious device leaks the honest device's view of their own keychain.
- Query leaks the answer, which is either the keychain of the specified user u at epoch t or $\perp$. If t is not the latest epoch, Query moreover leaks the epoch t' of the first statement on u's keychain with $t' > t$, or $\perp$ if no such statement exists.
- Update, unless there has been a compromise of the server's state, leaks (1) the number of honestly generated updates applied, (2) the set of users for which there is an honest update that is the first one since the user's keychain has been last leaked to the adversary, and (3) for each maliciously generated statement included in the update, whether the adversary knew that user's most recent keychain. If the server's state has been compromised, without PCSUpdate having been executed in the meantime, then it leaks the entire set of updates.
- PCSUpdate leaks parts (1) and (3) of the leakage of Update. Further, the case of a state compromise never applies.
- VerExtension leaks the two epochs for which the server computes the extension proof, i.e., it leaks the inputs to the server algorithm.
- Audit leaks which epoch numbers are being audited.
- Corrupting a device leaks: (1) the keychains for the latest time when any usernames were queried by that device as well as the epochs during which they were queried, and (2) the latest commitment known by that device.
- Compromising the server's state leaks the directory.

A more formal description of the above leakage, as well as a proof of the following theorem, are deferred to Appendices J and K.

Theorem 3. *Let* RZKS *be the Rotatable Zero Knowledge set construction from [8],* SIG *be a strongly unforgeable signature scheme,*

and hash *be modeled as a Random Oracle. Then, the* MVKD *construction of Section 3 satisfies zero-knowledge with leakage* $\mathcal{L}_{\text{MVKD}}$ *w.r.t.* RZKS*'s ideal objects and* hash.

5 IMPLEMENTATION AND PERFORMANCE

Here we describe our implementation and experimental results for ELEKTRA.[7] The performance of both updates and queries is critical for practitioners, especially as we expect these to be blocking operations in clients. For a practical example, we look at Keybase, the only currently deployed KT system. When Keybase users update their keychain, the app blocks until the server receives the update, updates the KT directory, signs the new commitment, and proves that the new commitment includes the client's update. Similarly, users do not consider another user's device valid until it is reflected in the KT directory. We conjecture that these operations (except for PCSUpdate, which is rare) would need to run in under a second in order for real-world systems to use an MVKD.

To implement our scheme, we instantiated the RZKS construction from [8] which, as detailed in Section 3.1, leverages a RVRF, commitment, OA, and AVC. We used the NIST P-256 elliptic curve for the RVRF, SHA256 with different context strings for all hash functions, and Ed25519 for the signature scheme SIG. All experiment results are averaged over 10 trials.

Our implementation is written in Go and relies on PostgreSQL and LevelDB for persistence. LevelDB is used to store tree nodes as it supports high write throughput and fast range queries. All other data such as keychain links, VRF rotation proofs, and tree roots are stored in PostgreSQL. We implemented the following optimizations:

- If more than 100 links are built in a single epoch, the VRF image of each label is computed concurrently.
- When building a new epoch, the server verifies all links concurrently.
- The first 29 levels of the Merkle tree are cached in-memory to improve read performance.
- The VRF proof and image corresponding to each label is precomputed and stored in the database at the time the label is inserted during MVKD.Update or MVKD.PCSUpdate, in order to speed up MVKD.Query and computing VRF rotation proofs during MVKD.PCSUpdate.
- The server stores keychain states for each user in the database so when new links are added, it does not have to re-validate all the previous links.
- The server offers an API to perform MVKD.VerExtension as well as MVKD.Query for multiple users at once, so network latency is amortized.
- The exponentiations necessary to compute and verify VRF rotation proofs are performed concurrently.
- In MVKD.Query, clients verify sigchains for each queried user concurrently.

There are several other promising optimizations that we did not implement, given the proof-of-concept nature of our prototype. In particular, these optimizations include:

- **Map-reduce**: During a MVKD.Update operation, the tree could be sharded into (for example) 16 disjoint subtrees, with each subtree root being recomputed in parallel and possibly in a distributed way; after, the 16 subroots could be combined by a single process to compute the final tree root.
- **Parallelization on the server**: Any operation that is being parallelized could be improved using SIMD intrinsics and/or distributed computing.
- **Replication**: All data in both PostgreSQL and LevelDB is append-only, so it is straightforward to use distributed read replicas to reduce query latency globally.
- **Sharding**: Sharding tree node storage could increase performance as each individual database instance is smaller, but would impose an increased coordination cost.
- **Message compression**: Many Merkle tree proofs requested by the client for the same epoch would share the same hashes of nodes closer to the root. Both general-purpose compression or more ad-hoc techniques could eliminate this redundancy (for example, by having clients cache the levels of the tree close to the root), resulting in bandwidth savings.

In our experiments, the server runs on an AWS EC2 `c5.4xlarge` instance with 16 3.00 GHz Intel Xeon Platinum CPU cores and 32 GB memory, and the client on a Google Pixel 6 (2021) Android smartphone with 8 CPU cores (on average 2 GHz each) and 8 GB memory connected over Wi-Fi.

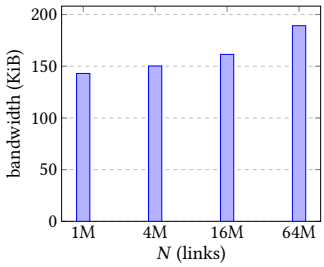
We experiment with directories consisting of up to 64M keychain links. This could represent, for example, a system with 16M users adding 4 links per year after one year. We believe that with additional optimizations as listed above, our implementation will easily scale to large real-world deployments.

Furthermore, for our experiments, we chose an epoch length of 1 second per epoch. We choose this parameter to capture a system larger than Keybase, which seems to generate new epochs as updates are necessary rather than on a fixed schedule. More concretely, Keybase begins a new epoch on average about every 15 seconds. The benefit of a smaller epoch length as we choose is that users wait less time to see their updates reflected in the system.

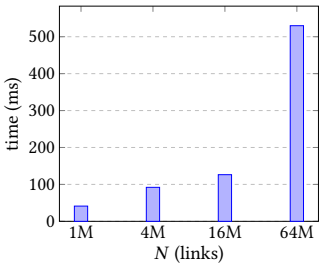
Performance of Queries. In a typical messaging application with MVKD, query performance significantly impacts actions such as starting a conversation with users one has not interacted with before, as it involves receiving and verifying proofs for their full keychain. Subsequent updates only require fetching any new links and inclusion proofs (as well as an absence proof at the end). A user first has to update their view of the MVKD using MVKD.VerExtension; then, the user can perform MVKD.Query for their conversation partners. Our implementation supports a single round-trip both for the Query and VerExtension algorithms, which reduces latency.

In this experiment, we simulate the MVKD queries necessary to join a small group conversation with 10 unknown users, each with 10 keychain links (100 links in total). The client (a Google Pixel 6 smartphone) is 300 epochs behind the server; assuming one epoch per second, it is 5 minutes behind. Figures 1a and 1d describe the performance and bandwidth of MVKD.VerExtension and MVKD.Query as the MVKD size grows. The entire operation of requesting, serving, receiving, and verifying can be performed in

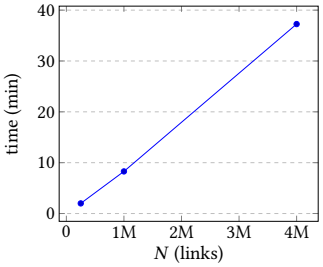
[7] The source code is available at https://github.com/zoom/elektra.



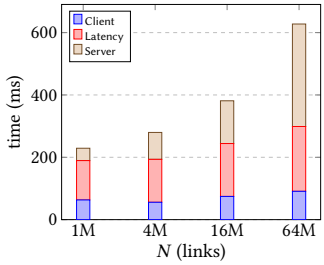
(a) MVKD.VerExtension **and** MVKD.Query **proof size by directory size** N.



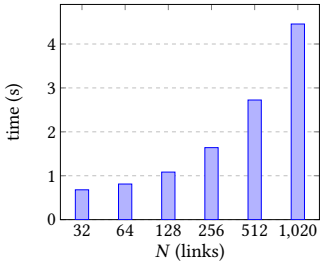
(b) MVKD.Update **runtime with 10 links by directory size** N **(measured in number of links).**



(c) MVKD.PCSUpdate **runtime by directory size** N.



(d) MVKD.VerExtension **and** MVKD.Query **runtime by directory size** N.



(e) MVKD.Update **runtime with** N **links in a directory of size 64M.**

Directory size	VRF time (min)	Build time (min)
250K	0.32	1.69
1M	1.31	6.97
4M	5.16	31.28

(f) Breakdown of MVKD.PCSUpdate **runtime.**

Figure 1: Performance for querying and updating our MVKD **construction ELEKTRA.**

under a second at minimal bandwidth cost by a smartphone even in a large directory containing 64M keychain links.

We divide the elapsed duration into three parts. Latency is the network latency for sending the request and receiving the response. Server is the time for the server to load keychains from the database and compute proofs[8]; it loads each requested user's data concurrently. Client is the time for the client to process the response from the server, verifying RZKS inclusion and extension proofs as well as checking the validity of the received keychains and their signatures; the client validates each sigchain concurrently.

Note that the bandwidth and computation costs of VerExtension are negligible with respect to the cost of Query operations; moreover, they grow logarithmically in the number of epochs between the client's view and the server's view. So, a larger period between calls to VerExtension would not significantly affect our results.

Performance of Updates. The efficiency of MVKD.Update operations is critical for users to quickly make changes to their keychain. It is a blocking step when installing the MVKD-using application on a new device: the user cannot communicate until their new device is reflected in the MVKD.

Figures 1b and 1e demonstrate the runtime of MVKD.Update in two ways. We assume the set of links to be added have already been received (but not verified) by the server.

Figure 1b shows the time to create one epoch of 10 random links in an MVKD of different sizes, where the x-axis is scaled logarithmically. We can see that even a large MVKD of 64M links supports adding 10 links in well under a second. A server load of 10 links per second could occur in, for example, a system with 16M users creating 4 links per year, which averages to 2 links per second, plus some buffer to support traffic bursts.

Conversely, Figure 1e shows the time to create one epoch of N links in a tree of constant size. Our implementation manages to add 128 links in about a second to a directory containing 64M links. Build time for larger epochs could be improved by sharding the tree into subtrees and computing each subtree root concurrently.

In an update, the server first validates the new links, checking for keychain semantics (ValidKeychain) and verifying client signatures (CheckAuth). For existing users, keychain states (i.e., the list of valid devices and public keys) are cached in the database, so the server does not need to revalidate previous links. The server then performs the underlying RZKS.Update operation: it computes the VRF image of each label (as well as precomputes the corresponding proof, to speed up future queries) and then inserts the new label-value pairs into the Merkle tree. While verifying signatures and VRF exponentiations are expensive operations, in our implementation the cost of MVKD.Update is entirely dominated by LevelDB database calls for reading and writing tree nodes.

Note that the time required to run MVKD.Audit when the commitment and proofs are produced by Update (as opposed to by PCSUpdate) is strictly smaller than the time required to compute the update. A simple auditor implementation performs the same hash computations that the server did to compute the new Merkle tree root but does not have to compute or verify VRF proofs.

Performance of PCSUpdate**.** PCSUpdate is an important operation that allows the server to recover from any state leak, such as the leak of the VRF secret key used by the underlying RZKS. However, we expect this more expensive operation to be performed

[8] Merkle tree proofs are assembled for each query, while VRF proofs are precomputed during updates and simply retrieved from the database during queries.

infrequently in comparison with Update: it should be run regularly over longer time intervals as a best practice, such as once every month, and additionally when infrastructure breaches are detected. The frequency of PCSUpdate could match or be chosen using analogous considerations as that of TLS certificates and other key rotations. Our implementation also incentivizes providers to run this operation because it publicly reveals whenever a new epoch is computed using PCSUpdate instead of Update. Thus, rotating keys infrequently might adversely affect a provider's reputation.

Figure 1c demonstrates that the time to compute a PCSUpdate (where the secret key material is rotated but no new links are added) is linear in the size of the MVKD, as it involves computing a new VRF image for each label (in parallel) and a VRF rotation proof, as well as building a new Merkle tree with the new labels. Note that our implementation also precomputes the VRF proofs for all the labels using the new key at this point in order to speed up future queries, resulting in a total of 5 exponentiations per link during the computation of the rotation proof. This computation could be further parallelized or distributed across multiple servers. Figure 1f shows that while computing the VRF rotation proof is a significant cost, PCSUpdate execution time is dominated by the time it takes to actually build the Merkle tree.

While the PCSUpdate operation takes a significant amount of time and scales linearly with directory size, it does not require halting regular updates for a significant period of time. The server could start by picking a new VRF key and rebuilding the Merkle tree while still accepting updates and creating epochs for the old VRF key. Once the new tree is built, the server could briefly halt updates, add any new links to the new tree, and then use the new tree for the next epoch, after which updates could be enabled again. The VRF rotation proof could be computed after-the-fact, though within a time frame required by auditors. Strictly speaking, this optimization is not captured in our privacy definition, where update operations are atomic, in an effort to limit complexity. However, we believe that in ELEKTRA, if the server state leaks after this PCSUpdate process has started but before it has completed (i.e. while the next VRF key has been selected, but updates are still using the old key), this would be equivalent to the leakage in our model for PCSUpdate completing at the epoch right before the corruption. This is due to the fact that PCSUpdate essentially just recomputes the tree using a new VRF key and that the old VRF secret key is still kept by the server to allow it to generate absence proofs w.r.t. previous epochs.

Auditing a PCSUpdate proof is more expensive than a proof produced by Update. A PCSUpdate proof consists of 2 elliptic curve group elements (i.e. the VRF images of each label under the old and new key) and a 32-byte commitment per keychain link in the directory, plus a constant factor of 2 elliptic curve group elements and one scalar. If the auditor checks the whole directory from the first epoch and holds on to the data received when auditing previous epochs, this can be reduced to 1 elliptic curve group element per link (representing the image under the new VRF key), plus the constant factor. This optimization is not explicitly captured by our API, as MVKD.Audit is stateless for the auditor, but the extension is straightforward. In practice, in a directory with 64M links, since P-256 elliptic curve points take 64 bytes to represent, this amounts to approximately 4 GiB. In terms of computation, verifying a PCSUpdate proof is cheaper than computing it, as verifying the VRF rotation proof requires 2 instead of 5 exponentiations per link (though both require building the tree from scratch). Implementing a stateless auditor which does not rely on the data from previous epochs would result in larger proof sizes and thus bandwidth costs, but reduced computation time as the auditor will not need to maintain a database of Merkle tree nodes.

6 RELATED WORK

Our system directly builds off Keybase [12, 13], SEEMless [7], and RZKS [8], as we describe in the introduction. CONIKS [19] was the first KT system proposed in the academic literature and formally introduced the notion of a system that periodically commits to a set of public keys. CONIKS relied on users to be online every epoch to monitor their keys and leaks the update histories of users, which as we note in the introduction can be sensitive metadata. SEEMless improves on these drawbacks by providing stronger privacy guarantees and by using an append-only sparse Merkle tree that enables users to monitor their key history at any time with cost related to the number of times their key has been updated.

Parakeet [18] is a KT system recently introduced that improves over SEEMless by adding better scalability through a verifiable append-only data structure that supports compaction and a consistency protocol for broadcasting commitments. But, as noted earlier, Parakeet does not capture the stronger security guarantees that our device cross-signing system provides.

Merkle^2 [11], VeRSA [26], and Verdict [27] do not target strong privacy as a goal, thereby leaking update patterns. Moreover, VeRSA and Verdict utilize RSA accumulators and SNARKs, which are more expensive to deploy in practice. Meanwhile, Merkle^2 also uses a form of signature chains but does not formalize this primitive and cannot provide soundness if key resets are allowed, which we stress is an important feature used in practice that should be modeled.

ACKNOWLEDGMENTS

The first author would like to acknowledge support from the National Science Foundation under awards CNS-2120651 and DGE–2139899.

REFERENCES

[1] [n. d.]. IETF Key Transparency (keytrans). https://datatracker.ietf.org/wg/keytrans/about/. Accessed: 2023-04-27.

[2] Shashank Agrawal and Srinivasan Raghuraman. 2020. KVaC: Key-Value Commitments for Blockchains and Beyond. In *Advances in Cryptology - ASIACRYPT 2020 (Lecture Notes in Computer Science)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer.

[3] Apple.com. [n. d.]. Apple Privacy. https://www.apple.com/privacy/features. Accessed: 2022-08-03.

[4] Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. 2004. Universally Composable Protocols with Relaxed Set-Up Assumptions. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS '04)*. IEEE Computer Society, USA. https://doi.org/10.1109/FOCS.2004.71

[5] David A. Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. 2014. ARPKI: Attack Resilient Public-Key Infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[6] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. 2022. E2E Encryption for Zoom Meetings. White Paper – Github Repository zoom/zoom-e2e-whitepaper, Version 3.2, https://github.com/zoom/zoom-e2e-whitepaper/blob/master/zoom_e2e.pdf.

[7] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. 2019. SEEMless: Secure End-to-End Encrypted Messaging with less Trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security CCS*. ACM.

[8] Brian Chen, Yevgeniy Dodis, Esha Ghosh, Eli Goldin, Balachandar Kesavan, Antonio Marcedone, and Merry Ember Mou. 2022. Rotatable Zero Knowledge Sets: Post Compromise Secure Auditable Dictionaries with application to Key Transparency. In *Advances in Cryptology - ASIACRYPT 2022*. Springer International Publishing, Cham. Full version: https://eprint.iacr.org/2022/1264.

[9] Sergej Dechand, Dominik Schürmann, Karoline Busse, Yasemin Acar, Sascha Fahl, and Matthew Smith. 2016. An Empirical Study of Textual Key-Fingerprint Representations. In *25th USENIX Security Symposium, USENIX Security 2016*. USENIX Association.

[10] Google. [n. d.]. Key Transparency Overview. https://github.com/google/keytransparency/blob/master/docs/overview.md. Accessed: 2022-10-06.

[11] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca A. Popa. 2021. Merkle2: A Low-Latency Transparency Log System. *2021 IEEE Symposium on Security and Privacy (SP)*, 285–303.

[12] Keybase.io. [n. d.]. Keybase Chat. https://book.keybase.io/docs/chat. Accessed: 2022-08-03.

[13] Keybase.io. 2014. Meet your sigchain (and everyone else's). https://book.keybase.io/docs/server#meet-your-sigchain-and-everyone-elses. Accessed: 2022-07-29.

[14] Keybase.io. 2019. Keybase is not softer than TOFU. https://keybase.io/blog/chat-apps-softer-than-tofu. Accessed: 2019-05-05.

[15] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil D. Gligor. 2013. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In *22nd International World Wide Web Conference, WWW '13*. International World Wide Web Conferences Steering Committee / ACM.

[16] Sean Lawlor and Kevin Lewi. 2023. Deploying key transparency at WhatsApp. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/. Accessed: 2023-04-27.

[17] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. 2022. Aardvark: An Asynchronous Authenticated Dictionary with Applications to Account-based Cryptocurrencies. In *31st USENIX Security Symposium, USENIX Security 2022*. USENIX Association.

[18] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean F. Lawlor. 2023. Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/parakeet-practical-key-transparency-for-end-to-end-encrypted-messaging/

[19] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. 2015. CONIKS: Bringing Key Transparency to End Users. In *24th USENIX Security Symposium, USENIX Security 2015*. USENIX Association, Washington, D.C., 383–398. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara

[20] Silvio Micali, Salil Vadhan, and Michael Rabin. 1999. Verifiable Random Functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*. IEEE Computer Society.

[21] microsoft.com. 2022. Teams End-to-End Encryption. https://docs.microsoft.com/en-us/microsoftteams/teams-end-to-end-encryption. Accessed: 2022-05-26.

[22] Signal. [n. d.]. What do I do if my phone is lost or stolen? https://support.signal.org/hc/en-us/articles/360007062452-What-do-I-do-if-my-phone-is-lost-or-stolen-. Accessed: 2023-02-13.

[23] signal.org. 2016. Technical information. https://www.signal.org/docs. Accessed: 2022-08-03.

[24] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. 2019. Transparency Logs via Append-Only Authenticated Dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM.

[25] Alin Tomescu, Yu Xia, and Zachary Newman. 2020. Authenticated Dictionaries with Cross-Incremental Proof (Dis)aggregation. Cryptology ePrint Archive, Paper 2020/1239. https://eprint.iacr.org/2020/123. https://eprint.iacr.org/2020/123

[26] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. 2022. VeRSA: Verifiable Registries with Efficient Client Audits from RSA Authenticated Dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[27] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. 2022. Transparency Dictionaries with Succinct Proofs of Correct Operation. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*.

[28] Webex.com. 2022. Webex End-to-End Encryption. https://help.webex.com/en-us/article/WBX44739/What-Does-End-to-End-Encryption-Do?. Accessed: 2022-05-26.

[29] Whatsapp.com. 2021. WhatsApp Encryption Overview. White Paper – https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Accessed: 2022-08-03.

A PRELIMINARIES

We denote with λ the security parameter. We represent maps as sets of label-value pairs $D = \{(a, b), (c, d), \dots\}$ with unique labels. We write $D[x]$ to access the value y assigned to label x, and use the special symbol $\perp$ in case D does not contain such a label, i.e., the check $D[x] \neq \perp$ returns true iff there exists some y such that $(x, y) \in D$. Similarly, for a finite list $C = (a, b, c, \dots)$, we use $C[i]$ to denote the i-th element of list C (1-indexed), $\mathsf{last}(C)$ for its last element, and $C \| x$ for appending an element x to a list.

For algorithms with multiple inputs, we assume that there is a well-defined way to serialize and deserialize such a tuple, and we use "**parse** a as $(a_1, \dots, a_n)$" to denote that an algorithm tries to unpack a tuple (producing a special symbol ERROR if this is not possible). Given a boolean b, we use "**ensure** b" as shorthand for "if not b, **return** ERROR". In a security game, we use "**require** b" as a shorthand to denote that if b is false the oracle call by the adversary is immediately ended without output, and "**assert** b" to denote that if b is false, the security experiment is immediately terminated with ERROR. When involving the special symbol ERROR in arithmetic expressions, we assume them to evaluate to false.

For an interactive (two-party) algorithm Alg, we denote its execution by $(y_1; y_2) \leftarrow \mathsf{Alg}(x_1; x_2)$, where x_1 and x_2 are the two parties' inputs, and y_1 and y_2 the outputs. The parties can interact using the **send** and **receive** keywords. In a security game, when executing the algorithm between one honest party and the adversary $\mathcal{A}$ (who in turn invoked the oracle) we denote this by $(y_1; \perp) \leftarrow \mathsf{Alg}(x_1; \mathcal{A})$ or $(\perp; y_2) \leftarrow \mathsf{Alg}(\mathcal{A}; x_2)$. Here, control is returned to the adversary whenever the honest party waits for input from the other party. At this point, the adversary may either provide this input — in which case the execution continues — or to start (or continue) a concurrent oracle execution.

B ROTATABLE ZERO KNOWLEDGE SET

In this section, we recall the formal definition and security properties of Rotatable Zero Knowledge Sets (RZKS), introduced in [8]. We first begin with a summary of the primitive and its security definitions. The more formal description and figures in this section are almost verbatim from [8]; see that paper for additional details and discussion, including a description of an efficient and practical instantiation of this primitive.

Summary: Rotatable Zero Knowledge Sets. Our protocol makes use of the RZKS primitive, which implements a privacy-preserving and authenticated append-only dictionary that employs three parties: a server (maintaining the dictionary), users (querying the dictionary), and auditors (ensuring the append-only property).

- RZKS.Init produces a commitment com to an empty datastore and an initial server state st. A datastore is a map from labels label to values (val, t), where t is an integer indicating that the tuple has been added to the as part of the t-th Update operation (called epoch).
- RZKS.Update and RZKS.PCSUpdate append a set of new (label, value) pairs to the datastore. The algorithms output an updated commitment com′ and server state st′, as well as a proof π_S, which is then verified using RZKS.VerifyUpd. The former variant is more efficient while the latter variant provides PCS, i.e., guarantees after a server compromise.
- RZKS.Query is run by the server to answer a query for label by a user at epoch t_u. If the datastore contains a tuple $(\mathsf{label}, \mathsf{val}, t)$, with $t \leq t_u$, it returns (val, t) and a proof π thereof. Else, it returns a non-membership proof. Either proof can then be verified against the commitment com_{t_u} using RZKS.Verify.
- RZKS.ProveExt produces a so-called extension proof, which can then be verified using RZKS.VerExt. Such a proof enable users to reduce the number of interactions with the auditors: if the auditor has the same commitment com_{t_1} for epoch t_1 as the user, it assures the user, a-posteriori, that it also has had the same sequence of commitments com_i, for $0 \leq i < t_1$, as the auditor.

Our construction, on a high-level, assumes the following security properties from a RZKS; see below for formal definitions thereof.

Completeness. Intuitively, all updates and queries should behave as expected by their descriptions above. Furthermore, all proofs produced by various updating or querying algorithms should verify when properly queried to the corresponding verification algorithms.

Soundness. The RZKS soundness guarantees that a (malicious) prover is unable to produce two verifying proofs for two different values for the same label with respect to the same commitment, and that it knew the entire collection of (label, value) pairs at the time it produced the commitment (i.e., the entire datastore can be extracted from the commitment). Additionally, RZKS guarantees consistency among the RZKS commitments produced over the epochs: each commitment to an epoch also binds the server to all previous commitments (i.e. they can be extracted from the former), and a valid extension proof between two commitments ensure that they bind to the same set of previous commitments (up to the earlier of the two epochs). Moreover, the server cannot produce an update proof between commitments that do not respect the append-only property.

Privacy. The privacy definition of RZKS models *zero-knowledge with leakage* in the real world/ideal world paradigm, and is parameterized over an instantiation-specific leakage function. Informally, this means the commitments and proofs generated by any sequence of calls to RZKS algorithms can be simulated given any (label, value) pair which is part of the answer to the calls and the output of the leakage function.

B.1 Formal Definition

For reference, we provide the formal description of RZKS; the following is almost verbatim from [8].

Definition 5 (Def. 1 from [8]). A Rotatable Zero Knowledge Set (RZKS) consists of algorithms RZKS = (RZKS.GenPP, RZKS.Init, RZKS.Update, RZKS.PCSUpdate, RZKS.VerifyUpd, RZKS.Query, RZKS.Verify, RZKS.ProveExt, RZKS.VerExt) defined as follows:

- $\mathsf{pp} \leftarrow \mathsf{RZKS.GenPP}(1^\lambda)$: This algorithm takes the security parameter and produces public parameter pp for the scheme.

- $(\mathsf{com}, \mathsf{st}) \leftarrow \mathsf{RZKS.Init}(\mathsf{pp})$: This algorithm produces a commitment com to an empty datastore $\mathsf{D}_0 = \{\}$ and an initial server state st. A datastore D is a map from labels to values (val, t), where t is an integer indicating that the tuple has been added to the datastore as part of the t-th Update or PCSUpdate operation (this is called epoch). Each server state st will contain a datastore and a digest, which is referred to as $\mathbf{D}(\mathsf{st})$ and $\mathbf{com}(\mathsf{st})$. Similarly, each commitment will include the epoch $\mathbf{t}(\mathsf{com})$ of the datastore to which it is referring. (Alternatively, these can be thought of as deterministic functions which are part of the scheme).
- $(\mathsf{com}', \mathsf{st}', \pi_S) \leftarrow \mathsf{RZKS.Update}(\mathsf{pp}, \mathsf{st}, S)$, $(\mathsf{com}', \mathsf{st}', \pi_S) \leftarrow \mathsf{RZKS.PCSUpdate}(\mathsf{pp}, \mathsf{st}, S)$:
  Both algorithms update the datastore with a set $S = \{(\mathsf{label}_1, \mathsf{val}_1), (\mathsf{label}_2, \mathsf{val}_2), \ldots, (\mathsf{label}_n, \mathsf{val}_n)\}$ of new (label, value) pairs to insert. The algorithm outputs an updated commitment to the updated datastore, a modified internal state st', and a proof π_S that the update has been done correctly and only appended new label-value pairs. Intuitively, com' is a commitment to the updated datastore $\mathbf{D}(\mathsf{st}')$ at epoch $\mathbf{t}(\mathsf{com}') = \mathbf{t}(\mathsf{com}) + 1$, which extends $\mathbf{D}(\mathsf{st})$ by also mapping each label_i in S to the pair $(\mathsf{val}_i, \mathbf{t}(\mathsf{st}'))$.
- $0/1 \leftarrow \mathsf{RZKS.VerifyUpd}(\mathsf{pp}, \mathsf{com}_{t-1}, \mathsf{com}_t, \pi_S)$: This deterministic algorithm takes in two commitments to the datastore output at successive invocations of Update, and verifies the above proof.
- $(\pi, \mathsf{val}, t) \leftarrow \mathsf{RZKS.Query}(\mathsf{pp}, \mathsf{st}, u, \mathsf{label})$: This algorithm takes as input a state st, an epoch $u \leq \mathbf{t}(\mathsf{st})$, and a label. If a tuple $(\mathsf{label}, \mathsf{val}, t) \in \mathbf{D}(\mathsf{st})$ and $t \leq u$, it returns val, t and a proof π. Else, it returns $\mathsf{val} = \perp$, $t = \perp$ and a non-membership proof π. In both cases, proofs are meant to be verified against the commitment com_u output during the u-th update.
- $1/0 \leftarrow \mathsf{RZKS.Verify}(\mathsf{pp}, \mathsf{com}, \mathsf{label}, \mathsf{val}, t, \pi)$: This deterministic algorithm takes a (label, val, t) tuple, and verifies the proof π with respect to the commitment com. If $\mathsf{val} = \perp$ and $t = \perp$, this is considered a proof that label is not part of the data structure at epoch $\mathbf{t}(\mathsf{com})$.
- $\pi_E \leftarrow \mathsf{RZKS.ProveExt}(\mathsf{pp}, \mathsf{st}, t_0, t_1)$: This algorithm takes the state of the prover and two epochs t_0, t_1, and returns a proof π_E that the datastore after the t_1-th update is an extension of the datastore after the t_0-th update. Proofs are meant to be verified against the commitments com_{t_0} and com_{t_1} output by Update during the t_0-th and t_1-th update. Here, "extension" means that the respective digest com_{t_1} binds to com_{t_0} and all other prior commitments, such that a user currently storing com_{t_0} can safely update to com_{t_1} after checking the proof, knowing that the server cannot prove that any commitment for epoch t_0 other than com_{t_0} is consistent with com_{t_1}.
- $1/0 \leftarrow \mathsf{RZKS.VerExt}(\mathsf{pp}, \mathsf{com}_{t_0}, \mathsf{com}_{t_1}, \pi_E)$: This deterministic algorithm takes two datastore commitments and a proof (generated by ProveExt) and verifies it.

RZKS has to satisfy the following security properties:

Completeness. A RZKS satisfies completeness if for all PPT adversaries $\mathcal{A}$, the probability that the game described in Figure 2 outputs 0 is negligible in 1^λ.

Intuitively, all updates and queries should behave as expected by their descriptions in the definition. Furthermore, all proofs produced by various updating or querying algorithms should verify when properly queried to the corresponding verification algorithms. More formally, an adversary only breaks completeness if it is able to construct a sequence of queries such that one of the assertions in Figure 2 fails. For example, the assertion $\mathbf{D}(\mathsf{st}') = \mathbf{D}(\mathsf{st}) \cup \{(\mathsf{label}_i, \mathsf{val}_i, t+1)\}_{i \in [j]}$ in $\mathsf{Update}(S)$ will only fail if the elements added in S are not correctly added to the state of the datastore. Similarly, in $\mathsf{Query}(\mathsf{label}, u)$ completeness assert that $\mathsf{RZKS.Verify}(\mathsf{com}_u, \mathsf{label}, \mathsf{val}', t', \pi)$ succeeds, where (val', t', π) are those produced by the corresponding call to RZKS.Query.

Soundness. A RZKS satisfies soundness if there exists an extractor Extract such that for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in distinguishing the two experiments described in Figure 3 is negligible in 1^λ. Note that all the algorithms executed in the experiment get implicit access to the Ideal oracle, as they might need to make, i.e., random oracle calls.

The extractor Extract is required to provide various functionalities based on its first input:

- $\mathsf{pp}', \mathsf{st} \leftarrow \mathsf{Extract}(\mathtt{Init})$: Samples public parameters indistinguishable from honestly generated public parameters such that extraction will be possible. Also generates an initial state.
- $\mathsf{D}_{\mathsf{com}} \leftarrow \mathsf{Extract}(\mathtt{Extr}, \mathsf{st}, \mathsf{com})$: Takes in the internal state and a commitment to the datastore. Outputs the set of (label, val, i) committed to.
- $C_{\mathsf{com}} \leftarrow \mathsf{Extract}(\mathtt{ExtrC}, \mathsf{st}, \mathsf{com})$: Takes in the internal state and a commitment to the datastore. Outputs the set of previous commitments, indexed by epoch.
- $out, \mathsf{st} \leftarrow \mathsf{Extract}(\mathtt{Ideal}, \mathsf{st}, in)$: Simulates the behavior of some ideal functionality (for example a random oracle or generic group). Takes in any input and produces an output indistinguishable from the output the ideal functionality would have on that input.

One small subtlety of the definition here is that it does not allow the extractor to update its state outside of Ideal calls. The only advantage that the extractor gets over an honest party, is its control over the ideal functionality. This allows for easier composition, since a larger primitive utilizing RZKS will not need to simulate extractor state.

An adversary breaks soundness if it either distinguishes answers to Ideal queries in the real game from those produced by the extractor, or if it causes some assertion to be false in the ideal game. Each assertion in the ideal game captures some way in which the extractor could be caught in an inconsistent state. For example, consider the assertion $\mathsf{D}[\mathsf{com}][\mathsf{label}] = (\mathsf{val}^*, i^*)$ in CheckVerD. This will be false if the adversary can provide a proof that (label, val*, i^*) is in the datastore with digest com, but the extractor expects this datastore to either not contain label or to contain (label, val, i) for some different (val, i).

This soundness definition strengthens the aZKS [7] one by providing extractability. aZKS soundness already guarantees that a (malicious) prover is unable to produce two verifying proofs for two different values for the same label with respect to an aZKS commitment it has already produced. However, that definition does not guarantee that the malicious prover knew the entire collection of (label, value) pairs at the time it produced the commitment. Extractability requires that by mandating that the entire datastore can be extracted from the commitment, except with negligible probability.

$\underline{\text{Completeness}^{\mathcal{A}}_{\text{RZKS}}}$:

$\text{pp}' \leftarrow \text{RZKS.GenPP}(1^\lambda)$
$(\text{com}', \text{st}') \leftarrow \text{RZKS.Init}(\text{pp}')$
assert $\mathbf{com}(\text{st}') = \text{com}'$ and $\mathbf{t}(\text{com}') = 0$ and $\mathbf{D}(\text{st}') = \{\}$
$\text{com}_0 \leftarrow \text{com}', \text{st} \leftarrow \text{st}', t \leftarrow 0, \text{pp} \leftarrow \text{pp}'$
$\mathcal{A}^{O\cdots}(\text{pp}, \text{com}_0)$
return 1

$\underline{\text{Oracles Update}(S) \text{ and PCSUpdate}(S)}$:

parse S as $(\text{label}_1, \text{val}_1), \ldots, (\text{label}_j, \text{val}_j)$
require $\text{label}_1, \ldots, \text{label}_j$ are distinct and do not already appear in $\mathbf{D}(\text{st})$
$(\text{com}', \text{st}', \pi) \leftarrow \text{RZKS.Update}(\text{st}, S)$ // resp. RZKS.PCSUpdate(st, S)
assert $\mathbf{com}(\text{st}') = \text{com}', \mathbf{t}(\text{com}') = t + 1$ and $\mathbf{D}(\text{st}') = \mathbf{D}(\text{st}) \cup \{(\text{label}_i, \text{val}_i, t + 1)\}_{i \in [j]}$
assert $y \leftarrow \text{RZKS.VerifyUpd}(\text{com}_t, \text{com}', \pi);\ y = 1$
$\text{com}_{t+1} \leftarrow \text{com}', \text{st} \leftarrow \text{st}', t \leftarrow t + 1$

$\underline{\text{Oracle Query}(\text{label}, u)}$:

require $0 \le u \le t$
$(\pi, \text{val}', t') \leftarrow \text{RZKS.Query}(\text{st}, u, \text{label})$
If $\text{label} \in \mathbf{D}(\text{st}), (\text{val}_\text{D}, u_\text{D}) \leftarrow \mathbf{D}(\text{st})[\text{label}]$ and $u_\text{D} \le u$:
  assert $(\text{val}', t') = (\text{val}_\text{D}, u_\text{D})$
Else
  assert $(\text{val}', t') = (\perp, \perp)$
assert $y \leftarrow \text{RZKS.Verify}(\text{com}_u, \text{label}, \text{val}', t', \pi);\ y = 1$

$\underline{\text{Oracle ProveExt}(t_0, t_1)}$:

require $0 \le t_0 \le t_1 \le t$
$\pi_E \leftarrow \text{RZKS.ProveExt}(\text{st}, t_0, t_1)$
assert $y \leftarrow \text{RZKS.VerExt}(\text{com}_{t_0}, \text{com}_{t_1}, \pi_E);\ y = 1$

Figure 2 (Fig. 1 from [8]): Completeness for RZKS**. In this experiment, the adversary can read all the game's state and the oracle's intermediate variables, such as** $com_i \forall i$, st, y**. The experiment returns 1 unless one of the assertions is triggered. These checks enforce that the data structure is updated consistently, that the outputs of query reflect the state of the data structure, and that honestly generated proofs pass verification as intended.**

RZKS also explicitly guarantee consistency among the RZKS commitments produced over epochs. Informally, consistency guarantees that each commitment to an epoch also binds the server to all previous commitments (i.e. these can be extracted from the former). In particular, when the client swaps a commitment com^a with a more recent one com^b by verifying an extension proof, and then checks with an auditor that com^b is legitimate, the client can be sure that any auditor who checked all consecutive audit proofs up to com^b must also have checked the same com^a for epoch a. This is modeled in the security game by the assertions in the ExtractC, CheckVerUpdC, and CheckVerExt oracles.

Zero Knowledge. A RZKS is zero-knowledge for leakage function $\mathcal{L} = (\mathcal{L}_{\text{Update}}, \mathcal{L}_{\text{PCSUpdate}}, \mathcal{L}_{\text{Query}}, \mathcal{L}_{\text{ProveExt}}, \mathcal{L}_{\text{LeakState}})$ if there exists a simulator $\mathcal{S}$ such that every PPT malicious client algorithm $\mathcal{A}$ has negligible advantage in distinguishing the two experiments of Figure 4.

The stateful simulator $\mathcal{S}$ is required to provide various functionalities:

- $\text{com}', \text{pp}' \leftarrow \mathcal{S}(\texttt{Init})$: samples public parameters and an initial commitment indistinguishable from honest public parameters such that it will be possible to simulate proofs.
- $(\text{com}', \pi) \leftarrow \mathcal{S}((\texttt{PCS})\texttt{Update}, l)$: Takes in some leakage l about an Update (or, analogously, PCSUpdate) query on input S, i.e. in the experiment $l \leftarrow \mathcal{L}_{\text{Update}}(S)$ (or $l \leftarrow \mathcal{L}_{\text{PCSUpdate}}(S)$). Outputs a commitment com' indistinguishable from a commitment to the previous datastore with the elements of S appended. Furthermore, it simulates a proof π that the update was done correctly.
- $(\pi, \text{val}', t') \leftarrow \mathcal{S}(\texttt{Query}, l)$: Takes in leakage $l \leftarrow \mathcal{L}_{\text{Query}}(u, \text{label})$ about the entry indexed by (u, label) in the datastore. Outputs val', t' which would have been returned by an honest query. Also simulates a proof π that $\text{D}[\text{label}] = (\text{val}', t')$, or an absence proof if $\text{label} \notin \text{D}$.
- $\pi \leftarrow \mathcal{S}(\texttt{ProveExt}, l)$: Takes in partial information $l \leftarrow \mathcal{L}_{\text{ProveExt}}(t_0, t_1)$ from a ProveExt query the between epochs t_0 and t_1. Outputs an extension proof that the commitment provided at epoch t_1 binds to the one at epoch t_0.
- $\text{st} \leftarrow \mathcal{S}(\texttt{Leak}, l)$: Takes in partial information $l \leftarrow \mathcal{L}_{\text{LeakState}}()$ about the datastore and outputs a simulated state consistent with the information given.
- $out \leftarrow \mathcal{S}(\texttt{Ideal}, in)$: Simulates the behavior of some ideal functionality. Takes in any input and produces an output indistinguishable from the output the ideal functionality would have on that input.

Note that the particular leakage given will be construction specific, but should be designed to be as minimal as possible. The choice of leakage in the RZKS construction [8] will be described in detail in Section B.2. In the experiment, the only information the simulator has access to is the output of the leakage function, as well as the queries made to the Ideal oracle. The simulator's ability to control the ideal oracle is crucial for security proofs to go through.

Informally, zero knowledge means, that the proofs generated by any sequence of calls to RZKS algorithms can be simulated given access to minimal information about the queries made. The adversary breaks zero knowledge if it is able to generate a sequence of queries such that it can distinguish the output of the simulator from honestly generated outputs and proofs. For example, if the simulator is

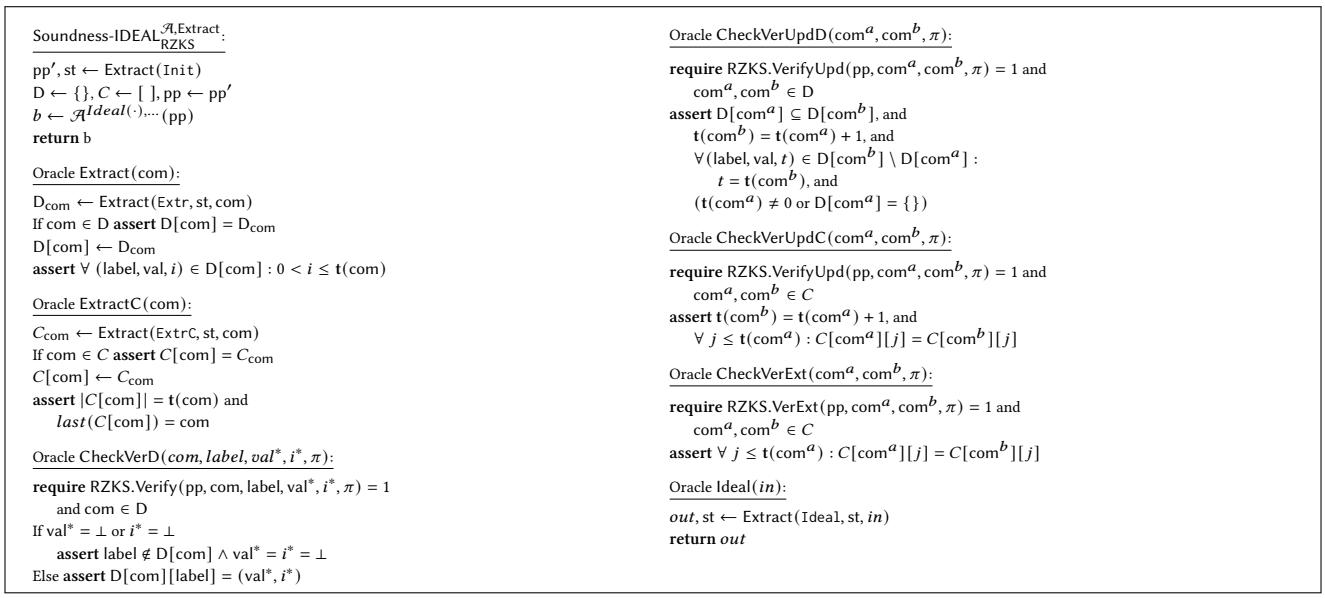


Figure 3 (Fig. 2 from [8]): Soundness for RZKS**. In the ideal world, the map** D **stores, for each commitment** com**, the datastore that the Extract algorithm output for that commitment. In addition the map C stores, for each commitment, the (ordered) list of commitments to previous epochs. When the adversary provides proofs, one requires that the proofs are consistent with such data structures. In the real world (not pictured), the public parameters are generated as** $\text{pp} \leftarrow \text{RZKS.GenPP}(1^\lambda)$**, and all the oracles do nothing and return no output, except for the Ideal oracle, which implements the ideal objects (such as random oracles) that the authors of [8] abstract to prove security of the primitives (and that are controlled by the extractor in the ideal world). In both cases,** RZKS**'s algorithms implicitly get access to the Ideal oracle as needed.**

unable to simulate query proofs, then an adversary could succeed by calling the Update({label, val}) oracle for some (label, val), then the $(\pi, \text{val}, 1) \leftarrow \text{Query}(\text{label}, 1)$ oracle, and running RZKS.Verify on π. Since the simulator can't simulate query proofs, π generated in the ideal world will not verify and so will be distinguished from π generated in the real world.

Post-compromise security is modeled by allowing for LeakState calls, which reveal the state in its entirety. When the adversary queries this oracle, the simulator is required to output a state that appears consistent with whatever proofs it has revealed before. Healing from compromise is modeled by having a dedicated leakage function for PCSUpdate (different from Update). Note that since all the leakage functions share state, calling LeakState or PCSUpdate might affect the leakage of other future queries.

B.2 Leakage for the RZKS construction in [8]

In [8], a practically efficient instantiation of this primitive is provided that satisfies correctness, soundness, as well as zero knowledge with respect to the following leakage function:

- The shared state consists of a set of labels X, a datastore D, a counter t for the current epoch (initialized to 0), a counter g for the current generation (i.e. the number of PCSUpdate operations performed, also starting at 0), a map G that matches each epoch to the respective generation, and a boolean *leaked* (initially *false*).
- $\mathcal{L}_{\textbf{Query}}(\text{label}, u)$: If $\exists(\text{label}, \text{val}, t') \in \text{D}$ such that $t' \leq u$, the function returns $(\text{label}, \text{val}, t', u)$. If $\exists(\text{label}, \text{val}, t') \in \text{D}$ such that $G[t'] = G[u]$, the function returns $(\text{label}, \perp, t', u)$. Otherwise, it returns $(\text{label}, \perp, \perp, u)$ and, if $G[u] = g$, adds label to X.
- $\mathcal{L}_{\textbf{Update}}(S)$: Parse $S = \{(\text{label}_i, \text{val}_i)\}$. If S contains any duplicate label, or any label which appears in D, this function returns $\perp$. Else, it increments t, sets $G[t] \leftarrow g$, and adds the pairs from S to the datastore D at epoch t. If *leaked*, it returns the labels in S. Else, it returns $|S|$ and the set of labels from S which are also in X.
- $\mathcal{L}_{\textbf{PCSUpdate}}(S)$: Parse $S = \{(\text{label}_i, \text{val}_i)\}$. If S contains any duplicate label, or any label which appears in D, this function returns $\perp$. Else, it increments t, adds the pairs from S to the datastore D at epoch t, and updates $X \leftarrow \{\}$, *leaked* $\leftarrow$ *false*, and $g \leftarrow g + 1$, $G[t] \leftarrow g$. It returns $|S|$.
- $\mathcal{L}_{\textbf{LeakState}}()$: Set *leaked* $\leftarrow$ *true*. **return** D.
- $\mathcal{L}_{\textbf{ProveExt}}(t_0, t_1)$: **return** (t_0, t_1).

The security of the construction depends on the random oracle model, the generic group model, and the Decisional Diffie Hellman assumption. See [8] for intuition on the leakage and further details.

C ADDITIONAL DEFINITIONS

In this section, we introduce some additional definitions and notation used for the formal MVKD security notions.

Keychain statements. First, associated to a keychain statement is a set of functions.

Definition 6. *For a keychain statement* s*, we define the following functions:*

- **type**(s) *returns the statement type representing the action to be performed (first component),*

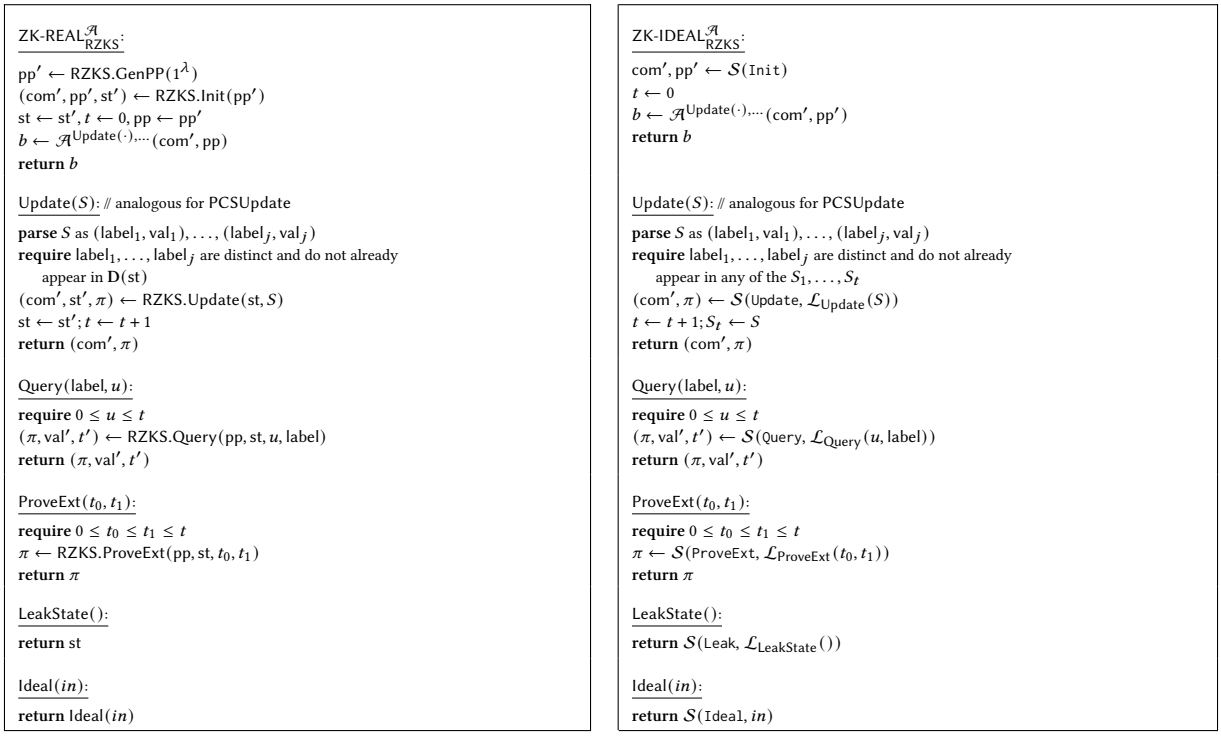


Figure 4 (Fig. 3 from [8]): Zero Knowledge (with leakage) security experiments for RZKS. $\mathcal{S}$ is a stateful algorithm (whose state is omitted here to simplify the notation). The leakage functions $\mathcal{L}_{\text{Update}}, \mathcal{L}_{\text{Query}}, \dots$ also share state among each other.

- $\mathbf{pk_a}(\text{s})$ *returns the public key of the device authorizing the statement (second component),*
- $\mathbf{data}(\text{s})$ *returns the public key or data on which the action is performed (third component),*
- $\mathbf{t}(\text{s})$ *returns the epoch the statement was inserted in the keychain (fourth component). The latter can be the special symbol* $\perp$*, in case the epoch has not yet been determined.*

Finally, we define the function $\text{AddEpoch}(\text{s}, t)$ *that takes a keychain statement with* $\mathbf{t}(\text{s}) = \perp$ *and an epoch number* t *and returns the modified statement with the same components except for the epoch set to* t*.*

Keychains. Similar to keychain statements, we associate with a keychain the function $\mathbf{u}(\mathbb{S})$ which returns the keychain's username. The *length* of keychain $\mathbb{S}$, i.e. its number of statements, is denoted with $|\mathbb{S}|$. For a statement s, we refer to $\mathbb{S}\|\text{s}$ as the new keychain resulting from adding s to the end of $\mathbb{S}$. We denote with $\mathbb{S}[i]$ the i-th statement in the list and with $\mathbb{S}[1 \dots i]$ the keychain that consists of the first i statements of $\mathbb{S}$ and has the same associated username. For an epoch t, let $\mathbf{prefix}(\mathbb{S}, t)$ be the prefix of all statements up to epoch t, i.e., $\mathbf{prefix}(\mathbb{S}, t) = \mathbb{S}[1 \dots i_t]$ when i_t denotes the largest index such that $\mathbf{t}(\mathbb{S}[i_t]) \le t$. For convenience, we sometimes treat $\mathbb{S}$ as a set and write $\text{s} \in \mathbb{S}$ to denote that s is a keychain statement in $\mathbb{S}$. When iterating over $\mathbb{S}$ in a loop, we assume this to happen in sequential order. We define an empty keychain for username u, meaning the keychain that has no statements, as $(u; ())$.

In the following, we provide a more formal definition for the validity of a keychain.

Definition 7 (Formal version of Definition 2). *We say that a keychain* $\mathbb{S}$ *is valid, denoted* $\text{ValidKeychain}(\mathbb{S})$*, iff the following requirements are met:*

- *The epoch component of each keychain statement* s *in a keychain must be an integer and strictly monotonically increasing for successive statements, except for the last statement, which can have* $\mathbf{t}(\text{s}) = \perp$*.*
- *A public key can only be added once: For each public key pk, there can exist only one* $\text{s} \in \mathbb{S}$ *with* $\mathbf{data}(\text{s}) = pk$ *and* $\mathbf{type}(\text{s}) \in \{\text{AddFirst,Add}\}$*.*
- *A public key can only be revoked if it has been previously added but not yet revoked (with an* AddFirst *statement acting as the revocation for all previously unrevoked keys): For each public key pk,* $\mathbb{S}$ *can have a statement* s *with* $\mathbf{data}(\text{s}) = pk$ *and* $\mathbf{type}(\text{s}) = \text{Revoke}$ *only if there was a previous statement* s' *with* $\mathbf{data}(\text{s}') = pk$ *and* $\mathbf{type}(\text{s}') \in \{\text{AddFirst, Add}\}$*, but no other statement* s'' *between* s' *and* s *with* $\mathbf{type}(\text{s}'') = \text{AddFirst}$*, or* $\mathbf{data}(\text{s}'') = pk$ *and* $\mathbf{type}(\text{s}'') = \text{Revoke}$*.*
- *For each statement (except for* AddFirst*, which is self-certifying) the authorizing key must be an unrevoked key of the keychain: For each keychain statement* s *in* $\mathbb{S}$*, if* $\mathbf{type}(\text{s}) \neq \text{AddFirst}$*,* $\mathbf{pk_a}(\text{s})$ *must have been previously added but not yet revoked (as defined above).*

MVKD scheme. As with keychains, we define a set of functions that expose certain values from a MVKD state. Alternatively, one can think w.l.o.g. that the honest participants' states have a representation that lets us identify certain values, and these functions are just notation to denote those values.

Definition 8. *For a* MVKD *scheme, we define the following functions:*

- $\mathbf{u}(st)$ *and* $\mathbf{pk}(st)$ *map a device state to the associated username and public key, respectively, as determined by the respective inputs and outputs to* DeviceSetup. *Neither must change over the run of the protocol.*
- $\mathbf{com}(st)$ *and* $\mathbf{com}(st^S)$ *map the device and server states, respectively, to the most recent commitment stored in the state.*
- $\mathbf{t}(\mathsf{com})$ *maps the commitment to its associated epoch number (or* ERROR *if the commitment is malformed). We further define* $\mathbf{t}(st)$ *and* $\mathbf{t}(st^S)$ *to be shorthands for* $\mathbf{t}(\mathbf{com}(st))$ *and* $\mathbf{t}(\mathbf{com}(st^S))$, *respectively.*
- $\mathbf{S}(st, u)$ *maps a device state and username to a pair* $(t^{\mathsf{last}}, \mathbb{S})$ *representing the epoch* t^{last} *the device held when it last queried* u*'s keychain, and the obtained keychain* $\mathbb{S}$. *For any users that have not been queried, the function returns* $t^{\mathsf{last}} = 0$, *with an empty keychain* $\mathbb{S} = (u; ())$. *If* u *had no keychain, it returns an empty keychain as well (along with the proper* t^{last}*).*

D ELEKTRA PROTOCOL DETAILS

ELEKTRA is defined using a signature scheme SIG, a Rotatable Zero Knowledge Set RZKS, and a hash function hash. The state of each client is a tuple $\mathsf{st} = (u, sk, pk, \mathsf{com}, \mathsf{Users})$ with the following components:

- The RZKS public parameters pp (which we omit from the description and implicitly pass to all the RZKS algorithms)
- u is the username of the user this device is assigned to
- sk, pk are this client's keys for SIG
- com stores the latest commitment that the client knows about
- Users is a dictionary that maps each username u' to a tuple $(\mathbb{S}, \mathsf{auth}, t^{\mathsf{last}})$, where $\mathbb{S}$ is the latest version of this username's keychain the client knows about, auth is the authenticator for the last link of this keychain (helpful to validate future extensions to the chain), and t^{last} is the epoch of the last time this username's chain was successfully queried for.

Note that if the client queries for a user that is not part of the directory and receives a valid non-inclusion proof, it will store an empty keychain for this user.
The server state $\mathsf{st}^S = (\mathsf{st}_{\mathsf{RZKS}}, \mathsf{Com}, \mathsf{Users}^S)$ consists of the following components:

- The RZKS public parameters pp (which we omit from the description and implicitly pass to all the RZKS algorithms)
- $\mathsf{st}_{\mathsf{RZKS}}$ is the server state of the RZKS instance
- Com is a dictionary that maps each epoch to the corresponding RZKS commitment and the audit proof that validates that commitment
- Users^S is a dictionary that maps each u to its keychain.

We refer to each component of the states either using functional notation, e.g. $\mathbf{com}(\mathsf{st})$, or "class member" notation, e.g. st.com. In addition, $\mathbf{S}(\mathsf{st}, u)$ returns $(t^{\mathsf{last}}, \mathbb{S})$ if $\mathsf{st.Users}[u] = (\mathbb{S}, \mathsf{auth}, t^{\mathsf{last}})$, or $(0, (u; ()))$ if $\mathsf{st.Users}[u] = \perp$. For the server state, $\mathbf{com}(\mathsf{st}^S)$ is defined as $\mathbf{com}(\mathsf{st}^S.\mathsf{st}_{\mathsf{RZKS}})$. Recall that for both states, $\mathbf{t}(\mathsf{st})$ is a shorthand for $\mathbf{t}(\mathbf{com}(\mathsf{st}))$.

The protocol is described in Figure 5.

E MVKD COMPLETENESS

In this section, we provide a formal definition of our completeness notion, which prescribes the desired functionality for the honest users interacting with an honest server. The game is presented in Figure 6. It also allows other dishonest parties to send malicious updates to the server to ensure this cannot indirectly affect the honest parties' output and state (beyond updating the relevant keychains). We make a few restrictions on what malicious clients may do, either because it is redundant or already addressed by other definitions. In more detail, Query and VerExtension do not modify the server state, so there is no gain from allowing malicious parties to execute these algorithms with the server in the definition. Furthermore, guarantees about direct interactions between honest and malicious clients as part of AddKey are covered by the soundness definition, and so are not covered by this definition.

The game maintains the current server state ServerState and device states ClientStates, a mapping Com from epoch numbers t to the respective commitment com_t, and the directory Dir. Here, $\mathsf{Dir}[u]$ represents the keychain of user u according to the latest update by the server. The adversary $\mathcal{A}$ is provided access to the specified oracles and is given as input security parameter 1^λ, public parameters pp, the initial server state, and the initial commitment. It wins the experiment if the game returns ERROR when the game initialization or an oracle call by $\mathcal{A}$ triggers an assertion. We define the advantage of $\mathcal{A}$ as the probability of winning the experiment, and we say that a scheme MVKD meets our completeness notion if every PPT adversary $\mathcal{A}$ has negligible such advantage.

Extending the keychains. Honest devices are created using the DeviceSetup oracle, which verifies basic consistency properties of the initialized state. A device can then be instructed to extend their keychain using the AddFirstKey, AddKey, RevokeKey, and AddExtra oracles. These oracles first verify that the operation is valid, with respect to the device's local view on their own user's keychain, and then execute the operation. If the device has an up-to-date view of their keychain, the resulting authenticator auth is added to $\mathsf{PendingUpdates}[u]$, registering it as an honestly generated update the server can execute.

The server is instructed to apply a given set of updates via either the Update or PCSUpdate oracle, which have the same effect in terms of functionality. It is noteworthy that the set of updates is chosen by the adversary and, thus, can include maliciously generated updates (to account for malicious devices). The respective algorithm might reject in such a case, but only if either (a) there is at least one update that has not been honestly generated and hence is not in PendingUpdates, or (b) there exists a collision in updates for a user, meaning that there is more than one update for a single user's keychain. If the algorithm accepts, the game updates its directory Dir, and clears PendingUpdates for the affected users, as concurrently generated updates are no longer applicable.

Querying the directory. A device can query another user's keychain. The Query oracle ensures the keychain output by the device matches the prefix of the keychain stored by the server that corresponds to the device's stored commitment. The operation aborts with an

CheckAuth*$(\text{auth}_{i-1}, t_{i-1}, u, \text{s}, \text{auth}_i)$:

parse $(h, z) \leftarrow \text{auth}_i$
parse $(\text{Type}, pk_a, d, t) \leftarrow \text{s}$
If **type**$(\text{s}) = \text{Add}$:
  parse $z \leftarrow (z_0, z_1)$
  ensure $\text{SIG.Ver}(\mathbf{pk_a}(\text{s}), 0\|h, z_0) = 1$ and $\text{SIG.Ver}(\mathbf{data}(\text{s}), 0\|h, z_1) = 1$
Else:
  ensure $\text{SIG.Ver}(pk, 0\|h, z) = 1$
ensure $h = \text{hash}(\text{auth}_{i-1}, t_{i-1}, u, \text{Type}, pk_a, d)$
return 1

AuthStatement*(st, Type, pk, d):

$(\mathbb{S}, \text{auth}, t) \leftarrow st.\text{Users}[\mathbf{u}(st)]$
If $(\mathbb{S}, \text{auth}, t) = \perp$:
  $\mathbb{S} \leftarrow (\mathbf{u}(\text{st}); ())$, $\text{auth} \leftarrow \varepsilon$
$e \leftarrow \mathbf{t}(\text{last}(\mathbb{S}))$ (or $e \leftarrow 0$ if $|\mathbb{S}| = 0$)
ensure $\text{ValidKeychain}(\mathbb{S}\|(\text{Type}, pk, d, \perp)) = 1$
ensure $\mathbf{pk}(st) = pk \vee (\text{Type} = \text{Add} \wedge \mathbf{pk}(st) = d)$
$h \leftarrow \text{hash}(\text{auth}, e, \mathbf{u}(\text{st}), \text{Type}, pk, d)$
$z \leftarrow \text{SIG.Sign}(\mathbf{sk}(st), 0\|h)$
return (h, z)

MVKD.GenPP(1^λ) :

return $\text{RZKS.GenPP}(1^\lambda)$

MVKD.ServerInit(pp) :

$(\text{com}_0, \text{st}_{\text{RZKS}}) \leftarrow \text{RZKS.Init}(\text{pp})$
$\text{Users}^S \leftarrow \{\}$; $\text{Com} \leftarrow \{(0, (\text{com}_0, \perp))\}$
$\text{st}^S \leftarrow (\text{st}_{\text{RZKS}}, \text{Com}, \text{Users}^S)$
return $(\text{st}^S, \text{com}_0)$

MVKD.DeviceSetup(pp, u) :

$(sk, pk) \leftarrow \text{SIG.KeyGen}()$
$\text{com} \leftarrow \perp$; $\text{Users} \leftarrow \{\}$
$st \leftarrow (u, sk, pk, \text{com}, \text{Users})$
return (st, pk)

MVKD.AddFirstKey(st) :

return $\text{AuthStatement}^*(st, \text{AddFirst}, \mathbf{pk}(\text{st}), \mathbf{pk}(\text{st}))$

MVKD.RevokeKey(st, pk) :

return $\text{AuthStatement}^*(st, \text{Revoke}, \mathbf{pk}(\text{st}), pk)$

MVKD.AddExtra(st, d):

return $\text{AuthStatement}^*(st, \text{Extra}, \mathbf{pk}(\text{st}), d)$

MVKD.AddKey$(st_0, pk_1 \,;\, st_1, pk_0)$:

Adding Client:
$(h, z) \leftarrow \text{AuthStatement}^*(\text{st}_0, \text{Add}, \mathbf{pk}(\text{st}_0), pk_1)$.
receive (h', z')
ensure $h = h'$ and $\text{SIG.Ver}(pk_1, 0\|h, z') = 1$
return $(h, (z, z'))$

Added Client:
$(h', z') \leftarrow \text{AuthStatement}^*(\text{st}_1, \text{Add}, pk_0, \mathbf{pk}(\text{st}_1))$.
If $(h', z') = \text{ERROR}$, **return** 0
send (h', z')
return 1

MVKD.Update(st^S, M) and MVKD.PCSUpdate(st^S, M):

ensure $\forall (u_0, \text{s}_0, \text{auth}_0), (u_1, \text{s}_1, \text{auth}_1) \in \text{M}$:
  $(u_0, \text{s}_0, \text{auth}_0) \neq (u_1, \text{s}_1, \text{auth}_1) \Rightarrow u_0 \neq u_1$
$t \leftarrow \mathbf{t}(\text{st}^S) + 1$
$S \leftarrow \{\}$; For $(u, \text{s}, \text{auth}') \in \text{M}$:
  $\mathbb{S} \leftarrow \text{st}^S.\text{Users}^S[u]$
  If $\mathbb{S} = \perp$: $\mathbb{S} \leftarrow (u; ())$, $\text{auth} \leftarrow \varepsilon$, $e \leftarrow 0$
  Else : $(\text{auth}, e) \leftarrow \mathbf{D}(\text{st}^S.\text{st}_{\text{RZKS}})[(u, |\mathbb{S}|)]$
  $\text{s} \leftarrow (\mathbf{type}(\text{s}), \mathbf{pk_a}(\text{s}), \mathbf{data}(\text{s}), t)$
  ensure $\text{ValidKeychain}(\mathbb{S}\|\text{s})$
  ensure $\text{CheckAuth}^*(\text{auth}, e, u, \text{s}, \text{auth}') = 1$
  $\text{st}^S.\text{Users}^S[u] \leftarrow \mathbb{S}\|\text{s}$
  $\text{label} \leftarrow (u, |\mathbb{S}| + 1)$; $S \leftarrow S \cup \{(\text{label}, \text{auth}')\}$
$(\text{com}, \text{st}^S.\text{st}_{\text{RZKS}}, \pi) \leftarrow \text{RZKS.Update}(\text{pp}, \text{st}^S.\text{st}_{\text{RZKS}}, S)$ // MVKD.Update
  or $(\text{com}, \text{st}^S.\text{st}_{\text{RZKS}}, \pi) \leftarrow \text{RZKS.PCSUpdate}(\text{pp}, \text{st}^S.\text{st}_{\text{RZKS}}, S)$ // MVKD.PCSUpdate
$\text{st}^S.\text{Com}[t] \leftarrow (\text{com}, \pi)$
return $(\text{st}^S, \text{com})$

MVKD.Query$(st, u \,;\, \text{st}^S, u, t)$:

Client:
$(\mathbb{S}, \text{auth}_j, t^{\text{last}}) \leftarrow st.\text{Users}[u]$
If $\mathbb{S} = \perp$:
  $\mathbb{S} \leftarrow (u, [\,])$, $j \leftarrow 0$, $\text{auth}_j \leftarrow \varepsilon$, $t^{\text{last}} \leftarrow 0$
Else:
  $j \leftarrow |\mathbb{S}|$
send t^{last}
receive a list of tuples $\mathbb{T} \leftarrow \{(\text{s}_i, \text{auth}_i, \pi_i)\}_{i=j+1}^{n+1}$. If $\mathbb{T} = \text{ERROR}$ or $|\mathbb{T}| = 0$, return ERROR.
If $|\mathbb{T}| > 1$:
  ensure $\mathbf{t}(\text{s}_{j+1}) > t^{\text{last}}$
For $i = j+1, \ldots, n+1$:
  $\text{label}_i \leftarrow (u, i)$; $t_i \leftarrow \mathbf{t}(\text{s}_i)$
  If $i = n+1$:
    ensure $\text{auth}_i = \perp, \text{s}_i = \perp$
  Else:
    ensure $\mathbf{t}(\text{s}_{i-1}) \leq \mathbf{t}(\text{s}_i)$ (with $\mathbf{t}(\text{s}_0) = 0$)
    ensure $\text{CheckAuth}^*(\text{auth}_{i-1}, t_{i-1}, u, \text{s}, \text{auth}_i) = 1$
  ensure $\text{RZKS.Verify}(\text{pp}, \mathbf{com}(st), \text{label}_i, \text{auth}_i, t_i, \pi_i) = 1$
$\mathbb{S}' \leftarrow \mathbb{S}\|\text{s}_{j+1}\| \ldots \|\text{s}_n$
ensure $\text{ValidKeychain}(\mathbb{S}')$
$st.\text{Users}[u] \leftarrow (\mathbb{S}', \text{auth}_n, \mathbf{t}(st))$
return $(st, \mathbb{S}')$

Server:
$\mathbb{S} \leftarrow \text{st}^S.\text{Users}^S[u]$
receive t^{last}
ensure $t^{\text{last}} \leq t$ and $t \leq \mathbf{t}(\text{st}_{\text{RZKS}})$, else **send** ERROR and **return** ERROR.
$j' \leftarrow |\{\text{s} : \text{s} \in \mathbb{S} \wedge \mathbf{t}(\text{s}) \leq t^{\text{last}}\}|$
$n \leftarrow |\{\text{s} : \text{s} \in \mathbb{S} \wedge \mathbf{t}(\text{s}) \leq t\}|$
For $i = j'+1, \ldots, n+1$:
  $\text{label}_i \leftarrow (u, i)$
  $(\pi_i, \text{val}_i, t_i) \leftarrow \text{RZKS.Query}(\text{pp}, \text{st}^S.\text{st}_{\text{RZKS}}, t, \text{label}_i)$
send $\{(\mathbb{S}[i], \text{val}_i, \pi_i)\}_{i=j'+1}^{n+1}$ (with $\mathbb{S}[n+1] = \perp$)

MVKD.VerExtension$(st, \text{com} \,;\, \text{st}^S, t', t'')$:

Client:
receive (com', π)
If $\mathbf{com}(\text{st}) = \perp$:
  $b \leftarrow 1$
Else if $\mathbf{t}(st) < \mathbf{t}(\text{com})$:
  $b \leftarrow \text{RZKS.VerExt}(\text{pp}, \mathbf{com}(st), \text{com}, \pi)$
Else:
  $b \leftarrow \text{RZKS.VerExt}(\text{pp}, \text{com}, \mathbf{com}(st), \pi)$
If $\text{com}' \neq \text{com}$:
  $b \leftarrow 0$
If $b = 1$ and ($\mathbf{t}(st) < \mathbf{t}(\text{com})$ or $\mathbf{com}(st) = \perp$):
  $\mathbf{com}(st) \leftarrow \text{com}$
return (st, b)

Server:
If $t' = \perp$:
  $\pi \leftarrow \perp$
Else if $t' < t''$:
  $\pi \leftarrow \text{RZKS.ProveExt}(\text{pp}, \text{st}^S.\text{st}_{\text{RZKS}}, t', t'')$
Else:
  $\pi \leftarrow \text{RZKS.ProveExt}(\text{pp}, \text{st}^S.\text{st}_{\text{RZKS}}, t'', t')$
Retrieve $(\text{com}, \cdot) \leftarrow \text{st}^S.\text{Com}[t'']$
send (com, π)

MVKD.Audit$(\text{com}_t, \text{com}_{t+1} \,;\, \text{st}^S, t')$:

Auditor:
receive π
return $\text{RZKS.VerifyUpd}(\text{pp}, \text{com}_t, \text{com}_{t+1}, \pi)$

Server:
$(\text{com}, \pi) \leftarrow \text{st}^S.\text{Com}[t'+1]$
send π

MVKD.Sign(st, m) :

return $\text{SIG.Sign}(\mathbf{sk}(st), 1\|m)$

MVKD.Verify(pk, m, σ) :

return $\text{SIG.Ver}(pk, 1\|m, \sigma)$

Figure 5 (MVKD Protocol): Detailed MVKD **protocol description. Procedures marked by an asterisk are internal predicates.**

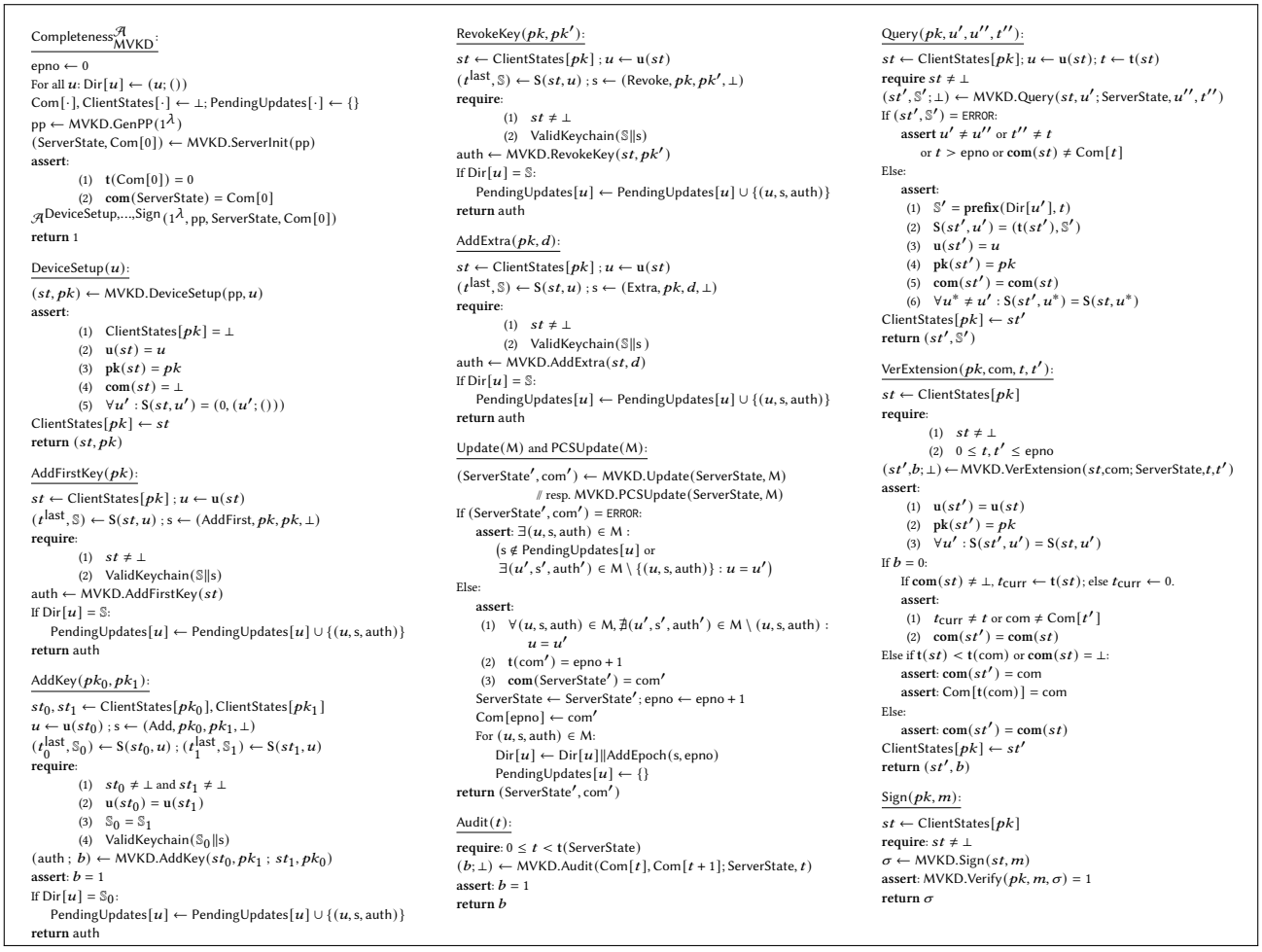


Figure 6: The MVKD **completeness experiment. The advantage of $\mathcal{A}$ is defined as the probability of the experiment returning** ERROR **by $\mathcal{A}$ triggering an assertion.**

ERROR only if there has been a mismatch with the server's view, i.e., either the input usernames do not match, the epoch input to the server does not correspond to the client's epoch, the client's epoch is greater than the server's current epoch, or the device uses a commitment that has never been output by the server. The device can further be instructed to update to the latest commitment using the VerExtension oracle. The game ensures that the algorithm only rejects if the device's and server's states or inputs mismatch. If the algorithm accepts, the game verifies that the commitment in the device's state is updated as expected.

Audits. The Audit oracle verifies that external auditors can successfully audit two consecutive commitments produced by the honest server.

Signatures. Finally, the game ensures that the associated signature is correct. That is, the Sign oracle verifies that a signature produced by the respective algorithm verifies under the device's public key.

F PROOF OF THEOREM 1

Sketch. The proof mainly follows by inspection. For instance, for the initial server state we observe that $\mathbf{com}$(ServerState) = Com[0] and $\mathbf{t}$(ServerState) immediately follow by completeness of the RZKS scheme. Analogously, the assertions in DeviceSetup mostly follow by inspection of the protocol, with the freshness of the public key following by correctness and unforgeability of the signature scheme. In the same vein, the assertion in Sign follows by correctness of the signature scheme and the signing key staying constant throughout the execution of the protocol.

Consider now Update (and PCSUpdate). Here, observe that the server accepts iff there is at most one update per user, all resulting keychains are valid (ValidKeychain), and their authenticators form a valid sigchain (CheckAuth). Hence, to conclude that Update (or PCSUpdate) does not erroneously abort, we observe that whenever PendingUpdates stores a keychain statement it is guaranteed that appending this statement does lead to a valid keychain (by definition of the game) and that the authenticator is valid (by correctness of the signature scheme and for the user's keychain not having changed since the statement having been produced). Next, observe that at the end of Update (or PCSUpdate) the game's Dir[·] dictionary matches the server's protocol state and that the authenticators validated by the Update (and PCSUpdate) algorithm are stored in the RZKS. Whenever a device executes the Query algorithm to query for a user u',

we are, thus, guaranteed that the returned keychain is valid and matches $\mathbf{prefix}(u', t)$ (the prefix up to the queried epoch t), and that the authenticators sent to the device — which by correctness of the RZKS scheme are the same ones as the server stored — check out. Hence, the device will output the keychain stored in Dir, and all remaining assertions from Query follow by simple inspection of the protocol.

Finally, for VerExtension, as long as the device's and server's input match, the completeness directly follows by the completeness of RZKS with respect to the RZKS.ProveExt and RZKS.VerExt algorithms, and analogously for Audit.

□

G MVKD SOUNDNESS

In this section, we provide a formal description of the MVKD soundness experiment. In particular, we give a formal descriptions of the four properties (introduced in Section 4.2) that our soundness definition captures: resiliency, unforgeability, consistency, and persistency.

Recall that we formalize soundness as a real-ideal world indistinguishability game. The ideal experiment Soundness-IDEAL$^{\mathcal{A},\text{Extract}}_{\text{MVKD}}$ is depicted in Figure 7 and uses a stateful extractor Extract for the ideal world, providing the following functionality:

Soundness-IDEAL$^{\mathcal{A},\text{Extract}}_{\text{MVKD}}$:

HonKeys, KS, PS, Sigs ← {}
D[·], C[·], ClientStates[·] ← ⊥
pp, st ← Extract(Init, 1^λ)
$b \leftarrow \mathcal{A}^{\text{DeviceSetup,Compromise,...,AddExtra}}(1^\lambda, \text{pp})$
return b

Commit(com):

$\text{D}_{\text{com}}, \text{C}_{\text{com}} \leftarrow$ Extract(Extr, st, com)
assert:
(1) D[com] ≠ ⊥ ⇒ D[com] = D_{com}, C[com] = C_{com}
(2) $\forall((u,i),\mathbb{S}) \in \text{D}_{\text{com}}$:
(a) $|\mathbb{S}| = i$, $\mathbf{u}(\mathbb{S}) = u$
(b) $\mathbf{t}(\text{com}) \geq \mathbf{t}(\text{last}(\mathbb{S})) > 0$
(c) ValidKeychain($\mathbb{S}$)
(d) HonestKeychain($\mathbb{S}$)
(3) $|\text{C}_{\text{com}}| = \mathbf{t}(\text{com})$ or $\mathbf{t}(\text{com}) =$ ERROR
(4) last(C_{com}) = com
D[com] ← D_{com}, C[com] ← C_{com}

HonestKeychain($\mathbb{S}$): // helper function, not an oracle

return $\forall i \in \{1,\ldots,|\mathbb{S}|\}$:
(1) $\mathbf{pk_a}(\mathbb{S}[i]) \in \text{HonKeys} \Rightarrow \mathbb{S}[1\ldots i] \in^* \text{KS}$
(2) $(\mathbf{type}(\mathbb{S}[i]) = \text{Add}$ and $\mathbf{data}(\mathbb{S}[i]) \in \text{HonKeys})$
$\Rightarrow \mathbb{S}[1\ldots i] \in^* \text{PS}$

DeviceSetup(u):

$(st, pk) \leftarrow$ MVKD.DeviceSetup(pp, u)
assert:
(1) ClientStates[pk] = ⊥
(2) $\mathbf{u}(st) = u$ and $\mathbf{pk}(st) = pk$
(3) $\mathbf{com}(st) = \bot$
(4) $\forall u' : \text{S}(st, u') = (0, (u'; ()))$
ClientStates[pk] ← st
HonKeys ← HonKeys ∪ {pk}
return pk

Query(pk, u):

$st \leftarrow$ ClientStates[pk] ; com ← $\mathbf{com}(st)$
require: $st \neq \bot$
$(t^{\text{last}}, \mathbb{S}) \leftarrow \text{S}(st, \mathbf{u}(st))$
$(st', \mathbb{S}'\ ;\ \bot) \leftarrow$ MVKD.Query($st, u\ ;\ \mathcal{A}$)
If $(st', \mathbb{S}') =$ ERROR: **return** ERROR
assert:
(1) extends($\mathbf{t}(st'), \mathbb{S}', t^{\text{last}}, \mathbb{S}$)
(2) $\mathbf{u}(st') = \mathbf{u}(st)$ and $\mathbf{pk}(st') = \mathbf{pk}(st)$ and $\mathbf{com}(st') =$ com
(3) $\text{S}(st', u) = (\mathbf{t}(st'), \mathbb{S}')$ and $\mathbf{u}(\mathbb{S}') = u$
(4) $\forall u' \neq u : \text{S}(st', u') = \text{S}(st, u')$
(5) For $i = |\mathbb{S}|+1, \ldots, |\mathbb{S}'|$:
$\mathbb{S}'[1\ldots i] = \text{D}[\text{com}][(u,i)]$
(6) $\text{D}[\text{com}][(u, |\mathbb{S}'|+1)] = \bot$
ClientStates[pk] ← st'
return $\mathbb{S}'$

Compromise(pk):

HonKeys ← HonKeys \ {pk}
return ClientStates[pk]

Audit($\text{com}_a, \text{com}_b$):

$(b\ ;\ \bot) \leftarrow$ MVKD.Audit($\text{com}_a, \text{com}_b\ ;\ \mathcal{A}$)
require: $b = 1$ and $\text{D}[\text{com}_a] \neq \bot$ and $\text{D}[\text{com}_b] \neq \bot$
assert:
(1) $\mathbf{t}(\text{com}_a) \neq$ ERROR and $\mathbf{t}(\text{com}_b) = \mathbf{t}(\text{com}_a) + 1$
(2) $\text{C}[\text{com}_b] = \text{C}[\text{com}_a] \| \text{com}_b$
(3) $\text{D}[\text{com}_a] \subseteq \text{D}[\text{com}_b]$
(4) $\forall((u,i),\mathbb{S}) \in \text{D}[\text{com}_b] \setminus \text{D}[\text{com}_a]$:
$\mathbf{t}(\text{last}(\mathbb{S})) = \mathbf{t}(\text{com}_b)$
(5) $\mathbf{t}(\text{com}_a) > 0$ or $\text{D}[\text{com}_a] = \{\}$

AddFirstKey(pk):

require: $pk \in$ HonKeys
$st \leftarrow$ ClientStates[pk] ; $(t^{\text{last}}, \mathbb{S}) \leftarrow \text{S}(st, \mathbf{u}(st))$
auth ← MVKD.AddFirstKey(st)
If auth ≠ ERROR: KS ← KS ∪ {$\mathbb{S}\|$(AddFirst, $pk, pk, \bot$)}
return auth

AddKey-Left(pk_0, pk_1):

require: $pk_0 \in$ HonKeys
$st_0 \leftarrow$ ClientStates[pk_0] ; $(t^{\text{last}}, \mathbb{S}) \leftarrow \text{S}(st_0, \mathbf{u}(st_0))$
auth ← MVKD.AddKey($st_0, pk_1\ ;\ \mathcal{A}$)
If auth ≠ ERROR : KS ← KS ∪ {$\mathbb{S}\|$(Add, $pk_0, pk_1, \bot$)}
return auth

AddKey-Right(pk_1, pk_0):

require: $pk_1 \in$ HonKeys
$st_1 \leftarrow$ ClientStates[pk_1] ; $(t^{\text{last}}, \mathbb{S}) \leftarrow \text{S}(st_1, \mathbf{u}(st_1))$
$b \leftarrow$ MVKD.AddKey($\mathcal{A}\ ;\ st_1, pk_0$)
If $b = 1$: PS ← PS ∪ {$\mathbb{S}\|$(Add, $pk_0, pk_1, \bot$)}
return b

RevokeKey(pk, pk_R):

require: $pk \in$ HonKeys
$st \leftarrow$ ClientStates[pk] ; $(t^{\text{last}}, \mathbb{S}) \leftarrow \text{S}(st, \mathbf{u}(st))$
auth ← MVKD.RevokeKey(st, pk_R)
If auth ≠ ERROR : KS ← KS ∪ {$\mathbb{S}\|$(Revoke, $pk, pk_R, \bot$)}
return auth

AddExtra(pk, d):

require: $pk \in$ HonKeys
$st \leftarrow$ ClientStates[pk] ; $(t^{\text{last}}, \mathbb{S}) \leftarrow \text{S}(st, \mathbf{u}(st))$
auth ← MVKD.AddExtra(st, d)
If auth ≠ ERROR : KS ← KS ∪ {$\mathbb{S}\|$(Extra, $pk, d, \bot$)}
return auth

VerExtension(pk, com'):

$st \leftarrow$ ClientStates[pk] ; com ← $\mathbf{com}(st)$
require: $st \neq \bot$ and $\text{D}[\text{com}'] \neq \bot$
$(st', b\ ;\ \bot) \leftarrow$ MVKD.VerExtension($st, \text{com}'\ ;\ \mathcal{A}$)
assert:
(1) $\mathbf{u}(st') = \mathbf{u}(st)$ and $\mathbf{pk}(st') = \mathbf{pk}(st)$
(2) $\forall u : \text{S}(st', u) = \text{S}(st, u)$
If $b = 1$ and ($\mathbf{t}(\text{com}') > \mathbf{t}(\text{com})$ or com = ⊥) :
assert: $\mathbf{com}(st') = \text{com}'$
Else:
assert: $\mathbf{com}(st') =$ com
If $b = 1$ and com ≠ ⊥:
For $j = 1, \ldots, \min(\mathbf{t}(\text{com}), \mathbf{t}(\text{com}'))$
assert: $C[\text{com}][j] = C[\text{com}'][j]$
ClientStates[pk] ← st'

Sign(pk, m):

$st \leftarrow$ ClientStates[pk]
require: $st \neq \bot$
$\sigma \leftarrow$ MVKD.Sign(st, m)
assert: MVKD.Verify(pk, m, σ) = 1
Sigs ← Sigs ∪ {(pk, m, σ)}
return σ

Forgery(pk, m, σ):

require: MVKD.Verify(pk, m, σ) = 1
assert: $pk \notin$ HonKeys or $(pk, m, \sigma) \in$ Sigs

Ideal(in):

out, st ← Extract(Ideal, st, in)
return out

Figure 7: The ideal MVKD **Soundness experiment.**

- pp, st ← Extract(Init, 1^λ): Samples public parameters (meant to be indistinguishable from honestly generated ones) to be given as input to the adversary, and initializes the extractor's state.
- out, st ← Extract(Ideal, st, in): Implements any ideal functionalities (such as Random Oracles) that the scheme depends on, and can also update its own state when answering these queries.
- $\text{D}_{\text{com}}, \text{C}_{\text{com}} \leftarrow$ Extract(Extr, st, com): On input a commitment com and its own state st, the extractor outputs a map D_{com} and a list C_{com}. D_{com} maps a username-integer pair (u, e) to a keychain for u with $e > 0$ statements. Intuitively, this map restricts what an adversary can force an honest client who holds com to output when querying for u, as we detail later. C_{com} is a list of $|\mathbf{t}(\text{com})|$ commitments, meant to correspond to previous epochs of the data structure that com commits to. Note that, when answering

these queries, the extractor cannot update its state. If com is malformed, $\mathsf{D}_{\mathsf{com}}$ can be an empty set and $\mathsf{C}_{\mathsf{com}}$ (or some of its list elements) can be ERROR.

The real experiment Soundness-REAL$^{\mathcal{A}}_{\mathsf{MVKD}}$ (not pictured) is defined similarly to the ideal one, with the following modifications: all assertions are removed (hence, triggering one in the ideal world makes the two worlds immediately distinguishable) and there is no extractor. Instead, the public parameters pp are sampled honestly at the beginning, the Ideal oracle is implemented according to the specification of each idealized functionality, and Commit queries on input com simply set $\mathsf{D}[\mathsf{com}] \leftarrow \varepsilon$ so that **require** statements have analogous behavior in both experiments. If no assertions are triggered, $\mathcal{A}$ can terminate its execution by outputting a bit, which is taken as the output of the experiment.

Definition 9. *We say that a* MVKD *satisfies Soundness (w.r.t. a set of idealized functionalities), if for any PPT adversary* $\mathcal{A}$*,*

$$\left|\Pr\left[\text{Soundness-IDEAL}^{\mathcal{A},\mathsf{Extract}}_{\mathsf{MVKD}} = 1\right] - \Pr\left[\text{Soundness-REAL}^{\mathcal{A}}_{\mathsf{MVKD}} = 1\right]\right| \leq \mathsf{negl}(\lambda)\,.$$

Formalizing the soundness game using indistinguishability ensures that the extractor must sample the public parameters and implement idealized functionalities in a way that is computationally indistinguishable from the real world. When the game itself executes one of the MVKD algorithms, that algorithm can also (implicitly) make Ideal oracle queries (say, to evaluate a Random Oracle hash). We limit $\mathcal{A}$ to not make more than one oracle call at a time involving the same honest device; this is acceptable as an honest device will only participate in one procedure at a time in practice.

Honest Devices. As in the completeness game, the adversary is allowed to create (honest) devices using the DeviceSetup oracle. Here, however, the adversary does not directly learn those devices' states, but instead can later corrupt them (using the Compromise oracle), with the game keeping track of all uncompromised keys in the HonKeys set. Whenever $\mathcal{A}$ uses an oracle to interact with an honest device in a way that updates the device's state, the game asserts that the components of the state which are not expected to change stay constant (which formalizes part of *resiliency*). Examples include assertions 2 and 4 in Query or 1 and 2 in VerExtension.

Signatures. The adversary can request honest devices to sign arbitrary messages m using the Sign oracle. The oracle verifies that the resulting signature is actually valid, enforcing that a malicious server cannot eradicate a device's ability to use their key (also part of *resiliency*). At the same time, the game also ensures that such signatures cannot be forged, allowing the adversary to win by submitting a valid forgery for an honest device's key to the Forgery oracle (*unforgeability* of signatures).

Commit oracle. Before the adversary can ask any client to update their commitment to com, the adversary has to "announce" it by invoking the Commit oracle (VerExtension requires $\mathsf{D}[\mathsf{com}'] \neq \bot$). This ensures that the extractor is executed before clients can perform any queries (or update their commitments), and therefore that we can test whether what the clients output matches the output of the extractor. As specified above, the extractor outputs a map $\mathsf{D}_{\mathsf{com}}$ of keychains, and a list of commitments to the previous epochs that com binds to. This oracle enforces some basic consistency properties of the extracted values: that the output is the same when the extractor is called on the same input com (but a potentially different state) more than once; that all keychains[9] in $\mathsf{D}_{\mathsf{com}}$ are valid and have the expected length; that the list $\mathsf{C}[\mathsf{com}]$ has the appropriate length and ends with com.

Querying the directory. The Query oracle formalizes the main guarantees of the soundness property. The adversary wins if, when interacting with an honest client in a Query for user u's keychain, it causes the client to either:

- output a keychain which does not extend the one priorly stored as part of the client's state, addressing *persistency* from a client's point of view, or
- output a keychain which does not match the output of the extractor, except for the following caveats, addressing *consistency*.

To formalize consistency, the Query oracle uses the following definition.

Definition 10. *Let* $\mathbb{S}_1, \mathbb{S}_2$ *be two keychains. We say that* $\mathbb{S}_2$ *extends* $\mathbb{S}_1$*, denoted* $\mathsf{extends}(\mathbb{S}_2, \mathbb{S}_1)$*, if* $\mathbf{u}(\mathbb{S}_2) = \mathbf{u}(\mathbb{S}_1)$*, and there exist statements* $\mathsf{s}_1, \ldots, \mathsf{s}_n$ *such that* $\mathbb{S}_2 = \mathbb{S}_1 \| \mathsf{s}_1 \| \ldots \| \mathsf{s}_n$*. We further say that keychain* $\mathbb{S}_2$ *extends* $\mathbb{S}_1$ *between epochs* t_1 *and* t_2*, denoted* $\mathsf{extends}(t_2, \mathbb{S}_2, t_1, \mathbb{S}_1)$ *iff* $\mathsf{extends}(\mathbb{S}_2, \mathbb{S}_1)$ *and for each* $j \in \{|\mathbb{S}_1|+1, \ldots, |\mathbb{S}_2|\}$ *it holds that* $\mathbf{t}(\mathsf{last}(\mathbb{S}_1)) \leq t_1 < \mathbf{t}(\mathbb{S}_2[j]) \leq t_2$.

Recall from Section 4.2 that we formalize a slightly weakened consistency property that accommodates the efficiency requirements of practical protocols. Simply put, we want protocols to have complexity that scales linearly in the number of changes to $\mathbb{S}$ since it was last queried given that, for most users, keychain updates are infrequent compared to queries.

Our protocol (cf. Section 3) for instance caches the keychain and only requests newly added links from the server, plus a proof that no further statement with index $|\mathbb{S}'| + 1$ exists. This protocol is, however, susceptible to the following attack: The adversary publishes a keychain for user u with 2 statements at epoch t, then extend it with a 3rd statement at epoch $t + 1$, while also replacing the first statement with something malicious and completely removing the 2nd statement from the RZKS.[10] Consequently, a client who queries for u at epoch t and then at epoch $t + 1$ would output the chain with all 3 honest statements. However, a different client who queries for u at epoch $t + 1$ for the first time (with the same commitment) would output a keychain with only the first malicious statement since, in our protocol, the server could provide an inclusion proof for that first statement and a non-membership proof that the directory contains no second statement.

As a consequence, we formalize the following weaker guarantees. Consider two users u_1 and u_2 that both query for u's keychain successfully (i.e., do not output an error) while holding the same commitment com. Let k and ℓ denote the length of their cached keychains for u, respectively.

[9]We provide more intuition for assertion 2 on HonestKeychain in the paragraph about keychain authentication.

[10]Note that this update is removing and altering existing data from the directory, so the adversary couldn't make Audit succeed between the two epochs. We, however, wish to formalize guarantees even if auditing is not performed.

- If $k = \ell$, then we enforce that either both users output the same keychain.
- Otherwise, if without loss of generality $k < \ell$, the adversary has two options: Either both users can output the same keychain, or u_1 can output one of length at most $\ell - 1$ that is inconsistent, i.e. not even necessarily a prefix with what u_2 outputs (or has output in prior queries).

To this end, the Query oracle enforces that for each new i-th statement output by a user ($k < i \leq |\mathbb{S}'|$, where k is the length of the cached keychain as above), the full prefix of the keychain until that statement must match the output of the extractor for that length, i.e., that $\mathbb{S}'[1 \ldots i] = \mathsf{D}_{\mathsf{com}}[(u, i)]$. Moreover, the oracle asserts that $\mathsf{D}_{\mathsf{com}}[(u, |\mathbb{S}'| + 1)] = \perp$, which confirms that the server isn't hiding any extensions (and matches the absence proof requested by clients in our protocol).

These assertions capture the property described above: if u_1 outputs a new link with index $i > \ell$, then that link must match the extractor and therefore u_2's output, and thus both oracle queries will execute the same checks on the entire prefixes, ensuring consistency.

Remark 1. *A trivial modification of our protocol without caching (where all clients always query for all statements until the first one which is proven not to be part of the directory) would satisfy the stronger consistency notion. Moreover, the two definitions offer the same guarantees if we assume all commitments have been audited (more precisely, if we assume that for each commitment* com *held by a client, there is a chain of successful audits starting at a commitment for epoch 0 and ending at* com*) as discussed below.*

Auditing. The Audit oracle ensures that, if $\mathcal{A}$ makes the MVKD.Audit on input com_a and com_b succeed, then persistency is satisfied. This, in particular, also ensures a stronger notion of consistency: By the time u_1 verifies that $\mathbb{S}'[1 \ldots i] = \mathsf{D}_{\mathsf{com}}[(u, i)]$, even for $i < \ell$, persistency ensures that this is equal to $\mathsf{D}_{\mathsf{com}'}[(u, i)]$ where com′ refers to the commitment u_2 had when retrieving and caching the i-th statement.

Keychain authentication. Again, akin to the completeness game, the adversary can instruct honest devices to authorize extending their keychain (through oracles AddFirstKey, RevokeKey, AddExtra, AddKey-Left and AddKey-Right). Importantly, for soundness the game does not sanitize inputs and requires the protocol to not violate security (but e.g. reject those inputs) even when instructed to perform bogus actions. If the device does accept the action, we then append the resulting keychain (obtained by appending the new action to the keychain currently in the device's state) to the KS set that tracks all keychains whose last statement has been authorized by the appropriate honest device. Note that the last statement in these chains has no epoch, as the server is expected to fill those in during an update.

Some comments are due for the AddKey algorithm. Since AddKey statements need to be authorized by both the device being added to a chain, and the one doing the addition (interacting with each other), the game exposes two oracles: AddKey-Left and AddKey-Right. In the former, the honest device pk_0 interacts with the adversary (for adding pk_1 which may or may not be honest) while in the latter the client pk_1 interacts in the role of the device being added with the adversary. We record approval of the device doing the addition in KS (as in all other cases), and the approval of the device being added in a dedicated set PS. We stress that since the adversary can interleave those oracle calls — whenever the interactive algorithm requires input from the adversary, they might instead call another oracle — the adversary can simply forward the respective messages. We do not make any assumption on the network model and account for the adversary potentially tampering with the messages or even trying to completely impersonate one of the parties. In addition, the game does not enforce that the two interactions happened in lockstep or even at the same time — which we consider to be outside the scope of this work. Such stronger liveness properties are conceivable and could e.g., be achieved by a protocol first establishing a secure channel between the two involved devices using their respective public keys.

During Commit oracle calls, the HonestKeychain assertion enforces that all keychains output by the extractor and whose last statement is expected to be authorized by a device which is honest *at that point in time* are part of KS and/or PS. The assertion uses a special $\in^*$ operator, which accounts for the fact that the adversary can choose the epoch on which updates in KS and PS are published, defined as follows:

Definition 11. *Two keychains $\mathbb{S}_1, \mathbb{S}_2$ are **weakly equal**, also denoted $\mathbb{S}_1 =^* \mathbb{S}_2$, if $\mathbf{u}(\mathbb{S}_1) = \mathbf{u}(\mathbb{S}_2)$; $|\mathbb{S}_1| = |\mathbb{S}_2| = \ell$ for some non-zero integer ℓ; $\forall i \in [\ell - 1]$, $\mathbb{S}_1[i] = \mathbb{S}_2[i]$; and $\mathbb{S}_1[\ell]$ and $\mathbb{S}_2[\ell]$ are equal except for the last component t. A keychain **weakly belongs** to a set $\mathcal{S}$ of keychains, denoted $\mathbb{S} \in^* \mathcal{S}$, if there exists $\mathbb{S}' \in \mathcal{S}$ such that $\mathbb{S} =^* \mathbb{S}'$.*

Observe that HonestKeychain formalizes a strong *unforgeability* property, and is one of the main advantages of stating a single soundness definition combining extractability and authorization. Checking unforgeability at the time of a commitment being announced — rather than upon the first use a keychain by an honest party — allows to formalize the following intuitive property: assume $\mathcal{A}$ publishes a commitment com today, expecting to corrupt a device pk tomorrow. Even after the corruption, $\mathcal{A}$ cannot convince an honest device holding com to output a forged keychain statement from pk.

Updating client commitments. The VerExtension oracle ensures that, if MVKD.VerExtension succeeds, then the list $\mathsf{C}[\mathsf{com}_n]$ extends $\mathsf{C}[\mathsf{com}_o]$, where com_o and com_n are respectively the commitments with the least and most recent epoch among the input com and the client state $\mathbf{com}(\mathsf{st})$. If the client state has no commitment, the check is skipped.

This ensures that the client can safely forget the old commitment, and keep only the most recent one, without losing the option to audit any past commitments. More in detail, by comparing com with an auditor who has audited the directory since its inception, the client implicitly ensures that every other older commitment they have been given must also match what the auditor checked. (Obviously, the client can also ask the server for all previous commitments and perform this audit themselves.) This ensures that properties enforced by Audit (such as that the server never removed anything from the directory) would hold for all those past commitments.

H PROOF OF THEOREM 2

Proof. This proof is partially adapted from the proof of RZKS Soundness (Theorem 4) in [8]. Let RZKS.Extract be the extractor from the RZKS soundness definitions. We can construct an extractor MVKD.Extract for the MVKD soundness game as follows:

MVKD.Extract :

- Internally run an instance of RZKS.Extract (with state RZKS.st). Set MVKD.st = (RZKS.st, H), where H is a map used to lazily implement the random oracle hash. If H ever contains collisions, i.e. for any two distinct x, x', we have $(x, y), (x', y) \in H$, the simulator will abort (i.e. it returns ERROR to all further queries). Therefore, when $(x, y) \in H$, we can write $y = H(x)$ and $x = H^{-1}(y)$.
- To facilitate the security analysis, we describe the extractor as keeping track of an additional set C0, which is updated both during Ideal and Extr queries. Note that, according to the definition, the extractor is not allowed to update its state (and therefore the set C0) during Extr queries. However, the extractor actually never reads from this set, so this detail can be ignored until it is leveraged in the analysis. Looking ahead, C0 will contain all the values which we do not want hash to sample.
- To handle Extract(Ideal, st, in):
  - **parse** st as (RZKS.st, H)
  - If in encodes a query for one of the RZKS ideal objects, then invoke the RZKS extractor to compute (out, RZKS.st′) ← RZKS.Extract(Ideal, RZKS.st, in). Afterwards, output out, (RZKS.st′, H)
  - Otherwise, if in encodes a hash query on input x:
    - If $H(x)$ is already defined, return it.
    - If x can be parsed as (auth′, t', u', Type, pk, d), and auth′ can be parsed as (h, z), then C0 ← C0 ∪ $\{h\}$
    - Sample $y \leftarrow\!\!\$\ \{0,1\}^\lambda$ uniformly at random. (If the same value was previously sampled, abort.)
    - Add (x, y) to H, y to C0.
    - **return** y, (RZKS.st, H).
- Define the helper function UnrollChain(H, auth, u, l, t) :
  - $S \leftarrow ()$
  - While auth ≠ ε:
    - **parse** auth as (h, z); add h to C0.
    - **parse** $H^{-1}(h)$ as (auth′, t', u', Type, pk, d)
    - **ensure** $u' = u$; let s ← (Type, pk, d, t)
    - **ensure** CheckAuth(auth′, t', u, s, auth) = 1
    - $S \leftarrow$ s‖S, auth ← auth′, $t \leftarrow t'$
  - **ensure** $|S| = l$ and $t = 0$
  - $\mathbb{S} \leftarrow (u; S)$
  - **ensure** ValidKeychain($\mathbb{S}$) = 1
  - **return** $\mathbb{S}$
- To handle Extract(Extr, st, com):
  - **parse** st as (RZKS.st, H)
  - $\mathsf{D}'_{\mathsf{com}} \leftarrow$ RZKS.Extract(Extr, RZKS.st, com)
  - $\mathsf{C}_{\mathsf{com}} \leftarrow$ RZKS.Extract(ExtrC, RZKS.st, com)
  - $\mathsf{D}_{\mathsf{com}} = \{\}$; For each (lbl, auth, t) ∈ $\mathsf{D}'_{\mathsf{com}}$:
    - **parse** lbl as (u, l), else **continue**.
    - $\mathbb{S} \leftarrow$ UnrollChain(H, auth, u, l, t).
    - If $\mathbb{S} \neq$ ERROR, set $\mathsf{D}_{\mathsf{com}}[(u, l)] \leftarrow \mathbb{S}$
  - **return** $\mathsf{D}_{\mathsf{com}}, \mathsf{C}_{\mathsf{com}}$

We want to prove that for all adversaries, the two $\mathsf{Sound}_{\mathsf{MVKD}}$ games instantiated with MVKD.Extract are computationally indistinguishable for any adversary $\mathcal{A}$. To do so, let's consider a sequence of hybrids:

- *Hyb*0. This is defined as Soundness-IDEAL$^{\mathcal{A},\mathsf{MVKD.Extract}}_{\mathsf{MVKD}}$.
- *Hyb*1. Defined as *Hyb*0, but we substitute HonestKeychain with a function that always returns 1 (i.e., it never triggers the assertion in Commit), remove all assertions from the Forgery and Sign oracles, and remove assertion 1 from DeviceSetup.
- *Hyb*2. Defined as the previous hybrid, but we remove all the assertions which are already checked by the MVKD algorithms in our construction, or those that cannot be triggered (information theoretically) because of how our extractor is defined. More precisely:
  - In the Commit oracle, we remove assertions 2a, 2c.
  - In DeviceSetup, we remove all (remaining) assertions.
  - In Query, we remove assertions 1 to 4.
  - In VerExtension, we remove all assertion except for the last one (i.e. $\mathsf{C}[\mathsf{com}][j] = \mathsf{C}[\mathsf{com}'][j]$)
- *Hyb*3. Defined as *Hyb*2, but the extractor keeps an additional map $\mathsf{D}_{\mathsf{RZKS}}$ to track the output of the RZKS extractor and make sure it is consistent across multiple calls (we stress that in *Hyb*2 the extractor is not allowed to update its state during Extr queries). More in detail:
  - During a Commit oracle call, let $\mathsf{D}'_{\mathsf{com}} \leftarrow$ RZKS.Extract(Extr, RZKS.st, com) be the output of the MVKD extractor (as in *Hyb*2). We add an assertion 1′: $\mathsf{D}_{\mathsf{RZKS}}[\mathsf{com}] \neq \perp \Rightarrow \mathsf{D}_{\mathsf{RZKS}}[\mathsf{com}] = \mathsf{D}'_{\mathsf{com}}, \mathsf{C}[\mathsf{com}] = \mathsf{C}_{\mathsf{com}}$. We also add assertion 2b′: $\forall((u, i), (\mathsf{auth}_i, t_i)) \in \mathsf{D}'_{\mathsf{com}}$, it must be $t_i \in \mathbb{N}$ and $\mathbf{t}(\mathsf{com}) \geq t_i > 0$. Then, at the end of the oracle query, we also set $\mathsf{D}_{\mathsf{RZKS}}[\mathsf{com}] \leftarrow \mathsf{D}'_{\mathsf{com}}$.
  - During Query oracle calls we add two extra assertion 5′ and 6′. Assertion 5′ requires that for $i = |\mathbb{S}|+1, \ldots, |\mathbb{S}'|$: $\mathsf{D}_{\mathsf{RZKS}}[\mathsf{com}][(u, i)] = (\mathsf{auth}_i, \mathbf{t}(\mathsf{s}_i))$, where $\mathsf{auth}_i, \mathsf{s}_i$ are the values sent by the adversary to the honest client (controlled by the game) as part of the execution of MVKD.Query. Assertion 6′ enforces that $\mathsf{D}_{\mathsf{RZKS}}[\mathsf{com}][(u, |\mathbb{S}'| + 1)] = \perp$.

  - During Audit queries, we add two extra assertion $3'$ and $5'$, which are analogous to 3 and 5 but with $\mathsf{D}_{\mathsf{RZKS}}[\mathsf{com}]$ instead of $\mathsf{D}[\mathsf{com}]$. Moreover, we add an assertion $4'$: $\forall((u,i),(\mathsf{auth}_i,t)) \in \mathsf{D}[\mathsf{com}_b] \setminus \mathsf{D}[\mathsf{com}_a] : t = \mathbf{t}(\mathsf{com}_b)$
- *Hyb*4. Defined as the previous hybrid, except:
  - `Ideal` queries for hash are answered with a "real" implementation of a random oracle (i.e., this oracle no longer aborts in case of collisions). The game no longer keeps track of the map D (but still keeps $\mathsf{D}_{\mathsf{RZKS}}$), so all assertions related to it are removed, as detailed below.
  - During Commit oracle calls, MVKD.Extract ignores the output of UnrollChain, and we remove assertion 1 and $2b$ (but keep $1'$ and $2b'$).
  - During Query oracle calls, we remove assertions 5 and 6 (but keep $5'$ and $6'$).
  - During Audit queries, we remove assertions 3 to 5 (but keep $3'$ to $5'$).
- *Hyb*5. This is defined as $\mathsf{Soundness\text{-}REAL}^{\mathcal{A}}_{\mathsf{MVKD}}$.

Any adversary distinguishing the first from the last game with non-negligible advantage must also have non-negligible advantage in distinguishing a couple of consecutive hybrids.

*Hyb*0 $\approx$ *Hyb*1**.** Any adversary who can distinguish *Hyb*0 from *Hyb*1 must trigger one of the assertions in *Hyb*0 which have been removed in *Hyb*1, and can therefore be used to break the correctness or unforgeability of the underlying signature scheme, with a straightforward reduction.

In particular, the assertion in Sign simply mandates that honestly generated signatures pass verification, which follows from the correctness of the signature scheme (note that in our scheme there is no way that an interaction with the adversary would cause an honest client to alter their signing keys after they are honestly generated).

Assume by contradiction that the adversary $\mathcal{A}$ can cause either the Forgery oracle to trigger an assertion, or the HonestKeychain function to return 0 in *Hyb*0, with better than negligible probability. We will show how to build an adversary $\mathcal{B}$ that leverages $\mathcal{A}$ to break the unforgeability of SIG.

Assume $\mathcal{A}$ triggers an assertion as a result of a Forgery query (pk^*, m^*, σ^*). Since $pk^* \in \mathsf{HonKeys}$, this key must have been returned to $\mathcal{A}$ as part of a DeviceSetup query. $\mathcal{B}$ runs $\mathcal{A}$, simulating an execution of the soundness game. $\mathcal{B}$ guesses which of the (polynomially many) setup queries will lead to the public key that $\mathcal{A}$ will use in its forgery, and instead of generating it honestly, $\mathcal{B}$ uses the one received from its unforgeability challenger. Accordingly, whenever a query from the adversary would require the game to produce a signature w.r.t. pk^* (either as part of Sign, AddKey-Left, AddKey-Right, AddFirstKey, RevokeKey or AddExtra queries), $\mathcal{B}$ uses its signing oracle to generate this signature. If $\mathcal{A}$ makes a compromise query for pk^*, or otherwise halts without making a forgery, $\mathcal{B}$ aborts. Otherwise, $\mathcal{B}$ can output the forgery $(0\|m^*, \sigma^*)$ to its own challenger and win the unforgeability game. Note that $\mathcal{A}$'s view in the simulation is the same as in an honest execution of the game, and moreover the $\mathcal{B}$'s output will be a valid forgery: the signature has to verify by construction, and it cannot have been returned to $\mathcal{B}$ by its own Sign oracle because all such signatures of messages which begin with 0 are part of the Sigs set (they are produced as a result of Sign queries), while the ones produced as part of other oracle calls begin with 1.

A similar argument can be used for the case where $\mathcal{A}$ causes HonestKeychain to return 0 in Commit. Our extractor guarantees that all prefixes of keychains which the extractor outputs come with valid signatures from the appropriate keys. In particular, CheckAuth (executed as part of UnrollChain by the extractor) ensures that the hash of the keychain prefix h has a valid signature from the appropriate (honest) public keys. Given that honest devices will only sign a keychain statement after having checked the whole keychain up until that statement (ensuring that all the h values inside auth have well defined preimages), and that hash is implemented without collisions by the extractor, if the adversary hasn't queried the appropriate honest oracles to obtain those signatures (which would cause the keychain prefixes to be added to KS or PS, contradicting the fact that the assertion is triggered), then those signatures must be forgeries, and we can construct an adversary $\mathcal{B}$ similar to the one of the previous case to exploit this.

Last, the assertion in DeviceSetup is only triggered if SIG.KeyGen can return the same public key twice. It is straightforward to see that if this happens with greater than negligible probability, then the signature scheme either cannot be correct or unforgeable: an adversary can just run the key generation algorithm multiple times. If the algorithm produces the same public key received from the challenger, either signatures simply don't verify under those keys or the adversary can use the corresponding secret key to trivially forge.

*Hyb*1 $\approx$ *Hyb*2**.** These two hybrids are perfectly indistinguishable, as the assertions that we are removing cannot actually be triggered in *Hyb*1:

- During Commit queries, the extractor checks internally that its output satisfies assertions 2a and 2c.
- In our construction, the state output by MVKD.DeviceSetup always satisfies the conditions asserted in the DeviceSetup oracle query.
- In Query, it must be that the first assertion $\mathsf{extends}\big(\mathbf{t}(st'), \mathbb{S}', t^{\mathsf{last}}, \mathbb{S}\big)$ cannot be triggered (or that, if it were to be triggered, assertion 5 would also be triggered simultaneously). Indeed, in our protocol MVKD.Query always outputs a keychain which extends the one in its state, ensuring that the epochs in each statement are monotonically increasing. In addition, MVKD.Query checks that the first new keychain link has an epoch greater than t^{last}. Finally, it must be that either the epoch of the last statement is $\leq \mathbf{t}(st')$, or assertion 5 is triggered: if assertion 5 is not triggered, it must be that $\mathbb{S}' = \mathsf{D}[\mathsf{com}][(u, |\mathbb{S}'|)]$. Consider the Commit oracle query that $\mathcal{A}$ made on input com before this one (if $\mathsf{com}(\mathsf{st}) = \mathsf{com}$, $\mathcal{A}$ must have made a successful query on input $\mathbf{pk}(\mathsf{st}), \mathsf{com}$ to VerExtension, and thus it must be $\mathsf{D}[\mathsf{com}] \neq \perp$). Either assertion 2b was triggered during that Commit query for $((u, |\mathbb{S}|), \mathbb{S})$ (in which case the game would have been halted earlier), or it must have been $\mathbf{t}(\mathsf{com}) \geq \mathbf{t}(\mathsf{last}(\mathbb{S}))$ with $\mathbf{com}(\mathsf{st}) = \mathsf{com}$, which is what we wanted.
- It is easy to check by inspection that assertions 2 to 4 in Query, as well as all assertions in VerExtension except for the last one, are never triggered by our instantiation of the protocol.

*Hyb*2 ≈ *Hyb*3. Assume by contradiction that there exists $\mathcal{A}$ that can distinguish *Hyb*2 from *Hyb*3. By construction, $\mathcal{A}$ needs to trigger one of the new assertions in *Hyb*3 with better than negligible probability. We can leverage $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the soundness of the RZKS. To do so, it is enough to construct $\mathcal{B}$ such that, if $\mathcal{A}$ triggers the new assertions in *Hyb*3, then $\mathcal{B}$ also triggers an assertion in Soundness-IDEAL$_{\mathsf{RZKS}}$ with at least the same probability. $\mathcal{B}$'s advantage in the RZKS soundness game bounds the probability that this assertion is triggered, which implies the former must also be negligible.

$\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of *Hyb*3, with the following modifications:

- $\mathcal{B}$ does not keep track of the RZKS extractor's state RZKS.st, but uses its own oracles instead. $\mathcal{B}$ does not enforce any assertions (beyond the ones its challenger might trigger as a result of $\mathcal{B}$'s queries), and keeps as internal state a set D' which keeps track of which commitments have been submitted to the Commit oracle. In addition, $\mathcal{B}$ does not keep track of the sets D and C, and all statements of the form "**require** $\mathsf{D}[\mathsf{com}] \neq \perp$" or "**require** $\mathsf{C}[\mathsf{com}] \neq \perp$" are replaced with "**require** $\mathsf{com} \in \mathsf{D}'$"
- Queries to the Ideal oracle for objects related to the RZKS are forwarded by $\mathcal{B}$ to its own challenger, while queries for hash are answered uniformly at random (aborting on collisions), and stored in a table H for consistency, as MVKD.Extract would.
- Commit oracle queries are handled by adding com to D', and making an ExtractD(com) and an ExtractC(com) oracle calls to its own challenger for the same value com, and returning.
- Query oracle calls are handled by $\mathcal{B}$ as in *Hyb*3, except that if MVKD.Query doesn't return ERROR, instead of the assertions, $\mathcal{B}$ makes a CheckVerD(**com**(st), label$_i$, auth$_i$, t_i, π_i) query to its challenger for each RZKS.Verify(pp, **com**(st), label$_i$, auth$_i$, t_i, π_i) operation performed during MVKD.Query as part of this oracle call.
- During Audit oracle queries, if MVKD.Audit returns b=1, instead of the assertions $\mathcal{B}$ queries its own challenger CheckVerUpdD(com$_a$, com$_b$, π), where π is the proof received from the adversary during MVKD.Audit. Then $\mathcal{B}$ returns.
- DeviceSetup, Compromise, AddFirstKey, AddKey-Left, AddKey-Right, RevokeKey, AddExtra, Sign, and Forgery oracle queries are handled by $\mathcal{B}$ as in *Hyb*2, but skipping all the assertions and replacing **require** statements as detailed above.
- When $\mathcal{A}$ halts, $\mathcal{B}$ halts with the same output.

Note that, when $\mathcal{B}$ is executed in Soundness-IDEAL$_{\mathsf{RZKS}}$, $\mathcal{A}$'s view up until the point where the game halts has the same distribution as in an execution of *Hyb*3. Indeed, ideal oracle queries for the RZKS idealized functionalities are answered by RZKS.Extract in both cases, either executed by the RZKS game in Soundness-IDEAL$_{\mathsf{RZKS}}$, or by the MVKD game (as part of MVKD.Extract) in *Hyb*3. Moreover, the state of the RZKS extractor has the same distribution (since it answers the same Ideal queries, while other queries do not change its state), and so its outputs are equally distributed too. Similarly, outputs of ideal oracle queries for hash have the same distribution, as they are sampled following the same algorithm, either by $\mathcal{B}$ or MVKD.Extract. Moreover, until the game is halted, all the answers given to $\mathcal{A}$ in response to its other oracle queries are sampled from exactly the same distribution. It is possible, since we are removing some assertions in $\mathcal{B}$'s simulation compared to *Hyb*3, that $\mathcal{A}$ might trigger an assertion there that is not enforced by $\mathcal{B}$, after which $\mathcal{A}$ might behave arbitrarily. However, here we only need to focus on those executions where one of the new assertions introduced in *Hyb*3 is triggered (which implies that none of the other assertions can be, since triggering an assertion ends the game).

It remains to show that if $\mathcal{A}$ triggers one of the newly added assertions in *Hyb*3, then $\mathcal{B}$ also triggers one in Soundness-IDEAL$_{\mathsf{RZKS}}$. To this end, first note that, by construction, the map $\mathsf{D}_{\mathsf{RZKS}}$ described in *Hyb*3 has the same content as the map D maintained by $\mathcal{B}$'s challenger, given that each map stores the output of RZKS.Extract queries in each game. If during a Commit query $\mathcal{A}$ triggers assertions $1'$ (but not 1) or $2b'$, then $\mathcal{B}$'s ExtractD query will cause the same assertion to be triggered in its game. Now, assume assertions $5'$ or $6'$ are triggered during a Query oracle call for some index i. By construction, since MVKD.Query doesn't return ERROR, it must be that RZKS.Verify succeeds in *Hyb*3. For each call to RZKS.Verify, $\mathcal{B}$ makes a call CheckVerD with analogous parameters, which causes the game to check if $\mathsf{D}[\mathsf{com}][(u,i)] = (\mathsf{auth}_i, \mathbf{t}(\mathsf{s}_i))$ (or $\mathsf{D}[\mathsf{com}][(u, |\mathbb{S}|+1)] = \perp$). Similarly, assertions $3'$, $4'$, and $5'$ in Audit queries would trigger analogous assertions during $\mathcal{B}$'s CheckVerUpdD call.

*Hyb*3 ≈ *Hyb*4. Denote with E the event that, during an execution of *Hyb*3 or *Hyb*4 and while answering an Ideal query for hash, we have that $y \in \mathsf{C0}$ (where y is the value sampled as an answer to the query, and C0 is the set defined as part of MVKD.Extract). By basic probability, one can check that

$$\left|\Pr\left[Hyb3(\mathcal{A})=1\right] - \Pr\left[Hyb4(\mathcal{A})=1\right]\right| \leq$$
$$\left|\Pr\left[Hyb3(\mathcal{A})=1|\neg\mathsf{E}\right] - \Pr\left[Hyb4(\mathcal{A})=1|\neg\mathsf{E}\right]\right| + \Pr[\mathsf{E}]$$

We have that Pr[E] is negligible. Indeed, the set C0 has polynomial size, since it starts empty and grows by an (amortized) constant number of elements for each of the polynomially many queries that $\mathcal{A}$ can make: Ideal queries grow C0 by at most 2 elements, and Extr queries by at most the number of repetitions of the while loop in UnrollChain, which is bounded by the size of H up to that point (and H itself grows by a constant size per oracle query). Since each y is sampled from an exponentially large space, the probability that a single sampled value falls in a polynomial size set is negligible. The games sample at most one y value per Ideal query, and the number of queries is again polynomially bounded, from which the result follows by a union bound.

To prove that the advantage in distinguishing the two hybrids is negligible, it is then enough to do so conditioned on ¬E. As in the previous hybrids, one can argue that, conditioned on not having random oracle collisions, $\mathcal{A}$'s view in both experiments has the same distribution until the point where an assertion is triggered, and therefore its output is the same if it returns without triggering an assertion. Therefore, it is enough to show that the probability of triggering an assertion in *Hyb*3 and in *Hyb*4 (conditioned on ¬E) is the same.

First note that, in either hybrid and conditioned on ¬E, UnrollChain always returns the same output when called multiple times on the same inputs. Indeed, UnrollChain is a deterministic algorithm whose output only depends on the inputs and the content of the H table internal to the extractor's state. We have that, once UnrollChain is called on a specific set of inputs, subsequent additions to H do not alter the output of the algorithm on those inputs. This is because, every time the extractor samples a value y to add a record to H, it also adds y

to CO, which will prevent it from being sampled again (and ensures that $H^{-1}(h)$ is well defined). Moreover, the extractor also adds h to CO whenever it checks if $H^{-1}(h)$ is defined, preventing h from being sampled in the future and thus preventing UnrollChain's answer from changing.

Given the above, it is easy to check that the two games throw assertions with exactly the same probability. In particular, assertions that are checked in both games are obviously triggered with the same probability, as the parts of the state that are shared between the two games have the same distribution (the only difference is that *Hyb*4 doesn't keep track of D). During Commit queries, *Hyb*4 doesn't enforce assertion 1. However, both games enforce assertion $1'$. If assertion $1'$ is not triggered, it must be that $\mathsf{C}[\mathsf{com}] = \mathsf{com}$ and $\mathsf{D}'[\mathsf{com}] = \mathsf{D}'_{\mathsf{com}}$. But since $\mathsf{D}[\mathsf{com}]$ is obtained by applying UnrollChain to all elements in $\mathsf{D}'_{\mathsf{com}}$, and we argued that UnrollChain always returns the same results, it follows that $\mathsf{D}[\mathsf{com}] = \mathsf{D}_{\mathsf{com}}$, and therefore assertion 1 cannot be triggered. Similarly, if assertion $2b'$ is not triggered in *Hyb*3, then assertion $2b$ can't be triggered either: by construction of our extractor $\forall((u,i),\mathbb{S}) \in \mathsf{D}_{\mathsf{com}}, \mathbf{t}(\mathsf{last}(\mathbb{S})) = t_i$, where $\mathsf{D}'_{\mathsf{com}}[(u,i)] = (\mathsf{auth}_i, t_i)$, and therefore $\mathbf{t}(\mathsf{com}) \geq \mathbf{t}(\mathsf{last}(\mathbb{S})) > 0$. Similarly, during Query oracle calls, *Hyb*4 does not enforce assertion 5 (but enforces $5'$). Assume assertion 5 (but not assertion $5'$) is triggered in *Hyb*3, and let i^* be the smallest index for which this happens. Observe that since assertion $5'$ isn't triggered, $D_{\mathsf{RZKS}}[\mathsf{com}][(u,i^*)]$ must equal the value (auth, t) that the client received from the adversary as part of the MVKD.Query execution. Since the execution did not return an error, it must also be that CheckAuth returned 1, and therefore that the hash of the i^*-th keychain statement computed by the client matched the one contained in auth. Since we are conditioning on no hash collisions, this implies that the output of the honest client must match the value $\mathsf{D}[\mathsf{com}][(u,i^*)]$ output by the extractor, contradicting that assertion 5 is triggered. We stress that assertions 5 and 6 imply that the output of the extractor matches the one of honest clients during Query, which is at the core of our extractability-based soundness. The proofs that assertion 6 in Query, and the other assertions removed from Audit oracle calls in *Hyb*4, cannot be triggered in *Hyb*3 are analogous.

*Hyb*4 $\approx$ *Hyb*5**.** We can prove that the two hybrids are indistinguishable with a straightforward reduction to the soundness of the RZKS. Assume by contradiction that there exists $\mathcal{A}$ that can distinguish *Hyb*4 from *Hyb*5. We can leverage $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the soundness of the RZKS. Our adversary $\mathcal{B}$ is defined exactly as the one used to argue that *Hyb*2 $\approx$ *Hyb*3, except that:

- To implement hash oracle queries, $\mathcal{B}$ no longer aborts on collisions (thus honestly implementing an ideal random oracle).
- During Audit queries from $\mathcal{A}$, in addition to the $\mathsf{CheckVerUpdD}(\mathsf{com}_a, \mathsf{com}_b, \pi)$ query to its own challenger, $\mathcal{B}$ also makes an additional $\mathsf{CheckVerUpdC}(\mathsf{com}_a, \mathsf{com}_b, \pi)$ query.
- During VerExtension oracle queries, if MVKD.VerExtension returns $b = 1$, then instead of the assertions $\mathcal{B}$ queries to its own challenger $\mathsf{CheckVerExt}(\mathsf{com}_a, \mathsf{com}_b, \pi)$, where $(\mathsf{com}_a, \mathsf{com}_b, \pi)$ are the inputs to the RZKS.VerExt call performed by $\mathcal{B}$ as part of MVKD.VerExtension. Then $\mathcal{B}$ returns.

As in the previous cases, we can argue that when $\mathcal{B}$ is executed in $\mathsf{Soundness\text{-}IDEAL}_{\mathsf{RZKS}}$, $\mathcal{A}$'s view up until the point where the game halts has the same distribution as in an execution of *Hyb*4.

Moreover, $\mathcal{A}$ triggers an assertion in *Hyb*4 with the same probability that the analogous assertion is triggered in $\mathsf{Soundness\text{-}IDEAL}_{\mathsf{RZKS}}$. Indeed, since the set $\mathsf{D}'_{\mathsf{RZKS}}$ in *Hyb*4 contains the same values as the set D in $\mathsf{Soundness\text{-}IDEAL}_{\mathsf{RZKS}}$, we have that:

- The conditions asserted during a Commit query in *Hyb*4 are $1', 2b', 3, 4$: assertions $1'$ and $2b'$ are analogous to those performed during an Extract query by $\mathsf{Soundness\text{-}IDEAL}_{\mathsf{RZKS}}$, while 3 and 4 match those in ExtractC queries. (We assume here that the error handling of RZKS.Extract matches the one we need in case of completely malformed commitments.)
- Assertions $5'$ and $6'$ in Query oracle calls are analogous to those performed during a CheckVerD query.
- Assertions $1, 3', 4', 5'$ in Audit oracle calls are analogous to those performed during a CheckVerUpdD query, and assertion 2 is analogous to that in CheckVerUpdC.
- The last assertion in VerExtension is analogous to that in CheckVerExt.

In short we have

$$\Pr[\mathit{Hyb}4(\mathcal{A}) = 1] = \Pr\left[\mathsf{Soundness\text{-}IDEAL}_{\mathsf{RZKS}}^{\mathcal{B},\mathsf{RZKS.Extract}} = 1\right].$$

Similarly, when $\mathcal{B}$ is executed in $\mathsf{Soundness\text{-}REAL}_{\mathsf{RZKS}}$, $\mathcal{A}$'s view until the game is halted has the same distribution as in *Hyb*5, and neither of the experiments throws any assertions, from which we have

$$\Pr[\mathit{Hyb}5(\mathcal{A}) = 1] = \Pr[\mathsf{Soundness\text{-}REAL}_{\mathsf{RZKS}}^{\mathcal{B}} = 1].$$

This implies that $\mathcal{A}$'s advantage in distinguishing the two hybrids can be bounded by $\mathcal{B}$'s advantage in the RZKS soundness game, which is negligible. □

I DETAILS ON PRIVACY

Here we provide the full details of the zero-knowledge with leakage game described in Section 4.3. We provide the detailed pseudocode of games $\mathsf{ZK\text{-}REAL}_{\mathsf{MVKD}}$ and $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ in Subsection I.3, with details on helper functions and predicates used in the games, and an overview of the tasks of the simulator, provided in Subsection I.1 and Subsection I.2, respectively.

Definition 12. *We say that a* MVKD *satisfies zero-knowledge with respect to a leakage function* $\mathcal{L} = (\mathcal{L}_{\mathsf{AddKey0}}, \mathcal{L}_{\mathsf{AddKey1}}, \mathcal{L}_{\mathsf{Query}}, \mathcal{L}_{\mathsf{Update}}, \mathcal{L}_{\mathsf{PCSUpdate}}, \mathcal{L}_{\mathsf{VerExt}}, \mathcal{L}_{\mathsf{Audit}}, \mathcal{L}_{\mathsf{Corr}}, \mathcal{L}_{\mathsf{LeakState}})$*, if there exists an efficient simulator* $\mathcal{S}$*, such that for any PPT adversary* $\mathcal{A}$

$$\left|\Pr\left[\ \mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}^{\mathcal{A},\mathcal{S}} = 1\ \right] - \Pr\left[\ \mathsf{ZK\text{-}REAL}_{\mathsf{MVKD}}^{\mathcal{A}} = 1\ \right]\right| \leq \mathsf{negl}(\lambda)\,.$$

I.1 Helper functions and predicates

Before defining the oracles, we define some helper functions and predicates that will simplify notation. Recall from Section 4.3 that our zero-knowledge game makes use of handles to refer to public keys, with devices being addressed by username-handle pairs. We assume in the following that ValidKeychain works for keychains involving handles, by treating handles as separate public keys. For example,

ValidKeychain would return 0 if a handle h gets added twice, but not detect if first a handle h and then the respective public key pk gets added.

The first helper algorithm SubstituteHandles canonicalizes symbolic keychain statements, which may contain a mixture of handles and public keys. (By symbolic we mean that the keychains can contain statements with handles in lieu of public keys.) It replaces known handles with their respective public keys for keychain statement s associated with username u, taking a mapping PK from known username-handle pairs to public keys as an argument and returning the updated statement s′. This mapping will be produced by the simulator, as described in the next section.

$$\mathsf{s}' \leftarrow \mathsf{SubstituteHandles}(\mathsf{PK}, u, \mathsf{s})$$

(1) Let s′ be a copy of s.
(2) If $\mathbf{pk_a}(\mathsf{s}) \in \mathcal{H}$ and $\mathsf{PK}[u, \mathbf{pk_a}(\mathsf{s})] \neq \perp$, then set $\mathbf{pk_a}(\mathsf{s}) \leftarrow \mathsf{PK}[u, \mathbf{pk_a}(\mathsf{s})]$.
(3) If $\mathbf{type}(\mathsf{s}) \in \{\mathsf{Add}, \mathsf{AddFirst}, \mathsf{Revoke}\}$, $\mathbf{data}(\mathsf{s}) \in \mathcal{H}$, and $\mathsf{PK}[u, \mathbf{data}(\mathsf{s})] \neq \perp$, then set $\mathbf{data}(\mathsf{s}) \leftarrow \mathsf{PK}[u, \mathbf{data}(\mathsf{s})]$.
(4) Return s′.

For a keychain $\mathbb{S}$, we further define $\mathbb{S}' \leftarrow \mathsf{SubstituteHandles}(\mathsf{PK}, \mathbb{S})$ to be the algorithm that creates a copy $\mathbb{S}'$ of $\mathbb{S}$ and sets $\mathbb{S}'[i] \leftarrow \mathsf{SubstituteHandles}(\mathsf{PK}, \mathbf{u}(\mathbb{S}), \mathbb{S}[i])$ for every $i = 1, \ldots, |\mathbb{S}|$.

The helper predicate Consistent is used in the ideal world game ZK-IDEAL$_{\mathsf{MVKD}}$ to check whether a list of proposed updates can in principle by applied to the directory by the server. It takes as input set M_{HonDev} of honest device updates, set M_{CorrDev} of malicious updates, directory Dir which maps usernames to keychains, and the mapping PK as described above. The first two checks ensure that all statements are well-formed. The third one ensures that each user has at most one update, while the fourth check ensures that no updates can lead to an invalid keychain. (Note that a protocol, by a combination of completeness and soundness, must reject an update failing the third or fourth checks anyway.)

$$b \leftarrow \mathsf{Consistent}(M_{\mathsf{HonDev}}, M_{\mathsf{CorrDev}}, \mathsf{Dir}, \mathsf{PK})$$

(1) If $\exists m \in M_{\mathsf{HonDev}}$ s.t. $m \neq (u, \mathsf{s}, \perp)$ for some username u and keychain statement s, return 0.
(2) If $\exists m \in M_{\mathsf{CorrDev}}$ s.t. $m \neq (u, \mathsf{s}, \mathsf{auth})$ for some username u, keychain statement s, and keychain statement authenticator auth, return 0.
(3) If $\exists\ (u, \mathsf{s}, \cdot), (u', \mathsf{s}', \cdot) \in M_{\mathsf{HonDev}} \cup M_{\mathsf{CorrDev}}$ s.t. $(u, \mathsf{s}, \cdot) \neq (u', \mathsf{s}', \cdot) \wedge u = u'$, return 0.
(4) $\forall (u, \mathsf{s}, \cdot) \in M_{\mathsf{HonDev}} \cup M_{\mathsf{CorrDev}}$, let $\mathbb{S}' \leftarrow \mathsf{SubstituteHandles}(\mathsf{PK}, \mathsf{Dir}[u] \| \mathsf{s})$ and verify that $\mathsf{ValidKeychain}(\mathbb{S}')$, else return 0.
(5) Return 1.

The helper predicate ValidAction is used in the ideal world game ZK-IDEAL$_{\mathsf{MVKD}}$ to decide whether an honest user can propose a given modification, encoded as keychain statement s, to their keychain. This is with respect to the honest device's current view on their keychain. The predicate takes as input the username u for which to apply the proposed modification and the keychain statement s which captures the proposed update. It also takes as input a table T used by game ZK-IDEAL$_{\mathsf{MVKD}}$ to map username-handle pairs to an ordered list of successful actions performed by the device. In particular, it looks for the last honest query entry made by the device for its own username, which reveals the device's latest view of its own keychain.

$$b \leftarrow \mathsf{ValidAction}(\mathsf{T}, u, \mathsf{s}, \mathsf{PK})$$

(1) Let $h = \mathbf{pk_a}(\mathsf{s})$ and retrieve in $\mathsf{T}_{(u,h)}$ the latest entry of the form $(\mathsf{HonQuery}, u, \mathsf{qepno}, \mathbb{S})$ for some qepno and $\mathbb{S}$. If no such entry exists, assign $\mathbb{S} \leftarrow (u; ())$.
(2) Let $\mathbb{S}' \leftarrow \mathsf{SubstituteHandles}(\mathsf{PK}, u, \mathbb{S} \| \mathsf{s})$.
(3) Return $\mathsf{ValidKeychain}(\mathbb{S}')$.

Finally, the following function applies a set of updates to the directory, initiating the new epoch epno. It takes in a directory Dir that maps usernames to keychains, a set M of updates to be applied to Dir, and the new epoch epno.

$$\mathsf{Dir}' \leftarrow \mathsf{Expand}(\mathsf{Dir}, M, \mathsf{epno})$$

(1) Initialize $\mathsf{Dir}' \leftarrow \mathsf{Dir}$
(2) $\forall (u, \mathsf{s}, \mathsf{auth}) \in M$:
  (a) If $u \notin \mathsf{Dir}$, initialize new keychain $\mathbb{S} \leftarrow (u; \mathsf{AddEpoch}(\mathsf{s}, \mathsf{epno}))$ and assign $\mathsf{Dir}'[u] \leftarrow \mathbb{S}$.
  (b) Else let $\mathbb{S} \leftarrow \mathsf{Dir}[u]$ and create new keychain $\mathbb{S}' \leftarrow \mathbb{S} \| \mathsf{AddEpoch}(\mathsf{s}, \mathsf{epno})$. Then update the directory $\mathsf{Dir}'[u] \leftarrow \mathbb{S}'$.
(3) Return Dir′

I.2 Simulator

The simulator $\mathcal{S}$ is given the output of the leakage function $\mathcal{L}$ and the output of the queries. It produces the values to be output to the adversary in the ideal-world experiment depicted in the next section, as well as an updated PK mapping that assigns public keys to username-handle pairs. More concretely, it provides the following functionality:

- $(\mathsf{com}_0, \mathsf{pp}) \leftarrow \mathcal{S}(\mathtt{Init}, 1^\lambda)$ is invoked at the beginning of the ideal-world experiment to provide the initial commitment com_0 as well as the scheme's public parameter pp.
- $(\sigma, \mathsf{PK}) \leftarrow \mathcal{S}(\mathtt{Sign}, u, h, m)$ is invoked whenever the adversary instructs an honest device, identified by (u, h) to sign the message m, outputting the signature σ, as well as the current mapping from username-handle pairs to public keys.

- $(\mathsf{auth}, \mathsf{PK}\ ;\ .) \leftarrow \mathcal{S}(\mathtt{AddKey0}, (u,h), pk, \mathcal{L}\ ;\ \mathcal{A})$ is invoked whenever an honest device (u,h) adds an adversarial public key pk. Here, the simulator acts on behalf of the honest adding device (which outputs auth) and interacts with the adversary acting as the device to be added.
- $(\ .\ ;\ \mathsf{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \mathtt{AddKey1}, pk, (u,h), \mathcal{L})$ is invoked whenever the adversary $\mathcal{A}$ adds an honest device (u,h). $\mathcal{S}$ thus simulates an honest device being added and interacts with the adversary acting as a malicious device performing the addition.
- $(st, \mathsf{PK}) \leftarrow \mathcal{S}(\mathtt{Corr}, u, h, \mathcal{L})$ is invoked whenever a device (u,h) is being corrupted. In this case, $\mathcal{S}$ must output the device's state st.
- $(\mathsf{com}, \mathsf{PK}) \leftarrow \mathcal{S}(\mathtt{Update}, M_{\mathsf{CorrDev}}, \mathcal{L})$ is invoked upon each server update. It takes the set of all adversarially generated updates M_{CorrDev} (and the subsequent specified leakage about honest updates) and outputs either the new commitment or error message com = ERROR to indicate that the updates should be rejected.
- $(\mathsf{com}, \mathsf{PK}) \leftarrow \mathcal{S}(\mathtt{PCSUpdate}, M_{\mathsf{CorrDev}}, \mathcal{L})$ is invoked upon each update with post-compromise security. It works analogously to the previous case.
- $(.\ ;\ \mathsf{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \mathtt{Query}, u, t, \mathcal{L})$ is invoked whenever the adversary $\mathcal{A}$ wants to run the interactive Query algorithm with the server, using as the server's input some user u and epoch t. $\mathcal{S}$ thus simulates the server's protocol messages towards the adversary $\mathcal{A}$.
- $(.\ ;\ \mathsf{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \mathtt{Audit}, \mathcal{L})$ is invoked whenever the adversary $\mathcal{A}$ audits the directory, with $\mathcal{S}$ simulating the server's protocol messages.
- $(.\ ;\ \mathsf{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \mathtt{VerExt}, \mathcal{L})$ is invoked whenever the adversary $\mathcal{A}$ runs VerExtension, with $\mathcal{S}$ simulating the server's protocol messages.
- $st^s \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \mathtt{LeakState}, \mathcal{L})$ is invoked whenever the adversary $\mathcal{A}$ compromises the server state.
- $out \leftarrow \mathcal{S}(\mathtt{Ideal}, in)$ is invoked whenever the adversary invokes the ideal oracle.

I.3 Oracle definitions

$\underline{\mathsf{ZK\text{-}REAL}^{\mathcal{A}}_{\mathsf{MVKD}}}$:
epno $\leftarrow 0$
$T_{\mathsf{HonDev}} \leftarrow \{\}; T_{\mathsf{CorrDev}} \leftarrow \{\}; T_{\mathsf{com}} \leftarrow \{\}$
qepno $\leftarrow \{\}$; T $\leftarrow \{\}$; st $\leftarrow \perp$
pp $\leftarrow$ MVKD.GenPP(1^λ)
$(st^s_0, \mathsf{com}_0) \leftarrow$ MVKD.ServerInit(pp)
$b \leftarrow \mathcal{A}^{\mathsf{DeviceSetup},\mathsf{AddFirstKey},\ldots}(\mathsf{com}_0, \mathsf{pp})$
return b

$\underline{\mathsf{DeviceSetup}(u)}$:
$h \leftarrow\$\ \mathcal{H}$
$(st, pk) \leftarrow$ MVKD.DeviceSetup(pp, u)
$\mathsf{qepno}_{(u,h)} \leftarrow 0; \mathsf{st}_{(u,h)} \leftarrow st$
$T_{\mathsf{HonDev}} \leftarrow T_{\mathsf{HonDev}} \cup (u,h)$
return h

$\underline{\mathsf{AddFirstKey}(u,h)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
auth $\leftarrow$ AddFirstKey$(\mathsf{st}_{(u,h)})$
require auth $\neq$ ERROR
s $\leftarrow$ (AddFirst, $h, h, \perp$)
$\mathsf{T}_{(u,h)}$.add((s, auth))

$\underline{\mathsf{RevokeKey}(u, h_0, h_1)}$:
require $(u,h_0), (u,h_1) \in T_{\mathsf{HonDev}}$
$st_0 \leftarrow \mathsf{st}_{(u,h_0)}; st_1 \leftarrow \mathsf{st}_{(u,h_1)}$
auth $\leftarrow$ MVKD.RevokeKey$(st_0, \mathbf{pk}(st_1))$
require auth $\neq$ ERROR
s $\leftarrow$ (Revoke, $h_0, h_1, \perp$)
$\mathsf{T}_{(u,h_0)}$.add((s, auth))

$\underline{\mathsf{RevokeKey}((u,h), pk)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
auth $\leftarrow$ MVKD.RevokeKey$(\mathsf{st}_{(u,h)}, pk)$
require auth $\neq$ ERROR
s $\leftarrow$ (Revoke, $h, pk, \perp$)
$\mathsf{T}_{(u,h)}$.add((s, auth))

$\underline{\mathsf{AddExtra}((u,h), d)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
auth $\leftarrow$ MVKD.AddExtra$(\mathsf{st}_{(u,h)}, d)$
require auth $\neq$ ERROR
s $\leftarrow$ (Extra, $h, d, \perp$)
$\mathsf{T}_{(u,h)}$.add((s, auth))

$\underline{\mathsf{Sign}(u,h,m)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
$\sigma \leftarrow$ MVKD.Sign$(\mathsf{st}_{(u,h)}, m)$
return σ

$\underline{\mathsf{ZK\text{-}IDEAL}^{\mathcal{A},\mathcal{S}}_{\mathsf{MVKD}}}$:
epno $\leftarrow 0$; Dir $\leftarrow \{\}$; PK $\leftarrow \{\}$
$T_{\mathsf{HonDev}} \leftarrow \{\}; T_{\mathsf{CorrDev}} \leftarrow \{\}; T_{\mathsf{com}} \leftarrow \{\}$
qepno $\leftarrow \{\}$; T $\leftarrow \{\}$
$(\mathsf{com}_0, \mathsf{pp}) \leftarrow \mathcal{S}(\mathtt{Init}, 1^\lambda)$
$b \leftarrow \mathcal{A}^{\mathsf{DeviceSetup},\mathsf{AddFirstKey},\ldots}(\mathsf{com}_0, \mathsf{pp})$
return b

$\underline{\mathsf{DeviceSetup}(u)}$:
$h \leftarrow\$\ \mathcal{H}$
$\mathsf{qepno}_{(u,h)} \leftarrow 0$
$T_{\mathsf{HonDev}} \leftarrow T_{\mathsf{HonDev}} \cup (u,h)$
return h

$\underline{\mathsf{AddFirstKey}(u,h)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
s $\leftarrow$ (AddFirst, $h, h, \perp$)
require ValidAction(T, u, s, PK) $= 1$
$\mathsf{T}_{(u,h)}$.add(s)

$\underline{\mathsf{RevokeKey}(u, h_0, h_1)}$:
require $(u,h_0), (u,h_1) \in T_{\mathsf{HonDev}}$
s $\leftarrow$ (Revoke, $h_0, h_1, \perp$)
require ValidAction(T, u, s, PK) $= 1$
$\mathsf{T}_{(u,h_0)}$.add(s)

$\underline{\mathsf{RevokeKey}((u,h), pk)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
s $\leftarrow$ (Revoke, $h, pk, \perp$)
require ValidAction(T, u, s, PK) $= 1$
$\mathsf{T}_{(u,h)}$.add(s)

$\underline{\mathsf{AddExtra}((u,h), d)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
s $\leftarrow$ (Extra, $h, d, \perp$)
require ValidAction(T, u, s, PK) $= 1$
$\mathsf{T}_{(u,h)}$.add(s)

$\underline{\mathsf{Sign}(u,h,m)}$:
require $(u,h) \in T_{\mathsf{HonDev}}$
$(\sigma, \mathsf{PK}) \leftarrow \mathcal{S}(\mathtt{Sign}, u, h, m)$
return σ

HonVerExt(u, h, com):

require $(u,h) \in T_{\text{HonDev}}$
$st \leftarrow st_{(u,h)}; t \leftarrow \mathbf{t}(\mathbf{com}(st)); t' \leftarrow \mathbf{t}(\text{com})$
$(st', b\ ;\ \perp) \leftarrow \text{MVKD.VerExtension}(st, \text{com}\ ;\ st^s_{\text{epno}}, t, t')$
require $b \neq 0$
$\text{st}_{(u,h)} \leftarrow st'; \text{qepno}_{(u,h)} \leftarrow \mathbf{t}(st')$

HonQuery(u, h, u'):

require $(u,h) \in T_{\text{HonDev}}$
$(st, \mathbb{S}\ ;\ \perp) \leftarrow \text{MVKD.Query}(\text{st}_{(u,h)}, u'\ ;\ st^s_{\text{epno}}, u', \text{qepno}_{(u,h)})$
require $(st, \mathbb{S}) \neq \texttt{ERROR}$
$\text{st}_{(u,h)} \leftarrow st$

AddKey(u, h, pk):

require $(u,h) \in T_{\text{HonDev}}$
$(\text{auth}\ ;\ .) \leftarrow \text{MVKD.AddKey}(\text{st}_{(u,h)}, pk\ ;\ \mathcal{A})$
return auth

AddKey(u, pk, h):

require $(u,h) \in T_{\text{HonDev}}$
$(.\ ;\ b) \leftarrow \text{MVKD.AddKey}(\mathcal{A}\ ;\ \text{st}_{(u,h)}, pk)$

AddKey(u, h_0, h_1):

require $(u,h_0), (u,h_1) \in T_{\text{HonDev}}$
$st_0 \leftarrow \text{st}_{(u,h_0)}; st_1 \leftarrow \text{st}_{(u,h_1)}$
$(\text{auth};\ b) \leftarrow \text{MVKD.AddKey}(st_0, \mathbf{pk}(st_1)\ ;\ st_1, \mathbf{pk}(st_0))$
require auth $\neq \texttt{ERROR}$ and $b = 1$
$\text{s} \leftarrow (\text{Add}, h_0, h_1, \perp)$
$\mathsf{T}_{(u,h_0)}.\text{add}((\text{s}, \text{auth}))$

CorrDev(u, h):

require $(u,h) \in T_{\text{HonDev}}$
$T_{\text{CorrDev}} = T_{\text{CorrDev}} \cup (u,h)$
$T_{\text{HonDev}} = T_{\text{HonDev}} \setminus (u,h)$
return $\text{st}_{(u,h)}$

Update$(M_{\text{HonDev}}, M_{\text{CorrDev}})$: // analogous for PCSUpdate

$M \leftarrow M_{\text{CorrDev}}$
$\forall (u, \text{s}, \cdot) \in M_{\text{HonDev}}$:
  $h_0 \leftarrow \mathbf{pk_a}(\text{s}); h_1 \leftarrow \mathbf{data}(\text{s})$
  require $(\text{s}, \cdot) \in \mathsf{T}_{(u,h_0)}$
  // Replace handles with keys
  $\text{PK} \leftarrow \{\}$
  if $h_0 \in \mathcal{H}$ **then** $\text{PK}[h_0] \leftarrow \mathbf{pk}(st_{(u,h_0)})$
  if $h_1 \in \mathcal{H}$ **then** $\text{PK}[h_1] \leftarrow \mathbf{pk}(st_{(u,h_1)})$
  $\text{s}' \leftarrow \text{SubstituteHandles}(\text{PK}, u, \text{s})$
  // Retrieve authenticator
  Retrieve $(\text{s}, \text{auth}) \in \mathsf{T}_{(u,h_0)}$
  $M \leftarrow M \cup \{(u, \text{s}', \text{auth})\}$
$(st^s, \text{com}) \leftarrow \text{MVKD.Update}(st^s_{\text{epno}}, M)$
require $(st^s, \text{com}) \neq \texttt{ERROR}$
$\text{epno} \leftarrow \mathbf{t}(\text{com}); st^s_{\text{epno}} \leftarrow st^s; T_{\text{com}}[t] \leftarrow \text{com}$

Query(u, t):

$\text{MVKD.Query}(\mathcal{A}\ ;\ st^s_{\text{epno}}, u, t)$

Audit(t):

$\text{MVKD.Audit}(\mathcal{A}\ ;\ st^s_{\text{epno}}, t)$

VerExtension(t, t'):

$\text{MVKD.VerExtension}(\mathcal{A}\ ;\ st^s_{\text{epno}}, t, t')$

LeakState():

return st^s_{epno}

Ideal(in):

return $\text{Ideal}(in)$

HonVerExt(u, h, com):

require $(u,h) \in T_{\text{HonDev}}$
require $\text{qepno}_{(u,h)} < \mathbf{t}(\text{com})$
require $T_{\text{com}}[\mathbf{t}(\text{com})] = \text{com}$
$\text{qepno}_{(u,h)} \leftarrow \mathbf{t}(\text{com})$
$\mathsf{T}_{(u,h)}.\text{add}((\text{HonVerExt}, \text{qepno}_{(u,h)}))$

HonQuery(u, h, u'):

require $(u,h) \in T_{\text{HonDev}}$
require $\text{qepno}_{(u,h)} \leq \text{epno}$
$\mathsf{T}_{(u,h)}.\text{add}((\text{HonQuery}, u', \text{qepno}_{(u,h)}, \text{Dir}_{\text{qepno}_{(u,h)}}[u']))$

AddKey(u, h, pk):

require $(u,h) \in T_{\text{HonDev}}$
$(\text{auth}, \text{PK}\ ;\ .) \leftarrow \mathcal{S}(\texttt{AddKey0}, (u,h), pk, \mathcal{L}_{\texttt{AddKey0}}(\mathsf{T}_{(u,h)})\ ;\ \mathcal{A})$
return auth

AddKey(u, pk, h):

require $(u,h) \in T_{\text{HonDev}}$
$(\ .\ ;\ \text{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \texttt{AddKey1}, pk, (u,h), \mathcal{L}_{\texttt{AddKey1}}(\mathsf{T}_{(u,h)}))$

AddKey(u, h_0, h_1):

require $(u,h_0), (u,h_1) \in T_{\text{HonDev}}$
$\text{s} \leftarrow (\text{Add}, h_0, h_1, \perp)$
require $\text{ValidAction}(\mathsf{T}, u, \text{s}, \text{PK}) = 1$
Let ℓ_0 be the last HonQuery for u in $\mathsf{T}_{(u,h_0)}$
Let ℓ_1 be the last HonQuery for u in $\mathsf{T}_{(u,h_1)}$
$(\text{HonQuery}, u, \cdot, \mathbb{S}_0) \leftarrow \ell_0$
$(\text{HonQuery}, u, \cdot, \mathbb{S}_1) \leftarrow \ell_1$
require $\mathbb{S}_0 = \mathbb{S}_1$
$\mathsf{T}_{(u,h_0)}.\text{add}(\text{s})$

CorrDev(u, h):

require $(u,h) \in T_{\text{HonDev}}$
$T_{\text{CorrDev}} = T_{\text{CorrDev}} \cup (u,h)$
$T_{\text{HonDev}} = T_{\text{HonDev}} \setminus (u,h)$
$L \leftarrow \mathcal{L}_{\texttt{Corr}}(u, h, \text{qepno}_{(u,h)}, \mathsf{T}_{(u,h)}, \text{PK}, \text{Dir}, \text{epno})$
$(st, \text{PK}) \leftarrow \mathcal{S}(\texttt{Corr}, u, h, L)$
return st

Update$(M_{\text{HonDev}}, M_{\text{CorrDev}})$: // analogous for PCSUpdate

require $\text{Consistent}(M_{\text{HonDev}}, M_{\text{CorrDev}}, \text{Dir}_{\text{epno}}, \text{PK}) = 1$
$M \leftarrow M_{\text{CorrDev}} \cup M_{\text{HonDev}}$
$\forall (u, \text{s}, \cdot) \in M_{\text{HonDev}}$:
  $h_0 \leftarrow \mathbf{pk_a}(\text{s})$
  require $\text{s} \in \mathsf{T}_{(u,h_0)}$
$L \leftarrow \mathcal{L}_{\texttt{Update}}(M_{\text{HonDev}}, M_{\text{CorrDev}}, \text{Dir}_{\text{epno}})$
$(\text{com}, \text{PK}) \leftarrow \mathcal{S}(\texttt{Update}, M_{\text{CorrDev}}, L)$
require $\text{com} \neq \texttt{ERROR}$
$\text{Dir}_{\mathbf{t}(\text{com})} \leftarrow \text{Expand}(\text{Dir}_{\text{epno}}, M, \mathbf{t}(\text{com}))$
$\text{epno} \leftarrow \mathbf{t}(\text{com}); T_{\text{com}}[t] \leftarrow \text{com}$

Query(u, t):

$(\ .\ ;\ \text{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \texttt{Query}, u, t, \mathcal{L}_{\texttt{Query}}(u, t, \text{Dir}, \text{epno}))$

Audit(t):

$(\ .\ ;\ \text{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \texttt{Audit}, \mathcal{L}_{\texttt{Audit}}(t))$

VerExtension(t, t'):

$(\ .\ ;\ \text{PK}) \leftarrow \mathcal{S}(\mathcal{A}\ ;\ \texttt{VerExt}, \mathcal{L}_{\texttt{VerExt}}(t, t'))$

LeakState():

return $\mathcal{S}(\texttt{LeakState}, \mathcal{L}_{\texttt{LeakState}}(\text{Dir}_{\text{epno}}))$

Ideal(in):

return $\mathcal{S}(\texttt{Ideal}, in)$

In the following, we provide some intuition on the above privacy game.

Game variables. Both experiments keep a number of state variables. First, they maintain T_{HonDev} and T_{CorrDev} as the set of honest and corrupted devices, respectively, storing pairs (u, h). Second, they store a mapping qepno, where $\mathsf{qepno}_{(u,h)}$ tracks the epoch the honest device (u, h) is currently in. For the server, epno stores the current epoch and $T_{\mathsf{com}}[t]$ stores the commitment output by the server (or the simulator) for epoch t. The real-world experiment $\mathsf{ZK\text{-}REAL}_{\mathsf{MVKD}}$ further tracks honest device's current state in $\mathsf{st}_{(u,h)}$, while in the ideal-world experiment $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ the directory Dir_t stores the (symbolic) directory of epoch t. Additionally, $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ stores a mapping PK from username-handle pairs to public keys chosen by the simulator.

Furthermore, both experiments maintain a mapping $\mathsf{T}_{(u,h)}$ storing the list of actions (successfully) performed by the device (u, h). In the real world it contains entries of the following format:

- $((\mathsf{AddFirst}, h, h, \bot), \mathsf{auth})$
- $((\mathsf{Revoke}, h_0, h_1, \bot), \mathsf{auth})$
- $((\mathsf{AddExtra}, h, d, \bot), \mathsf{auth})$
- $((\mathsf{Add}, h_0, h_1, \bot), \mathsf{auth})$
- $((\mathsf{Add}, h, pk, \bot), \mathsf{auth})$

while in the ideal world it contains the same entries but without auth, and the following additional entry types:

- $(\mathsf{HonVerExt}, t)$
- $(\mathsf{HonQuery}, u', t, \mathbb{S})$

Update. When the adversary instructs the server to update its directory, $\mathcal{A}$ provides two sets of keychain statements to be appended: M_{HonDev} contains honestly generated ones using the various oracles (for which the adversary does not know auth) while M_{CorrDev} contains keychain statements generated by the adversary. The former ones contain handles instead of public keys. Hence, for the server to be able to process them in the real world, the game first needs to substitute the handles with public keys. To this end, the game looks up the respective public key in the device's state. Further, it retrieves the keychain authenticator auth in $\mathsf{T}_{(u,h)}$.

In the ideal world the simulator is only given the maliciously generated updates (and some leakage on the honest ones). Hence, the experiment has to ensure itself that invalid updates are rejected, which it does using the Consistent helper predicate. After the simulator produces the updated commitment, and if it does not reject, the experiment updates its directory $\mathsf{Dir}_{\mathsf{epno}}$.

Post-compromise security. PCS is modeled by allowing for LeakState calls that reveal the current server state. Upon the adversary querying this oracle, the simulator must output an indistinguishable state, based on the leakage of $\mathcal{L}_{\mathtt{LeakState}}$. Note that since the leakage functions share a common state, this further can affect the leakage — and thus the privacy guarantees — of different operations.

Healing from compromise is then modeled by having a PCSUpdate oracle that works analogously to the Update one, except for the real world calling the respective algorithm instead and the ideal world using the dedicated leakage function $\mathcal{L}_{\mathtt{PCSUpdate}}$. Again, calling this function might affect other leakage functions via their shared state. In particular, observe that the more stringent privacy guarantees obtained by PCSUpdate compared to Update are formalized purely as part of the respective leakage functions $\mathcal{L}_{\mathtt{PCSUpdate}}$ and $\mathcal{L}_{\mathtt{Update}}$, and their effect on the other subsequent leakage function calls.

AddKey. Finally, observe that the game offers three different AddKey oracles, depending on which non-empty subset of the involved devices is honest. In the variant with two honest devices, the action gets recorded in $\mathsf{T}_{(u,h)}$ for the adding party if and only if both devices accept. (Recall that soundness mandates that otherwise the update anyway cannot be applied.) In contrast, if either the added party or adding party is malicious, then the update also counts as malicious. Notice that this is because interacting with a malicious party during AddKey enables the adversary to affect the outcome and learn auth.

J LEAKAGE IN OUR MVKD CONSTRUCTION

We now provide a formal definition of our protocol's leakage function $\mathcal{L} = (\mathcal{L}_{\mathtt{AddKey0}}, \mathcal{L}_{\mathtt{AddKey1}}, \mathcal{L}_{\mathtt{Query}}, \mathcal{L}_{\mathtt{Update}}, \mathcal{L}_{\mathtt{PCSUpdate}}, \mathcal{L}_{\mathtt{VerExt}}, \mathcal{L}_{\mathtt{Audit}}, \mathcal{L}_{\mathtt{Corr}}, \mathcal{L}_{\mathtt{LeakState}})$. See Section 4.4 for an intuitive description of the leakage.

The leakage function is described in Figure 8. The shared state consists of a boolean flag *leaked* (initialized to *false*), and a set K of username-keychain pairs which the adversary knows (initially empty). Note that K is irrelevant and not maintained while *leaked* = *true*, but patched up in $\mathcal{L}_{\mathtt{PCSUpdate}}$.

K PROOF OF THEOREM 3

Let $\mathcal{S}_{\mathsf{RZKS}}$ be the simulator from the RZKS zero knowledge property. In the following, we construct a simulator $\mathcal{S}_{\mathsf{MVKD}}$ for the MVKD zero knowledge game.

Note that the simulator $\mathcal{S}_{\mathsf{MVKD}}$ makes use of the protocol helper algorithms CheckAuth (for checking the keychain authenticator) and AuthStatement (for computing keychain authenticators) which are defined in Appendix D. Note that the hash function used by those algorithms is implemented as a random oracle by the simulator, in an honest manner. In other words, $\mathcal{S}_{\mathsf{MVKD}}$ samples new values at random and stores them in a table D so that repeated values get back the same response. We assume this is handled implicitly by CheckAuth and AuthStatement. The simulator will use the same table D to handle random oracle queries by the adversary.

We make use of the following helper algorithm that takes a symbolic keychain, that potentially contains handles, and transforms it into a sigchain by (1) replacing all handles with actual public keys, and (2) computing the keychain authenticators auth_i for each statement i. The algorithm directly accesses the simulator's directory Dir (to store the result), the mapping Keys from username-handle pairs to public/secret key pairs, and the mapping PK from username-handle pairs to public keys, returning the updated results. Note that Dir maps an epoch and username, using notation $\mathsf{Dir}_t[u]$, to a (keychain statement, authenticator) pair.

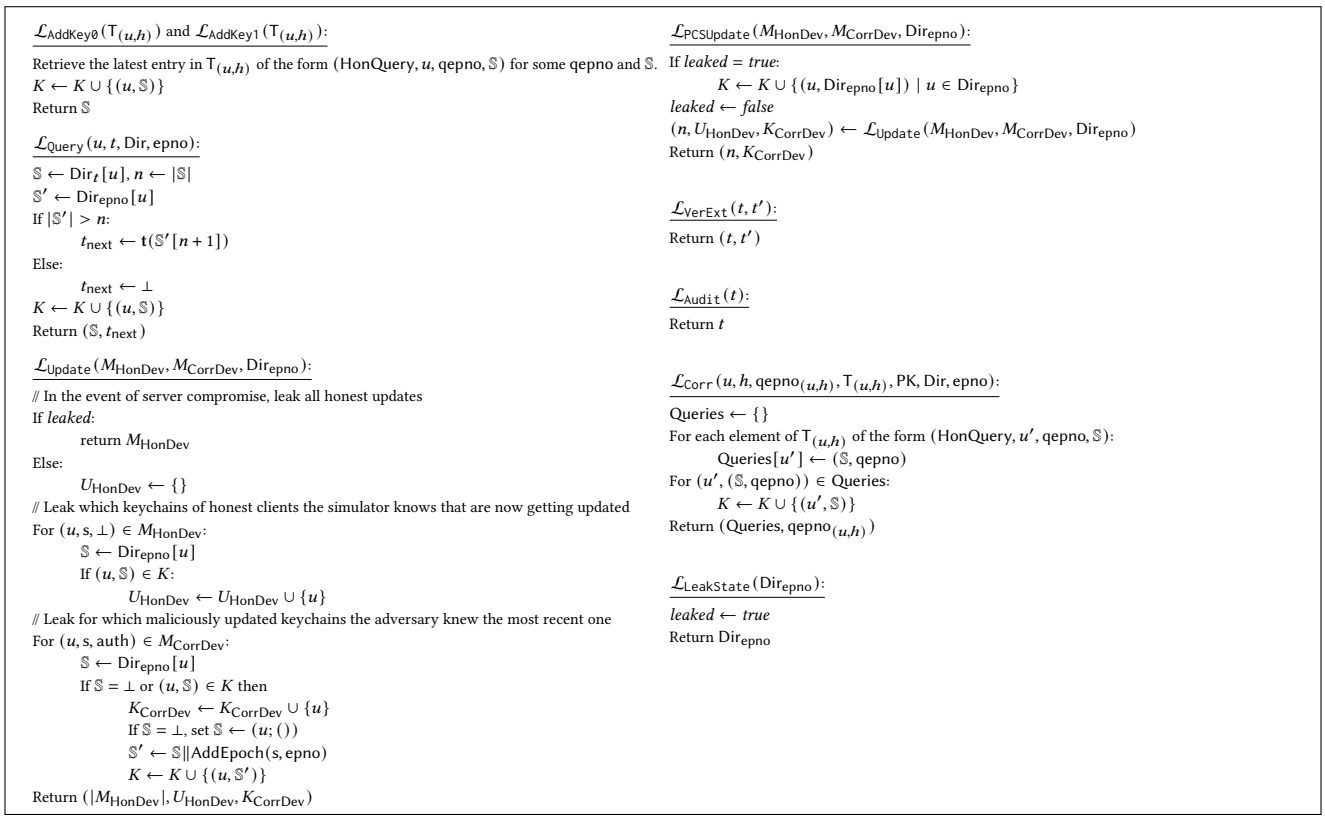


Figure 8 ELEKTRA's Leakage. Note that during Update **queries, we leak usernames for all the keychains of honest clients that the adversary knew which are getting updates as part of M_{HonDev}. For the proof to go through, we would strictly need only those updates which the adversary learned through a query (and not those learned through** AddKey **or corruptions), but we define it this way for simplicity.**

$$(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}, t)$$

- Let $n \leftarrow |\mathbb{S}|$ and $u \leftarrow \mathbf{u}(\mathbb{S})$. If $n = 0$ or $\mathsf{Dir}_t[u] \neq \perp$, then return Dir, Keys, PK.
- Let $t' \leftarrow \mathbf{t}(\mathsf{last}(\mathbb{S}))$. If $t' < t$, then $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}, t')$, set $\mathsf{Dir}_t[u] \leftarrow \mathsf{Dir}_{t'}[u]$, and return (Dir, Keys, PK).
- If $n > 1$:
  (1) $\mathbb{S}_{n-1} \leftarrow \mathbb{S}[1 \ldots n-1]$
  (2) $t_{n-1} \leftarrow \mathbf{t}(\mathsf{last}(\mathbb{S}_{n-1}))$
  (3) $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}_{n-1}, t_{n-1})$
  (4) $(\mathbb{S}'_{n-1}, \mathsf{auth}_{n-1}) \leftarrow \mathsf{Dir}_{t_{n-1}}[u]$.
- Else define $\mathbb{S}'_{n-1} \leftarrow (\mathbf{u}(\mathbb{S}); ())$ and $\mathsf{auth}_{n-1} \leftarrow \varepsilon$

// Replace handles by public keys in last link

- $\mathsf{s}_n \leftarrow \mathbb{S}[n]$
- $h_0 \leftarrow \mathbf{pk_a}(\mathsf{s}_n)$, $h_1 \leftarrow \mathbf{data}(\mathsf{s}_n)$
- If $h_0 \in \mathcal{H}$ and $\mathsf{Keys}[u, h_0] = \perp$:
  - $(sk, pk) \leftarrow \mathsf{SIG.KeyGen}()$
  - $\mathsf{Keys}[u, h_0] \leftarrow (sk, pk)$
  - $\mathsf{PK}[u, h_0] \leftarrow pk$
- If $h_1 \in \mathcal{H}$ and $\mathsf{Keys}[u, h_1] = \perp$:
  - $(sk, pk) \leftarrow \mathsf{SIG.KeyGen}()$
  - $\mathsf{Keys}[u, h_1] \leftarrow (sk, pk)$
  - $\mathsf{PK}[u, h_1] \leftarrow pk$
- $\mathsf{s}'_n \leftarrow \mathsf{SubstituteHandles}(\mathsf{PK}, u, \mathsf{s}_n)$

// Create fake client state to create authenticator

- Retrieve (sk, pk) from Keys such that $pk = \mathbf{pk_a}(\mathsf{s}'_n)$
- $\mathsf{Users}[u] \leftarrow (\mathbb{S}'_{n-1}, \mathsf{auth}_{n-1}, \cdot)$
- $st \leftarrow (u, sk, pk, \cdot, \mathsf{Users})$

// Compute authenticator

- $\mathsf{auth}_n \leftarrow \mathsf{AuthStatement}(st, \mathbf{type}(\mathsf{s}'_n), pk, \mathbf{data}(\mathsf{s}'_n))$
- If $\mathbf{type}(\mathsf{s}'_n) = \mathsf{Add}$:
  - Retrieve (sk', pk') from Keys such that $pk' = \mathbf{data}(\mathsf{s}'_n)$
  - $st' \leftarrow (u, sk', pk', \cdot, \mathsf{Users})$
  - $\mathsf{auth}'_n \leftarrow \mathsf{AuthStatement}(st', \mathsf{Add}, pk, pk')$
  - $(h, z) \leftarrow \mathsf{auth}_n$; $(h', z') \leftarrow \mathsf{auth}'_n$
  - $\mathsf{auth}_n \leftarrow (h, (z, z'))$
- $\mathbb{S}' \leftarrow \mathbb{S}'_{n-1} \| \mathsf{s}'_n$
- $\mathsf{Dir}_t[u] \leftarrow (\mathbb{S}', \mathsf{auth}_n)$
- Return Dir, Keys, PK

Finally, we introduce the following helper algorithm to fetch a user's sigchain from the directory (as known to the simulator) as of epoch t. If the simulator stored no sigchain for u up to epoch t, then the algorithm returns an empty keychain and the authenticator ϵ.

$$(\mathbb{S}, \mathsf{auth}) \leftarrow \mathsf{LatestSigchain}(\mathsf{Dir}, u, t)$$

- If $\mathsf{Dir}_t[u] \neq \perp$, return $\mathsf{Dir}_t[u]$
- Else if $t = 1$, return $((u; ()), \epsilon)$
- Else, return $\mathsf{LatestSigchain}(\mathsf{Dir}, u, t-1)$

Using the above helper algorithms, we now can define the actual simulator as follows.

$$\mathcal{S}_{\mathsf{MVKD}}:$$

- ▶ The simulator $\mathcal{S}_{\mathsf{MVKD}}$ maintains the following state:
  - The state for the RZKS simulator (implicit)
  - *leaked*: a bit which is set when the server state is leaked to the adversary (and unset upon PCSUpdate calls).
  - Dir_t: a dictionary mapping usernames to the keychain known to the simulator at epoch t
  - epno: the current epoch of the system
  - Keys: a dictionary mapping a username-handle pair to the key pair (sk, pk) for the corresponding device
  - Com: a dictionary mapping the epoch to the commitment and update proof for that epoch
  - g: the current generation, i.e., the number of PCSUpdate operations performed so far
  - G: a dictionary mapping epochs to their respective generation
  - Q: the set of RZKS labels that have been queried to the RZKS simulator before having been added to the RZKS data store
  - PK: a dictionary mapping a username-handle pair to the public key pk for the corresponding device
  - D: a table used by the simulator to implement the random oracle hash
- ▶ $(\mathsf{com}_0, \mathsf{pp}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathtt{Init}, 1^\lambda)$:
  - $\mathsf{Dir}_0 \leftarrow \{\}$; epno $\leftarrow 0$; Keys $\leftarrow [\,]$; Com $\leftarrow [\,]$; $g \leftarrow 0$; G $\leftarrow [\,]$; $Q \leftarrow \{\}$; PK $\leftarrow [\,]$; D $\leftarrow [\,]$
  - $(\mathsf{com}, \mathsf{pp}) \leftarrow \mathcal{S}_{\mathsf{RZKS}}(\mathtt{Init})$
  - $\mathsf{Com}[0] \leftarrow (\mathsf{com}, \perp)$
  - Return $(\mathsf{com}, \mathsf{pp})$
- ▶ $(\mathsf{com}, \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathtt{Update}, M_{\mathsf{CorrDev}}, L)$
  - If *leaked*, set $M_{\mathsf{HonDev}} \leftarrow L$. Else, parse L as $(n_{\mathsf{HonDev}}, U_{\mathsf{HonDev}}, K_{\mathsf{CorrDev}})$.

  `// Check validity of updates in` M_{CorrDev}
  - For $(u, \mathsf{s}', \mathsf{auth}') \in M_{\mathsf{CorrDev}}$:

    `// If the adversary has not compromised the server and does not know this username's keychain, then it cannot forge an update so return error`

    (1) If *leaked* = *false* and $u \notin K_{\mathsf{CorrDev}}$, then return (ERROR, PK).
    (2) $(\mathbb{S}, \mathsf{auth}) \leftarrow \mathsf{LatestSigchain}(\mathsf{Dir}, u, \mathsf{epno})$
    (3) If $|\mathbb{S}| \geq 1$, set $t \leftarrow \mathbf{t}(\mathsf{last}(\mathbb{S}))$, else $t \leftarrow 0$
    (4) If $\mathsf{CheckAuth}(\mathsf{auth}, t, u, \mathsf{s}', \mathsf{auth}') = 0$ or $\mathsf{ValidKeychain}(\mathbb{S} \| \mathsf{s}') = 0$, then return (ERROR, PK).

  `// Update the directory`
  - epno $\leftarrow$ epno + 1
  - $\mathsf{Dir}_{\mathsf{epno}} \leftarrow \{\}$
  - For each $(u, \mathsf{s}', \mathsf{auth}') \in M_{\mathsf{CorrDev}}$:
    (1) $(\mathbb{S}, \mathsf{auth}) \leftarrow \mathsf{LatestSigchain}(\mathsf{Dir}, u, \mathsf{epno} - 1)$
    (2) $\mathbb{S}' \leftarrow \mathbb{S} \| \mathsf{AddEpoch}(\mathsf{s}', \mathsf{epno})$
    (3) $\mathsf{Dir}_{\mathsf{epno}}[u] \leftarrow (\mathbb{S}', \mathsf{auth}')$
  - If *leaked*, for each $(u, \mathsf{s}', \perp) \in M_{\mathsf{HonDev}}$:
    (1) $(\mathbb{S}, \cdot) \leftarrow \mathsf{LatestSigchain}(\mathsf{Dir}, u, \mathsf{epno} - 1)$
    (2) $\mathbb{S}' \leftarrow \mathbb{S} \| \mathsf{AddEpoch}(\mathsf{s}', \mathsf{epno})$
    (3) $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}', \mathsf{epno})$

  `// Compute RZKS leakage`
  - If *leaked*:
    (1) $S \leftarrow \{\}$

(2) For $(u, \mathsf{s}, \cdot) \in M_{\mathsf{HonDev}} \cup M_{\mathsf{CorrDev}}$:
(a) $(\mathbb{S}, \cdot) \leftarrow \mathsf{Dir}_{\mathsf{epno}}[u]$
(b) $S \leftarrow S \cup \{(u, |\mathbb{S}|)\}$
(3) $L_{\mathsf{Update}} \leftarrow S$
– Else:
(1) $n \leftarrow n_{\mathsf{HonDev}} + |M_{\mathsf{CorrDev}}|$ // $|S|$
(2) $C \leftarrow \{\}$
(3) For $u \in (U_{\mathsf{HonDev}} \cup \{u' \mid (u', \cdot, \cdot) \in M_{\mathsf{CorrDev}}\})$:
(a) $(\mathbb{S}, \cdot) \leftarrow \mathsf{LatestSigchain}(\mathsf{Dir}, u, \mathsf{epno} - 1)$
(b) $\mathsf{label} \leftarrow (u, |\mathbb{S}| + 1)$
(c) If $\mathsf{label} \in Q$, then $C \leftarrow C \cup \{\mathsf{label}\}$
(4) $L_{\mathsf{Update}} \leftarrow (n, C)$
// Invoke RZKS simulator
– $(\mathsf{com}', \pi') \leftarrow \mathcal{S}_{\mathsf{RZKS}}(\mathsf{Update}, L_{\mathsf{Update}})$
– $\mathsf{Com}[\mathsf{epno}] \leftarrow (\mathsf{com}', \pi')$
– $\mathsf{G}[\mathsf{epno}] \leftarrow g$
– Return $(\mathsf{com}', \mathsf{PK})$

▶ $(\mathsf{com}, \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathsf{PCSUpdate}, M_{\mathsf{CorrDev}}, L)$
– Set *leaked* $\leftarrow$ *false*, $g \leftarrow g + 1$
– Set $Q \leftarrow \{\}$
– Parse L as $(n_{\mathsf{HonDev}}, K_{\mathsf{CorrDev}})$, set $U_{\mathsf{HonDev}} \leftarrow \{\}$, and define $L' \leftarrow (n_{\mathsf{HonDev}}, U_{\mathsf{HonDev}}, K_{\mathsf{CorrDev}})$
– Proceed analogous to $\mathcal{S}_{\mathsf{MVKD}}(\mathsf{Update}, M_{\mathsf{CorrDev}}, L')$, except that $\mathcal{S}_{\mathsf{MVKD}}$ invokes $\mathcal{S}_{\mathsf{RZKS}}(\mathsf{PCSUpdate}, L_{\mathsf{PCSUpdate}})$ instead of $\mathcal{S}_{\mathsf{RZKS}}(\mathsf{Update}, L_{\mathsf{Update}})$ (i.e., the respective RZKS simulator) with $L_{\mathsf{PCSUpdate}} \leftarrow n$, where $L_{\mathsf{Update}} = (n, C)$.

▶ $(. ; \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathcal{A} ; \mathsf{Query}, u, t, L)$
– Set $(\mathbb{S}, t_{\mathsf{next}}) \leftarrow L$ and $n \leftarrow |\mathbb{S}|$
– Receive t^{last}
– If $t >$ epno or $t^{\mathsf{last}} > t$, then send ERROR and return PK
– $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}, t)$
– $j' \leftarrow |\{\mathsf{s} : \mathsf{s} \in \mathbb{S} \wedge \mathbf{t}(\mathsf{s}) \leq t^{\mathsf{last}}\}|$
– For $i = j' + 1, \ldots, n + 1$:
(1) $\mathsf{label}_i \leftarrow (u, i)$
// Create RZKS leakage
(2) If $i \leq n$:
(a) $t_i \leftarrow \mathbf{t}(\mathbb{S}[i])$
(b) $(\cdot, \mathsf{auth}_i) \leftarrow \mathsf{Dir}_{t_i}[u]$
(c) $L_i \leftarrow (\mathsf{label}_i, \mathsf{auth}_i, t_i, t)$
(3) Else If $t_{\mathsf{next}} \neq \perp$ and $\mathsf{G}[t_{\mathsf{next}}] = \mathsf{G}[t]$:
(a) $L_i \leftarrow (\mathsf{label}_i, \perp, t_{\mathsf{next}}, t)$
(4) Else:
(a) $L_i \leftarrow (\mathsf{label}_i, \perp, \perp, t)$
(b) If $\mathsf{G}[t] = g$: $Q \leftarrow Q \cup \{\mathsf{label}_i\}$
// Simulate RZKS proof
(5) $(\pi_i, \mathsf{val}_i, t_i) \leftarrow \mathcal{S}_{\mathsf{RZKS}}(\mathsf{Query}, L_i)$
– $(\mathbb{S}', \mathsf{auth}_n) \leftarrow \mathsf{Dir}_t[u]$
– Send $\{(\mathbb{S}'[i], \mathsf{val}_i, \pi_i)\}_{i=j'+1}^{n+1}$ (with $\mathbb{S}'[n+1] = \perp$) to adversary
– Return PK

▶ $(. ; \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathcal{A} ; \mathsf{Audit}, L)$
– Parse $t \leftarrow L$
– $(\mathsf{com}, \pi) \leftarrow \mathsf{Com}[t + 1]$
– Send π to the adversary
– Return PK

▶ $(. ; \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathcal{A} ; \mathsf{VerExt}, L)$
– Parse $(t, t') \leftarrow L$
– If $t = \perp$, set $\pi \leftarrow \perp$
– Else if $t < t'$: set $\pi \leftarrow \mathcal{S}_{\mathsf{RZKS}}(\mathsf{ProveExt}, (t, t'))$
– Else: set $\pi \leftarrow \mathcal{S}_{\mathsf{RZKS}}(\mathsf{ProveExt}, t', t)$
– Retrieve $(\mathsf{com}, \cdot) \leftarrow \mathsf{Com}[t']$
– Send (com, π) to the adversary
– Return PK

▶ $(\sigma, \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathsf{Sign}, u, h, m)$

- If $\mathsf{Keys}[u,h] \neq \bot$, $(sk, pk) \leftarrow \mathsf{Keys}[u,h]$. Else sample $(sk, pk) \leftarrow \mathsf{SIG.KeyGen}()$ and set $\mathsf{Keys}[u,h] \leftarrow (sk, pk)$ as well as $\mathsf{PK}[u,h] \leftarrow pk$.
- Compute $\sigma \leftarrow \mathsf{SIG.Sign}(sk, 1 \| m)$
- Return (σ, PK)

▶ $(\mathsf{auth}', \mathsf{PK}\ ;\ .) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathtt{AddKey0}, (u,h), pk, L\ ;\ \mathcal{A})$

- Set $\mathbb{S} \leftarrow L$, $n \leftarrow |\mathbb{S}|$, $t \leftarrow \mathbf{t}(\mathsf{last}(\mathbb{S}))$
- $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}, t)$
- $(\mathbb{S}, \mathsf{auth}) \leftarrow \mathsf{LatestSigchain}(\mathsf{Dir}, u, t)$
- If $(\mathbb{S}, \mathsf{auth}) = \bot$, return $(\mathtt{ERROR}, \mathsf{PK})$
- If $\mathsf{Keys}[u,h] = \bot$:
  (1) $(sk, pk) \leftarrow \mathsf{SIG.KeyGen}()$
  (2) $\mathsf{Keys}[u,h] \leftarrow (sk, pk)$
  (3) $\mathsf{PK}[u,h] \leftarrow pk$
- $(sk_0, pk_0) \leftarrow \mathsf{Keys}[u,h]$; $pk_1 \leftarrow pk$

// Create fake client state to create authenticator

- $\mathsf{Users}[u] \leftarrow (\mathbb{S}, \mathsf{auth}, \cdot)$
- $st \leftarrow (u, sk_0, pk_0, \cdot, \mathsf{Users})$

// Compute authenticator

- $(h, z) \leftarrow \mathsf{AuthStatement}(st, \mathsf{Add}, pk_0, pk_1)$
- If $(h, z) = \mathtt{ERROR}$, return $(\mathtt{ERROR}, \mathsf{PK})$
- Receive (h', z') from the adversary
- If $h \neq h'$ or $\mathsf{SIG.Ver}(pk_1, 0 \| h, z') = 0$, return $(\mathtt{ERROR}, \mathsf{PK})$
- $\mathsf{auth}' \leftarrow (h, (z, z'))$
- Return $(\mathsf{auth}', \mathsf{PK})$

▶ $(.\ ;\ \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathcal{A}\ ;\ \mathtt{AddKey1}, pk, (u,h), L)$

- $\mathbb{S} \leftarrow L$, $n \leftarrow |\mathbb{S}|$, $t \leftarrow \mathbf{t}(\mathsf{last}(\mathbb{S}))$
- $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}, t)$
- $(\mathbb{S}, \mathsf{auth}) \leftarrow \mathsf{LatestSigchain}(\mathsf{Dir}, u, t)$
- If $(\mathbb{S}, \mathsf{auth}) = \bot$, return PK
- If $\mathsf{Keys}[u,h] = \bot$:
  (1) $(sk, pk) \leftarrow \mathsf{SIG.KeyGen}()$
  (2) $\mathsf{Keys}[u,h] \leftarrow (sk, pk)$
  (3) $\mathsf{PK}[u,h] \leftarrow pk$
- $pk_0 \leftarrow pk$; $(sk_1, pk_1) \leftarrow \mathsf{Keys}[u,h]$

// Create fake client state to create authenticator

- $\mathsf{Users}[u] \leftarrow (\mathbb{S}, \mathsf{auth}, \cdot)$
- $st \leftarrow (u, sk_1, pk_1, \cdot, \mathsf{Users})$

// Compute authenticator

- $(h', z') \leftarrow \mathsf{AuthStatement}(st, \mathsf{Add}, pk_0, pk_1)$
- If $(h', z') = \mathtt{ERROR}$, return PK
- Send (h', z') to the adversary
- Return PK

▶ $(st, \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathtt{Corr}, u, h, L)$

- $(\mathsf{Queries}, \mathsf{qepno}) \leftarrow L$
- If $\mathsf{Keys}[u,h] = \bot$:
  (1) $(sk, pk) \leftarrow \mathsf{SIG.KeyGen}()$
  (2) $\mathsf{Keys}[u,h] \leftarrow (sk, pk)$
  (3) $\mathsf{PK}[u,h] \leftarrow pk$
- $(sk, pk) \leftarrow \mathsf{Keys}[u,h]$
- $(\mathsf{com}, \pi) \leftarrow \mathsf{Com}[\mathsf{qepno}]$
- Initialize $\mathsf{Users} \leftarrow [\ ]$. For $u' \in \mathsf{Queries}$:
  (1) $(\mathbb{S}', \mathsf{qepno}') \leftarrow \mathsf{Queries}[u']$
  (2) $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, \mathbb{S}', \mathsf{qepno}')$
  (3) $(\mathbb{S}', \mathsf{auth}') \leftarrow \mathsf{Dir}_{\mathsf{qepno}'}[u']$
  (4) $\mathsf{Users}[u'] \leftarrow (\mathbb{S}', \mathsf{auth}', \mathsf{qepno}')$
- $st \leftarrow (u, sk, pk, \mathsf{com}, \mathsf{Users})$
- Return (st, PK)

▶ $(\mathsf{st}^S, \mathsf{PK}) \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathtt{LeakState}, L)$

- Set $leaked \leftarrow true$
- $\mathsf{Users}^S \leftarrow \{\}$
- For $u \in L$
  (1) $(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}) \leftarrow \mathsf{ComputeSigchain}(\mathsf{Dir}, \mathsf{Keys}, \mathsf{PK}, L[u], \mathsf{epno})$

(2) $(\mathbb{S}, \mathsf{auth}) \leftarrow \mathsf{Dir}_{\mathsf{epno}}[u]$
(3) $\mathsf{Users}^S[u] \leftarrow \mathbb{S}$

// Compute RZKS leakage

- $\mathsf{D} \leftarrow \{\}$
- For $u \in \mathsf{Dir}_{\mathsf{epno}}$
  (1) Let $(\mathbb{S}, \cdot) \leftarrow \mathsf{Dir}_{\mathsf{epno}}[u]$ and $n \leftarrow |\mathbb{S}|$
  (2) For $i = 1, \ldots, n$
    (a) $t_i \leftarrow \mathbf{t}(\mathbb{S}[i])$
    (b) $(\cdot, \mathsf{auth}_i) \leftarrow \mathsf{Dir}_{t_i}[u]$
    (c) $\mathsf{D} \leftarrow \mathsf{D} \cup \{((u, i), \mathsf{auth}_i, t_i)\}$
- $\mathsf{st}_{\mathsf{RZKS}} \leftarrow \mathcal{S}_{\mathsf{RZKS}}(\mathsf{Leak}, \mathsf{D})$
- $\mathsf{st}^S \leftarrow (\mathsf{st}_{\mathsf{RZKS}}, \mathsf{Com}, \mathsf{Users}^S)$
- Return $(\mathsf{st}^S, \mathsf{PK})$

▶ $out \leftarrow \mathcal{S}_{\mathsf{MVKD}}(\mathtt{Ideal}, in)$

- Queries to ideal objects related to the RZKS are answered by making the analogous query to $\mathcal{S}_{\mathsf{RZKS}}$.
- The random oracle hash is implemented by checking if an answer to the query was previously asked and is stored in table D. If so, that value is returned; otherwise, a value is sampled at random, stored in D and then returned.

Assume there exists an adversary $\mathcal{A}_{\mathsf{MVKD}}$ that can distinguish between games $\mathsf{ZK\text{-}REAL}^{\mathcal{A}_{\mathsf{MVKD}}}_{\mathsf{MVKD}}$ and $\mathsf{ZK\text{-}IDEAL}^{\mathcal{A}_{\mathsf{MVKD}}}_{\mathsf{MVKD}}$ with non-negligible probability (when ELEKTRA is instantiated with the RZKS from [8]). We provide the following adversary $\mathcal{A}_{\mathsf{RZKS}}$ that can distinguish between games $\mathsf{ZK\text{-}REAL}^{\mathcal{A}_{\mathsf{RZKS}}}_{\mathsf{RZKS}}$ and $\mathsf{ZK\text{-}IDEAL}^{\mathcal{A}_{\mathsf{RZKS}}}_{\mathsf{RZKS}}$ with non-negligible probability for scheme RZKS.

$\mathcal{A}^{\mathsf{Update}(\cdot),\mathsf{Query}(\cdot,\cdot),\mathsf{ProveExt}(\cdot,\cdot),\mathsf{LeakState}(),\mathsf{Ideal}(\cdot)}_{\mathsf{RZKS}}(\mathsf{com}', \mathsf{pp}')$

- $\mathcal{A}_{\mathsf{RZKS}}$ runs $\mathcal{A}_{\mathsf{MVKD}}$ and simulates oracle calls made by $\mathcal{A}_{\mathsf{MVKD}}$ according to how the $\mathsf{ZK\text{-}REAL}_{\mathsf{MVKD}}$ game responds to oracle calls. This means $\mathcal{A}_{\mathsf{RZKS}}$ makes calls to the MVKD protocol directly, except for the following oracles:
  - Initialization: Instead of calling MVKD.GenPP() and MVKD.ServerInit(), $\mathcal{A}_{\mathsf{RZKS}}$ follows what the ServerInit protocol does but uses com' and pp', which were given as input, in place of calling RZKS.GenPP() and RZKS.Init(). For the server state st^S, $\mathcal{A}_{\mathsf{RZKS}}$ only stores Com and Users^S and modifies Users^S to map the username to the keychain and the authenticator for the keychain.
  - $\mathsf{HonVerExt}(u, h, \mathsf{com})$: Instead of calling MVKD.VerExtension(), $\mathcal{A}_{\mathsf{RZKS}}$ verifies that the input com equals $T_{\mathsf{com}}[\mathbf{t}(\mathsf{com})]$, which is the commitment generated by a call to Update for the epoch associated with com. If this is true and $\mathsf{qepno}_{(u,h)} < \mathbf{t}(\mathsf{com})$, then $\mathcal{A}_{\mathsf{RZKS}}$ updates the device state itself by updating the commitment in the state to com.
  - $\mathsf{HonQuery}(u, h, u')$: Instead of calling MVKD.Query(), $\mathcal{A}_{\mathsf{RZKS}}$ gets the keychain and authenticator from the server state $(\mathbb{S}, \mathsf{auth}) \leftarrow \mathsf{st}^S.\mathsf{Users}^S[u]$, forms the keychain $\mathbb{S}'$ with statements up until the epoch $\mathsf{qepno}_{(u,h)}$ stored by the device, and updates $\mathsf{Users}[u'] \leftarrow (\mathbb{S}', \mathsf{auth}, \mathsf{epno})$ in the device state for (u, h).
  - $\mathsf{Query}(u)$: Instead of calling MVKD.Query(), $\mathcal{A}_{\mathsf{RZKS}}$ follows what the Query protocol does but calls its Query oracle in place of calling RZKS.Query().
  - $\mathsf{VerExtension}(t, t')$: Instead of calling MVKD.VerExtension(), $\mathcal{A}_{\mathsf{RZKS}}$ follows what the VerExtension protocol does but calls its ProveExt oracle in place of calling RZKS.ProveExt().
  - $\mathsf{Update}(M_{\mathsf{HonDev}}, M_{\mathsf{CorrDev}})$: Instead of calling MVKD.Update(), $\mathcal{A}_{\mathsf{RZKS}}$ follows what the Update protocol does but calls its Update oracle in place of calling RZKS.Update(). When updating $\mathsf{st}^S.\mathsf{Users}^S$, it also stores the authenticator in addition to the keychain.
  - $\mathsf{PCSUpdate}(M_{\mathsf{HonDev}}, M_{\mathsf{CorrDev}})$: Instead of calling MVKD.PCSUpdate(), $\mathcal{A}_{\mathsf{RZKS}}$ follows what the PCSUpdate protocol does but calls its PCSUpdate oracle in place of calling RZKS.PCSUpdate(). When updating $\mathsf{st}^S.\mathsf{Users}^S$, it also stores the authenticator in addition to the keychain.
  - LeakState: Instead of returning st^s_{epno}, $\mathcal{A}_{\mathsf{RZKS}}$ computes its own state $st^s \leftarrow (\mathsf{st}_{\mathsf{RZKS}}, \mathsf{Com}, \mathsf{Users}^S)$ and returns st^s. $\mathcal{A}_{\mathsf{RZKS}}$ computes $\mathsf{st}_{\mathsf{RZKS}}$ by querying its own oracle LeakState, Com from its own state it keeps track of, and Users^S also from its own state but only mapping username to the keychain instead of username to the keychain and authenticator.
  - $\mathsf{Ideal}(in)$: If in is an object related to the RZKS, then $\mathcal{A}_{\mathsf{RZKS}}$ answers with the response from the RZKS oracle $\mathsf{Ideal}(in)$. Otherwise, $\mathcal{A}_{\mathsf{RZKS}}$ simulates a random oracle via table D: it returns the value stored in D for in if available and otherwise samples a random output to return, which it also stores in D.
- $\mathcal{A}_{\mathsf{RZKS}}$ then returns whatever $\mathcal{A}_{\mathsf{MVKD}}$ returns.

When $\mathcal{A}_{\mathsf{RZKS}}$ is playing game $\mathsf{ZK\text{-}REAL}_{\mathsf{RZKS}}$, the oracles return outputs from the RZKS protocol. It is therefore easy to see that the view of $\mathcal{A}_{\mathsf{MVKD}}$ is identical to that in the $\mathsf{ZK\text{-}REAL}_{\mathsf{MVKD}}$ game, where the RZKS protocol is called when applicable, except for oracles HonVerExt, HonQuery, and Ideal. We argue that the view of $\mathcal{A}_{\mathsf{MVKD}}$ when calling these oracles is the same as in game $\mathsf{ZK\text{-}REAL}_{\mathsf{MVKD}}$:

- HonVerExt: Notice that MVKD.VerExtension() only fails for an honest device and server if the input commitment differs from the commitment stored by the server for the associated epoch. Furthermore, the algorithm only updates the device state to the input commitment if the commitment stored by the device is either $\perp$ or behind the input commitment. $\mathcal{A}_{\mathsf{RZKS}}$ therefore correctly simulates this behavior and updates the device state accordingly.

- HonQuery: Notice that MVKD.Query() only fails for an honest device and who has always interacted with the same honest server if the epoch the device sends to the server is greater than the epoch stored by the device or the epoch stored by the device is greater than the epoch stored by the server. Since the device is honest, neither case can occur. Furthermore, the algorithm updates the device state by updating Users for the queried username. $\mathcal{A}_{\mathsf{RZKS}}$ therefore correctly simulates this behavior and updates the device state accordingly.
- Ideal: For objects related to the RZKS, the RZKS oracle responds with the output from its own ideal object, just as it would in the real game. For simulating the hash function, $\mathcal{A}_{\mathsf{RZKS}}$ implements a random oracle; since we model the hash function as a random oracle, this is indistinguishable to $\mathcal{A}_{\mathsf{MVKD}}$.

We thus have that $\mathcal{A}_{\mathsf{RZKS}}$ returns 1 in game ZK-REAL$_{\mathsf{RZKS}}$ when $\mathcal{A}_{\mathsf{MVKD}}$ returns 1 in game ZK-REAL$_{\mathsf{MVKD}}$.

We now argue that when $\mathcal{A}_{\mathsf{RZKS}}$ is playing game ZK-IDEAL$_{\mathsf{RZKS}}$, the view of $\mathcal{A}_{\mathsf{MVKD}}$ is the same as that in the ZK-IDEAL$_{\mathsf{MVKD}}$ game, which we describe in more detail below.

- In the ZK-IDEAL$_{\mathsf{MVKD}}$ game during initialization, the MVKD simulator $\mathcal{S}_{\mathsf{MVKD}}$ calls the RZKS simulator $\mathcal{S}_{\mathsf{RZKS}}$ to get the public parameters pp′ and the initial commitment com′, which $\mathcal{A}_{\mathsf{MVKD}}$ expects to receive. When simulating this game, $\mathcal{A}_{\mathsf{RZKS}}$ gets com′ and pp′ from the RZKS simulator as input, so $\mathcal{A}_{\mathsf{MVKD}}$ will get back the commitment it expects.
- When simulating the DeviceSetup oracle for $\mathcal{A}_{\mathsf{MVKD}}$, $\mathcal{A}_{\mathsf{RZKS}}$ actually calls MVKD.DeviceSetup() to create the device key, while the ZK-IDEAL$_{\mathsf{MVKD}}$ game records in a table that a device was created. The view of $\mathcal{A}_{\mathsf{MVKD}}$ is the same, since it gets back the randomly-generated string that represents the device handle in both games.
- For the oracles AddFirstKey(u, h), RevokeKey(u, h_0, h_1), RevokeKey$((u, h), pk)$, and AddExtra$((u, h), d)$, the ideal-world game ZK-IDEAL$_{\mathsf{MVKD}}$ checks that these requests are valid by calling the ValidAction helper predicate and then records that these queries took place. Notice that $\mathcal{A}_{\mathsf{MVKD}}$ is not returned anything and the only action taken by the game is to record whether the algorithm succeeds. When $\mathcal{A}_{\mathsf{RZKS}}$ simulates this game, it calls the MVKD protocols, which also abort if there is an invalid request, and then records these queries took place. The MVKD protocols specified succeed when the submitted update is a valid keychain, which is the same as the check done by ValidAction.
- In the ZK-IDEAL$_{\mathsf{MVKD}}$ game oracle Sign(u, h, m), $\mathcal{S}_{\mathsf{MVKD}}$ is given the username u, device handle h, and message m to sign. The simulator stores a table Keys that maps a username and device handle to a key pair. If (u, h) already has a key pair, then that is used to sign m; otherwise, a new key pair is generated, stored, and used to sign. When $\mathcal{A}_{\mathsf{RZKS}}$ simulates this game, it calls MVKD.Sign(), which uses the key stored by the device. Since the keys remain consistent, the view of $\mathcal{A}_{\mathsf{MVKD}}$ is the same in both games.
- For oracle HonVerExt, it is clear by inspection that $\mathcal{A}_{\mathsf{MVKD}}$ performs the same checks as in ZK-IDEAL$_{\mathsf{MVKD}}$ and updates the honest device state when ZK-IDEAL$_{\mathsf{MVKD}}$ updates the epoch qepno for the device.
- For oracle HonQuery, it is clear by inspection that $\mathcal{A}_{\mathsf{MVKD}}$ performs the same checks as in ZK-IDEAL$_{\mathsf{MVKD}}$. It also updates the honest device state when ZK-IDEAL$_{\mathsf{MVKD}}$ updates the table T, with both updates storing the keychain stored in the directory updated during calls to Query. This ensures that, if the device is later corrupted, its state will be consistent.
- For oracle AddKey(u, h, pk), the honest device is the adding device while $\mathcal{A}_{\mathsf{MVKD}}$ acts as the device to be added. In order for what $\mathcal{A}_{\mathsf{MVKD}}$ sees when it plays this game to be consistent with what $\mathcal{A}_{\mathsf{RZKS}}$ returns from a call to MVKD.AddKey(), the simulator will need to ensure the following: (1) that the adding key is not revoked and (2) that pk is not a key already in the keychain for u. The leakage given to the simulator is the keychain from when the device last made a query, so the simulator can perform the above checks itself. Furthermore, ZK-IDEAL$_{\mathsf{MVKD}}$ views this as a malicious change and $\mathcal{A}_{\mathsf{MVKD}}$ gets the authenticator from ZK-IDEAL$_{\mathsf{MVKD}}$ — with $\mathcal{S}_{\mathsf{MVKD}}$ providing it to the game — so that it can submit the change as a malicious update to Update. This means that $\mathcal{S}_{\mathsf{MVKD}}$ does not need to later keep track of the authenticator computed by $\mathcal{A}_{\mathsf{MVKD}}$. Thus, the view of $\mathcal{A}_{\mathsf{MVKD}}$ remains the same in both games.
- For oracle AddKey(u, pk, h), the honest device is the new device getting added while $\mathcal{A}_{\mathsf{MVKD}}$ simulates the existing device doing the adding. Again, for what $\mathcal{A}_{\mathsf{MVKD}}$ sees when it plays this game to be consistent, the simulator will need to ensure the two properties the same as for oracle AddKey(u, h, pk) and additionally that the signature over the hash of the keychain statement that $\mathcal{A}_{\mathsf{MVKD}}$ receives is consistent with what the device to be added knows about its keychain. Since the leakage given to the simulator is the keychain from when the device last made a query, the simulator can perform these checks itself. Thus, the view of $\mathcal{A}_{\mathsf{MVKD}}$ remains the same in both games.
- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates the oracle AddKey(u, h_0, h_1), it calls MVKD.AddKey and, if the operation succeeds, it records this in $\mathsf{T}_{(u,h_0)}$. Note that MVKD.AddKey() succeeds if AuthStatement successfully creates the authenticator for both devices, the hashes in the authenticators match, and the signature over the hash provided by the added device verifies. Since only honest devices are used in this oracle, the hash is modeled as a Random Oracle, and the signature scheme SIG is strongly unforgeable, these checks should pass only if both devices have the same view of the keychain for u. Notice that this is precisely what the ZK-IDEAL$_{\mathsf{MVKD}}$ game checks for, so that $\mathcal{A}_{\mathsf{RZKS}}$ correctly simulates the view for $\mathcal{A}_{\mathsf{MVKD}}$.
- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates the oracle CorrDev(u, h), it simply returns the state of device (u, h) to $\mathcal{A}_{\mathsf{MVKD}}$. Note that the state of the device has the username, secret key, public key, the most recent commitment stored by the device, and the Users dictionary, which maps each username that the device knows about to the keychain, authenticator, and the epoch from when the device last queried for this user.
  The ZK-IDEAL$_{\mathsf{MVKD}}$ game simulates this by having the MVKD simulator $\mathcal{S}_{\mathsf{MVKD}}$ simulate the state using the leakage it is given. The key pair stored on the device is either generated if it has not been before, meaning $\mathcal{A}_{\mathsf{MVKD}}$ would not have seen anything from this key pair before, or it is looked up in the table Keys. The most recent commitment stored by the device is looked up since

part of the leakage is the epoch associated with the device by the game. Finally, Users can be simulated from the set Queries. Since both games are consistent, the view for $\mathcal{A}_{\mathsf{MVKD}}$ in each game is indistinguishable.

- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates the MVKD oracle Update($M_{\mathsf{HonDev}}, M_{\mathsf{CorrDev}}$), $\mathcal{A}_{\mathsf{RZKS}}$ calls MVKD.Update(), but substitutes the call to RZKS.Update() with its own Update oracle, which in the ZK-IDEAL$_{\mathsf{RZKS}}$ game returns the new commitment and update proof from the RZKS simulator. Before the Update oracle is called, $\mathcal{A}_{\mathsf{RZKS}}$ verifies that the update is valid by ensuring there is no more than one update per username, that each update results in a valid keychain, and that the authenticators verify according to CheckAuth.
  Notice that, assuming that ZK-IDEAL$_{\mathsf{MVKD}}$ (1) simulates the same checks performed in MVKD.Update and (2) causes $\mathcal{S}_{\mathsf{MVKD}}$ to provide $\mathcal{S}_{\mathsf{RZKS}}$ with the same leakage as in game ZK-IDEAL$_{\mathsf{RZKS}}$, then the view for $\mathcal{A}_{\mathsf{MVKD}}$ as simulated by $\mathcal{A}_{\mathsf{RZKS}}$ is indistinguishable to that in ZK-IDEAL$_{\mathsf{MVKD}}$. We now argue that both conditions are met when the server state has not been leaked. For Condition 1:
  - ZK-IDEAL$_{\mathsf{MVKD}}$ calls Consistent to verify that there are no duplicate updates for a single username and that all updates would result in a valid keychain.
  - For every malicious update in M_{CorrDev}, $\mathcal{S}_{\mathsf{MVKD}}$ checks whether it and $\mathcal{A}_{\mathsf{MVKD}}$ know the latest keychain for that user, by looking at K_{CorrDev} which it receives as part of the leakage. If it does not know the keychain, $\mathcal{A}_{\mathsf{MVKD}}$ does not know the keychain for the username either and is therefore submitting a fake update for which it must forge the authenticator. However, $\mathcal{A}_{\mathsf{MVKD}}$ cannot do this because of the strong unforgeability of SIG, so $\mathcal{S}_{\mathsf{MVKD}}$ can reject the update in this case. If $\mathcal{S}_{\mathsf{MVKD}}$ does know the latest keychain, then it verifies the update is valid by running CheckAuth and ValidKeychain and rejects otherwise.
  - Lastly, honest updates in M_{HonDev} are honestly generated and therefore would have correct authenticators that verify, so they do not need to be checked.

  For Condition 2, recall that the leakage provided to the RZKS simulator in game ZK-IDEAL$_{\mathsf{RZKS}}$ in the case where the server state has not been leaked is the number of elements to be added to the data structure and the set of labels in this update set which have previously been queried to the RZKS simulator when they had not yet been added to the data structure. For our MVKD protocol, labels that are not in the RZKS are queried each time a keychain is queried, since MVKD.Query() returns a non-inclusion proof for the next statement in the keychain to prove the server returns the complete keychain. $\mathcal{S}_{\mathsf{MVKD}}$ therefore keeps track of such labels in calls to Query through its set Q, which includes any labels that were queried after the last PCSUpdate, but were not present in the RZKS data structure at that time. ZK-IDEAL$_{\mathsf{MVKD}}$ leaks to $\mathcal{S}_{\mathsf{MVKD}}$ the set U_{HonDev} of usernames which receive honest updates and whose keychains $\mathcal{S}_{\mathsf{MVKD}}$ has previously seen. At this point, $\mathcal{S}_{\mathsf{MVKD}}$ also knows that it has the latest keychain for each malicious update since it checked for this earlier. For each username in the set of malicious updates and honest updates to keychains it knows, $\mathcal{S}_{\mathsf{MVKD}}$ can check whether the label to be added is in Q and, if so, add it to set C. We then have that $\mathcal{S}_{\mathsf{MVKD}}$ can call $\mathcal{S}_{\mathsf{RZKS}}$ with the total number of updates to be applied and the labels in C, which is the same leakage given to the RZKS simulator in game ZK-IDEAL$_{\mathsf{RZKS}}$ when played by $\mathcal{A}_{\mathsf{RZKS}}$. We therefore have that both conditions are met in the case that the server state has not been leaked.
  We also highlight that $\mathcal{S}_{\mathsf{MVKD}}$ keeps track of which keychains both it and $\mathcal{A}_{\mathsf{MVKD}}$ know to simulate responses. Notice that since $\mathcal{S}_{\mathsf{MVKD}}$ does not know the honest statements added, it now has an outdated view of the keychains for all usernames in U_{HonDev}. For the malicious updates, $\mathcal{S}_{\mathsf{MVKD}}$ can update $\mathsf{Dir}_{\mathsf{epno}}$ with the new keychains, since the update would only pass if $\mathcal{S}_{\mathsf{MVKD}}$ previously knew the up-to-date keychains. This allows future queries to also be consistent with what $\mathcal{A}_{\mathsf{RZKS}}$ would respond with.
  When the server state has been leaked, the simulator knows the current directory and receives the honest device updates as leakage, so the simulator can easily simulate the update algorithm itself. It verifies the malicious updates are valid as before. Recall that honest updates do not need to be checked because they must be valid by consequence of being honest updates. The expected leakage provided to the RZKS simulator in this case is the set of labels to be added to the data structure, which $\mathcal{S}_{\mathsf{MVKD}}$ can provide since it knows all the updates. Finally, $\mathcal{S}_{\mathsf{MVKD}}$ keeps track in the set K of which keychains both it and $\mathcal{A}_{\mathsf{MVKD}}$ know in order to simulate future responses.
- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates the MVKD oracle PCSUpdate($M_{\mathsf{HonDev}}, M_{\mathsf{CorrDev}}$), it calls MVKD.PCSUpdate(), but it replaces the call to RZKS.PCSUpdate() with its own PCSUpdate oracle. As for Update, notice that when ZK-IDEAL$_{\mathsf{MVKD}}$ (1) simulates the same checks performed in MVKD.PCSUpdate and (2) provides the same leakage to the RZKS simulator as in game ZK-IDEAL$_{\mathsf{RZKS}}$, then the view for $\mathcal{A}_{\mathsf{MVKD}}$ as simulated by $\mathcal{A}_{\mathsf{RZKS}}$ is indistinguishable to that in ZK-IDEAL$_{\mathsf{MVKD}}$.
  Since PCSUpdate functions nearly identically to Update except for the RZKS leakage, we only argue that Condition (2) is met. The RZKS leakage used in the game played by $\mathcal{A}_{\mathsf{RZKS}}$ is simply the number of updates. This number is directly leaked to and used by $\mathcal{S}_{\mathsf{MVKD}}$ in ZK-IDEAL$_{\mathsf{MVKD}}$, thereby showing that the necessary condition is met.
- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates oracle Query(u), it calls the interactive protocol MVKD.Query(), where $\mathcal{A}_{\mathsf{MVKD}}$ acts as the querying client, but it replaces the calls to RZKS.Query() with its own Query oracle. Notice that before calling its Query oracle it receives t^{last} from $\mathcal{A}_{\mathsf{MVKD}}$ and verifies that $t^{\mathsf{last}} \leq t$ and $t \leq \mathbf{t}(\mathsf{st}_{\mathsf{RZKS}})$ (or otherwise it halts and returns ERROR). When $\mathcal{A}_{\mathsf{RZKS}}$ does call its Query oracle for keychain statement $\mathbb{S}[i]$ for which it needs to retrieve the authenticator, the RZKS simulator is called with the leakage being:
  - $((u,i), \mathsf{val}_i, t_i, t)$ if $t_i \leq t$, where val_i is the associated auth for statement $\mathbb{S}[i]$ and t_i is the epoch when the statement was added
  - $((u,i), \perp, t_i, t)$ if $G[t_i] = G[t]$, meaning t_i and t are epochs during the same generation according to PCSUpdate
  - $((u,i), \perp, \perp, t)$ otherwise

Finally, $\mathcal{A}_{\mathsf{RZKS}}$ returns the keychain statement, authenticator, and proof for each statement added after the requested epoch by the client. We now show that $\mathcal{S}_{\mathsf{MVKD}}$ in $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ simulates this exact behavior so that the view for $\mathcal{A}_{\mathsf{MVKD}}$ as simulated by $\mathcal{A}_{\mathsf{RZKS}}$ is indistinguishable to that in $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$.
$\mathcal{S}_{\mathsf{MVKD}}$ performs equivalent checks to verify the epochs t, t^{last} are valid as $\mathcal{A}_{\mathsf{RZKS}}$ does. It next forms the RZKS leakage before querying the RZKS simulator on each label it needs to query. Note that it forms this leakage in the same way as does $\mathcal{A}_{\mathsf{RZKS}}$ and thus in either game $\mathcal{A}_{\mathsf{MVKD}}$ receives back the same outputs from the RZKS simulator since the leakage is consistent in either game. For the label not yet added, it adds the label to the set Q so that this can be used as part of the leakage for Update. Finally, it returns the keychain statement, authenticator, and proof for each statement added after the requested epoch by the client. Thus, the view for $\mathcal{A}_{\mathsf{MVKD}}$ in each game is indistinguishable.

- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates oracle $\mathsf{Audit}(t)$, it calls $\mathsf{MVKD.Audit}()$, where $\mathcal{A}_{\mathsf{MVKD}}$ acts as the auditor interacting with an honest server simulated by $\mathcal{A}_{\mathsf{RZKS}}$. $\mathcal{A}_{\mathsf{RZKS}}$ follows the MVKD protocol by sending $\mathcal{A}_{\mathsf{MVKD}}$ the update proof for the requested epoch. Since $\mathcal{A}_{\mathsf{RZKS}}$ is in the $\mathsf{ZK\text{-}IDEAL}_{\mathsf{RZKS}}$ game, this update proof was generated by the RZKS simulator.
  The $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ game calls $\mathcal{S}_{\mathsf{MVKD}}$, with the leakage being the epoch to be audited. $\mathcal{S}_{\mathsf{MVKD}}$ sends $\mathcal{A}_{\mathsf{MVKD}}$ the update proof for the requested epoch. Recall that this update proof also came from the RZKS simulator during Update. Thus, the view for $\mathcal{A}_{\mathsf{MVKD}}$ is the same in both games.
- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates oracle $\mathsf{VerExtension}(t, t')$, it calls $\mathsf{MVKD.VerExtension}()$, where $\mathcal{A}_{\mathsf{MVKD}}$ acts as the client interacting with an honest server simulated by $\mathcal{A}_{\mathsf{RZKS}}$. $\mathcal{A}_{\mathsf{RZKS}}$ follows the MVKD protocol by sending the result of ProveExt to $\mathcal{A}_{\mathsf{MVKD}}$, but instead of calling $\mathsf{RZKS.ProveExt}()$ it calls its own oracle ProveExt. Notice that this means the oracle verifies that the epochs queried are valid. Furthermore, since $\mathcal{A}_{\mathsf{RZKS}}$ is in the $\mathsf{ZK\text{-}IDEAL}_{\mathsf{RZKS}}$ game, the resulting proof was generated by the RZKS simulator, where the provided leakage is the queried epochs. This is identical to what the $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ game does, so that the view for $\mathcal{A}_{\mathsf{MVKD}}$ is the same in both games.
- When $\mathcal{A}_{\mathsf{RZKS}}$ simulates oracle LeakState, it returns the simulated server state st^s as described above. We now show that when $\mathcal{A}_{\mathsf{RZKS}}$ is in the $\mathsf{ZK\text{-}IDEAL}_{\mathsf{RZKS}}$ game, the simulated state provided to $\mathcal{A}_{\mathsf{MVKD}}$ by $\mathcal{A}_{\mathsf{RZKS}}$ is indistinguishable to the state provided by the $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ game. In particular, st^s computed by $\mathcal{A}_{\mathsf{RZKS}}$ contains three parts: $\mathsf{st}_{\mathsf{RZKS}}$, Com, Users^S.
  $\mathcal{A}_{\mathsf{RZKS}}$ computes $\mathsf{st}_{\mathsf{RZKS}}$ by querying its own oracle LeakState, which calls the RZKS simulator with the entire datastore D being the leakage. Notice that $\mathcal{S}_{\mathsf{MVKD}}$ in game $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ computes this in the same way by calling $\mathcal{S}_{\mathsf{RZKS}}$ with the same datastore, which $\mathcal{S}_{\mathsf{MVKD}}$ can compute because it is given the entire directory as leakage.
  $\mathcal{A}_{\mathsf{RZKS}}$ returns Com from its own state it keeps track of, which are the list of RZKS commitments computed by the RZKS simulator for each update. Since $\mathcal{S}_{\mathsf{MVKD}}$ is called for each update, it also knows and stores the same Com.
  Finally, $\mathcal{A}_{\mathsf{RZKS}}$ computes Users^S also from its own state but only mapping username to the keychain instead of username to the keychain and authenticator. Since $\mathcal{S}_{\mathsf{MVKD}}$ is given the entire directory as leakage, it can compute this same set Users^S. Therefore, the view for $\mathcal{A}_{\mathsf{MVKD}}$ is the same in both games.
- It is clear to see by inspection that when $\mathcal{A}_{\mathsf{RZKS}}$ is playing game $\mathsf{ZK\text{-}IDEAL}_{\mathsf{RZKS}}$ and simulates Ideal, the view of $\mathcal{A}_{\mathsf{MVKD}}$ is the same as that in the $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$ game.

We thus have that $\mathcal{A}_{\mathsf{RZKS}}$ returns 1 in game $\mathsf{ZK\text{-}IDEAL}_{\mathsf{RZKS}}$ when $\mathcal{A}_{\mathsf{MVKD}}$ returns 1 in game $\mathsf{ZK\text{-}IDEAL}_{\mathsf{MVKD}}$.