SoK: On the Security Goals of Key Transparency Systems

Nicholas Brandt[1], Mia Filić[2], and Sam A. Markelon[3]

[1] ETH Zurich, Zurich, Switzerland, crypto@nicholasbrandt.de
[2] ETH Zurich, Zurich, Switzerland, filicmia@gmail.com
[3] University of Florida, Gainesville, FL, USA, smarkelon@ufl.edu

Abstract. Key Transparency (KT) systems have emerged as a critical technology for adding verifiability to the distribution of public keys used in end-to-end encrypted messaging services. Despite substantial academic interest, increased industry adoption, and IETF standardization efforts, KT systems lack a holistic and formalized security model, limiting their resilience to practical threats and constraining future development. In this paper, we survey the existing KT literature and present the first cryptographically sound formalization of KT as an ideal functionality. Our work clarifies the underlying assumptions, defines core security properties, and highlights potential vulnerabilities in deployed KT systems. We prove in the Universal Composability framework that our concrete protocol achieves KT security as defined by our formalism. Our KT protocol builds on the latest trends in KT design, guided by the formalization.

Contents

1 Introduction

The security of end-to-end encrypted communication systems relies on the authenticity of the public keys of the participating parties. Traditionally, verifying the authenticity of another party's public key in secure communication systems required either physical meetings to exchange keys—a cumbersome process, especially with frequent key rotations and new device additions—or reliance on a third-party authority. Key Transparency (KT) systems address these challenges by providing an automated mechanism that allows users to verify they are receiving the correct public key, or at least one that is consistent with what other users are seeing from the same service, while preserving privacy. That is, unlike traditional public key infrastructure systems which require a trusted (third) party, KT systems aim to reduce or even remove such trust assumptions.

KT systems have not only attracted significant academic interest [MBB+15; Bon16; CDG+19; TBP+19; TKP+21; HHK+21; TFZ+22; CDG+22; MKS+23; LCG+23a; LCG+23b], but have also been implemented (or proposed to be implemented) by platforms such as Keybase [Mar23], Zoom [BBC+22], Google [HB20], WhatsApp [Lew23], Apple iMessage [App23], and Proton [GH24]. Complementing this industry adoption, the Internet Engineering Task Force (IETF) has formed the KEYTRANS working group [McM25; ML24] to both formalize and standardize KT systems.

Despite the growing body of research on KT, its core operational and security goals remain difficult to distill from the existing literature. Unlike Certificate Transparency (CT), which has been subjected to rigorous cryptographic analysis and has a well-defined set of security properties [DGH+16; CM16], KT research is still evolving in a fragmented manner. Various works focus on the efficiency and/or security of different components but there is no formal definition of what a KT system is or what *exactly* it should aim to achieve. This lack of a proper conceptualization of KT systems makes it impossible to clearly state the security assumptions and guarantees that a given KT system makes. To substantiate our claim about the lack of a holistic approach to KT we survey the current KT literature in Section 3; systematizing the common security goals and assumptions that support these goals.

1.1 Clear Exposition of Security Guarantees and Assumptions

Given the lack of a formal definition of KT systems, we argue that it is unclear how security guarantees (e.g., about certain building blocks) and assumptions (e.g., about the behavior of certain parties) made in the KT literature translate to the real world. Moreover, the gap between theoretical KT research and practical deployment further complicates efforts to extract clear operational and security goals. Without practical deployment insights, it is difficult to determine whether a proposed KT system provides security guarantees that are achievable in real-world settings or whether it is merely a theoretical construct with untested assumptions.

To support the claimed mismatch between theoretical modeling of KT systems and deployment we observe that Whatsapp's KT system [LL24b] is based on the SEEMless construction and security analysis of [CDG+19]. However, the security proof of the construction in [CDG+19] requires (as a building block) a *simulatable* verifiable random function (sVRF) [CL07] whereas WhatsApp's VRF implementation [LL24a] *cannot* be simulatable.[1] Since the security proof of [CDG+19] cannot be applied to Whatsapp's KT protocol, provable security cannot be claimed for their protocol on the basis of [CDG+19].[2]

Another example of the rather complicated state of KT literature is the fact that in state-of-the-art protocols a user cannot prevent their impersonation. Instead, the intuitive security guarantee is that an impersonated user can detect an attack against them by querying their own public key. We stress that [CDG+19] clearly states that their "soundness" notion requires a user to audit their own public key.[3] However, what happens if an incorrect public key is detected is left unspecified. Similarly, Melara, Blankstein, Bonneau, Felten, and Freedman [MBB+15] suggest a user to "whistleblow" the attack to auditors "via social media or other high-traffic sites". They "leave the complete specification of a whistleblowing protocol for future work"; we are not aware of any follow-up work. While such a security notion is formally valid, they fall short of what most (non-tech-savy) users would intuitively expect from a secure KT system. In particular, users generally assume that the system should prevent impersonation in the first place, not merely allow its detection. Indeed, Melara, Blankstein, Bonneau, Felten, and Freedman [MBB+15] already address this issue informally by distinguishing between a "default" and a "strict" policy. Using our formal definition of KT functionality one can show that for this limitation is inherent default mode user (those that cannot be assumed to remember a high-entropy secret).[4]

1.2 A Formal Framework for KT Systems

Previous works, [MBB+15; Bon16; CDG+19; TBP+19; TKP+21; HHK+21; TFZ+22; CDG+22; MKS+23; LCG+23a; LCG+23b], to varying degrees, have treated key transparency (KT) systems as a monolithic block, i.e., they fail to cleanly distinguish between the specification (the objective) of a KT systems, and the protocol that realizes that specification (or they don't give protocols at all). As a result, the security formalisms for KT systems—when they are considered at all—tend to focus on the properties of individual components, rather than establishing rigorous security guarantees for the system as a whole.

In our view, this complicates the analysis of KT systems and the interpretation of their security guarantees. This situation is somewhat reminiscent of the early stages of the development of other multi-party computation (MPC) applications [Yao82], like key exchange—where the word "key-exchange" was used synonymously with the concrete Diffie–Hellman key-exchange protocol [DH76]. In contrast, the

[1] In fact, the unconditional unique provability property of WhatsApp's `ECVRF-EDWARDS25519-SHA512-TAI` [GRP+23] VRF strongly contradicts (in a formal sense) the simulatability of a sVRF as required by [CDG+19].

[2] This does not necessarily mean that there is an attack against WhatsApp's protocol.

[3] For provable security every user would have to audit their own key at every epoch, which is not practical.

[4] On a technical level, a malicious server can always simulate a (targeted) user towards a third (honest) user, if the users don't share secret information.

field today draws a clear distinction between the abstract objective of key exchange—typically defined as an "ideal" *functionality*—the concept of a public-key encryption (PKE) *scheme*, and any specific key-exchange *protocol* that realizes the functionality using a PKE scheme.

To remedy the above issues we provide a unified and formal framework that captures the desired properties of KT systems (again, which we identify through a systematic analysis of the existing literature in Section 3) in the Universal Composability (UC) model [Can01]. In Section 4 we provide a formal definition of a KT system as an ideal functionality. Then, in Section 5 we realize this functionality using a protocol, with (UC-)proofs in Appendix C.2.

1.3 Perspective

We hope that this works serves to provide clarity on what the objectives and guarantees of KT systems are and that our formal framework can be used to analyze existing KT systems and future ones in a common and rigorous manner. Importantly, we emphasize that our findings are *not* meant as a criticism of previous works; each work provides valuable insights to the problem at hand. Rather, we feel that it is important to highlight conceptual problems as early and as clearly as possible to prevent their solidification in standardizations[5]; in particular in light of the standardization efforts of the KEYTRANS workgroup [McM25; ML24]. We believe that specifying what exactly an adversary is allowed to do (in form of an ideal functionality) contributes to the clarification of the necessary security and operational assumptions, as well as the resulting security guarantees. In turn, this aids servers and users in understanding what level of security is given under given explicit operational assumptions.

1.4 Contribution

We summarize the main contributions in this work:

- In Section 3, we survey the existing KT literature to systematically distill the common goals and operational assumptions of various KT schemes. In turn, we leverage this analysis to help us specify our idealized KT functionality.
- In Section 4, we model the (idealized) objective of a KT system as an ideal functionality and show that this idealized version cannot be achieved due to inherent attacks. We propose a weakened ideal functionality that still captures the essence of KT systems. This clarifies the assumptions, features and security guarantees of KT systems which have been somewhat recondite in previous literature.
- We formally prove (in the Universal Composability framework [Can01]) that a protocol (given in Figure 3) inspired by the (implicit) protocols in the literature [CDG+19; LCG+23b] realizes this functionality. We stress that obtaining a formal and composable (UC) proof for the implicit protocols in the literature is not straightforward. Because existing work focuses on security guarantees for components of a KT system, it requires significant effort to design a entire protocol and prove its security. Moreover, because we aim for UC-security the security properties that we require from the protocol components exceeds the capabilities of all previous works. In particular, our data structure for storing keys (KT scheme) is *extractable* and *simulatable* whereas [CDG+19] only achieves simulatability.
- Notably, our work represents the first systematization of knowledge (SoK) on KT, capturing the current state of the field. By clarifying the status quo, it lays a foundation for future progress and makes it easier for the community to reason about, refine or extend the objectives and design goals of KT. Should the objectives and design goals of KT systems evolve significantly, we still believe this SoK will serve as an important milestone.

[5] As was the case with standards such as early versions of SSL. [Sch22]

2 Related Work

2.1 Comparison with Certificate Transparency

Certificate transparency (CT) is a widely deployed system designed to enhance the security of the web public key infrastructure by providing an append-only, publicly auditable log of TLS certificates issued by certificate authorities (CAs) [Lau14]. This ensures that any mis-issued or maliciously issued certificates (e.g., a rogue CA signing a certificate for `google.com`) can be detected and revoked before causing harm. CT operates through public, verifiable logs where any entity—browsers, domain owners, or independent auditors—can monitor the issuance of certificates to detect anomalies.

While there is a natural relation to key transparency, CT logs fully public certificates, whereas KT must carefully balance verifiability with privacy. Publishing a global log of all users' public keys, as CT does for certificates, could expose sensitive metadata and enable surveillance. KT must also handle frequent key updates (e.g., when users switch devices or rotate keys), whereas CT primarily records relatively static certificate issuance events. While both systems rely on append-only logs to prevent equivocation, CT ensures that domain owners can detect unauthorized certificates, whereas KT ensures that users are communicating using the correct, i.e., globally consistent, public key—without needing to trust a central provider. KT provides a mechanism for users to efficiently query their own key state without downloading the full log. This portrays grounds for better scalability, and minimizes data exposure—making it more suitable for user-centric secure communication.

Moreover, unlike in the case of KT, there have been efforts to formally analyze the end-to-end security guarantees of CT. Most notable are the works of Dowling, Günther, Herath, and Stebila [DGH+16], and Chase and Meiklejohn [CM16]. Dowling, Günther, Herath, and Stebila [DGH+16] highlight the need for formal security analysis of CT beyond its practical deployment, aiming to define provable security goals and demonstrate that CT meets the goals under standard cryptographic assumptions. Prior work primarily described CT's mechanism but lacked rigorous security proofs, which this study addresses through a game-based formal model. The authors establish four key security properties for transparent logging schemes such as CT and formally prove that a ubiquitous CT protocol does achieve these properties. Their proofs rely on Merkle tree collision resistance and unforgeable signatures. However, they highlight that these CT formalizations are insufficient for KT protocols due to the lack of privacy guarantees and different key consistency-check requirements. In particular, they contrast CT with the CONIKS KT system [MBB+15].

Chase and Meiklejohn [CM16] introduce a generalized cryptographic model for transparency, defining transparency overlays as a cryptographic primitive called dynamic list commitments. This framework enables a formal security analysis of transparency systems, which the authors apply to CT and Bitcoin, proving their transparency properties. Similarly to [DGH+16], the formalization does not extend to KT due to its additional privacy requirements and efficiency constraints.

2.2 Key Transparency Literature we Survey.

We now briefly introduce the literature we survey in depth in Section 3. We select papers that aim to present full end-to-end KT protocols to do our systematic analysis on.

Melara, Blankstein, Bonneau, Felten, and Freedman introduce the first KT system, CONIKS. In particular, they introduce the idea of Merkle prefix trees to guarantee that all users see the same key bindings and enabled clients to periodically verify consistency proofs to detect tampering. Additionally, the system incorporates privacy-preserving mechanisms to limit unnecessary exposure of key data. Later, Chase, Deshpande, Ghosh, and Malvai [CDG+19] present their KT system, called SEEMless, which improves CONIKS in several ways. In particular, [CDG+19] takes a more modular approach than [MBB+15]. Concretely, Chase, Deshpande, Ghosh, and Malvai introduce two primitives called the "verifiable key directory" (VKD) and the "append-only zero-knowledge set" (aZKS), and use them in black-box way

in their solution. We view this as an important step toward advancing KT systems—both in terms of efficient constructions and minimizing the underlying assumptions. Moreover, Chase, Deshpande, Ghosh, and Malvai define formal security properties of their primitives and provide proof sketches for the security of their primitives. Overall, their work adopts a more formal approach compared to that of [MBB+15].

Building on these works, Malvai, Kokoris-Kogias, Sonnino, Ghosh, Oztürk, Lewi, and Lawlor [MKS+23] introduce Parakeet. Parakeet addresses scalability concerns by introducing a more efficient VKD for large-scale deployments. Further, they propose a compaction mechanism to limit the growth of the KT log over time and a novel gossip protocol to distribute the log's commitments and ensure consistency. We also analyze OPTIKS by Len, Chase, Ghosh, Laine, and Moreno [LCG+23b] which provides further optimizations and scalability improvements. Further, we examine ELEKTRA by Len, Chase, Ghosh, Jost, Kesavan, and Marcedone [LCG+23a] which is focused on the multi-device setting, in addition to improved privacy and post-compromise security for KT systems. Lastly, we also analyze the IETF KEYTRANS working group architecture draft [McM25] and protocol draft [ML24].

Further, there are a number of works that introduce valuable components (auditable logs, auditing protocols, etc.) that could be used in key transparency systems [Bon16; TBP+19; LGG+20; HHK+21; TKP+21; TFZ+22; CDG+22]. However, these papers do not attempt to put forth full KT protocols, so we do not include them in Section 3. Moreover, we stress that the IETF KEYTRANS working group architecture and protocol draft are strictly to be considered works in progress. We analyze the latest versions at the time of creation of this paper (architecture draft version 03 and protocol draft version 00), but again emphasize that they are subject to updates and improvements.

3 Distilling the Security Goals of KT Systems

We start by surveying the literature for the *desired security* goals of KT systems. We then systemize these goals, distilling them into a set of notions, categorized into cryptographic guarantees and the system-level assumptions that support these goals. For each goal, we pull direct quotes from the relevant literature supporting their inclusion in our analysis. Wherever possible, we support our analysis with direct quotations from the original sources. For the sake of readability, the full text of each quote is provided in Appendix F, and we reference them throughout our discussion.

As stated in Section 2, our analysis covers what we consider to be the core corpus of KT protocols: CONIKS [MBB+15], SEEMless [CDG+19], Parakeet [MKS+23], OPTIKS [LCG+23b], ELEKTRA [LCG+23a], and the IETF KEYTRANS architecture draft [McM25] and protocol draft [ML24]. These works present full or nearly full protocol designs and include formal definitions for key components. However, they stop short of clearly articulating full protocol specifications or holistic security guarantees. That is, while these papers formalize *primitives* such as verifiable key logs or authenticated dictionaries, they do not rigorously define the precise security properties a complete KT *protocol* should satisfy.We note one partial exception: the ongoing development of the IETF protocol draft [ML24], which includes a partial protocol description. However, at the time of writing, it does not yet include a formal treatment of security considerations and is a work in progress.

As an explicit example of this gap, we take a deeper look at SEEMless [CDG+19]. While the paper specifies the underlying KT scheme with relative clarity, it never formalizes the surrounding protocol or its intended security guarantees. In particular, the scheme does not inherently achieve even *weak consistency* (see below): users' views of the state of the key log can diverge indefinitely unless users explicitly perform key history checks very frequently (i.e., check for the authentic inclusion of their own key in the log every epoch). We present a detailed description of this issue in Appendix E. The authors acknowledge the importance of these checks, but leave the exact necessary frequency and enforcement of them to future work. This means that any consistency—or any protocol-level security property—depends entirely on how the scheme is used, yet no concrete protocol is proposed. Furthermore, their experimental evaluation assumes users only check their own key history after their own key updates, which undermines

the KT system's core goal of detecting equivocation in others' key bindings. Additionally, the paper's narrative, particularly the section in which the authors give an intuition for their construction, offers various optimizations without clarifying how they meaningfully improve the security of the system or what specific goals they aim to achieve. The result is a system that raises the cost of server misbehavior, but neither clearly prevents it nor clearly defines the (reduced) level of security it guarantees. Without a formal protocol definition—including specifics such as history check frequency—even the minimal KT guarantees appear to remain unsatisfied (or, at best, informally described).

We also observe that the security and operational goals of KT systems rely on a number of system-level assumptions. While we systemize these assumptions in detail below, we first provide a brief overview as presented in OPTIKS [LCG+23b], where they are most clearly articulated. In OPTIKS, the server is assumed to be fully malicious with respect to distributing incorrect keys to users, but it is still expected to enforce access control and preserve the privacy of the key log. User devices, on the other hand, may also be malicious, in the sense that they could attempt to access private information (e.g., the public keys of users they are not authorized to communicate with). OPTIKS further assumes the existence of a public bulletin board that allows users to receive a singular, consistent view of the key log, along with at least one honest auditor responsible for verifying the correctness of the server's commitments. An auditor is a trusted third party who posts a signed statement of correctness of a given commitment to the same bulletin board. In practice, a new commitment to the key log is posted at regular intervals, typically over a short, fixed period known as an *epoch*. During each epoch, the server processes a batch of key updates it has received, updates its internal key log accordingly, computes a commitment to the updated state, and publishes this commitment to the public bulletin board, where it can be audited and retrieved by users. The precise specification of this bulletin board is left as an open problem, although the authors note that a public blockchain could be used for this purpose. Additionally, it is assumed that users are able to track their devices and the approximate times of their own key updates. This enables them to compare the current state of the key log with their expected update history from a user-centric perspective. Finally, it is assumed that users, the server, auditors, and the bulletin board all maintain approximately synchronized clocks.

Given the current state of the literature, it is a nontrivial task to distill a coherent set of holistic, protocol-level goals for KT systems, along with the system-level assumptions required to support these goals. Nonetheless, this is a crucial step toward understanding what a formal model of KT protocols *should* capture. We undertake this task below. Specifically, we define the cryptographic guarantees that KT systems appear to aim for and examine what the literature explicitly says about these goals—supported by full textual quotes provided in Appendix F. We follow the same approach for identifying and analyzing the system-level assumptions underlying these protocols. Leveraging this analysis, we develop a formal framework for the security of KT protocols (Section 4), rigorously define a concrete protocol (Section 5), and formally prove its security with respect to our model (Appendix C.2). For the guarantees described below, where possible, we adopt terminology aligned with the current literature, while clearly distinguishing between different types of guarantees to avoid ambiguity.

Importantly, our goal here is not to over-interpret or extend the original works beyond what they themselves make explicit. Instead, we aim to faithfully reflect the level of specificity given in each source and present a consolidated view of the protocol-level guarantees that are either stated or suggested. Where contradictions or inconsistencies arise, we make them explicit. In some cases, we highlight how certain guarantees may not hold under plausible interpretations of the respective works. Ultimately, this effort is meant to clarify the current state of KT protocol design and serve as a step toward more rigorous, comprehensive modeling.

3.1 System-Level Assumptions

We start by analyzing the common system-level assumptions of KT systems as these inform what is possible to cryptographically guarantee in realistic settings.

Split-View Resistance: *The KT server should not be able to serve distinct, conflicting views of the state of the key directory to different users. That is, a broadcast functionality exists such that all parties are able to retrieve a consistent commitment to the key directory.*

Split-view resistance refers to the system-level assumption that users should be able to retrieve a singular and consistent commitment to the state of the key log at any given point in time. Without this property, it is not possible to bootstrap any meaningful cryptographic consistency guarantees. This requirement was recognized early by CONIKS (Quote 1) and has been reaffirmed in the IETF KEYTRANS architecture draft (Quote 30). In practice, achieving split-view resistance remains a challenge. Most works address it by assuming the existence of a public bulletin board—an abstraction that appears in SEEMless (Quote 7), OPTIKS (Quote 19), and ELEKTRA (Quote 27). We adopt this assumption in our work as well, as our focus is on the core cryptographic guarantees of KT protocols. Various mechanisms have been proposed to instantiate this bulletin board, including public blockchains and gossip protocols. However, Parakeet argues that such approaches are unnecessarily heavyweight and instead introduces a novel consensus-less protocol to achieve split-view resistance (Quote 14). Further research is needed to formalize the notion of a public bulletin board and to design efficient, robust protocols that can reliably implement this functionality.

Out-of-band Whistleblowing: A fundamental limitation of KT systems is that they cannot prevent misbehavior by a malicious server—they can only detect when such behavior has occurred. A subtle but critical assumption underlying all consistency properties is that when a user checks the validity of their own key, the result confirms that the key is indeed correct. If this check fails (i.e., the user detects that their key has been tampered with–then these consistency guarantees begin to break down). A significantly underexplored area in the KT literature is how the system should respond once misbehavior is actually detected. CONIKS identified this issue early and emphasized the need for a whistleblowing mechanism to expose server misbehavior to others (Quote 2). This need is similarly echoed in the IETF KEYTRANS architecture draft (Quote 31). However, neither CONIKS nor subsequent works provide a concrete or standardized mechanism for whistleblowing, beyond the observation that some form of authenticated, peer-to-peer, out-of-band communication is required. This gives rise to a subtle paradox: such a communication channel implicitly assumes the existence of a trustworthy public key infrastructure (PKI) to authenticate the whistleblower to others, yet the very motivation for KT systems is to serve as a secure alternative to traditional PKI. In other words, enabling users to securely report KT misbehavior appears to require a PKI that, in theory, KT is designed to replace.

Privacy via Access Control: *Users should only be able to query for public keys that they are authorized to access. This ensures that KT does not expose all registered identities in a way that could enable mass surveillance or enumeration attacks.*

Privacy via access control is the system-level assumption that underpins formal privacy guarantees in key transparency systems. It defines which set of users are permitted to issue queries about a particular user—or, more generally, how frequently users are allowed to interact with the system. In practice, this means that only users who appear in each other's contact lists should be allowed to issue KT queries about one another. In the absence of a contact-based model, rate limiting can be employed to prevent exhaustive enumeration of the key log. This assumption is explicitly stated in the literature, including by CONIKS (Quote 3), SEEMless (Quote 8), OPTIKS (Quote 20), and the IETF KEYTRANS architecture draft (Quote 32).

Censorship Resistance: *A KT provider should not be able to suppress or delay key queries or updates in a way that selectively prevents certain users from updating or verifying their keys. This prevents targeted denial-of-service attacks on specific identities.*

Numerous works have emphasized that a KT server should not be able to suppress updates—either by delaying the release of new commitments beyond the designated epoch interval or by selectively ignoring updates from certain users. This requirement is explicitly stated in CONIKS (Quote 4), SEEMless (Quote 9), Parakeet (Quote 15), and the IETF KEYTRANS architecture draft (Quote 33). Despite this, concrete mechanisms for enforcing such guarantees remain underexplored. In particular, distinguishing between malicious censorship by the server and benign failures—such as server outages or network disruptions—has yet to be rigorously addressed in the literature. Moreover, there is no proposed mechanism for dealing with such behavior, beyond the underspecified mechanism of whistleblowing.

3.2 Cryptographic Guarantees

We now put forth various notions of consistency and a notion of privacy for KT systems. These are derived from an intuitive sense of what one would desire from KT systems, the above system-level assumption, and direct evidence from the literature.

Strong Consistency: *A KT system satisfies strong consistency if, for any honest user U, querying the key transparency directory for the public key of another honest user V always returns the latest, unique, authentic public key registered by V. This prevents key substitution attacks by malicious identity providers.*

Strong consistency represents the gold standard of what can be cryptographically achieved and guaranteed by a key transparency (KT) protocol. However, in practice, realizing this property—particularly for all users within a system—is largely aspirational. This is because existing KT systems rely on the assumption that users frequently check the validity of their own public keys, ideally at each update to the system's database (e.g. epoch). In reality, such behavior is impractical: it would require users to remain continuously online and to run the KT-enabled application indefinitely.

This limitation is well acknowledged in the existing KT literature. Works that attempt to provide formal definitions typically define a notion analogous to our strong consistency property under the term soundness, as applied to their underlying verifiable log primitives [CDG+19; MKS+23; LCG+23b; LCG+23a] (SEEMless [CDG+19] refers to this primitive as a verifiable key directory, while OPTIKS [LCG+23b] describes it as a private authenticated history dictionary). As expected, these definitions carry an important caveat: strong consistency is only guaranteed if users check the validity of their own key within a time window that is "current" relative to when another user queries it. In addition, an honest auditor must verify the correctness of the log commitment published by the server to a public bulletin board. The burden thus falls primarily on the user. This requirement is explicitly stated in SEEMless (Quote 10) and reaffirmed in OPTIKS (Quote 21, Quote 22). Similarly, Parakeet claims to achieve strong consistency under these same conditions (Quote 16), while also highlighting the role of honest auditors in ensuring the property holds (Quote 17).

In reality, strong consistency is fundamentally unattainable in any setting where the server may be malicious. A server can always attempt a key substitution attack; the best a KT protocol can offer is the ability to detect such behavior. Thus, the security condition underlying strong consistency implicitly assumes that the user not only performs a validity check of their own key, but that the check *passes*. Without this, the consistency guarantee does not hold. Therefore, KT systems can only hope to achieve a frailer, detection-based variant of strong consistency: a KT system satisfies this relaxed form if, for any honest user U querying the directory for another honest user V, the response is either (i) the latest, unique, and authentic public key registered by V, or (ii) a response that allows U to detect that the key they received is not valid. However, as discussed in our section on whistleblowing, it remains unclear

how systems should handle detected misbehavior in practice, and what consequences should follow from such a detection.

Weak Consistency: *Given a commitment C to a KT log state, any two honest users querying the log should receive the same public key for a given identity. This ensures that different users observe the same key bindings when relying on the same transparency commitment.*

Weak consistency is a more realistic, albeit further from the "spiritual" consistency goal of KT systems. For systems like OPTIKS [LCG+23b] and ELEKTRA [LCG+23a], weak consistency can be achieved without requiring users to *ever* check for the validity of their own key in the log. This is also (on a high level) what we formally prove our KT protocol (Section 5) achieves. Moreover, we intuitively explain under what conditions our protocol would satisfy stronger consistency notions.

For works that provide formal security definitions, a notion akin to weak consistency can be extracted from primitives one level of abstraction below the verifiable log—namely, the (ordered) append-only zero-knowledge set (aZKS). The soundness definitions of aZKS structures [CDG+19; LCG+23b; LCG+23a] are essentially equivalent to our definition of weak consistency. An aZKS is a cryptographic data structure that underpins verifiable (label-value) logs. It can be viewed as an append-only variant of a Merkle tree [Mer87] that supports efficient membership and non-membership proofs, while also having additional privacy guarantees.

SEEMless [CDG+19] formalizes soundness for their aZKS primitive as the property that no malicious prover can generate two valid proofs for different values under the same label (user id and public key version tuple in SEEMless) with respect to a fixed commitment (Quote 11). While SEEMless's aZKS soundness definition might suggest that the protocol satisfies weak consistency by default—without additional behavioral assumptions on users—we demonstrate unequivocally that this is not the case in Appendix E. This underscores the importance of analyzing complex systems like KT from a holistic perspective.

Similarly, ELEKTRA claims to satisfy an analogous property, which they simply refer to as consistency (Quote 28). The IETF KEYTRANS architecture draft also explicitly names weak consistency (or rather, an equivalent notion) as a goal for KT systems (Quote 34), and the accompanying protocol draft claims that their proposed protocol satisfies this goal as well (Quote 37)—although as the work is still in early stages and no formal notions of security exist this claim is difficult to judge.

Relaxed Consistency: *A KT system satisfies relaxed consistency if a user querying for a public key receives an authenticated key under predefined conditions, such as temporal consistency constraints or network partition scenarios; or misbehavior by the server is detected. This allows for flexibility while ensuring that users eventually converge to the same authentic view.*

Relaxed consistency can be viewed as a middle ground between the goal of strong consistency (or more specifically the frailer notion that can only detect misbehavior) and the more practical goal of weak consistency. At a high level, it guarantees that a user will *eventually* receive the authentic public key of another user, contingent on a specific event. In practice, this means that a user can be confident in the authenticity of another user's public key, but only after that other user has verified the correctness of their own key. If the server has behaved maliciously, this misbehavior will be detected once the targeted user checks the validity of their own key. However, this notion does not impose strict timing guarantees—it merely assumes that the user will perform this check at some point in the future.

SEEMless acknowledges that requiring users to check their key every epoch is unrealistic. Instead, it suggests that consistency or misbehavior detection can still be achieved as long as users verify the validity of their own key sufficiently often (Quote 12, Quote 13). However, the precise meaning of "sufficiently often" is left undefined. Similarly, the IETF KEYTRANS architecture draft asserts that, assuming user devices do not remain permanently offline, any malicious behavior by the server will eventually be

detected within a bounded time frame (Quote 35). In this model, detection occurs when the targeted user eventually checks the validity of their own key.

Privacy: *The KT system should minimize information leakage to external observers. This means that an adversary who does not have explicit access rights should not be able to learn non-trivial information about key bindings or update patterns beyond what is explicitly made public.*

Unlike certificate transparency, KT systems explicitly aim to provide privacy as a core security goal. Specifically, KT systems are designed to hide key bindings and update patterns associated with individual users—including their very existence on the platform—from any party not explicitly authorized to communicate with them (see Privacy via Access Control below). This includes protection from other users, auditors, and external third parties. Such a notion of privacy is emphasized repeatedly in the literature, including in OPTIKS (Quote 23, Quote 25) and ELEKTRA (Quote 29).

In contrast to consistency guarantees—where the server is modeled as fully malicious—privacy definitions typically assume that the server is honest-but-curious. That is, while the server has full visibility into the log and could, in principle, leak this information, it is explicitly assumed not to do so. Instead, the server is expected to follow the protocol and attempt to minimize privacy leakage. OPTIKS makes this point explicitly (Quote 26).

CONIKS [MBB+15] claims to achieve an informal notion of consistency without requiring key bindings to be made public (Quote 5). However, SEEMless [CDG+19] later presented an attack that revealed CONIKS' actual leakage was significantly greater than originally claimed. This discrepancy, arising from the lack of formal security definitions in CONIKS, highlights the critical importance of adopting the provable security paradigm when analyzing protocols designed for adversarial settings. Formal definitions are essential for enabling precise and verifiable security claims.

SEEMless provides a formal leakage definition: auditors learn only the number of key updates per epoch, and key queries reveal only the latest value and the corresponding epoch of addition (Quote 6).[6] That is for key queries, a query for user U would solely reveal the public key associated with U (in addition to some metadata), and no information about other users. Parakeet adopts the same leakage profile as SEEMless (Quote 18). OPTIKS leaks similar information but also discloses the values and epochs of all past keys for a user when their most recent key is queried (Quote 24). A comparable privacy goal is stated in the IETF KEYTRANS architecture draft (Quote 36).

4 Modeling Key Transparency as Multi-Party Computation

Now that we have surveyed the existing literature on KT systems and established the need for a holistic and formal security model on the protocol level, we are ready to present our formal framework for KT systems.

Secure multi-party computation (MPC) enables several mutually distrusting parties to jointly compute a common function on their inputs without unnecessarily compromising the privacy of their inputs. Yao [Yao82] introduced the formal concept of MPC as a solution to the so-called "Millionaires' Problem" where two millionaires want to determine who is richer without revealing their actual wealth. General MPC is not restricted to secure function evaluation but it actually allows parties to perform any arbitrary *interactive*[7] computation securely, e.g., exchanging their public keys. Thus, the objective of a KT system is (quite self-evidently) an interactive multi-party computation; and—in our view—should be formally modeled as such. Before we can present our formal model of KT systems, we introduce some MPC basics and provide some intuition for how security is defined for MPC protocols.

[6] This is the same leakage as in our functionality.

[7] Sometimes also called "reactive".

4.1 Secure Multi-Party Computation

To reiterate, secure MPC [Yao82; Yao86] enables a group of mutually distrusting parties to jointly perform a (interactive) computation as if the computation was performed by a trusted third party, i.e., in the intended way. To ensure this guarantee, the security of an MPC protocol is not evaluated using game-based notions—i.e., by testing whether an adversary can win a specific predefined game. This is because it has shown itself difficult to anticipate and enumerate all the ways in which an adversary might misbehave in order to subvert the intended functionality [Can01; Lin17]. To address this, MPC security is established by showing that every real execution of the protocol, even in the presence of malicious parties, closely approximates an ideal execution where a trusted third party performs the computation.[8] This approach is known as the "real-ideal paradigm" or simulation-based security [GM84].

The Real-Ideal Paradigm The objective of a given MPC task is specified by a so-called *ideal functionality* (essentially corresponding to the code of the trusted third party) that the protocol tries to emulate. We say a protocol π securely realizes an ideal functionality $\mathcal{F}$ if the real execution of the protocol π is indistinguishable from the ideal execution of $\mathcal{F}$. Naturally, in the real execution the honest parties execute the protocol's code whereas the adversary controls the corrupted parties behavior. In contrast, in the ideal execution the (idealized) adversary, also called *simulator*, only has limited access to the ideal functionality $\mathcal{F}$ (as per its definition). Honest parties are simply dummy parties that forward all their inputs to the "ideal" functionality. Finally, given the transcript of either execution, a distinguisher must not be able to tell which execution it is.

The rational behind this paradigm is that whatever havoc the real adversary can cause in the real execution of the protocol, it cannot be too far from whatever the idealized adversary (simulator) can cause in the ideal execution, thanks to the indistinguishability requirement. However, because the simulator is allowed to behave arbitrarily, the indistinguishability requirement ensures that any action by the real adversary is indistinguishable from something explicitly allowed in the ideal world. For a more thorough introduction to MPC we refer the interested reader to the excellent monograph of Evans, Kolesnikov, and Rosulek [EKR18].

The UC Framework It is often the case that the security of a practical protocol is proven under the assumption—either implicit or explicit—that it runs as a single instance in isolation. As a result, the proven guarantees may break when the protocol is executed concurrently—either with itself or alongside other protocols—as is typically the case in real-world deployments.[9] This limitation can be sided by proving security within a composability framework, such as the Universal Composability (UC) framework [Can00; Can01], which ensures that security properties are preserved even under arbitrary composition. In other words, when designing protocol for functionalities that are themselves used in larger (a priori unknown) contexts, the above simulation-based security (or some other game-based security), might not suffice.

The UC framework has been used to analyze key exchange and secure channels [CK02], virtual smart cards [ABM16], RFID security [DKL$^+$10], and the Bitcoin blockchain protocol [BMT$^+$24], to name a few. In this work, we also conduct our analysis within the UC framework.

While the UC framework provides much stronger security guarantees than the standalone (simulation-based) model, it comes with some inherent restrictions. Unfortunately, it is known to be impossible to achieve universal composability without some kind of setup assumption[CF01; CKL06]. Therefore, in our proofs, we assume that protocol parties can agree on a common reference string (CRS), which is

[8] It is generally the case that a protocol is composed of smaller building blocks, whose individual game-based security notions are used to argue the indistinguishability of the two executions.

[9] A well-known example is that many zero-knowledge protocols fail to remain secure under parallel repetition.

a standard and widely accepted compromise in UC-based proofs. Moreover, the KT literature typically assumes a CRS in their proofs,[10] despite their security guarantees not being composable. We do not rely on random oracles, and our protocols are proven UC-secure under standard cryptographic assumptions.

4.2 Abstraction Layers of KT Systems

While KT was first introduced in [MBB+15], we accredit the first step towards a proper formalization of KT to Chase, Deshpande, Ghosh, and Malvai [CDG+19] through their notion of a verifiable key directory (VKD). Though, while formally defining several algorithms, their definition lacks the previously described distinction between the objective of the KT system and the protocol that realizes said objective. We argue that in order to properly formalize KT, we distinguish between three formal concepts:

- A KT *functionality* specifies the objective of a KT system (e.g. supplying users with previously registered keys). Formally, any ideal MPC functionality is defined by the code of a (virtual) trusted third party with whom the parties interact.
- A KT *protocol* realizes (in the real-ideal paradigm) the KT functionality, i.e., its execution should be indistinguishable from the execution of the ideal functionality. Formally, a protocol consists of a set of parties and defines the code of those (honest) parties that is executed during the run of the protocol.
- A KT *scheme* defines a set of algorithms (analogous to a PKE scheme). The purpose of the scheme is to simplify the description and security proof of the protocol in which it is used.

Remark 1. *A conceptually important distinction between a protocol and a scheme is that within the scope of a scheme there are no parties. For example, consider the notion of a digital signature scheme* $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vfy})$*. The scheme's signing algorithm* Sign *does not need the explicit or implicit notion of a signer* party*; it simply describes know the signing process works. In contrast, in a protocol a given party may want to sign a particular message and hence invoke the signing algorithm at a specific point in the protocol.*

In Appendix D we discuss to which degree previous literature (specifically SEEMless [CDG+19]) captures or fails to capture these levels of abstraction, and how it affects the security guarantees made by these works.

Why Worry About Layers of Abstraction? We argue that establishing a mathematically rigorous model of KT systems is a necessary requirement for a robust standardization (the need of which is reflected in the standardization efforts of the IETF [McM25; ML24]). Importantly, the soundness of the formalization (including the distinction between functionality, protocol and scheme) is not a mere academic exercise, but it in fact has real world consequences. As evidence of real-world implications of the currently insufficient state of KT formalization we observe a fundamental flaw in the security argument of Whatsapp's KT system [LL24b]. While Whatsapp's KT system is based on the construction and security analysis of [CDG+19] the security proof of the construction in [CDG+19] requires (as a building block) a *simulatable* verifiable random function (sVRF) [CL07]. However, WhatsApp's VRF implementation [LL24a] *cannot* be simulatable. We attribute this mismatch between the deployed VRF and one that would work in the security proof of [CDG+19] directly to the aforementioned lack of rigor in the existing KT literature.

4.3 Ideal KT functionality

First, we give a definition of an utopian KT functionality in Figure 1 that captures the features and security that we intuitively expect from a KT system. This functionality allows two high-level procedures:

[10] For example, per their use of simulatable verifiable random functions in their construction.

Functionality $\mathcal{F}_{\mathsf{idealKTS}}$

$\mathcal{F}_{\mathsf{idealKTS}}$ proceeds as follows, running with security parameter λ, n users $\mathcal{U} = \{\mathsf{id}_1, ..., \mathsf{id}_n\}$, server SP, and adversary $\mathcal{S}$. Messages not covered here are ignored. Initially, set the database $\mathsf{D}[\mathsf{id}] := []$ for each $\mathsf{id} \in [n]$.

- **RegisterKey**: When receiving a key k from user U_{id}, store $\mathsf{D}[\mathsf{id}][|\mathsf{D}[\mathsf{id}]| + 1] := \mathsf{k}$.
- **QueryKey**: When receiving query $(\textbf{QueryKey}, \mathsf{id}')$ from user U_{id}, retrieve $\mathsf{k} := \mathsf{D}[\mathsf{id}', |\mathsf{D}[\mathsf{id}']|]$. Send $(\textbf{QueryKeyResponse}, \mathsf{id}', \mathsf{k})$ to U_{id}.

Fig. 1: An overly optimistic functionality $\mathcal{F}_{\mathsf{idealKTS}}$ for a KT system that cannot be realized in practice.

1. Each user may register a new key at any point in time.
2. Each user U_{id} may query any user $\mathsf{U}_{\mathsf{id}'}$'s most recent key.

Unsurprisingly, we cannot realize $\mathcal{F}_{\mathsf{idealKTS}}$ if we assume a realistic communication model. Suppose that users only communicate directly with the server in an authenticated manner.[11] Now, suppose a server simply crashes. In this scenario no protocol can securely realize $\mathcal{F}_{\mathsf{idealKTS}}$. The simple reason is that no honest party can ever obtain any other (honest) party's registered key because all communication would be routed through the server. However, according to the functionality $\mathcal{F}_{\mathsf{idealKTS}}$, even in this scenario, any honest party querying another party's previously registered key should receive a correct response (from the assumed trusted third party that executed $\mathcal{F}_{\mathsf{idealKT}}$ in the ideal world).

The takeaway from this toy example is that we have to weaken the ideal KT functionality according to restrictions dictated by our model of the real world. However, in addition to a server crashing, there are a number of other real-world attacks to consider to actually obtain a sane KT functionality. Again, from here on out we assume that users only have authenticated channels to the server but not amongst each other.

1. Impersonation attack: If we accept that users lose their device, i.e., they lose their entire (secret) state, then we need to assume that the server is honest in order to make meaningful security guarantees. The reason is that if the user has no secret state, then that user has no authenticated channel to any other user, i.e., that user cannot communicate directly with other users. However, since the user has no state (and thus no shared secret with another user), a malicious server can impersonate the affected user simply by simulating the affected user.
2. Omission attack: If a user registers a new key with the server but that key is only sent to the server (i.e., the new key is not correlated with the view of any other party), then there is no guarantee that a (stop-fault) server will actually include the newly registered key in the next epoch update.

In addressing these attacks, Chase, Deshpande, Ghosh, and Malvai [CDG+19] define a "soundness" security property for VKD underlying their KT system. When satisfied, it guarantees that if a user has successfully verified their own key value in a given epoch, then all other users obtain the same key within the epoch. However, it is important to note that the property definition makes no statement about what happens if the check fails. Therefore, this VKD level property does not imply any security for its corresponding KT system if the server is malicious.

Take, for example, a malicious server that injects their own key for a particular user. While the system in this setting is clearly insecure according to common sense, the soundness notion is still satisfied. The authors of [CDG+19] argue that such malicious server can be detected by users and that the users should then complain "out-of-band". However, exactly how this complaint is communicated to all system users, what it entails, and why it would be acceptable in practical applications remains unspecified.

Even more critically, there is no mention of what happens if users are allowed to be completely reset—such as by losing their device. Intuitively, a complete reset means that a user has lost the means

[11] Assuming a priori authenticated channels between users defeats the purpose of creating a KT system in the first place.

to prove their identity within the system. Note that in this setting the above attack implies that the server is able to completely impersonate the user that is being reset without the means for the user to raise a credible accusation *within* the model. Even more concerning, relying solely on the above soundness notion from [CDG+19] would allow not only a malicious server to impersonate any user, but also enable a malicious *user* to impersonate any other user. Consequently, in a system that allows users to lose their device, the server must be assumed at least semi-honest.

We want to stress that some non-cryptographic techniques are considered to mitigate this problem in practice. For example Linker and Basin [LB24] formalize the notion of social authentication and implement a protocol for it. To tackle the problem of users impersonating each others, [CDG+19] relies on application-level access control. The IETF working group on KT systems [McM25; ML24] adapts the same approach. Obviously, these approaches may provide valuable security mechanisms in practice. Since, we are interested in modeling KT systems in a cryptographically sound way, we consider these approaches orthogonal to KT systems and thus out-of-scope.

4.4 Realizable KT Functionality

The functionality in Figure 2 captures real-world limitations described above. It allows for three procedures: 1) users can register keys, 2) users can query keys, and 3) the server can increment the epoch. Whenever a user U_{id} queries for the key of another user $\mathsf{U}_{\mathsf{id}'}$ in epoch τ, the functionality responds with the most recent key that was added up to the previous epoch $\tau - 1$. The resaon why $\mathcal{F}_{\mathsf{KT}}$ does not respond with the key stored in the *current* epoch is that the user $\mathsf{U}_{\mathsf{id}'}$ could still overwrite their key in the current epoch τ. In other words, the keys that users register in any given epoch only become persistent (or committed to) once the epoch has passed.
Another noteworthy aspect of our functionality is that if the server is corrupted, then the simulator (the adversary) must allow query responses before they are give to querying users, and the simulator gets to overwrite the list of updated keys (but importantly it does not get to specify the entire database). This behavior reflects the fault-stop attack discussed earlier. It also reflects the impersonation attack by allowing a malicious server to impersonate a user by updating their key via L_{SP} in **Update**.[12] Impersonation attacks are inherent to KT systems due to the need to support "default" users—those unable to permanently store high-entropy secrets—so they can recover from complete state loss (e.g., losing all devices) and reclaim their identity within the service [CDG+19; MBB+15].

In sum, our KT functionality in Figure 2 captures all issues described above, and we are able to eventually realize it with our protocol in Section 5.

The limitation enabling impersonation attacks is not unique to the domain of KT; in the web PKI ecosystem, certificate authorities (CAs) could issue certificates that bind a domain to an incorrect public key—effectively facilitating impersonation attacks [Lau14; LLK13]. As a result, the security guarantee offered by systems with this limitation shifts from prevention to detection: the impersonated entity (e.g., a service user) must be able to detect the attack (e.g., by querying their own public key). To this extent, the web PKI ecosystem introduced Certificate Transparency (CT), a system for publicly logging all issued certificates. KT adopt a similar strategy by (privately) logging all service public keys that receiving users may accept as valid.

This is reflected in our KT functionality through the enforcement of a persistent database state for each epoch τ, denoted by D^{τ}. D^{τ} is defined exactly once—immediately before the epoch begins, during the transition from epoch τ to $\tau + 1$. As a result, any query issued during an epoch is answered consistently, i.e., based on D^{τ} for the query input. Therefore, if a user queries for their own public key, they observe the same value that all other users would —whether it is correct or has been maliciously modified—thus providing means to detect impersonation attacks. As a remark, if detecting impersonation attacks with

[12] The server can also maliciously omit or overwrite key registrations via L_{SP}.

some delay of t is acceptable, our functionality can be readily extended to allow querying keys from previous epochs, i.e., in respect to $\mathsf{D}^{\tau'}, \tau' < \tau$,[13] thereby enabling retrospective detection of such attacks. For clarity of exposition, we omit this modification.

The append-only log maintained by a KT system must balance two competing goals: preserving user privacy while enabling transparency. In particular, a KT system should prevent *anyone* from learning which users or public keys are present in the system solely by inspecting the log. However, some degree of privacy leakage is necessary to enable efficient constructions while still ensuring the detectability of misbehavior. Consequently, the KT log should aim to reveal no more than the number of key registrations [CDG+19; LCG+23b]. This is reflected in our functionality by allowing the adversary (via the simulator) to learn the number of honest key registrations during each **Update**.

4.5 Our KT functionality and Consistency Notions in Section 3

Our KT functionality implies the notion of weak consistency as specified in Section 3—i.e., different users observe the same key bindings when relying on the same transparency log state (epoch commitment). Indeed, we observe by inspection that within each epoch (the same transparency log state), any two users receiving a query answer from the functionality, receive the same public key for a given identity, thereby ensuring weak consistency.

However, our functionality (unsurprisingly) does not imply strong consistency—i.e., ensuring that a user's query return contains the latest authentic key registered by the queried identity holder.

Nevertheless, such a alternative notion of strong consistency is achievable under certain system-level assumptions. Namely, either strong consistency holds or key owners are able to detect maliciously injected or omitted keys. For instance, we can modify our protocol such that users are forced to query for their current epoch key in every epoch and raise an alarm on inconsistent key responses. Then this protocol[14] fulfills the aforementioned notion of alternative strong consistency. That said, we stress the importance of not over-relying on specific system-level assumptions such as the uptime of users, in particular in the context of end-to-end communication.

5 Key Transparency Protocol

To realize our KT functionality $\mathcal{F}_{\mathsf{KT}}$ we propose a KT protocol that is inspired by the (implicit and incomplete) protocol descriptions of [CDG+19; LCG+23b]. For modularity our KT protocol uses our KT scheme (defined in Appendix B) as a building block—so that the concrete instantiation (Appendix C.1) of the KT scheme can be changed with ease (e.g. from a group-based to a post-quantum secure scheme). Moreover, the use of our KT scheme significantly simplifies the description and the UC-proof of our KT protocol.

To closely reflect real-world deployment scenarios and applications, we make the following assumptions (standard in the KT literature)

- Each user has a unique (permanent) identifier id (e.g., a username such as a phone number or an email address, or UUID).
- An asymmetric public-key infrastructure (PKI) exists. More concretely, the server holds two key pairs: $(\mathsf{pk}_{\Sigma}, \mathsf{sk}_{\Sigma})$ for a signature scheme, and $(\mathsf{pk}_{\mathsf{PKE}}, \mathsf{sk}_{\mathsf{PKE}})$ for public-key encryption scheme.
- Each user has an authenticated channel with the server. This would typically be realized by a key-exchange via two-factor authentication such as SMS or an authenticator app.

[13] Note that our extended functionality guarantees that any query to $\mathsf{D}^{\tau'}$, regardless of when it is made after or during epoch τ', would return a consistent result.

[14] Under assumption that users are maximally honest-but-curious and never suppress alarms.

Functionality $\mathcal{F}_{\mathsf{KT}}$

$\mathcal{F}_{\mathsf{KT}}$ proceeds as follows, running with security parameter λ, n users $\mathcal{U} = \{\mathsf{U}_1, ..., \mathsf{U}_n\}$ a server SP, and a simulator $\mathcal{S}$ that is to emulate the world to an adversary $\mathcal{A}$ that corrupts $\mathcal{C} \subseteq \mathcal{U} \cup \{\mathsf{SP}\}$. Messages not covered are ignored. Initially, set the epoch counter $\tau := 0$. For each user $\mathsf{id} \in [n]$ set the initial database as $\mathsf{D}^{-1}[\mathsf{id}] := (\mathsf{k} = \bot, \mathsf{v} = 0)$ (key–version pairs). For each epoch $\tau' \geq -1$ initialize an empty update list $\mathsf{L}_{\tau'}[\mathsf{id}'] := []$ (containing keys only).

- **RegisterKey**:
  1. Receive $(\textbf{RegisterKey}, \mathsf{id}, \mathsf{k} \neq \bot)$ from user U_{id}.
  2. Store $\mathsf{L}_\tau[\mathsf{id}] := \mathsf{k}$ (overwriting).
  3. If the server **is** corrupted, i.e., $\mathsf{SP} \in \mathcal{C}$, then
     (a) send $(\textbf{RegisterKey}, \mathsf{id}, \mathsf{k})$ to $\mathcal{S}$.
- **QueryKey**:
  1. Receive query $(\textbf{QueryKey}, \mathsf{id}')$ from user U_{id}.
  2. Retrieve the key $(\mathsf{k}, \mathsf{v}) := \mathsf{D}^{\tau-1}[\mathsf{id}']$,
  3. If the server **is not** corrupted, i.e., $\mathsf{SP} \notin \mathcal{C}$, then
     (a) send $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k})$ to U_{id}.
  4. If the server **is** corrupted, i.e., $\mathsf{SP} \in \mathcal{C}$, then
     (a) send $(\textbf{QueryKey}, \mathsf{id}, \mathsf{id}')$ to $\mathcal{S}$,
     (b) (possibly) receive $(\textbf{AllowQueryKey}, \tau, \mathsf{id}, \mathsf{id}')$ from $\mathcal{S}$.
        i. Send $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k})$ to user U_{id}.
- **Update**:
  1. Receive **Update** from SP.
  2. Initialize the next epoch database as $\mathsf{D}^\tau := \mathsf{D}^{\tau-1}$.
  3. Update the database with the honest updates as follows: for each $\mathsf{id} \in [n]$ let $(\mathsf{k}_{\tau-1,\mathsf{id}}, \mathsf{v}_{\tau-1,\mathsf{id}}) := \mathsf{D}^{\tau-1}[\mathsf{id}]$ be the key–version pair of user U_{id} in epoch $\tau - 1$, let $\mathsf{k}_{\tau,\mathsf{id}} := \mathsf{L}_\tau[\mathsf{id}]$ be the key that user U_{id} registered in epoch τ, set the new database entry $\mathsf{D}^\tau[\mathsf{id}] := (\mathsf{k}_{\tau,\mathsf{id}}, \mathsf{v}_{\tau,\mathsf{id}})$ with incremented version $\mathsf{v}_{\tau,\mathsf{id}} := \mathsf{v}_{\tau-1,\mathsf{id}} + 1$.
  4. Let $\ell := |\{\, \mathsf{id} \mid \mathsf{L}_\tau[\mathsf{id}] \neq \bot \,\}|$ be the number of honest users that updated their key in epoch τ. Send $(\textbf{Update}, \ell)$ to $\mathcal{S}$.
  5. If the server **is** corrupted, i.e., $\mathsf{SP} \in \mathcal{C}$, then
     (a) send **Update** to $\mathcal{S}$,
     (b) receive a list of id–key pairs as $(\textbf{Update}, \mathsf{L}_{\mathsf{SP}})$ from $\mathcal{S}$,
     (c) tamper with the next epoch database as follows: for each $\mathsf{id} \in [n]$ let $\widetilde{\mathsf{k}}_{\tau,\mathsf{id}} := \mathsf{L}_{\mathsf{SP}}[\mathsf{id}]$ be the adversarial key for user U_{id} in epoch τ,
        - if $\widetilde{\mathsf{k}}_{\tau,\mathsf{id}} = \bot$, reset the key that was registered by user U_{id} by setting $\mathsf{D}^\tau[\mathsf{id}] := \mathsf{D}^{\tau-1}[\mathsf{id}]$ (i.e., the user U_{id}'s key update is ignored, the version is not incremented),
        - if $\widetilde{\mathsf{k}}_{\tau,\mathsf{id}} \neq \bot$, set the adversarially chosen key as $\mathsf{D}^\tau[\mathsf{id}] := (\widetilde{\mathsf{k}}_{\tau,\mathsf{id}}, \mathsf{v}_{\tau,\mathsf{id}})$ (i.e., the user U_{id}'s key update is replaced by the adversarially chosen key, the version is incremented).
  6. Increment the epoch number $\tau := \tau + 1$.
  7. Send $(\textbf{Update}, \tau)$ to each user U_{id} for $\mathsf{id} \in [n]$.
  8. Ignore all further inputs $(\textbf{AllowQueryKey}, \tau', \cdot)$ for $\tau' \lneq \tau$.

Fig. 2: Our KT functionality: The database state in epoch τ denoted by the variable D^τ is *persistent*, i.e., it is defined only once during the update procedure from epoch τ to $\tau + 1$. In contrast, the update list L_τ is *volatile*, i.e., its entries can be overwritten by users and finally by the adversary during the same epoch τ. For the ease of exposition, we have concentrated on the case where the number of users $n \in \mathbb{N}$ is fixed and known to all parties.

- The server can broadcast messages (e.g., epoch commitments) to all users. The mechanism for this would typically entail the server signing the epoch commitment and then publishing it to a public append-only database that is accessible to all users. Proposed real-world mechanisms for this include blockchains [Bon16; TD17], gossip protocols [MKL+20], and a novel light-weight "consensusless" consistency protocol described in [MKS+23].

Our KT scheme. Our KT scheme is inspired by the VKD primitive of [CDG+19] but neither its construction nor its security proofs follow directly from [CDG+19]. We present the formal definition in Appendix B and the concrete instantiation in detail in Appendix C.1. In Appendix A we provide preliminaries that our scheme and the security proof of our protocol rely on. Instead of presenting a formal definition of our KT scheme at this point, we give a high-level overview of its construction and functionality to facilitate the understanding of how it is used in our KT protocol. Generally, our KT scheme relies on an sVRF, a collision-resistant hash function, and a non-interactive zero-knowledge proof (NIZK) system.
On a high level, our concrete KT scheme construction follows the structural approach of the VKD scheme in [CDG+19]. In each epoch, we store database elements using a Patricia trie (via $\mathsf{KTS.Commit}$), using a simulatable verifiable random function (sVRF) to map elements to the positions in the database. The root of the Patricia trie is the *epoch commitment* and commits to the entire database for that epoch. Using the database of the previous epoch and the list of updated keys, we can generate the next epoch commitment and prove (via $\mathsf{KTS.UpdateEpoch}$) that the database of the next epoch is consistent with the previous one. Then, following the approach in [LCG+23b], we respond to queries (via $\mathsf{KTS.QryKey}$) with an inclusion proof for the latest version of the queried user's key, along with non-inclusion proofs for the subsequent version. The querying user can then verify the query response via $\mathsf{KTS.VfyQry}$ using the epoch commitment and the query response. Finally, to achieve consistency *between* epochs, we attach a non-interactive zero-knowledge proof (NIZK) to the epoch update proofs that ensures that the databases contained in the epoch commitment are "well-formed", e.g. they don't miss any intermediate version of keys, and for each identity at most one new key was added relative to the database of the previous database. This approach differs substantially from [CDG+19; LCG+23b], which do not employ NIZKs and thus do not achieve what we call "inter-epoch consistency" in Appendix B. Naturally, our KT scheme also achieve "intra-epoch consistency" (essentially weak consistency in Section 3) which is inherited from the technique of [CDG+19]. Moreover, our KT scheme is private in a sense that an adversary cannot learn any information about the keys of the users that it did not explicitly query for. Lastly, a major difference to the VKD scheme of [CDG+19] and technical hurdle is the fact that our KT scheme is extractible, meaning that with a trapdoor associated with the CRS one can extract (via $\mathsf{KTS.ExtKeys}$) the database that is committed to in the epoch commitment. This is crucial for our UC-proof to work but extraction is only used in the UC-proof—not in the construction itself.[15] While extractability at first seems at odds with the succinctness of the epoch commitment (as a Patricia trie root is highly compressing), we actually extract the database not only from the epoch commitment but also from the epoch update proofs that the server publishes with each epoch commitment.

Our KT protocol. Our KT protocol is presented in Figure 3. Thanks to the usage of our KT scheme the protocol itself (although formally well-defined) is relatively simple for a task as complex as KT. Note that we are in the $(\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{crs}})$-hybrid model, but that only the server needs to (upon epoch updates) utilize the broadcast functionality $\mathcal{F}_{\mathsf{BC}}$ to publish the epoch commitment and the update proofs. Initially, each party (users and server) obtain the CRS from the $\mathcal{F}_{\mathsf{crs}}$ functionality. A user that wants to register a key in epoch τ simply sends that request to the server who stores the key in the list of current updates L_τ.

[15] The reader familiar with UC may recall the UC commitment problem that require a commitment scheme to be both simulatable and extractable.

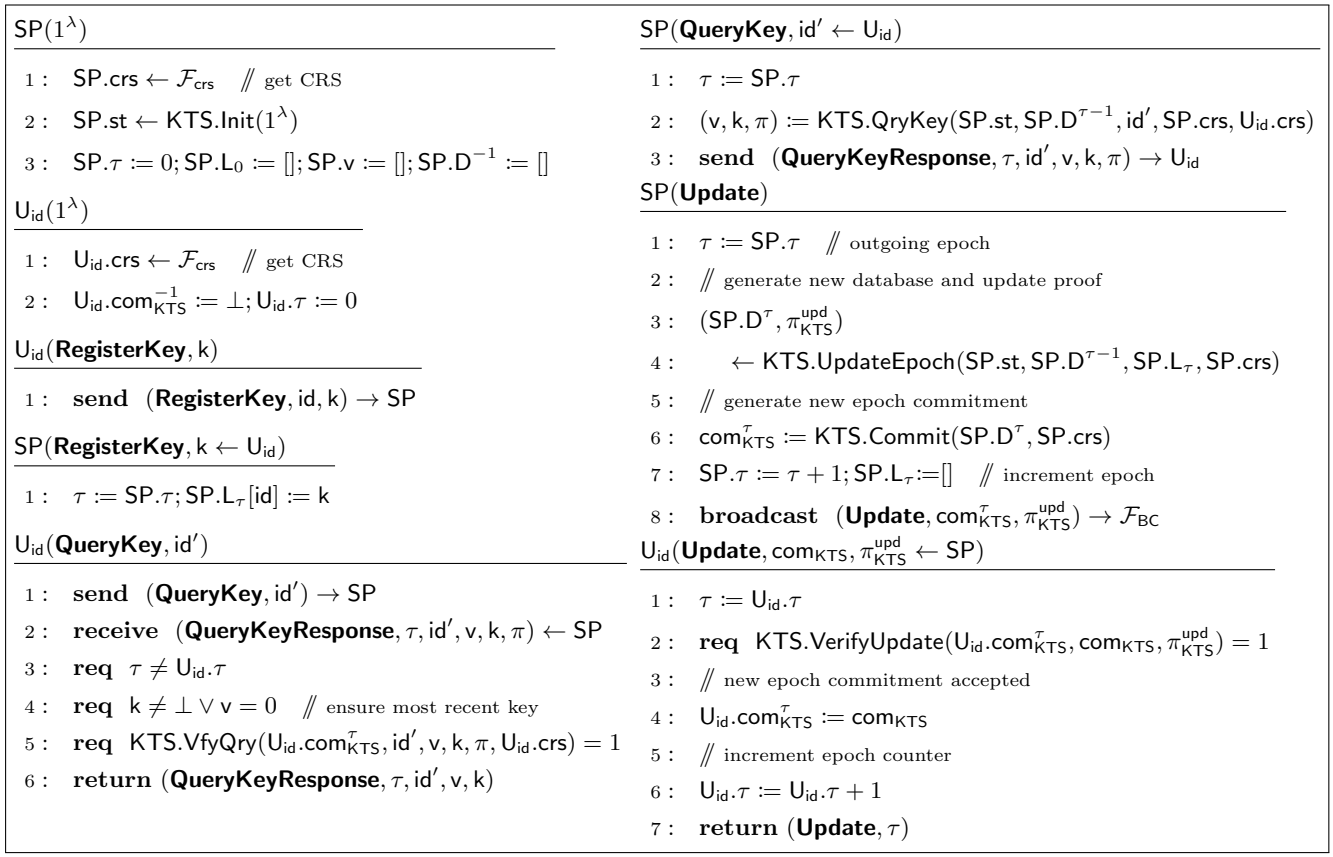


Fig. 3: Our KT protocol π_{KT} using a KT scheme KTS to realize the functionality $\mathcal{F}_{\mathsf{KT}}$ in the $\{\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{crs}}\}$-hybrid model. In the argument of the procedures we denote messages from other parties/functionality by $\leftarrow$; if $\leftarrow$ is not present, then it is input from the environment.

When the server increments the epoch it first computes the new database and update proof via $\mathsf{KTS.UpdateEpoch}$ and then commits to the new database via $\mathsf{KTS.Commit}$. Finally, the server broadcasts the new epoch commitment and the update proof to all users.
To query a user $\mathsf{U}_{\mathsf{id}'}$'s current key a user U_{id} simply sends that request to the server who queries the KTS (via $\mathsf{KTS.QryKey}$) to obtain the stored key and query response proof. Then the server simply send the key and the the proof back to the user U_{id}. The user U_{id} now verifies the query response and performs some sanity-checks to handle edge cases.

Remark 2. *Our protocol requires each user to independently verify epoch updates. However, it could be adapted so that a single honest (potentially more powerful) user performs the verification for each epoch and broadcasts the results to others. Such a variant would be secure under the assumption that the verifying entity is at most honest-but-curious (as assumed in [LCG+23b; CDG+19]). In contrast, our approach avoids reliance on trust between users, enabling security guarantees in the presence of fully malicious users.*

Our main formal result is the following:

Theorem 1. *Let* KTS *be a KT scheme (as defined in Definition 5). Then the protocol* π_{KT} *securely UC-realizes the functionality* $\mathcal{F}_{\mathsf{KT}}$ *in the* $\{\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{crs}}\}$*-hybrid model.*

We prove that our KT Protocol (Figure 3) securely UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{KT}}$ (Figure 2) in the UC framework. For space reasons, we defer the full proof to Appendix C, but provide a high-level and intuitive overview here.
Our proof—like most UC proofs—shows that the real and the ideal execution are indistinguishable to an efficient enviroment via sequence of intermediate hybrids games. For each two consecutive hybrid games we argue that the environments outputs distribution only changes negligibly. This is done via a reduction from some property of the KT scheme to the two hybrid games; if there output of the enviroment changes, then that environment can be used to break the property of the KT scheme.

Proof Sketch. We start in the real protocol execution where protocol parties communicate with the server. As a first modification, if the server is honest, we switch the CRS to simulation mode (by mode indistinguishability of KTS) which means that we can simulate arbitrary proofs and are no longer bound by any soundness requirements. Next, the (uncorrupted) server simulates the proofs for epoch updates and key queries (by simulation indistinguishability of KTS). This is possible exactly because we switched to simulation mode before. Next, the (uncorrupted) server does not commit to the actual database in the epoch commitment but instead to a dummy database of the same size. This modification is indistinguishable to the environment by the privacy of KTS; recall the the proofs are now simulated, so the committed database can be arbitrary. Next, we introduce the ideal $\mathcal{F}_{\mathsf{KT}}$ functionality and dummy parties for each honest party. Moreover, we fork all inputs from the environment to an honest protocol party to the resp. dummy party with forwards it to $\mathcal{F}_{\mathsf{KT}}$. However, all outputs from $\mathcal{F}_{\mathsf{KT}}$ to dummy parties are (as of yet) dropped. Thus, this change does not affect the games output distribution at all. Now, we abort the game when the outputs of the honest protocol parties don't exaclty match the dummy parties' dropped outputs. We argue that this abort only happens with negligible probability. An environment that could induce an abort would (by reduction) break the inter-epoch or intra-epoch consistency of the KT scheme. This step crucially relies on the extractability of KTS. Namely, if the corrupted server broadcasts an epoch commitment the simulator needs to extract the database from the epoch commitment and the update proof to feed into $\mathcal{F}_{\mathsf{KT}}$.[16] Finally, we switch to the actual ideal exection with only dummy parties but without protocol parties. Recall that—conditioned on no abort—the outputs of the honest protocol parties are exactly the same as the outputs of the dummy parties.

[16] This mechanism is analogous to the extractability of UC commitments in [CF01].

6 Conclusion

Despite their growing importance in securing end-to-end encrypted communication, the protocol-level understanding of KT systems remains surprisingly underdeveloped. In this work, we took a step back to ask a basic but overlooked question: what *exactly* are these systems trying to achieve?

We surveyed the core KT literature and found a landscape rich with clever constructions but lacking in clarity. Protocol goals are inconsistently stated, system assumptions are often implicit, and guarantees are difficult (if not impossible) to verify across contexts. Our work systematically distills these protocol-level properties, classifies their assumptions, and pinpoints contradictions and gaps. We the presented the first formalization of KT as an ideal functionality, capturing the security guarantees and system-level assumptions suggested (but rarely made explicit) by the literature.

Our formalization provides a unified framework for reasoning about KT security and operational guarantees. We show that several idealized goals—particularly strong consistency—cannot be achieved in practice without unrealistic assumptions. Instead, we offer a realizable functionality that models KT systems more accurately, accounting for real-world threats such as key substitution, omission attacks, and delayed detection. We then construct and prove the security of a concrete KT protocol in the UC framework, providing the first composable security guarantees for a KT system.

Our work is not a critique of any individual system, nor an academic exercise in formalism, but rather a call for greater clarity, precision, and rigor in KT protocol design. We hope our formal framework provides a useful foundation for the ongoing standardization efforts by IETF and inspires future work to build KT systems that are not only practical, but also provably secure.

References

[ABM16] M. Abdalla, F. Benhamouda, and P. MacKenzie. Virtual smart cards: how to sign with a password and a server. In *Public-Key Cryptography – PKC 2016*, pages 353–381. Springer International Publishing, 2016.

[App23] Apple Security Engineering and Architecture (SEAR). Imessage contact key verification. https : //security.apple.com/blog/imessage − contact − key − verification, 2023. Accessed: 2023-04-01.

[BBC+22] J. Blum, S. Booth, B. Chen, O. Gal, M. Krohn, J. Len, K. Lyons, A. Marcedone, M. Maxim, M. E. Mou, et al. Zoom cryptography whitepaper, 2022.

[BMT+24] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: a composable treatment. *Journal of Cryptology*, 37, 2024.

[Bon16] J. Bonneau. Ethiks: using ethereum to audit a coniks key transparency log. In *International Conference on Financial Cryptography and Data Security*, pages 95–105. Springer, 2016.

[Can00] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000.

[Can01] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CDG+19] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai. Seemless: secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1639–1656, 2019.

[CDG+22] B. Chen, Y. Dodis, E. Ghosh, E. Goldin, B. Kesavan, A. Marcedone, and M. E. Mou. Rotatable zero knowledge sets: post compromise secure auditable dictionaries with application to key transparency. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 547–580. Springer, 2022.

[CF01] R. Canetti and M. Fischlin. Universally composable commitments. In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Berlin, Heidelberg, August 2001.

[CK02] R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. In L. R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 337–351, Berlin, Heidelberg. Springer Berlin Heidelberg, 2002.

[CKL06] R. Canetti, E. Kushilevitz, and Y. Lindell. On the limitations of universally composable two-party computation without set-up assumptions. *Journal of Cryptology*, 19(2):135–167, 2006.

[CL07] M. Chase and A. Lysyanskaya. Simulatable VRFs with applications to multi-theorem NIZK. In A. Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 303–322. Springer, Berlin, Heidelberg, August 2007.

[CM16] M. Chase and S. Meiklejohn. Transparency overlays and applications. In *Proceedings of the 2016 acm sigsac conference on computer and communications security*, pages 168–179, 2016.

[DGH+16] B. Dowling, F. Günther, U. Herath, and D. Stebila. Secure logging schemes and certificate transparency. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II 21*, pages 140–158. Springer, 2016.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DKL+10] D. N. Duc, D. M. Konidala, H. Lee, and K. Kim. A survey on rfid security and provably secure grouping-proof protocols. *International Journal of Internet Technology and Secured Transactions*, 2(3–4):222–249, 2010.

[DP07] Y. Dodis and P. Puniya. Feistel networks made public, and applications. In M. Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 534–554. Springer, Berlin, Heidelberg, May 2007.

[EKR18] D. Evans, V. Kolesnikov, and M. Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[GH24] T. Göbel and D. Huigens. Proton key transparency whitepaper, 2024.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GMW91] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, July 1991.

[GRP+23] S. Goldberg, L. Reyzin, D. Papadopoulos, and J. Včelák. Verifiable Random Functions (VRFs). RFC 9381, August 2023.

[HB20] R. Hurst and G. Belvin. Google/keytransparency. https : //github.com/google/keytransparency/, 2020.

[HHK+21] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa. Merkle 2: a low-latency transparency log system. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 285–303. IEEE, 2021.

[HIL+99] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[Lau14] B. Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[LB24] F. Linker and D. A. Basin. SOAP: A social authentication protocol. In D. Balzarotti and W. Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[LCG+23a] J. Len, M. Chase, E. Ghosh, D. Jost, B. Kesavan, and A. Marcedone. Elektra: efficient lightweight multi-device key transparency. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2915–2929, 2023.

[LCG+23b] J. Len, M. Chase, E. Ghosh, K. Laine, and R. C. Moreno. Optiks: an optimized key transparency system. *Cryptology ePrint Archive*, 2023.

[Lew23] K. Lewi. Whatsapp key transparency. In *Proceedings of the 2023 USENIX Conference on Privacy Engineering Practice and Respect (PEPR '23)*, Santa Clara, CA, 2023.

[LGG+20] D. Leung, Y. Gilad, S. Gorbunov, L. Reyzin, and N. Zeldovich. Aardvark: a concurrent authenticated dictionary with short proofs. *IACR Cryptol. ePrint Arch.*, 2020:975, 2020.

[Lin17] Y. Lindell. *How to simulate it – a tutorial on the simulation proof technique*. In *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*. Y. Lindell, editor. Springer International Publishing, Cham, 2017, pages 277–346.

[LL24a] S. Lawlor and K. Lewi. Akd. https : //github.com/facebook/akd, version v0.11.0, 2024.

[LL24b] S. Lawlor and K. Lewi. Whatsapp key transparency, 2024. Real World Cryptography 2024.

[LLK13] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, June 2013.

[Mar23] A. Marcedone. Key transparency at keybase and zoom. Presentation at IETF 116 Meeting, March 2023. https : //datatracker.ietf.org/meeting/116/materials/slides − 116 − keytrans − keybase − and − zoom − 00.pdf.

[MBB+15] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. {Coniks}: bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[McM25] B. McMillion. Key Transparency Architecture. Internet-Draft draft-ietf-keytrans-architecture-03, Internet Engineering Task Force, February 2025. 23 pages. Work in Progress.

[Mer87] R. C. Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[MKL+20] S. Meiklejohn, P. Kalinnikov, C. S. Lin, M. Hutchinson, G. Belvin, M. Raykova, and A. Cutter. Think global, act local: gossip and client audits in verifiable data structures. *arXiv preprint arXiv:2011.04551*, 2020.

[MKS+23] H. Malvai, L. Kokoris-Kogias, A. Sonnino, E. Ghosh, E. Oztürk, K. Lewi, and S. Lawlor. Parakeet: practical key transparency for end-to-end encrypted messaging. *NDSS Symposium 2023*, 2023.

[ML24] B. McMillion and F. Linker. Key Transparency Protocol. Internet-Draft draft-ietf-keytrans-protocol-00, Internet Engineering Task Force, December 2024. 34 pages. Work in Progress.

[Sch22] J. Schwenk. *A short history of tls*. In *Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications*. Springer International Publishing, Cham, 2022, pages 243–265.

[TBP+19] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas. Transparency logs via append-only authenticated dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1299–1316, 2019.

[TD17] A. Tomescu and S. Devadas. Catena: efficient non-equivocation via bitcoin. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 393–409. IEEE, 2017.

[TFZ+22] N. Tyagi, B. Fisch, A. Zitek, J. Bonneau, and S. Tessaro. Versa: verifiable registries with efficient client audits from rsa authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2793–2807, 2022.

[TKP+21] I. Tzialla, A. Kothapalli, B. Parno, and S. Setty. Transparency dictionaries with succinct proofs of correct operation. *Cryptology ePrint Archive*, 2021.

[Yao82] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[Yao86] A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

A Preliminaries

A.1 Notation and Conventions

Given an integer $m \in \mathbb{Z}^+$, we write $[m]$ to mean the set $\{1, 2, ..., m\}$. We consider all logarithms to be in base 2. Within our pseudocode we use the notation $:=$ for deterministic assignment, and $\leftarrow$ for assignment according to a distribution or randomized algorithm. We index into arrays using $[\cdot]$ notation. For a k-dimensional array A, the entry at position $(i_1, i_2, \ldots, i_k)$ is denoted $A[i_1, i_2, \ldots, i_k]$. Similarly, if F is a function returning a k-dimensional array, we write $F(x)[i_1, i_2, \ldots, i_k]$ to access the corresponding element at those coordinates. For an array A we define the length of the array $|A| := \max\{\, i \mid A[i] \neq \perp \,\}$ as the maximal index i at which a (non-$\perp$) entry is stored.

For any randomized algorithm alg, we may denote the coins that alg can use as an extra argument $r \in \mathcal{R}$ where $\mathcal{R}$ is the set of possible coins, and write $\mathsf{output} =: \mathsf{alg}(\mathrm{input}_1, \mathrm{input}_2, ..., \mathrm{input}_l; r)$. We may

also suppress coins whenever it is notationally convenient to do so. If an algorithm is deterministic, we allow setting r to $\perp$. We remark that the output of a randomized algorithm can be seen as a random variable over the output space of the algorithm.

A.2 Verifiable Random Functions

A verifiable random function (VRF) is the public-key analog to the traditional pseudorandom function [GGM86]. Then evaluating the VRF using the secret key it is possible to generate an acompanying proof that the output was correctly computed. Using the public verification key, the preimage, a purposed image, and a proof anyone can verify that image indeed corresponds to the preimage. A *simulatable* VRF (sVRF) [CL07] also has public parameters (or a common reference string, or alternatively is defined relative to a (random) oracle). These parameters can be set up in two modes: a extraction mode and a simulation mode. In the extraction mode, for each verificaton key and each preimage there exists at most one image that verifies. In the simulation mode, given the trapdoor (or the ability to program the oracle), one can generate valid proofs for any preimage-image pair. Moreover, both modes are computationally indistinguishable. We mention in passing the folklore cosntruction of a simulatable VRF from a standard PRF (or OWF via [HIL+99; GGM86]) and a non-interactive zero-knowledge proof system.

We emphasize that—as Chase, Deshpande, Ghosh, and Malvai [CDG+19]—we require a *simulatable* VRF. This is crucial for the formal security proof of our KT protocol (as well as the protocol of [CDG+19]). Chase, Deshpande, Ghosh, and Malvai [CDG+19] omit the algorithms $\mathsf{Setup}, \mathsf{SimSetup}, \mathsf{SimQryKey}, \mathsf{SimUpdateDS}$ in the aZKS definition but consider them in a proof in the appendix of their work.

Definition 1 (Simulatable Verifiable Random Function). *A simulatable verifiable random function (sVRF) is a tuple of polynomial-time algorithms* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Vfy}, \mathsf{SimSetup}, \mathsf{SimEval})$ *where*

- $\mathsf{Setup}(1^\lambda)$ *on input security parameter* 1^λ*, outputs a CRS* crs*,*
- $\mathsf{Gen}(1^\lambda, \mathsf{crs})$ *on input security parameter* 1^λ *and CRS* crs*, outputs a verification key* vk *and a secret key* sk*,*
- $\mathsf{Eval}(\mathsf{sk}, x, \mathsf{crs})$ *on input secret key* sk*, preimage* x *and CRS* crs*, outputs an image* $y \in \{0,1\}^{\ell_y(\lambda)}$ *and a proof* π*,*
- $\mathsf{Vfy}(\mathsf{vk}, x, y, \pi, \mathsf{crs})$ *on input verification key* vk*, preimage* x*, image* y*, proof* π *and CRS* crs*, outputs* 1 *if the proof is valid and* 0 *otherwise,*
- $\mathsf{SimSetup}(1^\lambda)$ *on input security parameter* 1^λ*, outputs a simulated CRS* crs *and a trapdoor* td*,*
- $\mathsf{SimEval}(\mathsf{vk}, x, y, \mathsf{crs}, \mathsf{td})$ *on input verification key* vk*, preimage* x*, image* y*, CRS* crs*, and trapdoor* td*, outputs a proof* π*.*

Setup *and* Gen *are necessarily probabilistic,* Eval *and* Vfy *are deterministic.*

Correctness. *We say a VRF* VRF *is* α*-correct if for all* $\lambda \in \mathbb{N}$ *it holds that*

$$\Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}\big(^\lambda, \mathsf{crs}\big) \\ (y, \pi) \leftarrow \mathsf{Eval}(\mathsf{sk}, x) \end{array} : \mathsf{Vfy}_\lambda(\mathsf{vk}, x, y, \pi) = 1\right] \geq \alpha(\lambda)\ . \tag{1}$$

Unique provability. *We say a VRF* VRF *is* ν*-uniquely provable, iff for all* $\lambda \in \mathbb{N}$ *it holds that*

$$\Pr\left[\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) : \forall \mathsf{vk}, x, y_1, y_2, \pi_1, \pi_2 : \begin{array}{r} \mathsf{Vfy}(\mathsf{vk}, x, y_1, \pi_1) = 1 \\ \wedge\ \mathsf{Vfy}(\mathsf{vk}, x, y_2, \pi_2) = 1 \\ \Longrightarrow y_1 = y_2 \end{array}\right] \geq \nu(\lambda)\ . \tag{2}$$

We say a VRF VRF *is perfectly uniquely provable for* $\nu(\lambda) = 1$*.*

Pseudorandomness. *We say a VRF* VRF *is (adaptively) pseudorandom if for each non-uniform*

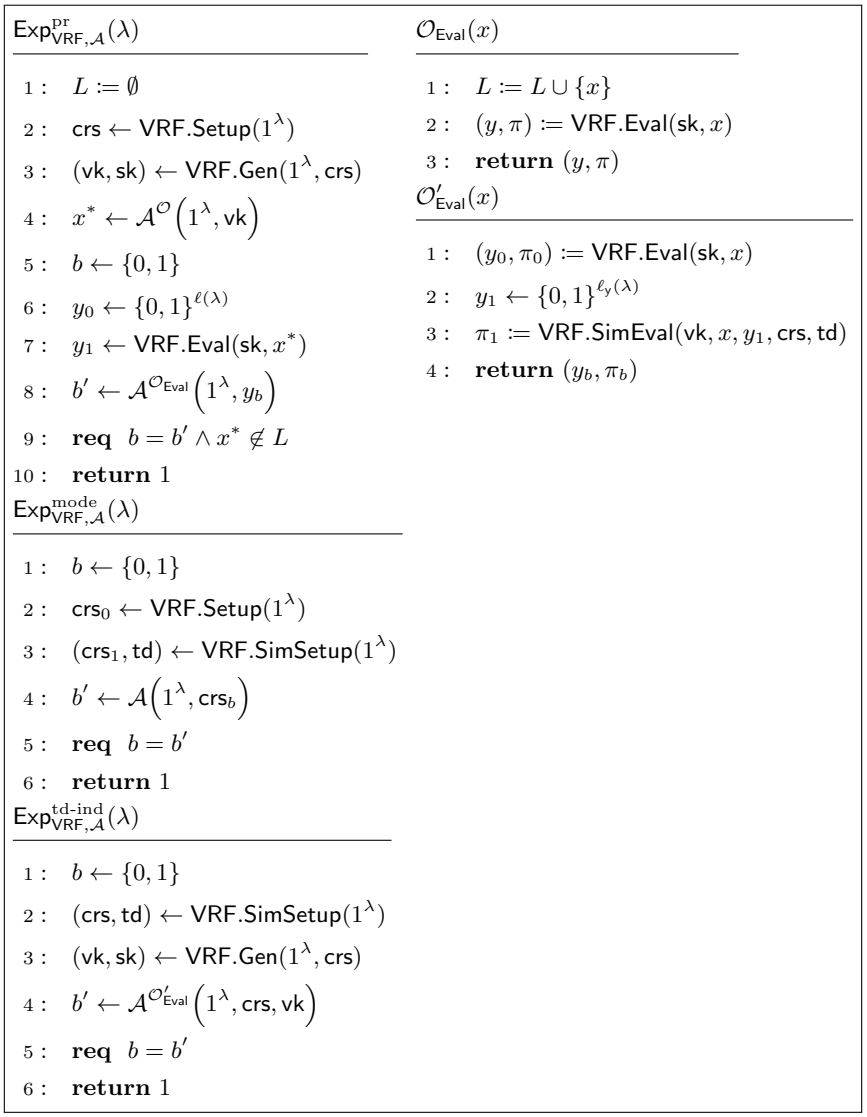


Fig. 4: $\mathsf{Exp}^{\mathrm{pr}}_{\mathsf{VRF},\mathcal{A}}$ denotes the adaptive pseudorandomness game, $\mathsf{Exp}^{\mathrm{mode}}_{\mathsf{VRF},\mathcal{A}}$ denotes the mode indistinguishability game, and $\mathsf{Exp}^{\text{td-ind}}_{\mathsf{VRF},\mathcal{A}}$ denotes the trapdoor indistinguishability game [CL07] for an sVRF VRF and (stateful) adversary $\mathcal{A}$. We assume w.l.o.g. that the adversary $\mathcal{A}$ never queries the same value twice.

polynomial-time-bounded (stateful) adversary $\mathcal{A}$ its advantage is bounded by

$$\left|\Pr\left[\mathsf{Exp}^{\mathrm{pr}}_{\mathsf{VRF},\mathcal{A}}(\lambda)=1\right]-1/2\right|\leq \mathrm{negl}(\lambda) \tag{3}$$

with the pseudorandomness game defined in Figure 4.

Mode indistinguishability. *We say a VRF* VRF *is CRS-indistinguishable if for all non-uniform polynomial-time-bounded adversaries $\mathcal{A}$ its advantage is bounded by*

$$\left|\Pr\left[\mathsf{Exp}^{\mathrm{mode}}_{\mathsf{VRF},\mathcal{A}}(\lambda)=1\right]-1/2\right|\leq \mathrm{negl}(\lambda) \tag{4}$$

with the mode indistinguishability game defined in Figure 4.

Invertability. *We say a VRF* VRF *is invertible if there exists a (deterministic) algorithm* Inv *such that for each preimage x it holds that*

$$\Pr\left[\begin{array}{c}\mathsf{crs}\leftarrow\mathsf{Setup}(1^\lambda)\\ (\mathsf{vk},\mathsf{sk})\leftarrow\mathsf{Gen}\big(^\lambda\big)\\ (y,\pi)\leftarrow\mathsf{Eval}(\mathsf{sk},x)\end{array}:\mathsf{Inv}(\mathsf{sk},y,\mathsf{crs})=x\right]=1\ . \tag{5}$$

Remark 3. *W.l.o.g. we assume that the VRF is invertable. Dodis and Puniya [DP07] show how to construct a verifiable random permutation (VRP) from any VRF. Because their VRP is based in a Feistel construction, the secret key can be used to invert the VRP. Although, they do not state their result formally for simulatable VRFs, their construction does work with simulatable VRFs.*
Invertability will be useful in the proof of extractability of our KT scheme in Appendix C.1 to extract the database from the labels of the Patricia trie in the epoch commitment updates.

A.3 Non-Interactive Zero-Knowledge Proofs

A non-interactive zero-knowledge proof (NIZK) system [GMR85; GMW91] is a cryptographic primitive that allows a prover to convince a verifier of the validity of a statement without revealing any additional information about the statement.

Definition 2 (Non-Interactive Zero-Knowledge Proof System). *A non-interactive zero-knowledge proof system (NIZK) for an* NP*-relation R is a tuple of polynomial-time algorithms* $\mathsf{NIZK}=(\mathsf{Setup},\mathsf{Prove},\mathsf{Vfy})$ *where*

- Setup *on input security parameter 1^λ outputs a (sound) CRS* crs*,*
- Prove *on input CRS* crs*, statement $\mathbb{x}$ and a witness w outputs a proof π,*
- Vfy *on input CRS* crs*, statement $\mathbb{x}$, and a proof π outputs a bit b indicating the validity of the proof,*
- SimSetup *on input security parameter 1^λ outputs a (simulation) CRS* crs *and a (simulation) trapdoor* td*,*
- SimProve *on input CRS* crs*, statement $\mathbb{x}$ and a trapdoor* td *outputs a proof π.*

We define several properties for NIZKs.

Perfect completeness. *For each statement–witness pair $(\mathbb{x},\mathbb{w})\in R$ it holds that*

$$\Pr\big[\mathsf{crs}\leftarrow\mathsf{Setup}(1^\lambda),\pi\leftarrow\mathsf{Prove}(\mathbb{x},\mathbb{w}):\mathsf{Vfy}(\mathbb{x},\pi,\mathsf{crs})=1\big]=1\ . \tag{6}$$

Statistical soundness. *We say* NIZK *is statistically ϵ_{SND}-sound, iff for each statement $\mathbb{x}\notin L_R$ it holds that*

$$\Pr\big[\mathsf{crs}\leftarrow\mathsf{Setup}(1^\lambda):\exists\pi:\mathsf{Vfy}(\mathbb{x},\pi,\mathsf{crs})=1\big]\leq\epsilon_{\mathsf{SND}}(\lambda)\ . \tag{7}$$

Adaptive computational zero-knowledge. *We say* NIZK *is ϵ-zero-knowledge if there exists a simulator $\mathcal{S}$ such that for each polynomially time-bounded malicious verifier $\mathcal{A}$ the two following distributions*

are ϵ-indistinguishable to polynomially time-bounded distinguishers:

$$\left\{ \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathbb{x}, \mathbb{w}, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{crs}) \\ \mathbb{x} \in L_R \\ \pi \leftarrow \mathsf{Prove}(\mathbb{x}, \mathbb{w}, \mathsf{crs}) \end{array} : (\pi, \mathsf{st}) \right\} \approx_\epsilon \left\{ \begin{array}{c} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda) \\ (\mathbb{x}, \mathbb{w}, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{crs}) \\ \mathbb{x} \in L_R \\ \pi \leftarrow \mathsf{SimProve}(\mathbb{x}, \mathsf{td}, \mathsf{crs}) \end{array} : (\pi, \mathsf{st}) \right\} \tag{8}$$

We require that the malicious verifier produce a valid statement $\mathbb{x} \in L_R$.

CRS indistinguishability. *We say* NIZK *is ϵ-CRS-indistinguishable if the two following distributions are ϵ-indistinguishable to any polynomially time-bounded distinguisher:*

$$\left\{ \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) : (\mathsf{crs}) \right\} \approx_\epsilon \left\{ (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda) : (\mathsf{crs}) \right\} . \tag{9}$$

A.4 Hash Functions

Definition 3. *A hash function is a tuple of polynomial-time algorithms* $(\mathsf{Setup}, \mathsf{Eval})$ *where*
- $\mathsf{Setup}(1^\lambda)$ *on input security parameter* 1^λ*, outputs a CRS* crs*,*
- $\mathsf{Eval}(x, \mathsf{crs})$ *on input CRS* crs *and input* x*, outputs a hash value* y*.*

Setup *is necessarily probabilistic,* H *is deterministic.*

Collision-Resistance. *We say a hash function* H *is* ϵ_{cr}*-collision-resistant if for all non-uniform polynomial-time-bounded adversaries* $\mathcal{A}$ *it holds that*

$$\Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (x_1, x_2) \leftarrow \mathcal{A}(1^\lambda, \mathsf{crs}) \end{array} : x_1 \neq x_2 \wedge \mathsf{Eval}(x_1, \mathsf{crs}) = \mathsf{Eval}(x_2, \mathsf{crs}) \right] \leq \epsilon_{\mathsf{cr}}(\lambda) . \tag{10}$$

A.5 Patricia Tries

A Patricia trie is a data structure that allows for efficient storage and retrieval of key-value pairs. We use essentially the Patricia trie as described in [CDG+19] and refer the interested reader to [CDG+19] for a more detailed description of the implementation.

Definition 4. *A Patricia trie is a tuple of polynomial-time algorithms* $\mathsf{PTrie}_\mathsf{H} = (\mathsf{Eval}_\mathsf{H}, \mathsf{Copath}_\mathsf{H}, \mathsf{Vfy}_\mathsf{H}, \mathsf{Delim}_\mathsf{H})$ *using the hash function* H *where*
- Eval_H *on input a set of label–value pairs* $\{(\ell_1, h_1), \ldots, (\ell_n, h_n)\}$ *and a hash CRS, outputs the root hash value* h_ε *of the Patricia trie,*
- $\mathsf{Copath}_\mathsf{H}$ *on input a set of label–value pairs* $\{(\ell_1, h_1), \ldots, (\ell_n, h_n)\}$*, a label* ℓ*, and hash CRS* crs*, outputs the copath* cp *of the label* ℓ *in the Patricia trie,*
- Vfy_H *on input root hash* h_ε*, node hash value* h*, copath* cp*, and hash CRS* crs*, outputs* 1 *if the node value* h *hashes to the root hash along the copath, and* 0 *otherwise,*
- $\mathsf{Delim}_\mathsf{H}$ *on input a set of label–value pairs* $\{(\ell_1, h_1), \ldots, (\ell_n, h_n)\}$*, a (intermediate) label* ℓ*, and hash CRS* crs*, outputs the label–hash pairs of the parent* $(\ell_{\mathsf{parent}}, h_{\mathsf{parent}})$ *(the node this the longest common prefix),* $(\ell_{\mathsf{left}}, h_{\mathsf{left}})$ *(the left child of the parent), and* $(\ell_{\mathsf{right}}, h_{\mathsf{right}})$ *(the right child of the parent).*

B Key Transparency Scheme

In this section we formally introduce the notion of a key transparency scheme (KT scheme), similar to a VKD scheme [CDG+19]. Our KT scheme will be used in the construction of our KT protocol (Appendix C.2). The purpose of the KT scheme is twofold:
- It extremely simplifies the protocol description.

- Its formal security properties distill sufficient properties for the formal security proof of our KT protocol.
- It allows to modularly switch between different implementation of KT schemes without the need to overhaul the entire protocol (e.g. switch to post-quantum instantiations).

Definition 5 (Dual-Mode Key Transparency Scheme). *A dual-mode key transparency (KT) scheme is a tuple of efficient algorithms* $\mathsf{KTS} = (\mathsf{ExtSetup}, \mathsf{Init}, \mathsf{Commit}, \mathsf{QryKey}, \mathsf{VfyQry}, \mathsf{UpdateEpoch}, \mathsf{VerifyUpdate}, \mathsf{SimSetup}, \mathsf{SimQryKey}, \mathsf{SimUpdateEpoch}, \mathsf{ExtKeys})$ *where*

- $\mathsf{ExtSetup}(1^\lambda)$ *on input security parameter* 1^λ*, outputs a CRS* crs *and an extraction trapdoor* td*.*
- $\mathsf{Init}(1^\lambda, \mathsf{crs})$ *on input security parameter* 1^λ *and CRS* crs*, outputs a server state* st*,*
- $\mathsf{Commit}(\mathsf{st}, \mathsf{D}, \mathsf{crs})$ *on input server state* st*, database* D*, and a common reference string (CRS)* crs*, outputs a commitment* com *to the server state.*
- $\mathsf{QryKey}(\mathsf{st}, \mathsf{D}, \mathsf{id}, \mathsf{v}, \mathsf{crs})$ *on input server state* st*, database* D*, user identifier* id*, version* v*, and CRS* crs*, outputs the key* k *associated with* id *at version* v*, and a proof* π *of correctness,*
- $\mathsf{VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi, \mathsf{crs})$ *on input commitment* com*, user identifier* id*, version* v*, key* k*, proof* π*, and CRS* crs*, outputs a bit indicating the correctness of the key relative to the commitment* com*.*
- $\mathsf{UpdateEpoch}(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{crs})$ *on input server state* st*, database* D*, list of key updates* $\mathsf{L} = (\mathsf{id}_i, \mathsf{k}_i)_i$*, and CRS* crs*, outputs the new database* D' *and a proof* π *of correctness.*
- $\mathsf{VerifyUpdate}(\mathsf{com}, \mathsf{com}', \pi, \mathsf{crs})$ *on input commitments* $\mathsf{com}, \mathsf{com}'$*, proof* π*, and CRS* crs*, outputs a bit indicating the validity of the new commitment relative to the previous commitment.*
- $\mathsf{SimSetup}(1^\lambda)$ *on input security parameter* 1^λ*, outputs a CRS* crs *and a simulation trapdoor* td*.*
- $\mathsf{SimQryKey}(\mathsf{com}, S, \mathsf{k}, \mathsf{crs}, \mathsf{td})$ *on input commitment* com*, set of identifier–version–key triples* $S = \{(\mathsf{id}, \mathsf{v}, \mathsf{k})\}$*, CRS* crs*, and simulation trapdoor* td*, outputs a list of proof* $(\pi_{\mathsf{id},\mathsf{v}})_{\mathsf{id},\mathsf{v}}$*.*
- $\mathsf{SimUpdateEpoch}(\mathsf{com}, \mathsf{com}', \mathsf{crs}, \mathsf{td})$ *on input commitments* $\mathsf{com}, \mathsf{com}'$*, CRS* crs*, and simulation trapdoor* td*, outputs a proof* π*.*
- $\mathsf{ExtKeys}(\mathsf{com}^1_{\mathsf{KTS}}, ..., \mathsf{com}^\tau_{\mathsf{KTS}}, \pi^1_{\mathrm{upd}}, ..., \pi^\tau_{\mathrm{upd}}, \mathsf{crs}, \mathsf{td})$ *on input commitments* com_i*, update proofs* π_i*, CRS* crs*, and extraction trapdoor* td*, outputs the database* D *associated with the commitment* com*.*

In particular the algorithms Commit*,* QryKey*,* $\mathsf{VerifyUpd}$ *and* $\mathsf{ExtKeys}$ *are deterministic.*

We define several correctness and security properties for KT systems which games formalized in Figure 5.

Correctness. *We say a KT scheme is (perfectly) correct iff for every (unbounded) stateful adversary* $\mathcal{A}$*, we have that* $\Pr[\mathsf{Exp}^{\mathrm{corr}}_{\mathsf{KTS},\mathcal{A}}(\lambda) = 1] = 0$ *where* $\mathsf{Exp}^{\mathrm{corr}}_{\mathsf{KTS},\mathcal{A}}$ *is the correctness game defined in Figure 5. Correctness guarantees that (verifying) responses match the underlying database.*

Intra-Epoch Consistency. *We say a KT scheme is* $\epsilon_{\mathrm{intra}}$*-intra-epoch consistent iff for every (unbounded) adversary* $\mathcal{A}$*, we have that* $\Pr[\mathsf{Exp}^{\mathrm{intra}}_{\mathsf{KTS},\mathcal{A}}(\lambda) = 1] \leq \epsilon_{\mathrm{intra}}(\lambda)$ *where* $\mathsf{Exp}^{\mathrm{intra}}_{\mathsf{KTS},\mathcal{A}}$ *is the intra-epoch consistency game defined in Figure 5. Intra-Epoch consistency guarantees that within a single epoch (a single commitment* com*) only a single key verifies for a given query, i.e., database label.*

Inter-Epoch Consistency. *Let us define a partial order* $\leq$ *on the set of databases. For each two databases* $\mathsf{D}^1, \mathsf{D}^2$ *we write* $\mathsf{D}^1 \leq \mathsf{D}^2$ *(*D^2 *is a valid extension of* D^1*), iff* D^2 *contains all (non-*$\perp$*) keys of* D^1 *plus at most one new key for each user* id*. Formally,*[17]

$$\forall \mathsf{id}, \mathsf{v} \neq \left|\mathsf{D}^1[\mathsf{id}]\right| + 1 : \mathsf{D}^2[\mathsf{id}][\mathsf{v}] = \mathsf{D}^1[\mathsf{id}][\mathsf{v}] \ . \tag{11}$$

We say a KT scheme is $\epsilon_{\mathrm{inter}}$*-inter-epoch consistent iff for every (unbounded) adversary* $\mathcal{A}$*, we have that* $\Pr[\mathsf{Exp}^{\mathrm{inter}}_{\mathsf{KTS},\mathcal{A}}(\lambda) = 1] \leq \epsilon_{\mathrm{inter}}(\lambda)$ *where* $\mathsf{Exp}^{\mathrm{inter}}_{\mathsf{KTS},\mathcal{A}}$ *is the inter-epoch consistency game defined in Figure 5. This notion essentially corresponds to "VKD soundness" in [CDG+19]; it ensures once a given key version has been added, its valid response remains consistent across epochs.*

Privacy. *We say a KT scheme is* ϵ_{priv}*-private iff for all polynomially time-bounded adversaries* $\mathcal{A}$ *its*

[17] Note that our partial order $\leq$ is *not* transitive.

$\mathsf{Exp}^{\mathrm{corr}}_{\mathsf{KTS},\mathcal{A}}(\lambda)$

$(\mathsf{st}, \mathsf{crs}, \mathsf{D}, \mathsf{id}, \mathsf{v}, \mathsf{L}) \leftarrow \mathcal{A}(1^\lambda)$
// honestly generate commitment
$\mathsf{com} \coloneqq \mathsf{KTS.Commit}(\mathsf{st}, \mathsf{D}, \mathsf{crs})$
// honestly generate epoch update
$(\mathsf{D}', \pi_{\mathrm{upd}}) \coloneqq \mathsf{KTS.UpdateEpoch}(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{crs})$
// honestly generate next commitment
$\mathsf{com}' \coloneqq \mathsf{KTS.Commit}(\mathsf{st}, \mathsf{D}', \mathsf{crs})$
// honestly generate query response
$(\mathsf{k}, \pi) \coloneqq \mathsf{KTS.QryKey}(\mathsf{st}, \mathsf{D}, (\mathsf{id}, \mathsf{v}), \mathsf{crs})$
// adversary wins if
// honest query response doesn't match database
if $\mathsf{D}[\mathsf{id}, \mathsf{v}] \neq \mathsf{k}$ **return** 1
// honest query response doesn't verify
if $\mathsf{KTS.VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi, \mathsf{crs}) = 0$
  return 1
// honest update epoch doesn't verify
if $\mathsf{KTS.VerifyUpdate}(\mathsf{com}, \mathsf{com}', \pi_{\mathrm{upd}}, \mathsf{crs}) = 0$
  return 1
if $\mathsf{D} \not\leq \mathsf{D}'$ **return** 1 // honest D' doesn't extend D
// updates not exactly incorporated
if $\exists \mathsf{id} : \mathsf{D}'[\mathsf{id}][|\mathsf{D}[\mathsf{id}]+1|] \neq \mathsf{L}[\mathsf{id}]$ **return** 1
return 0

$\mathsf{Exp}^{\mathrm{sim}}_{\mathsf{KTS},\mathcal{A}}(\lambda)$

$(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{KTS.SimSetup}(1^\lambda)$
$\mathsf{st} \leftarrow \mathsf{KTS.Init}(1^\lambda, \mathsf{crs})$
$(\mathsf{D}, \mathsf{L}) \leftarrow \mathcal{A}_1(1^\lambda, \mathsf{crs}, \mathsf{td}, \mathsf{st})$
// generate update proofs
$(\mathsf{D}', \pi^0_{\mathrm{upd}}) \coloneqq \mathsf{KTS.UpdateEpoch}(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{crs})$
$\pi^1_{\mathrm{upd}} \coloneqq \mathsf{KTS.SimUpdateEpoch}(\mathsf{com}_{\mathsf{KTS}},$
$\qquad \mathsf{com}'_{\mathsf{KTS}}, |\mathsf{D}'|, \mathsf{crs}, \mathsf{td})$
// generate query responses
$i \coloneqq 1; N \coloneqq |\mathsf{D}'|$
$\mathsf{com}_{\mathsf{KTS}} \coloneqq \mathsf{KTS.Commit}(\mathsf{st}, \mathsf{D}, \mathsf{crs})$
for $\mathsf{id}, \mathsf{v} : \mathsf{D}[\mathsf{id}][\mathsf{v}] \neq \bot$
  $(\mathsf{k}, \pi^0_{\mathsf{id},\mathsf{v}}) \coloneqq \mathsf{KTS.QryKey}(\mathsf{st}, \mathsf{D}, \mathsf{id}, \mathsf{v}, \mathsf{crs})$
  $\pi^1_{\mathsf{id},\mathsf{v}} \coloneqq \mathsf{KTS.SimQryKey}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, i, N, \mathsf{crs}, \mathsf{td})$
  $i \coloneqq i + 1$
$b \leftarrow \{0,1\}$
$P^b \coloneqq \{(\mathsf{id}, \mathsf{v}, \pi^b_{\mathsf{id},\mathsf{v}}) \mid \mathsf{D}[\mathsf{id}][\mathsf{v}] \neq \bot\}$
$b' \leftarrow \mathcal{A}_2(1^\lambda, \mathsf{crs}, \mathsf{td}, \mathsf{st}, \mathsf{D}, P^b, \pi^b_{\mathrm{upd}})$
if $b = b'$ **return** 1
else return 0

$\mathsf{Exp}^{\mathrm{mode}}_{\mathsf{KTS},\mathcal{A}}(\lambda)$

$b \leftarrow \{0,1\}$
$(\mathsf{crs}_0, \mathsf{td}_0) \leftarrow \mathsf{KTS.ExtSetup}(1^\lambda)$
$(\mathsf{crs}_1, \mathsf{td}_1) \leftarrow \mathsf{KTS.SimSetup}(1^\lambda)$
$b' \leftarrow \mathcal{A}(\mathsf{crs}_b)$
req $b = b'$
return 1

$\mathsf{Exp}^{\mathrm{intra}}_{\mathsf{KTS},\mathcal{A}}(\lambda)$

$(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{KTS.ExtSetup}(1^\lambda)$
$(\mathsf{com}^1, ..., \mathsf{com}^\tau, \pi^1_{\mathrm{upd}}, ..., \pi^\tau_{\mathrm{upd}}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi) \leftarrow \mathcal{A}(\mathsf{crs}, \mathsf{td})$
$\mathsf{com}^0 \coloneqq \bot$
for $i \in \{1, ..., d\}$
  req $\mathsf{KTS.VerifyUpd}(\mathsf{com}^{i-1}, \mathsf{com}^i, \pi^i_{\mathrm{upd}}, \mathsf{crs}) = 1$
req $\mathsf{KTS.VfyQry}(\mathsf{com}^\tau, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi, \mathsf{crs}) = 1$
$\mathsf{D}^\tau \coloneqq \mathsf{KTS.ExtKeys}(\mathsf{com}^1, ..., \mathsf{com}^\tau, \pi^1_{\mathrm{upd}}, ..., \pi^\tau_{\mathrm{upd}}, \mathsf{crs}, \mathsf{td})$
req $\mathsf{D}^\tau[\mathsf{id}][\mathsf{v}] \neq \mathsf{k}$
return 1

$\mathsf{Exp}^{\mathrm{inter}}_{\mathsf{KTS},\mathcal{A}}(\lambda)$

$(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{KTS.ExtSetup}(1^\lambda)$
$(\mathsf{com}^1, ..., \mathsf{com}^\tau, \pi^1_{\mathrm{upd}}, ..., \pi^\tau_{\mathrm{upd}}) \leftarrow \mathcal{A}(\mathsf{crs}, \mathsf{td})$
$\mathsf{com}^0 \coloneqq \bot$
$\mathsf{D}^0 = [[], ..., []]$ // empty database of n users
for $i \in \{1, ..., d\}$
  req $\mathsf{KTS.VerifyUpd}(\mathsf{com}^{i-1}, \mathsf{com}^i, \pi^i_{\mathrm{upd}}, \mathsf{crs}) = 1$
  $\mathsf{D}^i \coloneqq \mathsf{KTS.ExtKeys}(\mathsf{com}^1, ..., \mathsf{com}^i, \pi^1_{\mathrm{upd}}, ..., \pi^i_{\mathrm{upd}}, \mathsf{crs}, \mathsf{td})$
req $\exists \iota : \mathsf{D}^{\iota-1} \not\leq \mathsf{D}^\iota$ // D^ι is not a valid extension of $\mathsf{D}^{\iota-1}$
return 1

$\mathsf{Exp}^{\mathrm{priv}}_{\mathsf{KTS},\mathcal{A}}(\lambda)$

$(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{KTS.SimSetup}(1^\lambda)$
$(\mathsf{D}^0, \mathsf{D}^1) \leftarrow \mathcal{A}(\mathsf{crs})$
req $|\mathsf{D}^0| = |\mathsf{D}^1|$
$\mathsf{st} \leftarrow \mathsf{KTS.Init}(1^\lambda)$
$b \leftarrow \{0,1\}$
$\mathsf{com} \coloneqq \mathsf{KTS.Commit}(\mathsf{st}, \mathsf{D}^b, \mathsf{crs})$
$b' \leftarrow \mathcal{A}^{\mathsf{KTS.SimQryKey}(\cdot,\cdot,\cdot,\mathsf{crs},\mathsf{td}),\mathsf{KTS.SimUpdateEpoch}(\cdot,\cdot,\cdot,\mathsf{crs},\mathsf{td})}(\mathsf{crs}, \mathsf{com})$
req $b = b'$
return 1

Fig. 5: Games for our definition of KT schemes.

advantage is bounded by

$$\left|2\Pr\left[\mathsf{Exp}^{\mathrm{priv}}_{\mathsf{KTS},\mathcal{A}}(\lambda)=1\right]-1\right|\leq\epsilon_{\mathrm{priv}}(\lambda) \tag{12}$$

where $\mathsf{Exp}^{\mathrm{priv}}_{\mathsf{KTS},\mathcal{A}}$ *is the privacy game defined in Figure 5. Privacy guarantees that an adversary cannot distinguish between two databases (in the simulation mode).*
The following three properties are used only in the proof of security of our KT protocol.
Mode Indistinguishability. *We say a KT scheme* KTS *is* ϵ_{priv}*-CRS-indistinguishable if for all non-uniform polynomially time-bounded adversaries* $\mathcal{A}$ *its advantage is bounded by*

$$\left|2\Pr\left[\mathsf{Exp}^{\mathrm{mode}}_{\mathsf{KTS},\mathcal{A}}(\lambda)=1\right]-1\right|\leq\epsilon_{\mathrm{priv}}(\lambda) \tag{13}$$

with the mode indistinguishability game defined in Figure 5.
Simulation correctness. *We say a KT scheme is simulation-correct if for each* $\mathsf{com},\mathsf{com}',\ell,\mathsf{id}_1,\mathsf{v}_1,\mathsf{k}_1,...,\mathsf{id}_\ell,\mathsf{v}_\ell,\mathsf{k}_\ell,\mathsf{L}$ *we require that*

$$\Pr\left[\begin{array}{l}(\mathsf{crs},\mathsf{td})\leftarrow\mathsf{SimSetup}(1^\lambda)\\(\pi_i)_{i\in[\ell]}\coloneqq\mathsf{SimQryKey}(\mathsf{com},(\mathsf{id}_i,\mathsf{v}_i)_{i\in[\ell]},\mathsf{crs},\mathsf{td})\end{array}:\forall i\in[\ell]:\mathsf{VfyQry}(\mathsf{com},\mathsf{id}_i,\mathsf{v}_i,\mathsf{k}_i,\pi_i,\mathsf{crs})=1\right]=1 \tag{14}$$

and

$$\Pr\left[\begin{array}{c}(\mathsf{crs},\mathsf{td})\leftarrow\mathsf{SimSetup}(1^\lambda)\\\pi_{\mathrm{upd}}\coloneqq\mathsf{SimUpdateEpoch}(\mathsf{com},\mathsf{com}',\mathsf{L},\mathsf{crs},\mathsf{td})\end{array}:\mathsf{VerifyUpdate}(\mathsf{com},\mathsf{com}',\pi_{\mathrm{upd}},\mathsf{crs})=1\right]=1\ . \tag{15}$$

Perfect simulation indistinguishability. *We say a KT scheme is* ϵ_{sim}*-simulation-indistinguishable if for all non-uniform (unbounded) adversaries* $\mathcal{A}_1,\mathcal{A}_2$ *its advantage is bounded by*

$$\left|2\Pr\left[\mathsf{Exp}^{\mathrm{sim}}_{\mathsf{KTS},\mathcal{A}}(\lambda)=1\right]-1\right|\leq\epsilon_{\mathrm{sim}}(\lambda) \tag{16}$$

where $\mathsf{Exp}^{\mathrm{sim}}_{\mathsf{KTS},\mathcal{A}}$ *is the privacy game defined in Figure 5. We use perfect simulation indistinguishability* $\epsilon_{\mathrm{sim}}=0$ *merely for ease of exhibition; computational indistinguishability suffices.*

The keen reader will notice that our notion of a KT scheme is an adaptation of the verifiable key directory scheme (VKD) in [CDG+19]. At their core both primitives provide the necessary interface to store and verify data in a persistent manner. Because we deem it insightful for the further standarization process of KT systems, we discuss the relation between our KT scheme and the VKD scheme of [CDG+19] in Appendix D.

C Formal Constructions and Proofs

For our security analysis we assume adversaries to be non-uniform polynomial-time (or unbounded) Turing machines. We denote by negl some negligible function (existentially quantified).

C.1 KT Scheme Instantiation

In this section we provide an instantiation of a KT scheme as introduced in Definition 5.

Our KT scheme is inspired by the VKD primitive of [CDG+19]. Whereas the VKD scheme of [CDG+19] is based on another primitive, called append-only zero-knowledge set (aZKS) [CDG+19], we build our KT scheme directly from (simulatable) VRFs and collision-resistant hash functions. The reasons are twofold:

- In comparison to a VKD the security properties that we need from our KT scheme are stronger (e.g. we need extractability).

- The VKD provides a simulation-type privacy notion, whereas for our purposes a simple indistinguishability notion suffices.

Construction 1 (KT Scheme Instantiation)**.** *Let* VRF *be an sVRF. Let* H *be a hash function. Let* NIZK *be a NIZK. We define a KT scheme* KTS *in Figure 5 with the following NP relation: The statement about the consistency of the epoch commitment* $\mathbb{x}_{\mathsf{upd}} = (\mathsf{com}_{\mathsf{KTS}}, \mathsf{com}'_{\mathsf{KTS}}, W, W')$ *and the witness* $\mathbb{w}_{\mathsf{upd}} = (\mathsf{sk}, (\mathsf{id}_i, \mathsf{v}_i, \mathsf{k}_i)_{i \in |W' \setminus W|})$ *is defined as follows:*

- *there exists a VRF secret key* sk *corresponding the public key* vk*, i.e.,* $(\mathsf{vk}, \mathsf{sk}) \in \mathsf{VRF.Gen}(1^\lambda)$*,*
- *for each (added) label–value pair* $(\ell'_i, h'_i) \in W' \setminus W$ *there exists a tuple* $(\mathsf{id}_i, \mathsf{v}_i, \mathsf{k}_i, h_i)$ *such that* $\mathsf{v}_i = 1$ *or* $(\ell_i, h_i) \in W$ *was already contained in the old commitment where* $(\ell_i, \cdot) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}, \mathsf{id}_i || \mathsf{v}_i - 1)$*.*

Our construction follows the blueprint of the VKD scheme of [CDG+19]. That is we use a Patricia trie to store the elements in the database. Proofs for queries contain a membership for the most current version of an identity's key, and a non-inclusion proof for the next version. Proofs for inter-epoch consistency contain all new key-value pairs, and a proof that only those pairs were added to the Patricia trie (and no previous entry modified).

Lemma 1. *If* VRF *is uniquely provable and pseudorandom,* H *is collision-resistant, and* NIZK *is zero-knowledge, sound and mode-indistinguishable, then the KT scheme from Construction 1 fulfills all properties specified in Definition 5.*

Proof Sketch of Lemma 1. We prove that the KT scheme from Construction 1 fulfills all properties specified in Definition 5.

Correctness. The (perfect) correctness and simulation correctness of Construction 1 follows directly from the perfect correctness of the sVRF and the perfect completeness of the NIZK.

Intra-Epoch consistency. The intra-epoch consistency of Construction 1 follows from the collision-resistance of the hash function, and the uniqueness of the sVRF. Breaking intra-epoch consistency means that an adversary produces two different *valid* query responses for the same id and version, but with different keys $\mathsf{k} \neq \mathsf{k}'$. Assuming the hash function is collision-resistant, then the root value of the Patricia trie fixes all leaf values, in particular all $\ell_{\mathsf{id},\mathsf{v}}$ and $y_{\mathsf{id},\mathsf{v}}$ values. In turn, the the unique provability of the sVRF then fixes $\mathsf{id}, \mathsf{v}, \mathsf{k}$ for every leaf. Hence, an adversary cannot produce two different valid query responses for the same id and version, but with different keys $\mathsf{k} \neq \mathsf{k}'$. This is conceptually similar to the soundness notion in the VKD scheme of [CDG+19].

Inter-Epoch consistency. The inter-epoch consistency of Construction 1 follows from the soundness of the NIZK, the collision-resistance of the hash function, and the uniqueness of the sVRF. Breaking inter-epoch consistency means that an adversary produces two different *valid* epoch commitments $\mathsf{com}^{\tau-1}_{\mathsf{KTS}}, \mathsf{com}^{\tau}_{\mathsf{KTS}}$ whose underlying databases $\mathsf{D}^{\tau-1}, \mathsf{D}^{\tau}$ are not well-ordered; in the sense of the $\leq$ relation in Definition 5. Again, assuming the collision-resistance of the hash function and the uniqueness of the sVRF, then the root value of the Patricia trie fixes the entire database (of an epoch) relative to its epoch commitment. Now, the NIZK statement $\mathbb{x}_{\mathsf{upd}}$ essentially enforces the well-ordering of the database $\mathsf{D} \leq \mathsf{D}'$. Concretely, it enforces that all old keys of the old database $\mathsf{D}^{\tau-1}$ remain unmodified in the new database D^{τ}. Moreover, it enforces that for each newly added key under identity id the previous version already existed in the old database $\mathsf{D}^{\tau-1}$. Hence, by the soundness of the NIZK no adversary can produce two different valid epoch commitments $\mathsf{com}^{\tau-1}_{\mathsf{KTS}}, \mathsf{com}^{\tau}_{\mathsf{KTS}}$ whose underlying databases $\mathsf{D}^{\tau-1}, \mathsf{D}^{\tau}$ are not well-ordered. This is differs from the soundness notion in the VKD scheme of [CDG+19] which does not offer such "temporal" consistency.

$\mathsf{ExtSetup}(1^\lambda)$

$\mathsf{crs}_{\mathsf{VRF}} \leftarrow \mathsf{VRF.Setup}(1^\lambda)$
$\mathsf{crs}_{\mathsf{H}} \leftarrow \mathsf{H.Setup}(1^\lambda)$
$\mathsf{crs}_{\mathsf{NIZK}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$
$\mathsf{crs}_{\mathsf{KTS}} \coloneqq (\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}, \mathsf{crs}_{\mathsf{NIZK}})$
$(\mathsf{vk}', \mathsf{sk}') \leftarrow \mathsf{VRF.Gen}(1^\lambda, \mathsf{crs}_{\mathsf{VRF}})$ // could be PRF as well
$\mathsf{td}_{\mathsf{KTS}} \coloneqq \mathsf{sk}'$
return $(\mathsf{crs}_{\mathsf{KTS}}, \mathsf{td}_{\mathsf{KTS}})$

$\mathsf{Init}(1^\lambda)$

$(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{VRF.Gen}(1^\lambda, \mathsf{crs}_{\mathsf{VRF}})$
return $\mathsf{st}_{\mathsf{KTS}} \coloneqq (\mathsf{vk}, \mathsf{sk})$

$\mathsf{Commit}(\mathsf{st}_{\mathsf{KTS}}, \mathsf{D}, \mathsf{crs}_{\mathsf{KTS}})$

$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}, \mathsf{crs}_{\mathsf{NIZK}}) \coloneqq \mathsf{crs}_{\mathsf{KTS}}$
$(\mathsf{vk}, \mathsf{sk}) \coloneqq \mathsf{st}_{\mathsf{KTS}}$
for id
  for $\mathsf{v} \in [|\mathsf{D}[\mathsf{id}]|]$
    $\mathsf{k} \coloneqq \mathsf{D}[\mathsf{id}][\mathsf{v}]$
    $(\ell_{\mathsf{id},\mathsf{v}}, \pi_{\mathsf{id},\mathsf{v}}) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}||\mathsf{v}, \mathsf{crs}_{\mathsf{VRF}})$
    $(y_{\mathsf{id},\mathsf{v}}, \pi'_{\mathsf{id},\mathsf{v}}) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}||\mathsf{v}||\mathsf{k}, \mathsf{crs}_{\mathsf{VRF}})$
// list of labels and leaf values
$W \coloneqq \{(\ell_{\mathsf{id}',\mathsf{v}'}, y_{\mathsf{id}',\mathsf{v}'}) \mid \mathsf{D}[\mathsf{id}'][\mathsf{v}'] \neq \bot\}$
$h_\varepsilon \coloneqq \mathsf{PTrie_H.Eval_H}(W)$
return $\mathsf{com}_{\mathsf{KTS}} \coloneqq (\mathsf{vk}, h_\varepsilon)$

$\mathsf{QryKey}(\mathsf{st}_{\mathsf{KTS}}, \mathsf{D}, \mathsf{id}, \mathsf{v}, \mathsf{crs}_{\mathsf{KTS}})$

$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}, \mathsf{crs}_{\mathsf{NIZK}}) \coloneqq \mathsf{crs}_{\mathsf{KTS}}$
$(\mathsf{vk}, \mathsf{sk}) \coloneqq \mathsf{st}_{\mathsf{KTS}}$
for $\mathsf{id}' \in [n], \mathsf{v}' \in [1, ..., \left|\mathsf{D}[\mathsf{id}']\right|]$
  $\mathsf{k}' \coloneqq \mathsf{D}[\mathsf{id}'][\mathsf{v}']$
  $(\ell_{\mathsf{id}',\mathsf{v}'}, \pi_{\mathsf{id}',\mathsf{v}'}) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}'||\mathsf{v}', \mathsf{crs}_{\mathsf{VRF}})$
  $(y_{\mathsf{id}',\mathsf{v}'}, \pi'_{\mathsf{id}',\mathsf{v}'}) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}'||\mathsf{v}'||\mathsf{k}', \mathsf{crs}_{\mathsf{VRF}})$
$W \coloneqq (\ell_{\mathsf{id},\mathsf{v}}, y_{\mathsf{id},\mathsf{v}})_{\mathsf{id},\mathsf{v}}$
// inclusion proof of current version
$\mathsf{cp} \coloneqq \mathsf{PTrie_H.Copath_H}(W, \ell_{\mathsf{id},\mathsf{v}}, \mathsf{crs}_{\mathsf{H}})$
// non-inclusion of next version
$(\ell_{\mathsf{parent}}, h_{\mathsf{parent}}, \ell_{\mathsf{left}}, h_{\mathsf{left}}, \ell_{\mathsf{right}}, h_{\mathsf{right}})$
  $\coloneqq \mathsf{PTrie_H.Delim_H}(W, \ell_{\mathsf{id},\mathsf{v}}, \mathsf{crs}_{\mathsf{H}})$
$\mathsf{cp}' \coloneqq \mathsf{PTrie_H.Copath_H}(W, \ell_{\mathsf{parent}}, \mathsf{crs}_{\mathsf{H}})$
$\pi \coloneqq (\ell_{\mathsf{id},\mathsf{v}}, \pi_{\mathsf{id},\mathsf{v}}, y_{\mathsf{id},\mathsf{v}}, \pi'_{\mathsf{id},\mathsf{v}}, \mathsf{cp},$
  $\ell_{\mathsf{parent}}, h_{\mathsf{parent}}, \ell_{\mathsf{left}}, h_{\mathsf{left}}, \ell_{\mathsf{right}}, h_{\mathsf{right}}, \mathsf{cp}')$
return (k, π)

$\mathsf{VfyQry}(\mathsf{com}_{\mathsf{KTS}}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi, \mathsf{crs}_{\mathsf{KTS}})$

$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}, \mathsf{crs}_{\mathsf{NIZK}}) \coloneqq \mathsf{crs}_{\mathsf{KTS}}$
$(\mathsf{vk}, h_\varepsilon) \coloneqq \mathsf{com}_{\mathsf{KTS}}$
$(\ell_{\mathsf{id},\mathsf{v}}, \pi_{\mathsf{id},\mathsf{v}}, y_{\mathsf{id},\mathsf{v}}, \pi'_{\mathsf{id},\mathsf{v}}, \mathsf{cp},$
$\ell_{\mathsf{parent}}, h_{\mathsf{parent}}, \ell_{\mathsf{left}}, h_{\mathsf{left}}, \ell_{\mathsf{right}}, h_{\mathsf{right}}, \mathsf{cp}') \coloneqq \pi$
req $\mathsf{k} \neq \bot$
// verify inclusion
req $\mathsf{VRF.Vfy}(\mathsf{vk}, \mathsf{id}||\mathsf{v}, \ell_{\mathsf{id},\mathsf{v}}, \pi_{\mathsf{id},\mathsf{v}}, \mathsf{crs}_{\mathsf{VRF}}) = 1$
req $\mathsf{VRF.Vfy}(\mathsf{vk}, \mathsf{id}||\mathsf{v}||\mathsf{k}, y_{\mathsf{id},\mathsf{v}}, \pi'_{\mathsf{id},\mathsf{v}}, \mathsf{crs}_{\mathsf{VRF}}) = 1$
req $\mathsf{PTrie_H.Vfy}(h_\varepsilon, h_{\mathsf{id},\mathsf{v}}, \mathsf{cp}, \mathsf{crs}_{\mathsf{H}}) = 1$
// verify non-inclusion
req $h_{\mathsf{parent}} = \mathsf{H.Eval}(y_{\mathsf{left}}||y_{\mathsf{right}}, \mathsf{crs}_{\mathsf{H}})$
req $\mathsf{PTrie_H.Vfy}(h_\varepsilon, h_{\mathsf{parent}}, \mathsf{cp}', \mathsf{crs}_{\mathsf{H}}) = 1$
  $L_{\mathsf{left}} \coloneqq |\ell_{\mathsf{left}}|$ // $\ell_{\mathsf{left}} \in \{0,1\}^{L_{\mathsf{left}}}$
  $L_{\mathsf{right}} \coloneqq |\ell_{\mathsf{right}}|$ // $\ell_{\mathsf{right}} \in \{0,1\}^{L_{\mathsf{right}}}$
  $L \coloneqq |\ell_{\mathsf{id},\mathsf{v}}|$ // $L = \ell_{\mathsf{y}}(\lambda)$ is the VRF output length
  // $\ell_{\mathsf{id},\mathsf{v}}$ is between ℓ_{left} and ℓ_{right}
  req $\mathrm{int}(\ell_{\mathsf{left}}||1^{L-L_{\mathsf{left}}}) \lneq \mathrm{int}(\ell_{\mathsf{id},\mathsf{v}}) \lneq \mathrm{int}(\ell_{\mathsf{right}}||0^{L-L_{\mathsf{right}}})$
return 1

$\mathsf{UpdateEpoch}(\mathsf{st}_{\mathsf{KTS}}, \mathsf{D}, \mathsf{L}, \mathsf{crs}_{\mathsf{KTS}})$

$\mathsf{D}' \coloneqq \mathsf{D}$
for id
  $\mathsf{k} \coloneqq \mathsf{L}[\mathsf{id}]$
  if $\mathsf{k} \neq \bot$
    $\mathsf{v} \coloneqq |\mathsf{D}[\mathsf{id}]|$
    $\mathsf{D}'[\mathsf{id}][\mathsf{v}+1] \coloneqq \mathsf{k}$ // insert new key at next version
for $\mathsf{id} \in [n], \mathsf{v} \in [\left|\mathsf{D}'[\mathsf{id}]\right|]$
  $\mathsf{k} \coloneqq \mathsf{D}'[\mathsf{id}][\mathsf{v}]$
  $(\ell_{\mathsf{id},\mathsf{v}}, \pi_{\mathsf{id},\mathsf{v}}) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}||\mathsf{v}, \mathsf{crs}_{\mathsf{VRF}})$
  $(y_{\mathsf{id},\mathsf{v}}, \pi'_{\mathsf{id},\mathsf{v}}) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{id}||\mathsf{v}||\mathsf{k}, \mathsf{crs}_{\mathsf{VRF}})$
$W \coloneqq \{(\ell_{\mathsf{id}',\mathsf{v}'}, y_{\mathsf{id}',\mathsf{v}'}) \mid \mathsf{D}[\mathsf{id}'][\mathsf{v}'] \neq \bot\}$
$W' \coloneqq \{(\ell_{\mathsf{id}',\mathsf{v}'}, y_{\mathsf{id}',\mathsf{v}'}) \mid \mathsf{D}'[\mathsf{id}'][\mathsf{v}'] \neq \bot\}$
$\Delta W \coloneqq W' \setminus W$
$\mathsf{com}_{\mathsf{KTS}} \coloneqq \mathsf{KTS.Commit}(\mathsf{st}_{\mathsf{KTS}}, \mathsf{D}, \mathsf{crs}_{\mathsf{KTS}})$
$\mathsf{com}'_{\mathsf{KTS}} \coloneqq \mathsf{KTS.Commit}(\mathsf{st}_{\mathsf{KTS}}, \mathsf{D}', \mathsf{crs}_{\mathsf{KTS}})$
// statement of properly added keys
$\mathbb{x}_{\mathsf{upd}} \coloneqq \left(\mathsf{com}_{\mathsf{KTS}}, \mathsf{com}'_{\mathsf{KTS}}, W, W'\right)$
// witness for the statement
$\mathbb{w}_{\mathsf{upd}} \coloneqq \left(\mathsf{sk}, \{(\mathsf{id}, \mathsf{v}, \mathsf{k}) \;\middle|\; \mathsf{D}[\mathsf{id}][\mathsf{v}] = \bot \wedge \mathsf{k} \coloneqq \mathsf{D}'[\mathsf{id}][\mathsf{v}] \neq \bot\}\right)$
// generate the NIZK proof
$\pi_{\mathsf{NIZK}} \leftarrow \mathsf{NIZK.Prove}(\mathbb{x}_{\mathsf{upd}}, \mathbb{w}_{\mathsf{upd}}, \mathsf{crs}_{\mathsf{NIZK}})$
$\pi_{\mathsf{KTS}} \coloneqq (\Delta W, \pi_{\mathsf{NIZK}})$
return $(\mathsf{D}', \pi_{\mathsf{KTS}})$

$\mathsf{VerifyUpdate}(\mathsf{com}^1_{\mathsf{KTS}}, ..., \mathsf{com}^\tau_{\mathsf{KTS}}, \pi^1_{\mathrm{upd}}, ..., \pi^\tau_{\mathrm{upd}}, \mathsf{crs}_{\mathsf{KTS}})$

$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}, \mathsf{crs}_{\mathsf{NIZK}}) \coloneqq \mathsf{crs}_{\mathsf{KTS}}$
$\mathsf{com}^0_{\mathsf{KTS}} \coloneqq \bot; W^0 \coloneqq \emptyset$
for $\iota \in [\tau]$
  $(\Delta W^\iota, \pi^\iota_{\mathsf{NIZK}}) \coloneqq \pi^\iota_{\mathrm{upd}}$
  $W^\iota \coloneqq W^{\iota-1} \cup \Delta W^\iota$
  $(\mathsf{vk}, h_\varepsilon) \coloneqq \mathsf{com}^{\iota-1}_{\mathsf{KTS}}$
  $(\mathsf{vk}', h'_\varepsilon) \coloneqq \mathsf{com}^{\iota}_{\mathsf{KTS}}$
  req $\mathsf{vk} = \mathsf{vk}'$
  req $h_\varepsilon = \mathsf{PTrie}_{\mathsf{H}}.\mathsf{Eval}_{\mathsf{H}}(W^{\iota-1}, \mathsf{crs}_{\mathsf{H}})$
  req $h'_\varepsilon = \mathsf{PTrie}_{\mathsf{H}}.\mathsf{Eval}_{\mathsf{H}}(W^{\iota}, \mathsf{crs}_{\mathsf{H}})$
  $\mathbb{x}_{\mathsf{upd}} \coloneqq (\mathsf{com}^{\iota-1}_{\mathsf{KTS}}, \mathsf{com}^{\iota}_{\mathsf{KTS}}, W^{\iota-1}, W^\iota)$
  req $\mathsf{NIZK.Vfy}(\mathbb{x}_{\mathsf{upd}}, \pi^\iota_{\mathsf{NIZK}}, \mathsf{crs}_{\mathsf{NIZK}}) = 1$
return 1

$\mathsf{SimSetup}(1^\lambda)$

$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{td}_{\mathsf{VRF}}) \leftarrow \mathsf{VRF.SimSetup}(1^\lambda)$
$\mathsf{crs}_{\mathsf{H}} \leftarrow \mathsf{H.Setup}(1^\lambda)$
$(\mathsf{crs}_{\mathsf{NIZK}}, \mathsf{td}_{\mathsf{NIZK}}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda)$
$\mathsf{crs}_{\mathsf{KTS}} \coloneqq (\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}, \mathsf{crs}_{\mathsf{NIZK}})$
$(\mathsf{vk}', \mathsf{sk}') \leftarrow \mathsf{VRF.Gen}(1^\lambda, \mathsf{crs}_{\mathsf{VRF}})$
$\mathsf{td}_{\mathsf{KTS}} \coloneqq (\mathsf{sk}', \mathsf{td}_{\mathsf{NIZK}})$
return $(\mathsf{crs}_{\mathsf{KTS}}, \mathsf{td}_{\mathsf{KTS}})$

$\mathsf{ExtKeys}(\mathsf{com}^1_{\mathsf{KTS}}, ..., \mathsf{com}^\tau_{\mathsf{KTS}}, \pi^1_{\mathrm{upd}}, ..., \pi^\tau_{\mathrm{upd}}, \mathsf{crs}_{\mathsf{KTS}}, \mathsf{td}_{\mathsf{KTS}})$

$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}) \coloneqq \mathsf{crs}_{\mathsf{KTS}}$
$\mathsf{sk}' \coloneqq \mathsf{td}_{\mathsf{KTS}}$
$\mathsf{com}^0_{\mathsf{KTS}} \coloneqq \bot; W^0 \coloneqq \emptyset$
for $\iota \in [\tau]$
  $(\Delta W^\iota, \pi^\iota_{\mathsf{NIZK}}) \coloneqq \pi^\iota_{\mathrm{upd}}$
  $W^\iota \coloneqq W^{\iota-1} \cup \Delta W^\iota$
$\mathsf{D} \coloneqq [[], ..., []]$
for $(\ell, y) \in W^\tau$
  $\mathsf{id}||\mathsf{v}||\mathsf{k} \coloneqq \mathsf{VRF.Inv}(\mathsf{sk}', y, \mathsf{crs}_{\mathsf{VRF}})$
  $\mathsf{D}[\mathsf{id}][\mathsf{v}] \coloneqq \mathsf{k}$
return D

$\mathsf{SimQryKey}(\mathsf{com}_{\mathsf{KTS}}, \mathsf{id}, \mathsf{v}, \mathsf{k}, i, N, \mathsf{crs}, \mathsf{td}_{\mathsf{KTS}})$

req $i \in [N] \wedge b \in \{0, 1\}$
$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}) \coloneqq \mathsf{crs}_{\mathsf{KTS}}$
$(\mathsf{sk}', \mathsf{td}_{\mathsf{NIZK}}) \coloneqq \mathsf{td}_{\mathsf{KTS}}$
for $\iota \in [N]$
  $(\ell_\iota, \cdot) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}', \iota||0)$
  $(y_\iota, \cdot) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}', \iota||1)$
  $h_\iota \coloneqq \mathsf{H.Eval}(\ell_\iota||y_\iota, \mathsf{crs}_{\mathsf{H}})$
$(\ell_i, \cdot) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}', \mathsf{id}||\mathsf{v})$
$\pi_i \coloneqq \mathsf{VRF.SimEval}(\mathsf{sk}', \mathsf{id}||\mathsf{v}, \ell_i, \mathsf{crs}_{\mathsf{VRF}}, \mathsf{td}_{\mathsf{VRF}})$
$\pi'_i \coloneqq \mathsf{VRF.SimEval}(\mathsf{sk}', \mathsf{id}||\mathsf{v}||\mathsf{k}, y_i, \mathsf{crs}_{\mathsf{VRF}}, \mathsf{td}_{\mathsf{VRF}})$
$W \coloneqq \{(\ell_\iota, y_\iota) \mid \iota \in [N]\}$
// inclusion proof of current version
$\mathsf{cp} \coloneqq \mathsf{PTrie}_{\mathsf{H}}.\mathsf{Copath}_{\mathsf{H}}(W, \ell_i, \mathsf{crs}_{\mathsf{H}})$
// non-inclusion of next version
$(\ell_{\mathsf{parent}}, h_{\mathsf{parent}}, \ell_{\mathsf{left}}, h_{\mathsf{left}}, \ell_{\mathsf{right}}, h_{\mathsf{right}})$
  $\coloneqq \mathsf{PTrie}_{\mathsf{H}}.\mathsf{Delim}_{\mathsf{H}}(W, \ell_i, \mathsf{crs}_{\mathsf{H}})$
$\mathsf{cp}' \coloneqq \mathsf{PTrie}_{\mathsf{H}}.\mathsf{Copath}_{\mathsf{H}}(W, \ell_{\mathsf{parent}}, \mathsf{crs}_{\mathsf{H}})$
return $\pi \coloneqq (\ell_i, \pi_i, y_i, \pi'_i, \mathsf{cp},$
  $\ell_{\mathsf{parent}}, h_{\mathsf{parent}}, \ell_{\mathsf{left}}, h_{\mathsf{left}}, \ell_{\mathsf{right}}, h_{\mathsf{right}}, \mathsf{cp}')$

$\mathsf{SimUpdateEpoch}(\mathsf{com}_{\mathsf{KTS}}, \mathsf{com}'_{\mathsf{KTS}}, N', \mathsf{crs}, \mathsf{td}_{\mathsf{KTS}})$

$(\mathsf{crs}_{\mathsf{VRF}}, \mathsf{crs}_{\mathsf{H}}) \coloneqq \mathsf{crs}_{\mathsf{KTS}}$
$(\mathsf{sk}', \mathsf{td}_{\mathsf{NIZK}}) \coloneqq \mathsf{td}_{\mathsf{KTS}}$
// generate dummy labels and leaf values
for $\iota \in [N']$
  $(\ell_\iota, \cdot) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}', \iota||0)$
  $(y_\iota, \cdot) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}', \iota||1)$
  $W_\iota \coloneqq \{(\ell_{\hat{\iota}}, y_{\hat{\iota}}) \mid \hat{\iota} \in [\iota]\}$
$(\mathsf{vk}, h_\varepsilon) \coloneqq \mathsf{com}_{\mathsf{KTS}}$
// find number of entries in commitment $\mathsf{com}_{\mathsf{KTS}}$
let N s.t. $h_\varepsilon = \mathsf{PTrie}_{\mathsf{H}}.\mathsf{Eval}_{\mathsf{H}}(W_N, \mathsf{crs}_{\mathsf{H}})$
$W' \coloneqq \{(\ell_{\hat{\iota}}, y_{\hat{\iota}}) \mid \hat{\iota} \in [N']\}$
$\mathbb{x}_{\mathsf{upd}} \coloneqq (\mathsf{com}_{\mathsf{KTS}}, \mathsf{com}'_{\mathsf{KTS}}, W_N, W')$
$\pi_{\mathsf{NIZK}} \leftarrow \mathsf{NIZK.SimProve}(\mathbb{x}_{\mathsf{upd}}, \mathsf{crs}_{\mathsf{NIZK}}, \mathsf{td}_{\mathsf{NIZK}})$
$\Delta W \coloneqq W' \setminus W_N$
return $(\Delta W, \pi_{\mathsf{NIZK}})$

Fig. 5: Our KT scheme instantiation inspired by the append-only zero-knowledge set from [CDG+19]. We denote the canonical Patricia trie by $\mathsf{PTrie}_{\mathsf{H}} = (\mathsf{Eval}_{\mathsf{H}}, \mathsf{Copath}_{\mathsf{H}}, \mathsf{Vfy}_{\mathsf{H}}, \mathsf{Delim}_{\mathsf{H}})$ using the hash function H.

Privacy. The privacy of Construction 1 simply follows from the privacy of the sVRF. In VRF simulation mode, the leaf labels $\ell_{\mathsf{id},\mathsf{v}}$ and values $y_{\mathsf{id},\mathsf{v}}$ are essentially independent of the actual id–version–key triples. Because those ℓ, y values are pseudorandom they don't reveal any information (other than the total size of the database) about the database.

Mode indistinguishability. The mode indistinguishability of Construction 1 follows direclty from the mode indistinguishability of the NIZK and the sVRF.

Simulation indistinguishability. The simulation indistinguishability of Construction 1 follows from the (trapdoor-)indistinguishability of the sVRF. In the case of perfect indistinguishability the distribution of honestly generated query responses and proofs is simply identical because honestly generated VRF proofs and simulated VRF proofs are perfectly indistinguishable. □

C.2 KT Protocol

In this section we present our KT protocol formally and prove that is securely UC-realizes out KT functionality $\mathcal{F}_{\mathsf{KT}}$ in the $\mathcal{F}_{\mathsf{crs}}, \mathcal{F}_{\mathsf{BC}}$-hybrid model.

Theorem 1. *Let* KTS *be a KT scheme (as defined in Definition 5). Then the protocol* π_{KT} *securely UC-realizes the functionality* $\mathcal{F}_{\mathsf{KT}}$ *in the* $\{\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{crs}}\}$*-hybrid model.*

Proof. Figure 6 shows the simulator for our protocol. As usual we prove UC security via a sequence of hybrid games. The first one is the "real" execution of the protocol in the $\mathcal{F}_{\mathsf{crs}}$-hybrid model, and the last one is the "ideal" execution of the ideal functionality with dummy parties. By showing that each pair of subsequent games are computationally indistinguishable, we can conclude that the real and the ideal execution are computationally indistinguishable, and thus the protocol is UC secure. For the reader's convenience we depict the hybrids in Figure 7.

Game 0 **(Real exection):** This is the real execution of the protocol. Here, parties obtain inputs from (and make outputs to) the environment and execute the protocol code for sending messages to each other (or when receiving messages from other parties). In particular, the hybrid functionality $\mathcal{F}_{\mathsf{crs}}$ samples the CRS honestly in the extraction mode $(\mathsf{crs}, \mathsf{td}_{\mathrm{ext}}) \leftarrow \mathsf{KTS.ExtSetup}(1^\lambda)$. Note that (as usually in UC security) the environment does not get to interact with the hybrid functionality directly.

Game 1 **(Switch CRS to simulation mode):** In this hybrid, if the server SP is honest, the CRS functionality $\mathcal{F}_{\mathsf{crs}}$ outputs a CRS in simulation mode $(\mathsf{crs}, \mathsf{td}_{\mathrm{sim}}) \leftarrow \mathsf{KTS.SimSetup}(1^\lambda)$ instead of the extraction mode. This step is justified by the mode indistinguishability property of the KT scheme KTS.

Game 2 **(Simulate query responses and epoch update proofs):** In this hybrid, the honest protocol server SP computes the query responses with $\mathsf{KTS.SimQryKey}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \mathsf{crs}, \mathsf{td}_{\mathrm{sim}})$ instead of $\mathsf{KTS.QryKey}(\mathsf{st}, \mathsf{D}, \mathsf{id}, \mathsf{v}, \mathsf{crs})$. Also, the server generates the epoch update proofs as $\pi_{\mathrm{upd}} \coloneqq \mathsf{KTS.SimUpdateEpoch}(\mathsf{com}, \mathsf{com}'$ instead of $(\mathsf{D}', \pi_{\mathrm{upd}}) \coloneqq \mathsf{KTS.UpdateEpoch}(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{crs})$ where $\mathsf{com}' \coloneqq \mathsf{KTS.Commit}(\mathsf{st}, \mathsf{D}', \mathsf{crs})$. This step is justified by the perfect simulation indistinguishability property of the KT scheme KTS.

Game 3 **(Simulate database):** Let D be some database with a total number of $N \coloneqq \sum_{\mathsf{id}} |\mathsf{D}[\mathsf{id}]|$ entries. In this hybrid, whenever the honest server would commit to a database D via $\mathsf{KTS.Commit}(\mathsf{SP.st}, \mathsf{D}, \mathsf{SP.crs})$, the server instead commits to the dummy database D' where the first entity has N dummy entries $\mathsf{D}'[1] = [0^\lambda, ..., 0^\lambda] \in (\{0,1\}^\lambda)^N$ and all other entries are empty $\mathsf{D}[\mathsf{id}] = []$ for all $\mathsf{id} \in \{2, ..., n\}$. This step is justified by the privacy of the KT scheme KTS.

Game 4 **(Introduce** $\mathcal{F}_{\mathsf{KT}}$**):** In this hybrid, we introduce the ideal functionality $\mathcal{F}_{\mathsf{KT}}$. As before the environment interacts directly with the honest (protocol) parties. Additionally, messages from the environment to honest parties are duplicated (forked off) to dummy parties that forward them to the ideal functionality $\mathcal{F}_{\mathsf{KT}}$. However, in contrast to the actual ideal world these (intermediate) dummy parties

$\mathcal{S}(1^\lambda)$

$\mathcal{S}.\tau \coloneqq 0$
if $\mathsf{SP} \in \mathcal{C}$
  $(\mathsf{SP.crs}, \mathsf{SP.td}_{\text{ext}}) \leftarrow \mathsf{KTS.ExtSetup}(1^\lambda)$
  $\mathcal{S}.\mathsf{L}_\tau \coloneqq \{\}$
else
  $(\mathsf{SP.crs}, \mathsf{SP.td}_{\text{sim}}) \leftarrow \mathsf{KTS.SimSetup}(1^\lambda)$
$\mathcal{S}.\mathsf{st} \leftarrow \mathsf{KTS.Init}(1^\lambda, \mathsf{SP.crs})$
$\mathcal{S}.\mathsf{com}^0_{\mathsf{KTS}} \coloneqq \bot$
$\mathcal{S}.\mathsf{D}^0 \coloneqq [[], ...[]]$

$\mathcal{S}(\textbf{RegisterKey}, \mathsf{id}, \mathsf{k} \leftarrow \mathcal{F}_{\mathsf{KT}})$

$\mathcal{S}.\mathcal{I}[\mathsf{id}] \coloneqq \mathsf{k}$

$\mathcal{S}(\textbf{QueryKey}, \mathsf{id}, \mathsf{id}' \leftarrow \mathcal{F}_{\mathsf{KT}})$

$\tau \coloneqq \mathcal{S}.\tau$
$\mathcal{S}.\mathcal{Q}_\tau[\mathsf{id}, \mathsf{id}'] \coloneqq 1$

$\mathcal{S}(\textbf{Update}, N \leftarrow \mathcal{F}_{\mathsf{KT}})$

$\mathcal{S}.N_\tau \coloneqq N$

$\mathcal{S}(\textbf{Update} \leftarrow \mathcal{F}_{\mathsf{KT}})$

$\tau \coloneqq \mathcal{S}.\tau$
send $(\textbf{Update}, \mathcal{S}.\mathsf{L}_\tau) \rightarrow \mathcal{F}_{\mathsf{KT}}$

$\mathcal{S}.\mathcal{F}_{\mathsf{crs}}(1^\lambda)$

for $\mathsf{id} \in [1, ..., n]$ **do** $\mathcal{S}.\mathsf{crs} \rightarrow \mathsf{U}_{\mathsf{id}}$
$\mathcal{S}.\mathsf{crs} \rightarrow \mathsf{SP}$

$\Delta(\mathsf{D}^{\tau-1}, \mathsf{D}^\tau)$

for $\mathsf{id} \in [1, ..., n]$
  $\mathsf{v}_\tau \coloneqq |\mathsf{D}^\tau[\mathsf{id}]|$
  // check if the key under id id has a new version
  if $\mathsf{v}_\tau = \left|\mathsf{D}^{\tau-1}[\mathsf{id}]\right| + 1$
    // add that new key to the list of new keys
    $\mathsf{L}_\Delta[\mathsf{id}] \coloneqq \mathsf{D}^\tau[\mathsf{id}][\mathsf{v}]$
return L_Δ

$\mathcal{S}.\mathsf{U}_{\mathsf{id}}(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k}, \pi \leftarrow \mathsf{SP})$

// validate the query response
req $\mathsf{k} \neq \bot \vee \mathsf{v} = 0$
req $\mathsf{KTS.VfyQry}(\mathcal{S}.\mathsf{com}^\tau_{\mathsf{KTS}}, \mathsf{id}', \mathsf{v}, \mathsf{k}, \pi, \mathcal{S}.\mathsf{crs}) = 1$
if $\mathcal{S}.\mathcal{Q}_\tau[\mathsf{id}, \mathsf{id}'] = 1$ // validate if query issued in the epoch
  send $(\textbf{AllowQueryKey}, \tau, \mathsf{id}, \mathsf{id}') \rightarrow \mathcal{F}_{\mathsf{KT}}$ **as** $\mathcal{S}$

$\mathcal{S}.\mathcal{F}_{\mathsf{BC}}(\textbf{Update}, \mathsf{com}_{\mathsf{KTS}}, \pi_{\text{upd}} \leftarrow \mathsf{SP})$

// validate the epoch update
$\tau \coloneqq \mathcal{S}.\tau$
req $\mathsf{KTS.VerifyUpdate}(\mathcal{S}.\mathsf{com}^{\tau-1}_{\mathsf{KTS}}, \mathsf{com}_{\mathsf{KTS}}, \pi^{\mathsf{upd}}_{\mathsf{KTS}}, \mathcal{S}.\mathsf{crs}) = 1$
$\mathcal{S}.\tau \coloneqq \tau + 1; \mathcal{S}.\mathsf{com}^\tau_{\mathsf{KTS}} \coloneqq \mathsf{com}_{\mathsf{KTS}}; \mathcal{S}.\pi^i_{\text{upd}} \coloneqq \pi_{\text{upd}}; \mathcal{S}.i \coloneqq 0$
if $\mathsf{SP} \in \mathcal{C}$
  $\mathsf{L}_{\mathsf{SP}} \coloneqq \Delta(\mathcal{S}.\mathsf{D}^{\tau-1}, \mathcal{S}.\mathsf{D}^\tau)$
  // trigger new epoch via $\mathcal{F}_{\mathsf{KT}}$
  send $(\textbf{Update}, \mathsf{L}_{\mathsf{SP}}) \rightarrow \mathcal{F}_{\mathsf{KT}}$ **as** $\mathcal{S}$
else
  $\mathcal{S}.\mathsf{D}^\tau \coloneqq \mathsf{KTS.ExtKeys}(\mathcal{S}.\mathsf{com}^1_{\mathsf{KTS}}, ..., \mathcal{S}.\mathsf{com}^\tau_{\mathsf{KTS}},$
    $\mathcal{S}.\pi^1_{\text{upd}}, ..., \mathcal{S}.\pi^\tau_{\text{upd}}, \mathcal{S}.\mathsf{crs}, \mathcal{S}.\mathsf{td}_{\text{ext}})$

$\mathcal{S}.\mathsf{SP}(\textbf{RegisterKey}, \mathsf{id}, \mathsf{k} \leftarrow \mathsf{U}_{\mathsf{id}})$

req $\mathsf{k} \neq \bot$
$\tau \coloneqq \mathcal{S}.\tau$
$\mathcal{S}.\mathsf{D}^\tau[\mathsf{id}] \coloneqq \mathsf{k}$
send $(\textbf{RegisterKey}, \mathsf{id}, \mathsf{k}) \rightarrow \mathcal{F}_{\mathsf{KT}}$ **as** U_{id}

$\mathcal{S}.\mathsf{SP}(\textbf{QueryKey}, \mathsf{id}' \leftarrow \mathsf{U}_{\mathsf{id}})$

$N_\tau \coloneqq \mathcal{S}.N_\tau$
send $(\textbf{QueryKey}, \mathsf{id}') \rightarrow \mathcal{F}_{\mathsf{KT}}$ **as** U_{id}
receive $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k}) \leftarrow \mathcal{F}_{\mathsf{KT}}$ **as** SP
if $\mathsf{k} \neq \bot$
  $b \coloneqq 0$ // trigger inclusion proof
  $\mathcal{S}.i \coloneqq \mathcal{S}.i + 1$ // increment (inclusion) query counter
else $b \coloneqq 1$ // trigger non-inclusion proof
$\pi \coloneqq \mathsf{KTS.SimQryKey}(\mathcal{S}.\mathsf{com}^\tau_{\mathsf{KTS}}, \mathsf{id}', \mathsf{v}, \mathsf{k}, i, N_\tau, b, \mathcal{S}.\mathsf{crs}, \mathcal{S}.\mathsf{td}_{\text{sim}})$
send $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k}, \pi) \rightarrow \mathsf{U}_{\mathsf{id}}$

$\mathcal{S}.\mathsf{SP}(\textbf{Update}, \tau \leftarrow \mathcal{F}_{\mathsf{KT}})$

$N_\tau \coloneqq \mathcal{S}.N_\tau$
$\mathcal{S}.\mathsf{D}^\tau \coloneqq [[0^\lambda, ..., 0^\lambda], ...[]]$ // $\mathcal{S}.\mathsf{D}^\tau[1] \in \{\,0^\lambda\,\}^{N_\tau}$
$\mathcal{S}.\mathsf{com}^\tau_{\mathsf{KTS}} \coloneqq \mathsf{KTS.Commit}(\mathcal{S}.\mathsf{st}, \mathcal{S}.\mathsf{D}^\tau, \mathcal{S}.\mathsf{crs})$
$\pi_{\text{upd}} \coloneqq \mathsf{KTS.SimUpdateEpoch}(\mathcal{S}.\mathsf{com}^{\tau-1}_{\mathsf{KTS}}, \mathcal{S}.\mathsf{com}^\tau_{\mathsf{KTS}}, N_\tau, \mathcal{S}.\mathsf{crs}, \mathcal{S}.\mathsf{td}_{\text{sim}})$
send $(\textbf{Update}, \mathcal{S}.\mathsf{com}^\tau_{\mathsf{KTS}}, \pi^{\mathsf{upd}}_{\mathsf{KTS}}) \rightarrow \mathcal{S}.\mathcal{F}_{\mathsf{BC}}$

Fig. 6: $\mathcal{S}_{\mathsf{KT}}$ for our KT protocol π_{KTS}. By $\mathcal{S}.\mathcal{F}_{\mathsf{BC}}$ (resp. $\mathcal{S}.\mathcal{F}_{\mathsf{crs}}$) we denote the simulated $\mathcal{F}_{\mathsf{BC}}$ (resp. $\mathcal{F}_{\mathsf{crs}}$) functionality. By $\mathcal{S}.\mathsf{U}_{\mathsf{id}}$ we denote the simulated (honest) user U_{id} and by $\mathcal{S}.\mathsf{SP}$ the simulated (honest) server SP. Note that the code of $\mathcal{S}.\mathsf{SP}$ is only executed if the server is honest.

do not forward any messages that they receive *from* $\mathcal{F}_{\mathsf{KT}}$ back to the environment, those message are simply dropped. Together with $\mathcal{F}_{\mathsf{KT}}$ we introduce our simulator $\mathcal{S}_{\mathsf{KT}}$ (defined in Figure 6). Messages from the environment to the adversary are also duplicated to the simulator, but—as with dummy parties—messages that the simulator receives from $\mathcal{F}_{\mathsf{KT}}$ not forwarded back to the environment. Because neither the dummy parties nor the simulator actually output anything to the environment we don't change the output distribution at all; the change is merely syntactical.

Game 5 **(Abort):** In this hybrid, we abort the game if the outputs of the honest (modified) protocol parties does not match the (so far dropped) outputs of the ideal functionality to the dummy parties, or if the adversary's output does not match the (so far dropped) simulator's output. We prove in Claim 2 that the abort only happens with negligible probability.

Game 6 **(Ideal execution):** In this hybrid, the environment interacts with (honest) dummy parties that relay all messages to and from the ideal functionality $\mathcal{F}_{\mathsf{KT}}$. Conditioned on the non-abort event, this hybrid game has exactly the same output distribution as the previous one by definition.

Claim 1. *If the KT scheme* KTS *is* $\epsilon_{\text{priv}}(\lambda)$*-private, then any efficient PPT distinguisher between Game 2 and Game 3 (in particular the environment) as advantage at most* $\text{poly}(\lambda) \cdot \epsilon_{\text{priv}}(\lambda)$.

Proof. If the server is corrupted, then Game 2 and Game 3 are identical. Henceforth, assume that the server is honest and that the CRS is set up in simulation mode.

We can give a straightforward reduction $\mathcal{R}$ from the privacy of the KT scheme KTS to the indistinguishability of the two games. Let $\widehat{\tau} \leq \text{poly}(\lambda)$ be an upper bound on the number of epochs that the environment $\mathcal{Z}$ issues. We proceed by at most $\widehat{\tau}+1$ hybrids, where in hybrid $j \in \{0, ..., \widehat{\tau}\}$ the first j epoch commitments are dummy commitments, the rest are computed as in Game 2. For each intermediate game j we define a reduction $\mathcal{R}_j$ as follows: it plays (as the adversary) the privacy game with the KT scheme KTS where it first obtains the (simulation mode) CRS. Then it runs the environment $\mathcal{Z}$ and simulates the first j epochs with dummy commitments. Whenever the reduction need to generate query responses or epoch updates, it uses its $\mathsf{KTS.SimQryKey}$ and $\mathsf{KTS.SimUpdateEpoch}$ oracles. Then it obtains the j-th database D and generates the dummy commitment D' with the same number of entries N as D. Then it gives the two databases $\mathsf{D}^0 := \mathsf{D}$ and $\mathsf{D}^1 = \mathsf{D}'$ to the privacy challenger who responds with a commitment $\mathsf{com}_{\mathsf{KTS}}$ to the database D or D'. The reduction $\mathcal{R}_j$ finishes the protocol with the environment $\mathcal{Z}$, and the environment finally outputs a bit b' (its guess whether it is in the real or ideal world). Finally, the reduction $\mathcal{R}$ outputs b'. Now, note that if the privacy challenger chooses its bit $b = 0$, then the reduction perfectly simulates intermediate game j to the environment $\mathcal{Z}$. If the privacy challenger chooses its bit $b = 1$, then the reduction perfectly simulates intermediate game $j + 1$ to the environment $\mathcal{Z}$. Hence, the probability that the environment $\mathcal{Z}$ outputs 0 can change at most by $\epsilon_{\text{priv}}(\lambda)$ between Game 2 and Game 3, i.e.,

$$\epsilon_{\text{priv}}(\lambda) \geq |2\Pr[\mathcal{Z} = b] - 1| = |\Pr[\mathcal{Z} = 0 \mid b = 0] + \Pr[\mathcal{Z} = 1 \mid b = 1] - 1| \tag{17}$$

$$= |\Pr[\mathcal{Z} = 0 \mid b = 0] - \Pr[\mathcal{Z} = 0 \mid b = 1]| = |\Pr_{\text{Game } j}[\mathcal{Z} = 0] - \Pr_{\text{Game } j+1}[\mathcal{Z} = 0]| \,. \tag{18}$$

■

Claim 2. *If the KT scheme* KTS *is* $\epsilon_{\text{inter}}(\lambda)$*-inter-epoch-consistent and* $\epsilon_{\text{intra}}(\lambda)$*-intra-epoch-consistent, then the abort probability in Game 5 is at most* $\epsilon_{\text{inter}}(\lambda) + \epsilon_{\text{intra}}(\lambda)$.

Proof. If the server is honest, then the abort will never happen because $\mathcal{F}_{\mathsf{KT}}$ will not consult the simulator for the list of adversarial key updates $\mathsf{L_{SP}}$. Moreover, whenever an honest protocol party has output $(\textbf{Update}, \tau)$ the honest protocol server must have send a new epoch commitment (plus proof) via $\mathcal{F}_{\mathsf{BC}}$ upon receiving $(\textbf{Update})$ from the environment. In this case, the (duplicate) dummy server forwards that $(\textbf{Update})$ message from the environment to the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ which then triggers a message $(\textbf{Update}, \tau)$ to all dummy users. Henceforth, assume that the server is corrupted.

C.3 Intuition.

To induce differences between the outputs of the honest protocol parties and the dummy parties the environment $\mathcal{Z}$ has to make the (corrupted) server behave in a way that cannot be mimiked by the simulator. To this end the environment has two options:

- The environment can break inter-epoch consistency by broadcasting two (e.g. consecutive) epoch commitments $\mathsf{com}^1_{\mathsf{KTS}}, \mathsf{com}^2_{\mathsf{KTS}}$ (plus a valid update proof), such that the databases $\mathsf{D}^1, \mathsf{D}^2$ (contained in those commitments) violate temporal consistency. An example could be that in the second database an entry has been deleted, i.e., $|\mathsf{D}^2[\mathsf{id}]| = |\mathsf{D}^1[\mathsf{id}]| - 1$ for some user id id. The environment can let a third party $\mathsf{U}_{\mathsf{id}'}$ issue a query for the user U_{id} relative to each epoch commitment, receiving two key–version pairs $(\mathsf{k}_1, \mathsf{v}_1)$ and $(\mathsf{k}_2, \mathsf{v}_2)$ respectively. In contrast, the simulator cannot enforce this behavior because the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ (by definition) will never output a version $\mathsf{v}_2 \lneq \mathsf{v}_1$—no matter what the simulator inputs into $\mathcal{F}_{\mathsf{KT}}$.
- The environment can break intra-epoch consistency by letting the (corrupted) server sent a valid query response $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{k}, \mathsf{v}, \pi)$ to an honest protocol party U_{id} such that the triple of $(\mathsf{id}', \mathsf{v}, \mathsf{k})$ is inconsistent ($\mathsf{D}[\mathsf{id}'][\mathsf{v}] \neq \mathsf{k}$) with the database D stored internally in $\mathcal{F}_{\mathsf{KT}}$. In that case, the honest protocol party U_{id} would output $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{k}, \mathsf{v})$ to the environment. However, the simulator cannot force the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ to output $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{k}, \mathsf{v})$ to the dummy party U_{id} because the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ will only ever output $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{k}', \mathsf{v}')$ where $\mathsf{v}' \coloneqq |\mathsf{D}[\mathsf{id}']|$ and $\mathsf{k}' \coloneqq \mathsf{D}[\mathsf{id}'][\mathsf{v}']$.

C.4 Formal analysis.

We will show setp-by-step that real and ideal messages in Game 5 are consistent. Let us denote by $y^{\text{real}}_{\mathsf{id},j}$ the j-th output of (honest) party U_{id} in Game 5 and let $y^{\text{ideal}}_{\mathsf{id},j}$ denote the j-th output of $\mathcal{F}_{\mathsf{KT}}$ to the dummy party U_{id} in Game 5.

C.5 Epoch update outputs.

First, we note that iff $y^{\text{real}}_{\mathsf{id},j} = (\textbf{Update}, \tau)$, then $y^{\text{ideal}}_{\mathsf{id},j} = (\textbf{Update}, \tau)$ as well. By the definition of our protocol an honest protocol party U_{id} only outputs $(\textbf{Update}, \tau)$ if it received via the $\mathcal{F}_{\mathsf{BC}}$ broadcast a (valid) $(\textbf{Update}, \mathsf{com}_{\mathsf{KTS}}, \pi^{\mathsf{upd}}_{\mathsf{KTS}})$ from the server SP. We distinguish two cases:

- If the server is corrupted, then the corrupted server must have broadcasted a valid $(\textbf{Update}, \mathsf{com}_{\mathsf{KTS}}, \pi^{\mathsf{upd}}_{\mathsf{KTS}})$ via the (simulated) $\mathcal{F}_{\mathsf{BC}}$. In this case, (by definition) the simulator inputs **Update** into the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ (in the name of the server). Then $\mathcal{F}_{\mathsf{KT}}$ sends **Update** to $\mathcal{S}$ who (by definition) responds with $(\textbf{Update}, \mathsf{L}_{\mathsf{SP}})$ for some list of adversarial updates L_{SP}. In turn, $\mathcal{F}_{\mathsf{KT}}$ sends $(\textbf{Update}, \tau)$ to the honest party U_{id} which outputs $(\textbf{Update}, \tau)$.
- If the server is not corrupted, then it must have received the input **Update** from the environment. In this case, the input **Update** is directly forwarded to the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ via the (honest) dummy server. Consequently, the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ sends $(\textbf{Update}, \tau)$ to each honest party U_{id} which outputs $(\textbf{Update}, \tau)$.

The opposite direction also holds; if an honest party outputs $y^{\text{ideal}}_{\mathsf{id},j} = (\textbf{Update}, \tau)$ in Game 5, then it outputs $y^{\text{real}}_{\mathsf{id},j} = (\textbf{Update}, \tau)$ in Game 4 as well.

- If the server is corrupted, then the simulator must have input **Update** into the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ (in the name of the server). The simulator (by definition) only does this if the corrupted server broadcasted a valid $(\textbf{Update}, \mathsf{com}_{\mathsf{KTS}}, \pi^{\mathsf{upd}}_{\mathsf{KTS}})$ via the (simulated) $\mathcal{F}_{\mathsf{BC}}$. In this case, the honest (protocol) party U_{id} also outputs $(\textbf{Update}, \tau)$.
- If the server is not corrupted, then the ideal functionality $\mathcal{F}_{\mathsf{KT}}$ must have gotten input **Update** from from the dummy server. Then the (honest) protocol server would broadcast a valid $(\textbf{Update}, \mathsf{com}_{\mathsf{KTS}}, \pi^{\mathsf{upd}}_{\mathsf{KTS}})$ and honest party U_{id} would output $(\textbf{Update}, \tau)$ as well.

■

C.6 Query response outputs.

The only other output that honest parties generate is of the form $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k})$. We observe that whenever $y^{\text{real}}_{\mathsf{id},j} = (\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k})$, then $y^{\text{ideal}}_{\mathsf{id},j} = (\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}', \mathsf{k}')$ for some $(\mathsf{v}', \mathsf{k}')$ (which might be (v, k) or not).
Suppose an honest protocol party U_{id} outputs $(\textbf{QueryKeyResponse}, \mathsf{id}', \mathsf{v}, \mathsf{k})$, then it must have received input $(\textbf{QueryKey}, \mathsf{id}')$ from the environment beforehand. Hence, the corresponding dummy party U_{id} inputs $(\textbf{QueryKey}, \mathsf{id}')$ into the functionality $\mathcal{F}_{\mathsf{KT}}$. Moreover, the protocol party U_{id} must have received a valid $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k}, \pi)$ from SP. In case $\mathsf{SP} \in \mathcal{C}$, (by definition) the simulator inputs $(\textbf{AllowQueryKey}, \tau, \mathsf{id}, \mathsf{id}')$ into $\mathcal{F}_{\mathsf{KT}}$. Because $\mathcal{F}_{\mathsf{KT}}$ received $(\textbf{QueryKey}, \mathsf{id}')$ from U_{id} before, $\mathcal{F}_{\mathsf{KT}}$ sends $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}', \mathsf{k}')$ to the dummy party U_{id} which forwards it to the environment. (Here k' and v' are the unique key and version stored in the database of $\mathcal{F}_{\mathsf{KT}}$.)
Analogously, we observe that whenever $y^{\text{ideal}}_{\mathsf{id},j} = (\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}', \mathsf{k}')$, then $y^{\text{real}}_{\mathsf{id},j} = (\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k})$ for some (v, k) (which might be $(\mathsf{v}', \mathsf{k}')$ or not). Suppose a dummy party U_{id} outputs $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k})$, then it must have forwarded $(\textbf{QueryKey}, \mathsf{id}')$ to $\mathcal{F}_{\mathsf{KT}}$ from the environment beforehand. Hence, the corresponding protocol party U_{id} on input $(\textbf{QueryKey}, \mathsf{id}')$ sent $(\textbf{QueryKey}, \mathsf{id}')$ to the server SP. If $\mathsf{SP} \notin \mathcal{C}$ is honest, then the protocol server must have sent a valid $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}', \mathsf{k}', \pi)$ to the protocol party U_{id}. If $\mathsf{SP} \in \mathcal{C}$ is corrupted, the simulator must have $(\textbf{AllowQueryKey}, \tau, \mathsf{id}, \mathsf{id}')$ into $\mathcal{F}_{\mathsf{KT}}$. This only happens (by definition) only if the corrupted protocol server sent a valid $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}', \mathsf{k}', \pi)$ to the protocol party U_{id}. That is, in both cases the honest protocol party U_{id} must have received a valid $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}', \mathsf{k}', \pi)$ from the server SP which it outputs to the environment.

Now, we know that real and ideal query responses always occur in pairs where (v, k) denotes the real version and key, and $(\mathsf{v}', \mathsf{k}')$ denotes the ideal version and key. It remains to show that which overwhelming probability the two pairs are equal. We do so by a reduction to the inter-epoch and intra-epoch consistency of the KT scheme KTS. Recall that if the server is honest, then $y^{\text{real}}_{\mathsf{id},j} = y^{\text{ideal}}_{\mathsf{id},j}$ for all j, simply because the honest protocol server sends the same version and key to protocol party U_{id} as $\mathcal{F}_{\mathsf{KT}}$ sends to the dummy party U_{id}.

Let $\mathsf{D}^0 = [[], ..., []], \mathsf{D}^1, ..., \mathsf{D}^\tau$ be the databases extracted by the simulator (recall that the server is corrupted). First, we want to show that the databases extracted by the simulator match the databases stored in $\mathcal{F}_{\mathsf{KT}}$. This ensures that the databases extracted by the simulator match the databases stored in the ideal functionality. Suppose the environment (corrupting the server) makes the malicious server broadcast epoch commitments $\mathsf{com}^1_{\mathsf{KTS}}, ..., \mathsf{com}^\tau_{\mathsf{KTS}}$ (plus valid update proofs $\pi^1_{\text{upd}}, ..., \pi^\tau_{\text{upd}}$) s.t. $\mathsf{D}^{\tau-1} \not\leq \mathsf{D}^\tau$ where τ is the first epoch where this happens. This means the environment created (valid) epoch commitments that contain non-well-ordered databases. An environment that causes this event breaks the inter-epoch consistency of the KT scheme KTS. Thus, any polynomially time-bounded environment $\mathcal{Z}$ can cause this event with probability at most $\epsilon_{\text{inter}}(\lambda)$. Henceforth, we can assume that all extracted databases D^i are well-ordered. This ensures that the databases extracted by the simulator match the databases stored in $\mathcal{F}_{\mathsf{KT}}$.

Next, we what to show that the query responses given by the corrupted server match the ones given by the ideal functionality. Now, suppose that the environment (corrupting the server) makes the malicious server broadcast epoch commitments $\mathsf{com}^1_{\mathsf{KTS}}, ..., \mathsf{com}^\tau_{\mathsf{KTS}}$ (plus valid update proofs $\pi^1_{\text{upd}}, ..., \pi^\tau_{\text{upd}}$), and send a valid $(\textbf{QueryKeyResponse}, \tau, \mathsf{id}', \mathsf{v}, \mathsf{k}, \pi)$ to an honest protocol party U_{id} s.t. $\mathsf{D}^\tau[\mathsf{id}'][\mathsf{v}] \neq \mathsf{k}$. This means that the enviroment created (valid) epoch commitments and a (valid) query response that is inconsistent with the most current database. An environment that causes this event breaks the intra-epoch consistency of the KT scheme KTS. Thus, any polynomially time-bounded environment $\mathcal{Z}$ can cause this event with probability at most $\epsilon_{\text{intra}}(\lambda)$. Hence, we can assume that all query responses by the corrupted protocol server match the ones given $\mathcal{F}_{\mathsf{KT}}$.

C.7 Conclusion.

Let $\mathcal{Z}$ be a PPT environment. Overall, the environment $\mathcal{Z}$'s advantage to distinguish between the real and the ideal execution is bounded by

$$|\Pr_{\mathbf{Game}\ 0}[\mathcal{Z} \to 1] - \Pr_{\mathbf{Game}\ 6}[\mathcal{Z} \to 1]| \leq \epsilon_{\text{mode}}(\lambda) + \text{poly}(\lambda)\epsilon_{\text{priv}}(\lambda) + \epsilon_{\text{inter}}(\lambda) + \epsilon_{\text{intra}}(\lambda)\ . \tag{19}$$

□

D Comparison with SEEMless [CDG+19]

To compare our work with [CDG+19], we have to cast their contributions into our framework of a functionality, protocol and scheme.

First, we observe that Chase, Deshpande, Ghosh, and Malvai are the first to "formalize the security and privacy requirements of a verifiable key directory service, [...] in terms of a new primitive that we call Verifiable Key Directories (VKD)." We see the notion of a VKD (as a scheme) as a step in the right direction towards a proper formalization of KT systems. Nevertheless, Chase, Deshpande, Ghosh, and Malvai conflate the concept of a scheme (in the sense of a tuple of algorithms) and the concept of a protocol (in the sense of code for parties interactiving with each other). We substantiate this view by the observing that Chase, Deshpande, Ghosh, and Malvai state:

- "Definition 1. A VKD consists of three types of parties: an identity provider or server, clients or users and external auditors", which indicates that their VKD is a protocol, and
- "A Verifiable Key Directory is comprised of the algorithms [...]", which formally defines VKD as a scheme.

D.1 Our KT Scheme vs. VKD

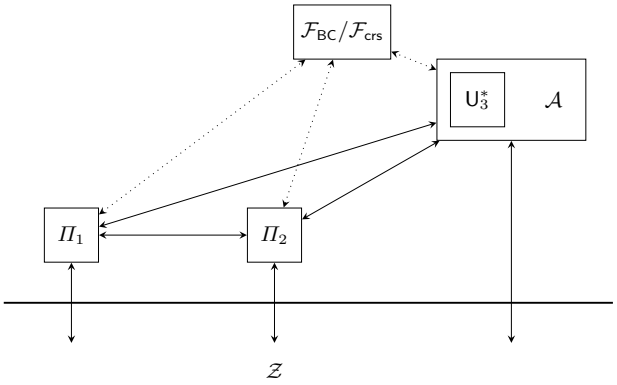
Given the formal definition of a VKD we compare the VKD to our KT scheme. The main difference is that our KT scheme makes many aspects explicit that are left unspecified in the VKD. For example, [CDG+19] explicitly omit any specification of "system parameters". Moreover, for their proof of VKD privacy Chase, Deshpande, Ghosh, and Malvai need a simulator that is only mentioned in the privacy proof in Appendix B of [CDG+19]. Our KT scheme makes this need for a simulator explicit by requiring algotithms SimSetup, SimQryKey, and SimUpdateEpoch for the so-called simulation mode. A conceptually novelty of our KT scheme is that we not only require the simulation mode but we observe that to prove UC security of our protocol we need our KT scheme to also be *extractable*—hence our KT scheme is dual-mode: simulation mode and extraction mode.[18] We remark that the extraction property is not necessary in [CDG+19] because they do not consider UC security.

D.2 Protocols

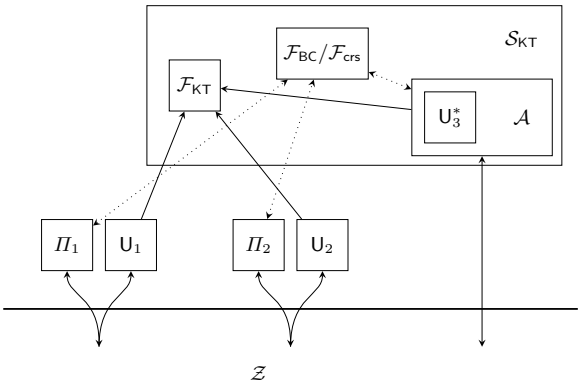
First, we observe that in [CDG+19] the protocol participants consist of a server, users and auditors. In contrast, we do not consider auditors for reasons outlined below. We claim that the protocol description in [CDG+19] lacks many (necessary) details; just two examples are:

- "The client checks each membership or non-membership proof, and the hash chain. Also check that version α as part of proof is less than current epoch t". However, it is not specified what the client should do if the verification check succeeds or fails.

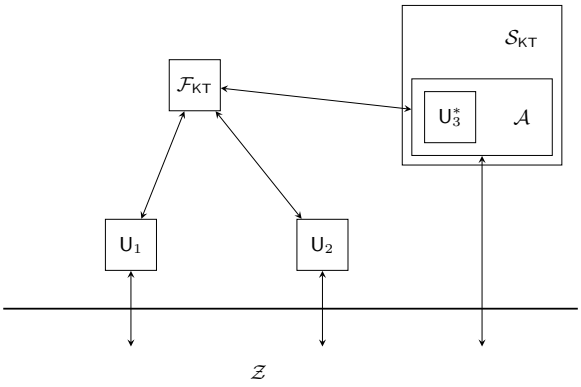
[18] The dual-mode requirement is a direct consequence of the UC commitment problem [CF01] with requires a commitment to be both extractable as well as equivocable (i.e. simulatable).



(a) Game 0 through 3.



(b) Game 4 and 5.



(c) Game 6.

Fig. 7: Graphical representation of our UC proof with three examplary parties where one party U_3 is corrupted. Honest protocol parties are denoted by Π_i, dummy parties are denoted by U_i, and corrupted parties are denoted by U_i^*.

- "Auditors will audit the commitments and proofs to make sure that no entries ever get deleted in either aZKS. They do so by verifying the update proofs Π^{Upd} output by the server. They also check that at each epoch both aZKS commitments are added to the hash chain." However, it is not specified what the auditors should do if the verification check succeeds or fails.

We argue that this lack of detail inhibits the adoption of KT protocols in practice. Worse even, it opens the door for security vulnerabilities due to mismatches between the theoretical protocol design and the actual implementation.
In contrast, our protocol in Figure 3 is fully specified in terms of the input/output behavior of the parties.

D.3 Security Guarantees in [CDG+19] vs. UC security

Our usage of the UC framework to prove security of our protocol means that the type of security guarantee is well understood from a theoretical standpoint. Moreover, we argue that our security notion is also more intuitive that the ones in [CDG+19]. The reason is that if one asks whether a particular scenario would be possible in the real world, then one only has to replay the scenario in the ideal world to see if it is possible to come up with an environmental strategy that leads to the scenario in question. If that is possible, then the scenario is secure, *by definition* of $\mathcal{F}_{\mathsf{KT}}$, and thus is allowed to happen in the real world as well.

In contrast, in [CDG+19] the security guarantees *on the protocol level* are essentially implications of the type: if a party P does A, then party P' cannot do B. Let us consider the soundness notion in [CDG+19]: "VKD soundness guarantees that if Alice has verified the update history of her key [...], whenever Bob queried [...] he would have received Alice's key value that is consistent with the key value reported in Alice's history query." In simpler terms, the guarantee is that if Alice verifies her own key history, she can be sure that any other party will only ever be served her keys. Notice here, that the guarantee is of the form: if party P does A, then party P' cannot do B; here P is Alice, A is verifying her key history, P' is the server, and B is serving a third party a key that is inconsistent with Alice's key history. The obvious problem with this type of guarantee is that it gives no security if the condition is violated

A similar issue holds for the privacy guarantee in [CDG+19]. Namely, a third party may learn the entire key history of every user, if he simply queries each user's key in each epoch, although [CDG+19] does state that a server *should* enforce some form of access control.

E Lack of Weak Consistency in SEEMless [CDG+19]

Our notion of weak consistency in Section 3 represents the most achievable notion of consistency for real-world key transparency (KT) systems. Assuming all parties maintain a consistent view of the commitment to the key log, this property can be realized—without placing any burden on users—even in the presence of a fully malicious server, as we demonstrate in Appendix C.

A surface-level reading of SEEMless [CDG+19] may suggest that it achieves a similar guarantee under the same assumptions. For instance, SEEMless states:

- "We also assume some way of ensuring that all parties have consistent views of the current root commitment or at least that they periodically compare these values to make sure that their views have not forked."
- "We require the following security properties of an append-only ZKS: Soundness: For soundness we want to capture two things: First, a malicious prover A^* algorithm should not be able to produce two verifying proofs for two different values for the same label with respect to a com. Second, since the aZKS is append-only, a malicious server should be unable to modify an existing label."

However, we show that SEEMless's protocol does not satisfy weak consistency as we define it unless the user explicitly checks the validity of their own key. In its current form, the server can serve divergent views of the key log to different users without being detected unless the affected user later performs a key history check. This issue is hinted at within the SEEMless text itself: "This does not prevent the server from showing Bob a version where version is much higher than Alice's real latest update. One approach to prevent this would be to have the server provide a that version is not in the aZKS_{all} for any higher version than Alice's current version. However, this potentially has a very high cost for Alice, proportional to the total possible number of updates, i.e. the number of epochs. Instead, we want to achieve the same guarantee but reduce Alice's work. To do this, we have the server add a 'marker' node, on every 2^ith update Alice makes. When Bob queries for Alice's key, he will also check that the previous marker node is in the aZKS_{all}. When Alice performs a KeyHistory check, she will also check that no higher marker nodes are included in the aZKS_{all}. Because we mark only the 2^i th updates, this cost is now only logarithmic in the number of epochs."

While this mitigation attempts to reduce the user's burden, a concrete discussion of the consequences of omitting such checks is missing from SEEMless. We make this explicit here through a simple example that assumes familiarity with the construction outlined in Section 4.3 of [CDG+19].

Suppose that Alice has legitimately added three versions of their key. Then, under honest opertion, aZKS_{all} should contain $\{(\text{alice}|1, \text{PK}_{a,1}), (\text{alice}|2, \text{PK}_{a,2}), (\text{alice}|3, \text{PK}_{a,3}), (\text{alice}|\text{mark}|0, \ldots)(\text{alice}|\text{mark}|1, \ldots)\}$ and aZKS_{old} should contain $\{(\text{alice}|1, \text{null}), (\text{alice}|2, \text{null})\}$. Now suppose the server is malicious. It inserts a fabricated key version $(\text{alice}|11, \text{PK}_{a,11})$ along with a forged marker node $(\text{alice}|\text{mark}|8, \ldots)$ into aZKS_{all} and commits to this altered log on the public bulletin board. Bob and Charlie each query for Alice's key. According to SEEMless's query and verification procedure, the server can present version 3 (a legitimate key) to Bob and version 11 (a fabricated key) to Charlie, and both will verify successfully. Neither user can detect this inconsistency—unless Alice herself later performs a key history check. Only at that point would the discrepancy be exposed.

In summary, for SEEMless to achieve consistency equivalent to our notion of weak consistency, it is necessary for users to check the validity of their own keys. This requirement is unsatisfying, as it places a substantial burden on users to achieve even a minimal consistency guarantee. By contrast, our protocol shows that weak consistency in adversarial settings can be achieved *without* requiring any action on the part of the user.

F Direct Paper Quotes

Here we list the full text quotes we reference from each paper of the core literature in Section 3.

F.1 CONIKS [MBB+15]

- "An identity provider may attempt to equivocate by presenting diverging views of the name-to-key bindings in its namespace to different users. Because CONIKS providers issue signed, chained 'snapshots' of each version of the key directory, any equivocation to two distinct parties must be maintained forever or else it will be detected by auditors who can then broadcast non-repudiable cryptographic evidence, ensuring that equivocation will be detected with high probability." [1]
- "If a client ever discovers two inconsistent STRs (for example, two distinct versions signed for the same epoch time), they should notify the user and whistleblow by publishing them to all auditors they arr able to contact. For example, clients could include them in messages sent to other clients, or they could explicitly send whistleblowing messages to other identity providers. We also envision out-of-band whistleblowing in which users publish inconsistent

STRs via social media or other high-traffic sites. We leave the complete specification of a whistleblowing protocol for future work." [2]

- "CONIKS clients may only query for individual usernames (which can be rate-limited and/or authenticated)..." [3]
- "CONIKS servers may attempt to hide malicious behavior by ceasing to respond to queries. We provide flexible defense against this, as servers may also simply go down. Servers may publish an expected next epoch number with each STR in the policy section P. Clients must decide whether they will accept STRs published at a later time than previously indicated." [4]
- "CONIKS servers do not need to make any information about their bindings public in order to allow consistency verification. Specifically, an adversary who has obtained an arbitrary number of consistency proofs at a given time, even for adversarially chosen usernames, cannot learn any information about which other users exist in the namespace or what data is bound to their usernames." [5]

F.2 SEEMLESS [CDG+19]

- "A party who only acts as an auditor learns only the numbers of keys added and keys updated each epoch. If that party additionally acts as a user (Alice) performing KeyHistory queries, the combined leakage may reveal when her keys are updated (even if she does not perform more KeyHistory queries), but that is expected to be something Alice knows since she is the one requesting the updates. If Alice additionally queries for Bob's key, the leakage reveals the version number of Bob's current key and the epoch when it was last updated, and may reveal when that key is no longer valid (because Bob performed an update), but will not reveal anything about subsequent or previous updates." [6]
- "We also assume some way of ensuring that all parties have consistent views of the current root commitment or at least that they periodically compare these values to make sure that their views have not forked." [7]
- "We also assume that the server enforces some kind of access control on the public key directory, for example only responding to Bob's query for Alice's key if he is on her contact list." [8]
- "The updates should be sufficiently frequent, so that the user keys are not out-of-date for long. The exact interval between these server updates, or epochs has to be chosen as a system parameter." [9]
- "Note that the onus is on the user, Alice, to make sure that the server is giving out the most recent and correct value for her key. Soundness guarantees that under the circumstances described above, Bob will always see a key consistent with what Alice has audited. But Alice needs to verify that her key as reported in the history query is consistent with the actual key that she chose." [10]
- "...a malicious prover algorithm should not be able to produce two verifying proofs for two different values for the same label with respect to a commitment." [11]
- "The functionality VKD.KeyHistory warrants further discussion since this is not a functionality that one usually expects from a key directory service. In a privacy-preserving verifiable key directory service, intuitively, we expect the server to be able to prove to Bob that he is seeing Alice's latest key without leaking any additional information about the directory. This is trivial to achieve if we assume that Alice can always sign her new public key with her old secret key. But this is a completely unreasonable assumption from an average user who may lose her device or re-install the software, thereby losing her previous secret key; the user will only have access to her latest secret key which is stored on her latest device. It is

crucial to not assume that Alice or Bob can remember any cryptographic secret. Under this constraint, we need Alice to monitor her key sufficiently often to make sure her latest key is in the server directory. Only then, we can talk about Bob getting Alice's latest key in a meaningful way. Alice could of course check every epoch to make sure that her key is being correctly reported, but this becomes costly, particularly when epochs are short. Instead, we allow Alice to query periodically and retrieve a list of all the times her key has changed and the resulting values using the VKD.KeyHistory interface." [12]

- "In addition to that, we want Alice's client to check her key history sufficiently often. This entails running KeyHistory query in the background periodically and notifying Alice if it is not consistent with the updates she has made. keyver must also be run when Alice changes her device or reinstalls the client software (thereby forcing a key update) in which cases the software would display to Alice a list of the times when her key was updated." [13]

F.3 Parakeet [MKS+23]

- "One way to avoid these issues is having auditors run consensus on each update of the IdP, essentially replacing the trust assumptions of the blockchain with a custom-made blockchain just for the transparency layer. However, this is simply an overkill. As a final contribution, Parakeet shows a consensus-less protocol that achieves all the desired properties for defending against split-view attacks." [14]
- "We provide an optional enhancement of the consistency protocol presented in Section V providing censorship resistance from malicious IdP. This protocol allows users to tie the liveness of their update requests with the liveness of the entire system effectively defending against selective censorship attacks." [15]
- "As stated before, the soundness definition used in this paper is that of non-equivocation, i.e., if, at an epoch t, Bob accepts a value val as Alice's key, the server cannot convince Alice that her key was val-prime for the same epoch t with val-prime 6 = val. Note, this is in the presence of auditors." [16]
- "In other words, in the presence of auditing and witnesses, Alice and Bob must always agree on the view of val Alice—the value associated with the label Alice." [17]
- "The leakage functions of this construction match those of [SEEMLESS's] construction exactly." [18]

F.4 OPTIKS [LCG+23b]

- "The server posts the commitments to its directory on a public bulletin board to which other participants of the system have access. The bulletin board should be tamper proof as well as append-only and also all participants should have a consistent view of its contents." [19]
- "Although we do not model this, we assume that the server enforces some kind of access control for clients querying its system, e.g. rate limiting key lookups or executing key lookups only if the requesting user is a contact of the user whose key is being queried." [20]
- "Our system relies on users being able to verify the history of their key updates. Therefore, users must have some way of keeping track of their devices and the approximate times of their key updates. This is an assumption made of other KT systems like SEEMless. One way to facilitate this is to enable users to add notes to their key updates, such as "added new laptop." Also crucial to our system is that clients must be online to check their key history each time period. We utilize this assumption as part of our scalability optimizations, which we discuss in Section 5. Given that time periods are long, we expect most clients will achieve this in practice and, indeed, this is a common assumption of KT systems. Moreover, this

is an improvement over many KT systems which assume that a client must be online each epoch to check their keys [18]." [21]

- "Assuming that all epochs are audited, the server cannot lie about a key's value during a lookup without the inconsistency being caught during a history check." [22]
- "A KT system should maintain privacy for the users of the system and updates to their keys. We model this with a definition that says participants of the system (excluding the server) should not learn anything from queries to the server except for some well-defined leakage function. For instance, a key lookup for a user should not leak anything about the keys of other users of the system." [23]
- "Both lookups and history checks leak the value and epoch of addition for each version of a key. Our leakage profile is therefore nearly the same as that for SEEMless and Parakeet, except that key lookups in their protocols leak only the version number for the key and the value and epoch of addition for the latest key version." [24]
- "The client devices can be malicious in that they may aim to learn private information (public keys, ho often a certain user changes her key, etc.) about other clients who are not on their contact list." [25]
- "However, the server is trusted to exercise access control and not give out every client's public key to everyone else. In other words, the server is trusted for privacy." [26]

F.5 ELEKTRA [LCG+23a]

- "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." [27]
- "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." [28]
- "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." [29]

F.6 IETF KEYTRANS Architecture Draft [McM25]

- "It is sometimes possible for a Transparency Log to present forked views of data to different users. This means that, from an individual user's perspective, a log may appear to be operating correctly in the sense that all of a user's requests succeed and proofs verify correctly. However, the Transparency Log has presented a view to the user that's not globally consistent with what it has shown other users. As such, the log may be able to change a label's value without the label's owner becoming aware." [30]
- "This provides ample opportunity for users to detect when a fork has been presented, but isn't in itself sufficient for detection. To detect forks, users must either use peer-to-peer communication or anonymous communication with the Transparency Log. With peer-to-peer communication, two users gossip with each other to establish that they both have the same view of the log's data. This gossip is able to happen over any supported out-of-band channel, even if it is heavily bandwidth-limited, such as scanning a QR code or talking over the phone. With anonymous communication, a single user accesses the Transparency Log

over an anonymous channel and tries to establish that the log is presenting the same view of data over the anonymous channel as it does over authenticated channels. In the event that a fork is successfully detected, the user is able to produce non-repudiable proof of log misbehavior which can be published." [31]

- "Applications determine the privacy of data in KT by relying on these properties when they enforce access control policies on the queries issued by users, as discussed in Section 3." [32]
- "When a user modifies a label, they're guaranteed that other users will see the modification the next time they search for the label." [33]
- "A user that correctly verifies a proof from the Transparency Log (and does any required monitoring afterwards) receives a guarantee that the Transparency Log operator executed the label-value lookup correctly, and in a way that's globally consistent with what it has shown all other users. That is, when a user searches for a label, they're guaranteed that the result they receive represents the same result that any other user searching for the same label would've seen." [34]
- "In short, assuming that the underlying cryptographic primitives used are secure, any deployment-specific assumptions hold (such as non-collusion), and that user devices don't go permanently offline, then malicious behavior by the Transparency Log is always detected within a bounded amount of time." [35]
- "In the event that a third-party auditor or manager is used, there's additional information leaked to the third-party that's not visible to outsiders. In the case of a third-party auditor, the auditor is able to learn the total number of distinct changes to the log. It is also able to learn the order and approximate timing with which each change was made. However, auditors are not able to learn the plaintext of any labels or values. This is because labels are masked with a VRF, and values are only provided to auditors as commitments. They are also not able to distinguish between whether a change represents a label being created for the first time or being updated, or whether a change represents a "real" change from an end-user or a "fake" padding change.In the case of a third-party manager, the manager generally learns everything that the service operator would know. This includes the total set of plaintext labels and values and their modification history. It also includes traffic patterns, such as how often a specific label is looked up." [36]

F.7 IETF KEYTRANS Protocol Draft [ML24]

- "This document describes a protocol that enables a group of users to ensure that they all have the same view of the public keys associated with each other's accounts. Ensuring a consistent view allows users to detect when unauthorized public keys have been associated with their account, indicating a potential compromise." [37]