A Formal Treatment of Key Transparency Systems with Scalability Improvements

Nicholas Brandt
ETH Zurich
Zurich, Switzerland
nicholas.brandt@inf.ethz.ch

Mia Filić
ETH Zurich
Zurich, Switzerland
mia.filic@inf.ethz.ch

Sam A. Markelon
University of Florida
Gainesville, FL, USA
smarkelon@ufl.edu

Abstract—**Key Transparency (KT) systems have emerged as a critical technology for securely distributing and verifying the correctness 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 introduce the first cryptographically sound formalization of KT as an ideal functionality, clarifying the assumptions, security properties, and potential vulnerabilities of deployed KT systems. We identify a significant security concern—a possible impersonation attack by a malicious service provider—and propose a backward-compatible solution. Additionally, we address a core scalability bottleneck by designing and implementing a novel, privacy-preserving verifiable Bloom filter (VBF) that significantly improves KT efficiency without compromising security. Experimental results demonstrate the effectiveness of our approach, marking a step forward in both the theoretical and practical deployment of scalable KT solutions.**

1. Introduction

Key Transparency (KT) systems offer a solution to the challenges of public key distribution in end-to-end encrypted communication platforms. KT systems are vital to preventing trivial man-in-the-middle attacks. 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. 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, as evidenced by works such as [33, 4, 9, 39, 41, 20, 40, 10, 28, 24, 25], but have also been implemented by platforms such as Keybase [29], Zoom [3], Google [21], WhatsApp [26], Apple iMessage [1], and Proton [17]. Complementing this industry adoption, the Internet Engineering Task Force (IETF) has formed the KEYTRANS working group [30] to both formalize and standardize KT systems. Our findings are aimed at supporting these standardization efforts by addressing crucial security and scalability challenges.

Despite this momentum, current KT systems lack a rigorous, formalized security framework, which limits both our understanding of these systems and our ability to protect them against real-world adversaries. Previous works have conflated the objective of KT systems with their proposed protocol that realizes this objective. This absence of a precise formalism restricts security analysis to isolated aspects of KT protocols, leaving several potential vulnerabilities undetected and unaddressed. For example, existing protocols do not fully address scenarios where a malicious service provider could manipulate key states to compromise user privacy and impersonate users, particularly if users have limited trust in their service providers. Moreover, without a formalized model, it is challenging to build upon current KT implementations or make meaningful advancements that can be seamlessly integrated into past or future systems. In this paper, we present the first cryptographically sound formalization of KT systems as an ideal functionality. This foundational step clarifies the underlying assumptions, goals, and security guarantees for KT systems, offering a more complete framework for evaluating and enhancing these protocols.

Scalability also remains a pressing issue for KT systems, particularly for real-world large scale deployments like WhatsApp's KT protocol [23]. Such deployments have over a billion active users and demand a frequency of key queries and updates that far exceed the limits of what past academic work has considered. Existing KT architectures face performance bottlenecks due to their reliance on centralized, resource-intensive operations to handle key queries and updates, making them increasingly costly to deploy at scale.

Addressing this challenge in previous KT research has largely focused on making this centralized core KT protocol more efficient, optimizing elements like data structures or reducing computation within the primary system to handle increased loads. While these approaches improve performance, they hit a clear ceiling: as user bases grow and the volume of key queries rises, the core KT protocol alone struggles to meet demand without substantial resource consumption. Some works have suggested to address scalability by distributing the KT protocol, yet these approaches treat distribution as an afterthought, overlooking the unique challenges posed by KT's stringent security requirements. Unlike traditional distributed systems, where achieving eventual consistency is often sufficient, KT protocols demand strict, synchronized consensus to ensure that every user receives accurate, verifiable responses at all times. This level of security makes straightforward distribution infeasible, as KT systems cannot tolerate inconsistencies or delayed synchronization without compromising user trust and protocol integrity.

Our work tackles the scalability challenge in KT systems by introducing a structured distribution model that maintains rigorous security guarantees while enabling efficient, scalable query handling across decentralized nodes. At the core of this solution is a new primitive we call the verifiable Bloom filter (VBF), designed specifically to handle key queries in a privacy-preserving and resource-efficient manner. Unlike conventional approaches that centralize query processing on a primary server, the VBF allows key queries to be processed on distributed, light-weight edge devices. This shift significantly reduces the computational burden on the central provider, allowing it to focus on critical functions while edge devices handle the bulk of routine query processing in real time.

The VBF achieves this by integrating verifiability directly into its design through cryptographic guarantees, ensuring that responses from edge devices can be trusted without requiring constant synchronization with the central server. This verifiability is crucial in maintaining the KT protocol's high security standards, as it allows users to verify the accuracy of responses independently, even in a decentralized context. Moreover, our VBF-based approach is modular, allowing it to be seamlessly integrated with existing KT frameworks. This adaptability means that the VBF can enhance scalability and flexibility in current KT systems without necessitating extensive changes to their core architecture. In essence, the VBF acts as an efficient, privacy-preserving summary layer that complements traditional KT protocols, extending their reach and performance capacity across larger user bases. Through this design, our approach not only addresses the scalability limits of centralized KT systems but also introduces a robust solution for secure, distributed key management that scales with demand.

Our specific contributions are thus threefold:

- We give the first cryptographically sound formalization of KT systems in the form of an ideal functionality. This clarifies the assumptions, features and security guarantees of KT systems which have been somewhat recondite in previous literature.
- We point out a security concern (an impersonation attack by a malicious service provider) that was so far (implicitly) considered unavoidable. We suggest a (backwards-compatible) solution for coping with malicious service providers. Moreover, we observe that implementations of KT protocols based on Meta's AKD library [22] exhibit no formal privacy guarantees!
- We identify a performance bottleneck in the current architecture of KT systems, and propose a scalable and privacy-preserving solution based on a new primitive that we call *verifiable Bloom filter*. We implement our protocol and provide experimental evidence of its performance improvements over the current state-of-the-art. In fact, at the scale of two billion users our scalable protocol reduces the total computation time for processing queries by 64% in typical deployment conditions versus a protocol not using our VBF mechanism. Additionally, we discuss how the VBF's modularity enables it to be combined with other KT protocols, extending its applicability across diverse deployment scenarios.

In summary, this paper not only provides a rigorous framework for understanding and advancing KT systems but also offers practical solutions to the core challenges of security and scalability. We anticipate that our findings will inform the design of future KT protocols and contribute to the ongoing IETF standardization efforts, with the potential to make decentralized, transparent key management a reliable, scalable feature in secure communication platforms. Future work will explore extending the formalism of VBFs and refining additional elements of the KT model to further enhance scalability and security in real-world applications.

2. Preliminaries

2.1. 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 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 $\mathsf{output} =: \mathsf{alg}(\mathsf{input}_1, \mathsf{input}_2, ..., \mathsf{input}_l; r)$. 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.

2.2. Append-Only Zero-Knowledge Sets

A zero-knowledge set (ZKS) [34] is a cryptographic primitive that allows for the efficient verification of set-membership queries on a represented set without revealing more information than the membership itself. An append-only zero-knowledge set (aZKS) [9] is a variant of ZKS that allows for updates to the represented set in an append-only manner, i.e., no existing entries are ever modified or deleted. This construction carries the additional leakage of the size of the represented set. We give a formal definition of aZKS in Appendix A and refer the reader to [9] for the formal security guarantees.

2.3. Bloom Filters

A Bloom filter [2] (BF) is the ubiquitous example of a compressing probabilistic data structure. A BF provides a compact representation of a given set, and admits set-membership queries (e.g., *is element x in the set?*). The BF structure is a length m bit-array. To represent a given set, each element in the set is run through k pairwise independent hash functions that map from $\{0,1\}^* \to [m]$, then for every value output by these hash functions the corresponding bit in the bit-array is set to 1 (or left set to 1 if already set). To make a set-membership query to the BF for an element x, x is run through these k hash functions and positive response 1 is returned iff all the corresponding bits in the bit-array are set to 1 (0 otherwise). We give an algorithmic description of the BF and more details in Appendix B.

The bit-array representation is typically *much* smaller than the number of bits needed to fully represent the elements in the set; thus, the responses to set-membership queries are only approximately correct. In particular, while Bloom filters do not have false-negative errors, they do suffer false-positive errors at a rate that is a function of the bit-array size (m), the number of hash functions (k), and the size of underlying set.

2.4. Verifiable Random Functions

A verifiable random function (VRF) [35] is the public-key analog of the pseudorandom functions primitive (PRF) [19], with the additional feature that one can verify that its outputs have been correctly computed. Looking ahead, we will use VRFs to transform the Bloom filter into a verifiable version of the structure that allows for verification of its query responses with respect to a commitment on the bit-array representation. Moreover, a VRF is an integral component of the construction of the aZKS used in most KT protocols. We give a formal definition of a (simulatable) VRF and its security properties in Appendix C.

3. Modeling Key Transparency

Previous works, [33, 9, 28, 25, 24], to varying degrees, have treated key transparency (KT) systems as a monolithic block, i.e., they fail to differentiate between the specification (the objective) of a KT system, and the protocol that realizes that specification. This significantly complicates the analysis of these protocols and the interpretation of their derived security guarantees. We notice that this was also the case in the early stages of the development of other multi-party computation (MPC) applications [44], like key exchange—where the word "key-exchange" was used synonymously with the Diffie–Hellman key-exchange protocol [12]. However, today there is a clear distinction between a public-key encryption (PKE) *scheme*[1] and a key-exchange *protocol*. In modern cryptography, the objective of an MPC is captured in the form of an *ideal* functionality that specifies how the system behaves in an idealized setting. A protocol for a given functionality is then judged to be secure, if it realizes the functionality in the real-ideal simulation paradigm [43, 5]. Since KT is inherently an MPC, we deem it appropriate and important to formally specify an ideal functionality for KT. Due to space restriction we refer the interested reader to the monograph of Evans, Kolesnikov, and Rosulek [13] for more details about MPC. While KT was first introduced in [33], we accredit the first step towards a proper formalization of KT to Chase et al. [9] 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 need to distinguish between three formal concepts:

- a KT *functionality* that specifies what the formal objective of a KT system is (e.g. supplying users with previously registered keys),
- a KT *protocol* that realizes (in the real-ideal paradigm) the KT functionality (using a KT scheme), and
- a KT *scheme* that defines a set of algorithms (analogous to a PKE scheme).

First, we give a definition of an optimal KT functionality that captures the features and security that we intuitively expect from a KT system.

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

$\mathcal{F}_{\mathsf{idealKT}}$ proceeds as follows, running with security parameter λ, n users $\mathcal{U} = \{\mathsf{id}_1, ..., \mathsf{id}_n\}$, service provider S, and adversary $\mathcal{S}$. Messages not covered here are ignored. Initially, set the epoch counter $\tau := 0$, the update list $\mathsf{L}_\tau := []$, the database $\mathsf{D}[\tau, \mathsf{id}] := \perp$.

- **RegisterKey**: When receiving a key k from user U_{id}, store $\mathsf{D}[\tau, \mathsf{id}] := \mathsf{k}$.
- **QueryKey**: When receiving query $(\textbf{QueryKey}, \mathsf{id}')$ from user U_{id}, retrieve $\mathsf{k} := \mathsf{D}[\tau - 1, \mathsf{id}']$. Send $(\textbf{QueryResponse}, \mathsf{id}', \mathsf{k})$ to U_{id}.
- **EpochUpdate**: When receiving **EpochUpdate** from S, for each id if $\mathsf{D}[\tau, \mathsf{id}] = \perp$, then $\mathsf{D}[\tau, \mathsf{id}] := \mathsf{D}[\tau - 1, \mathsf{id}]$, and increment $\tau := \tau + 1$. Send $(\textbf{EpochUpdate}, \tau)$ to each user $\mathsf{U}_{\mathsf{id}_1}, ..., \mathsf{U}_{\mathsf{id}_n}$.

This functionality allows three high-level procedures:

1) In each epoch each user may register a new key.
2) In each epoch τ each user U_{id} may query any user $\mathsf{U}_{\mathsf{id}'}$'s most recent key that was registered in (a previous) epoch $\tau' \lneq \tau$.
3) At any point the service provider may update the epoch.

1. A formal PKE scheme does not impose a notion of a sender or receiver, only encryption and decryption (and key generation).

This functionality models the behavior of an optimal KT system. Unfortunately, we cannot realize $\mathcal{F}_{\mathsf{idealKT}}$ in the plain model[2] because it implies a commitment functionality which is impossible without setup [6]. This impossibility is the reason why previous protocols employ the additional help of (to some degree) trusted auditors, and it explains why their security guarantees are somewhat convoluted. Thus, to obtain a *realizable* KT functionality, we need to grant the adversary more abilities; optimally to the smallest extent possible. On the other hand, to match the real-world deployment of KT systems, we need to make some restrictions on the possible protocol that would realize a (weakened) KT functionality. These restrictions depend on the deployment scenario, and roughly speaking, the greater user convenience the protocol provides the greater restrictions we need to impose.

On malicious service providers and insufficient security notions. Let us consider a few attack scenarios:

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 service provider 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 service provider can impersonate the affected user simply by simulating the affected user.
2) Omission attack: If a user registers a new key with the service provider but that key is only sent to the service provider (i.e., the new key is not correlated with the view of any other party), then there is no guarantee that a (stop-fault) service provider will actually include the newly registered key in the next epoch update.

In addressing these attacks, [9] defines 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 service provider is malicious.

Take, for example, a malicious service provider that injects their own key for a particular user. While the system in this setting is clearly insecure, the soundness notion is still satisfied. The authors of [9] argue that such malicious service provider 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 to prove their identity within the system. Note that in this setting the above attack implies that the service provider 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 [9] would allow not only a malicious service provider 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 service provider must be assumed at least semi-honest.

We contextualize this observation with the current development of auditing mechanisms in KT systems. These auditors should guarantee that the service provider acts honestly, but if the service provider is presumed to be corruptable, then the underlying security guarantees are void. Put plainly, if the service provider is assumed to be honest, then auditors are superfluous; if the service provider is assumed to be malicious, then even honest auditors cannot prevent impersonation attacks. Thus, the use of auditors only makes sense if out-of-band communication is assumed.

We want to stress that some non-cryptographic techniques are considered to mitigate this problem in practice. For example Linker and Basin [27], formalizes the notion of social authentication and implement a protocol for it. To tackle the problem of users impersonating each others, [9] relies on application-level access control. The IETF working group on KT systems [31] 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.

Coping with malicious service providers. For some high-risk users (such as journalists or activists), the assumption of a semi-honest service provider may not be acceptable, e.g. due to government coercion. In light of the inherent impersonation potential for reset users, we propose to let users opt-in to a "high-security" mode, in which they are expected to retain a high-entropy passphrase even when losing their device.[3] The advantage of this mode is that it extends security guarantees to protect against a malicious service provider. However, if the passphrase is compromised, the user's identity within the system is compromised. Further, selecting this mode, the user needs to be aware that if they forget their passphrase, they are no longer able to update public keys related to their identity, e.g. register new devices. We stress that this high-security mode is backwards-compatible with existing implementations. In this mode, a public key can only be updated if it is signed by the user's (persistent) private key, which is derived from the passphrase.

Realizable KT functionality. We now define a functionality $\mathcal{F}_{\mathsf{KT}}$ in Fig. 1 that can be realized but still provides a

2. In the universal composibility framework of Canetti [5] without setup assumptions.

3. Given that this is mostly relevant for high-risk users, we consider this a reasonable assumption.

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

$\mathcal{F}_{\mathsf{KT}}$ proceeds as follows, running with security parameter λ, n users $\mathcal{U} = \{\mathsf{id}_1, ..., \mathsf{id}_n\}$, and a service provider S, and adversary $\mathcal{S}$ that corrupts $\mathcal{C} \subseteq \mathcal{U} \cup \{\mathsf{S}\}$. Messages not covered here are ignored. Initially, set the epoch counter $\tau := 0$, for all $\tau' \geq -1$, set the update list $\mathsf{L}_{\tau'} := []$ and database $\mathsf{D}_{\tau'}[\mathsf{id}, \mathsf{v}] := \bot$ for each id and version v. Set the latest version list $\widehat{\mathsf{v}}[\mathsf{id}] = -1$ for each id. Let $\rho : \{0,1\}^* \to \{0,1\}^{\ell(\lambda)}$ be a random function.

- **RegisterKey**: When receiving a key k from user U_{id}, if the service provider is corrupted, i.e., $\mathsf{S} \in \mathcal{C}$, send (**RegisterKey**, id, k) to $\mathcal{S}$. If $\mathsf{S} \notin \mathcal{C}$ or after receiving **AllowRegisterKey** from $\mathcal{S}$, store $\mathsf{L}_\tau[\mathsf{id}, \widehat{\mathsf{v}}[\mathsf{id}'] + 1] := \mathsf{k}$.
- **QueryKey**: When receiving query (**QueryKey**, id') from user U_{id}, retrieve the key $\mathsf{k}{:=}\mathsf{D}_{\tau-1}[\mathsf{id}', \widehat{\mathsf{v}}[\mathsf{id}']]$. if the service provider is corrupted, i.e., $\mathsf{S} \in \mathcal{C}$, send (**QueryKey**, id, id', k) to $\mathcal{S}$. If $\mathsf{S} \notin \mathcal{C}$ or after receiving **AllowQueryKey** from $\mathcal{S}$, send (**QueryResponse**, τ, id', k, R) to user U_{id} with leakage $R := \rho(\mathsf{id}', \widehat{\mathsf{v}}[\mathsf{id}'])$.
- **EpochUpdate**: When receiving **EpochUpdate** from S, update the database with the entries of L_τ as $\mathsf{D}_\tau[\mathsf{label}] := \mathsf{L}_\tau[\mathsf{label}]$ and $\mathsf{D}_\tau[\mathsf{label}] := \mathsf{D}_{\tau-1}[\mathsf{label}]$ for all label $= (\mathsf{id}, \mathsf{v})$ in $\mathsf{D}_{\tau-1}$, increment the counter $\tau := \tau + 1$. Send leakage $R := (\tau, (\rho(\mathsf{label}))_{\mathsf{label}:\mathsf{L}_{\tau-1}[\mathsf{label}] \neq \bot})$ to $\mathcal{S}$. Send (**EpochUpdate**, τ) to each user $\mathsf{U}_{\mathsf{id}_1}, ..., \mathsf{U}_{\mathsf{id}_n}$.

Figure 1: The functionality $\mathcal{F}_{\mathsf{KT}}$ that models the objective of a system.

sufficient[4] level of security. For ease of exposition, we concentrate on the case where the number of users is fixed and known to all parties. In this functionality the adversary must greenlight key registration requests (if the service provider is corrupted) – reflecting the omission attack described in Attack 2. However, the (potentially malicious) service provider cannot impersonate users, as the functionality $\mathcal{F}_{\mathsf{KT}}$ only registers a key for user U_{id} if that key stems from user U_{id}. Moreover, to reflect unavoidable information obtained by the adversary in the real world, the functionality $\mathcal{F}_{\mathsf{KT}}$ provides leakage to the adversary. In particular, upon an epoch update (we assume that the epoch update and proof is public) the adversary learns a random function of the user identifiers and the versions of the keys that were updated in the previous epoch. This leakage ultimately stems from the KT protocol of Chase et al. [9], namely, the epoch update commitment contains VRF evaluations of $(\mathsf{id}, \mathsf{v})$ for each updating user U_{id}. Similarly, when querying for the (most recent) key of a user $\mathsf{U}_{\mathsf{id}'}$, the querying user learns the VRF image corresponding to $(\mathsf{id}', \mathsf{v}')$. Thus, if the adversary queries all users in each epoch, the adversary can trivially learn which user updated their key in which epoch.

KT Scheme. Before we describe our protocol that realizes this functionality, let us introduce the notion of a KT scheme.[5]

```
$\mathsf{Exp}^{\mathrm{corr}}_{\mathsf{KT},\mathcal{A}}(\lambda)$
1 : $\sigma := 0; \mathsf{D} \leftarrow \emptyset$
2 : $\mathsf{st} \leftarrow \mathsf{KT.Setup}(1^\lambda)$
3 : $\mathsf{crs} \leftarrow \mathcal{A}(\mathsf{st})$
4 : $\mathcal{A}^{\mathsf{QryKey}', \mathsf{UpdateEpoch}'}(\mathsf{st}, \mathsf{crs})$
5 : return $\sigma$

$\mathsf{QryKey}'(\mathsf{id}, \mathsf{v})$
1 : $(\mathsf{k_v}, \pi) \leftarrow \mathsf{KT.QryKey}(\mathsf{st}, \mathsf{id}, \mathsf{v}, \mathsf{crs})$
2 : $b := \mathsf{KT.VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k_v}, \pi, \mathsf{crs})$
3 : if $b = 1 \wedge \mathsf{D}[\mathsf{id}, \mathsf{v}] \neq \mathsf{k_v}$ then $\sigma := 1$

$\mathsf{UpdateEpoch}'(\mathsf{L})$
1 : $(\mathsf{st}', \mathsf{D}', \pi) \leftarrow \mathsf{KT.UpdateEpoch}(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{crs})$
2 : $\mathsf{com} := \mathsf{KT.Commit}(\mathsf{st}, \mathsf{D}, \mathsf{crs})$
3 : $\mathsf{com}' := \mathsf{KT.Commit}(\mathsf{st}', \mathsf{D}', \mathsf{crs})$
4 : $b := \mathsf{KT.Audit}(\mathsf{com}, \mathsf{com}', \pi, \mathsf{crs})$
5 : if $b = 0$ then $\sigma := 1$
6 : $\mathsf{st} := \mathsf{st}'; \mathsf{D} := \mathsf{D}'$

$\mathsf{Exp}^{\mathrm{inter}}_{\mathsf{KT},\mathcal{A}}(\lambda)$
1 : $(\mathsf{com}, \mathsf{com}', \pi_{\mathsf{com}}, \mathsf{id}, \mathsf{v}, \mathsf{v}', \mathsf{k}, \mathsf{k}', \pi_{\mathsf{v}}, \pi_{\mathsf{v}+1}, \pi', \mathsf{crs})$
      $\leftarrow \mathcal{A}(1^\lambda)$
2 : req $\mathsf{KT.Audit}(\mathsf{com}, \mathsf{com}', \pi_{\mathsf{com}}, \mathsf{crs}) = 1$
3 : req $\mathsf{KT.VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi_{\mathsf{v}}, \mathsf{crs}) = 1$
4 : / no skipping versions
5 : if $\mathsf{KT.VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}+1, \bot, \pi_{\mathsf{v}+1}, \mathsf{crs}) = 1$
6 :     $\wedge\, \mathsf{KT.VfyQry}(\mathsf{com}', \mathsf{id}, \mathsf{v}', \mathsf{k}', \pi', \mathsf{crs}) = 1$
7 :     $\wedge\, \mathsf{v}' \geq \mathsf{v} + 2 \wedge \mathsf{k}' \neq \bot$ then return 1
8 : / no modification of existing keys
9 : if $\mathsf{KT.VfyQry}(\mathsf{com}', \mathsf{id}, \mathsf{v}, \mathsf{k}', \pi', \mathsf{crs}) = 1$
10 :     $\wedge\, \mathsf{k} \notin \{\mathsf{k}', \bot\}$ then return 1
11 : return 0

$\mathsf{Exp}^{\mathrm{incl}}_{\mathsf{KT},\mathcal{A}}(\lambda)$
1 : $(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi_{\mathsf{v}}, \pi_{\mathsf{v}+1}, \pi', \mathsf{crs}) \leftarrow \mathcal{A}(1^\lambda)$
2 : req $(\mathsf{id}, \mathsf{k}) \in \mathsf{L} \wedge \mathsf{k} \neq \bot$
3 : $(\mathsf{st}', \mathsf{D}', \pi_{\mathsf{com}}) := \mathsf{KT.UpdateEpoch}(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{crs})$
4 : $\mathsf{com} := \mathsf{KT.Commit}(\mathsf{st}, \mathsf{D}, \mathsf{crs})$
5 : $\mathsf{com}' := \mathsf{KT.Commit}(\mathsf{st}', \mathsf{D}', \mathsf{crs})$
6 : req $\mathsf{KT.Audit}(\mathsf{com}, \mathsf{com}', \pi_{\mathsf{com}}, \mathsf{crs}) = 1$
7 : req $\mathsf{KT.VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi_{\mathsf{v}}, \mathsf{crs}) = 1$
8 : req $\mathsf{KT.VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}+1, \bot, \pi_{\mathsf{v}+1}, \mathsf{crs}) = 1$
9 : req $\mathsf{KT.VfyQry}(\mathsf{com}', \mathsf{id}, \mathsf{v}+1, \mathsf{k}, \pi', \mathsf{crs}) = 1$
10 : return 1
```

Figure 2: Games for our definition of KT schemes.

4. Optimal w.r.t. the previously discussed attacks.

5. This notion has some similarity with the notion of a verifiable key directory (VKD) in [9]. However, [9] explicitly omit any specification of "system parameters".

Definition 1 (Key Transparency Scheme). *A key transparency (KT) scheme is a tuple of efficient algorithms* $\mathsf{KT} = (\mathsf{Setup}, \mathsf{Init}, \mathsf{Commit}, \mathsf{QryKey}, \mathsf{VfyQry}, \mathsf{UpdateEpoch}, \mathsf{Audit}, \mathsf{SimSetup})$ *where*

- $\mathsf{Setup}(1^\lambda)$ *on input security parameter* 1^λ*, outputs a common reference string (CRS)* crs*,*
- $\mathsf{Init}(1^\lambda, \mathsf{crs})$ *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 a new server state* st'*, a new database* D'*, and a proof* π *of correctness.*
- $\mathsf{Audit}(\mathsf{com}, \mathsf{com}', \pi, \mathsf{crs})$ *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* τ*.*
- $\mathsf{SimQryKey}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \mathsf{crs}, \tau)$ *on input user identifier* id*, version* v*, CRS* crs*, and simulation trapdoor* τ*, outputs a key* k *associated with* id *at version* v*, and a proof* π*.*
- $\mathsf{SimUpdateEpoch}(\mathsf{vk}, \mathsf{com}, \mathsf{com}', \mathsf{L}, \mathsf{crs}, \tau)$ *on input commitments* $\mathsf{com}, \mathsf{com}'$*, list of key updates* $\mathsf{L} = (\mathsf{id}_i, \mathsf{k}_i)_i$*, CRS* crs*, and simulation trapdoor* τ*, outputs a proof* π*.*

Setup, $\mathsf{SimSetup}$ *and* Init *are probabilistic, the other algorithms are deterministic.*

We define several correctness and security properties for KT systems.

Correctness. *For every (unbounded) stateful adversary* $\mathcal{A}$*, we have that* $\Pr[\mathsf{Exp}^{\mathsf{corr}}_{\mathsf{KT},\mathcal{A}}(\lambda) = 1] = 0$ *where* $\mathsf{Exp}^{\mathsf{corr}}_{\mathsf{KT},\mathcal{A}}$ *is the correctness game defined in Fig. 2. Correctness guarantees that (verifying) responses match the underlying database.*

Intra-Epoch Consistency. *For every commitment* com*, user identifier* id*, version* v*, keys* $\mathsf{k}^0, \mathsf{k}^1$*, and proofs* π^0, π^1*, we have that*

$$\mathsf{VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}^0, \pi^0) = \mathsf{VfyQry}(\mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}^1, \pi^1) \implies \mathsf{k}^0 = \mathsf{k}^1 \ .$$

Intra-Epoch consistency guarantees that within a single epoch (a single commitment com*) only a single key verifies for a given query.*

Inter-Epoch Consistency. *For every (unbounded) adversary* $\mathcal{A}$*, we have that* $\Pr[\mathsf{Exp}^{\mathsf{inter}}(\lambda) = 1] = 0$ *where* $\mathsf{Exp}^{\mathsf{inter}}_{\mathsf{KT},\mathcal{A}}$ *is the inter-epoch consistency game defined in Fig. 2. This notion essentially corresponds to "VKD soundness" in [9]; it ensures once a given key version has been added, its valid response remains consistent across epochs.*

Inclusivity. *For every (unbounded) adversary* $\mathcal{A}$*, we have that* $\Pr[\mathsf{Exp}^{\mathsf{incl}}_{\mathsf{KT},\mathcal{A}}(\lambda) = 1] = 0$ *where* $\mathsf{Exp}^{\mathsf{incl}}_{\mathsf{KT},\mathcal{A}}$ *is the inclusivity game defined in Fig. 2. Inclusivity guarantees that a key* k *is a valid response for a query* $(\mathsf{id}, \mathsf{v})$ *if an (honest) auditor has verified an epoch update that contains* $(\mathsf{id}, \mathsf{k}) \in \mathsf{L}$*.*

Privacy. *For every (unbounded) adversary* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}_\mathcal{A}$ *such that for each database* D *it holds that*

$$\left|\Pr\left[\mathcal{A}(1^\lambda, \mathsf{com}) = 1\right] - \Pr\left[\mathcal{S}^{\mathsf{D}}_{\mathcal{A}}(1^\lambda) = 1\right]\right| \leq \operatorname{negl}(\lambda) \quad (1)$$

where $\mathsf{st} \leftarrow \mathsf{Setup}(1^\lambda)$ *and* $\mathsf{com} := \mathsf{Commit}(\mathsf{st}, \mathsf{D})$*. Privacy guarantees the KT scheme leaks at most as much information about the database as can be obtained by oracle access to the database.*[6]

The following three properties are used only in the proof of security of our KT protocol.

CRS indistinguishability. *The two distributions* $\{\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) : \mathsf{crs}\}$ *and* $\{(\mathsf{crs}, \tau) \leftarrow \mathsf{SimSetup}(1^\lambda) : \mathsf{crs}\}$ *are computationally indistinguishable.*

Query simulation. *For each* $\mathsf{vk}, \mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}$ *we require that* $\Pr[(\mathsf{crs}, \tau) \leftarrow \mathsf{SimSetup}(1^\lambda), (\mathsf{k}, \pi) := \mathsf{SimQryKey}(\mathsf{vk}, \mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{crs}, \tau) : \mathsf{VfyQry}(\mathsf{vk}, \mathsf{com}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi, \mathsf{crs}) = 1] = 1$*. In other words, if the CRS is in simulation mode, arbitrary proofs can be generated using the trapdoor.*

Epoch update simulation. *For each* $\mathsf{vk}, \mathsf{com}, \mathsf{com}', \mathsf{L}$ *we require that* $\Pr[(\mathsf{crs}, \tau) \leftarrow \mathsf{SimSetup}(1^\lambda), \pi := \mathsf{SimUpdateEpoch}(\mathsf{vk}, \mathsf{com}, \mathsf{com}', \mathsf{L}, \mathsf{crs}, \tau) : \mathsf{Audit}(\mathsf{vk}, \mathsf{com}, \mathsf{com}', \pi, \mathsf{crs}) = 1] = 1$.

Having established the definition of a KT scheme, we now present our construction of a KT scheme based on the append-only zero-knowledge set from [9].

Definition 2 (KT Scheme Instantiation). *Let* PRF *be a PRF. Let* VRF *be a VRF. Let* aZKS *be an append-only zero-knowledge set. We define a KT scheme* KT *in Fig. 3.*

Our KT scheme instantiation essentially provides a minimal API for parties to use in our KT protocol. To assure deterministic algorithms, we derandomize Commit, QryKey and $\mathsf{UpdateEpoch}$ by using a pseudorandom function [18]. It resembles the verifiable key directory (VKD) of [9], but disentangles the underlying scheme from the KT protocol. For example, a service provider in a KT protocol (not part of the scheme itself) would generate its internal state $\mathsf{st}_{\mathsf{KT}} \leftarrow \mathsf{KT}.\mathsf{Init}(1^\lambda)$, and then use the $\mathsf{KT}.\mathsf{UpdateEpoch}$ function to generate a new epoch update proof $\pi^{\mathsf{upd}}_{\mathsf{KT}}$ and update its internal state.

3.1. Comparison with Previous Work

We remark that the scheme instantiation in Fig. 3 represents the minimal components necessary to satisfy Definition 1 and its corresponding correctness and security problems. It is similar, but less complex than the instantiation

6. To guarantee functionality of the KT scheme, any adversary must at least have oracle access to the database.

```
Setup($1^\lambda$)
1 : return $\mathsf{crs} \leftarrow \mathsf{aZKS.Setup}(1^\lambda)$

Init($1^\lambda$)
1 : $\mathsf{k_{PRF}} \leftarrow \mathsf{PRF.Gen}(1^\lambda); \mathsf{st_{aZKS}} := \bot$
2 : return $\mathsf{st_{KT}} := (\mathsf{k_{PRF}}, \mathsf{st_{aZKS}})$

Commit($\mathsf{st_{KT}}, \mathsf{D}, \mathsf{crs}$)
1 : $(\mathsf{k_{PRF}}, \mathsf{st_{aZKS}}) := \mathsf{st_{KT}}$
2 : $r := \mathsf{PRF.Eval}(\mathsf{k_{PRF}}, \mathsf{st_{aZKS}}||\mathsf{D}||\mathsf{crs})$
3 : $(\mathsf{com_{aZKS}}, \mathsf{sk_{aZKS}}) := \mathsf{aZKS.Gen}(1^\lambda, \mathsf{D}, \mathsf{crs}; r)$
4 : return $\mathsf{com_{KT}} := \mathsf{com_{aZKS}}$

QryKey($\mathsf{st_{KT}}, \mathsf{D}, \mathsf{id}, \mathsf{v}, \mathsf{crs}$)
1 : $(\mathsf{k_{PRF}}, \mathsf{st_{aZKS}}) := \mathsf{st_{KT}}$
2 : $r := \mathsf{PRF.Eval}(\mathsf{k_{PRF}}, \mathsf{st_{aZKS}}||\mathsf{D}||\mathsf{id}||\mathsf{v}||\mathsf{crs})$
3 : $(\pi_{\mathsf{aZKS}}, \mathsf{k}) := \mathsf{aZKS.Qry}(\mathsf{st_{aZKS}}, \mathsf{D}, \mathsf{id}||\mathsf{v}, \mathsf{crs}; r)$
4 : $\pi_{\mathsf{KT}} := \pi_{\mathsf{aZKS}}$
5 : return $(\mathsf{k}, \pi_{\mathsf{KT}})$

VfyQry($\mathsf{com_{KT}}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \pi_{\mathsf{KT}}, \mathsf{crs}$)
1 : $\mathsf{com_{aZKS}} := \mathsf{com_{KT}}; \pi_{\mathsf{aZKS}} := \pi_{\mathsf{KT}}$
2 : return $\mathsf{aZKS.Vfy}(\mathsf{com_{aZKS}}, \mathsf{id}, \mathsf{k}, \pi_{\mathsf{aZKS}}, \mathsf{crs})$

UpdateEpoch($\mathsf{st_{KT}}, \mathsf{D}, \mathsf{L}, \mathsf{crs}$)
1 : $(\mathsf{k_{PRF}}, \mathsf{st_{aZKS}}) := \mathsf{st_{KT}}$
2 : $r := \mathsf{PRF.Eval}(\mathsf{k_{PRF}}, \mathsf{st_{aZKS}}||\mathsf{D}||\mathsf{L}||\mathsf{crs})$
3 : $(\pi^{\mathsf{upd}}_{\mathsf{aZKS}}, \mathsf{st}'_{\mathsf{aZKS}}) := \mathsf{aZKS.UpdateDS}(\mathsf{st_{aZKS}}, \mathsf{D}, \mathsf{L}, \mathsf{crs}; r)$
4 : $\mathsf{st}'_{\mathsf{KT}} := (\mathsf{k_{PRF}}, \mathsf{st}'_{\mathsf{aZKS}}); \mathsf{D}' := \mathsf{D}; \pi^{\mathsf{upd}}_{\mathsf{KT}} := \pi^{\mathsf{upd}}_{\mathsf{aZKS}}$
5 : for $(\mathsf{id}, \mathsf{k}) \in \mathsf{L}$ do
6 :     $\mathsf{D}'[\mathsf{id}, \widehat{\mathsf{v}}[\mathsf{id}]] := \mathsf{k}$
7 : return $(\mathsf{st}'_{\mathsf{KT}}, \mathsf{D}', \pi^{\mathsf{upd}}_{\mathsf{KT}})$

Audit($\mathsf{com_{KT}}, \mathsf{com}'_{\mathsf{KT}}, \pi^{\mathsf{upd}}_{\mathsf{KT}}, \mathsf{crs}$)
1 : $\mathsf{com_{aZKS}} := \mathsf{com_{KT}}; \mathsf{com}'_{\mathsf{aZKS}} := \mathsf{com}'_{\mathsf{KT}}; \pi^{\mathsf{upd}}_{\mathsf{aZKS}} := \pi^{\mathsf{upd}}_{\mathsf{KT}}$
2 : return $\mathsf{aZKS.VerifyUpd}(\mathsf{com_{aZKS}}, \mathsf{com}'_{\mathsf{aZKS}}, \pi^{\mathsf{upd}}_{\mathsf{aZKS}}, \mathsf{crs})$

SimSetup($1^\lambda$)
1 : return $(\mathsf{crs}, \tau) \leftarrow \mathsf{aZKS.SimSetup}(1^\lambda)$

SimQryKey($\mathsf{com_{KT}}, \mathsf{id}, \mathsf{v}, \mathsf{k}, \mathsf{crs}, \tau$)
1 : $\mathsf{com_{aZKS}} := \mathsf{com_{KT}}$
2 : return $\pi_{\mathsf{aZKS}} \leftarrow \mathsf{aZKS.SimQryKey}(\mathsf{com_{aZKS}}, \mathsf{id}||\mathsf{v}, \mathsf{k}, \mathsf{crs}, \tau)$

SimUpdateEpoch($\mathsf{com_{KT}}, \mathsf{com}'_{\mathsf{KT}}, \mathsf{L}, \mathsf{crs}, \tau$)
1 : $\mathsf{com_{aZKS}} := \mathsf{com_{KT}}; \mathsf{com}'_{\mathsf{aZKS}} \leftarrow \mathsf{com}'_{\mathsf{KT}}$
2 : return $\pi_{\mathsf{aZKS}} := \mathsf{aZKS.SimUpdateDS}(\mathsf{com_{aZKS}},$
                $\mathsf{com}'_{\mathsf{aZKS}}, \mathsf{L}, \mathsf{crs}, \tau)$
```

Figure 3: Our KT scheme instantiation building on the append-only zero-knowledge set from [9].

one would get from SEEMless [9] if one were to distill the presentation of their scheme via our methods. For instance, SEEMless uses an additional data structure called history tree to facilitate the reconstruction of the internal state of the service provider from any arbitrary epoch. This is to facilitate an additional operation called KeyHistoryQry, that allows a user to view the history of all their key updates to a system. While potentially useful, we omit this to target the core functionality desired by KT—delivering the latest key of one user to another user.

The lineage of papers following [9] keep the same trend observed above. That is, while similar to our KT scheme instantiation in Fig. 3 they extend our baseline scheme (and that of their previous works') to offer more functionality or to provide greater efficiency (sometimes at the expense of slightly relaxed security guarantees). For instance, Parakeet [28] notes that the construction in [9] is not scalable enough for large, real-world deployments. In response, they present: a more efficient VKD construction based on a primitive called an ordered zero-knowledge set (a generalization of the aZKS), a compaction mechanism that allows for the purging of obsolete entries, and a more efficient method for broadcasting commitments. Similarly, OPTIKS [25] improves on the simplicity and scalability of [28], while ELEKTRA [24] adds support for the setting in which users may register multiple devices with a particular service.

4. Verifiable Summary

As discussed in Section 1 current KT protocols suffer from a major scalability issue: namely, all query requests must be processed using the service provider's base KT scheme. As responding to key queries is the most frequent operation [7], it is sensible to focus on optimizing this operation. In a recent presentation concerning the WhatsApp KT deployment (based on the SEEMless protocol [9]) it was stated that their verifiable key directory has 50 billion nodes (implying an internal aoZKS depth of 34 and a username-key pair count in the billions). Moreover, the system collects over $150,000$ updates per epoch. At such a scale it is not possible to store the key directory in RAM and external storage layers must be used [23]. This means that to answer a query, it is possible that information needs to be retrieved off of hard-disk storage. This can make responding to the typical per-epoch query load prohibitively expensive.

An obvious method to reduce load on a single key directory and eliminate a single point of failure is to distribute the key transparency service. SEEMless [9] states: "We have described the service in terms of a single server, but in practice, the 'server' can be implemented using a distributed network of servers, for reliability and redundancy. Our model captures the server as a single logical entity, so it can accommodate a distributed implementation fairly easily." However, *is this actually true?* We contend that this is—in fact—not the case, due to theoretical as well as practical reasons. From the theoretical side, as eluded to before, it seems challenging to maintain synchronization among all parties in a distributed KT system. More practically, we feel that the apparent lack of deployed (or even proposed) distributed KT systems is a strong indicator that this is a non-trivial problem; especially considering the strong incentive for it.

Consider that unlike other distributed services, a consistency guarantee that all nodes will *eventually* (for some definition of eventually) receive and process all updates is not sufficient. To maintain correctness, inter-epoch consistency, and inclusivity, it is crucial that every server in a distributed KT service has the same exact state at the beginning of a new epoch. To ensure this is the case, and that a single consistent commitment is produced per epoch, service providers, like WhatsApp [23], use a single "write-path". That is, updates are processed by a single server and before a new epoch is triggered, the set of updates that will be included in the next update is sent to all "read-path" nodes in the distributed KT system. These nodes are then able to update their internal state with a consistent set of updates, and verifiably respond to queries when the next epoch is triggered with answers that are consistent with all other nodes in the system.

However, this distributed KT implementation still suffers from a number of issues. For one, it requires communication cost that is linear in the size of the updates to be sent to each distributed KT server. Further, each of these distributed nodes are required to have sufficient storage capacity to store the entire state of the service provider, and above all it does not alleviate the burden of processing queries using the expensive full KT scheme. Instead, consider the common design pattern of using a Bloom filter (BF) as a compact data summary to reduce the load on a data store that is expensive to access [36, 42, 14]. In the usual setting, the BF sits on top of a large key-value store and the BF contains all the keys in a store. A user first queries the key to the BF and if the answer is negative, the query's response is $\perp$. Only if the BF's answer is positive, the full data store is queried, whose response is then the definitive query response.

We propose a similar solution for solving the scalability issues of KT systems by outsourcing the task of processing key queries to multiple *summary nodes* controlled by the service provider. These can be stored on comparatively lightweight edge devices that essentially act as an approximate cache for the service provider's database; they store a compact list (in the form of a Bloom filter) of users that updated their key since the last epoch. Consider the case where a user id queries all their contacts' identifiers id' after an epoch update. Most of those contacts will not have updated their keys. Now, the user id will query some edge device with an identifier id to learn whether the user id' has updated their key. With high probability, the edge device will return a negative response, as the user id' has not updated their key. Hence, the user id does not need to query the service provider's central server. Only in the case where the edge device returns a positive response,[7] will the user id query the service provider's server.

As a first naive solution attempt, consider having the central update server publish a publicly available BF containing a list of all user identifiers that updated their key since the last epoch. Then users that have remained online between epochs can first query this Bloom filter, and only in the event of a positive response the query is forwarded to the full KT scheme. However, this clearly compromises privacy, because an adversary can now learn the identities of users that updated their key in a given epoch by a simple exhaustive attack. That is, without needing to query the service provider an adversary could build a set of users who likely updated their key in the last epoch by simply running local queries on the publicly available BF.

7. Note that Bloom filters have no false negatives.

To protect privacy, consider instead having the central update server create a secretly-keyed BF [15] and distributing them to the summary nodes. However, then a malicious service provider could provide inconsistent responses to different users (i.e. violate intra-epoch consistency in Definition 1). The reason is that—even if the underlying KT scheme is consistent—the user has no means of verifying whether the BF response is correct (as this would require the BF secret key).

Therefore, our complete solution is to use a verifiable version of a Bloom filter, where the hash functions are replaced by a verifiable random function (VRF) [35, 8]. In this way, the user can verify the correctness of the BF response.

Let us now formalize the notion of a verifiable Bloom filter (VBF).

Definition 3 (Verifiable Bloom Filter). *A verifiable Bloom filter (VBF) is a tuple of algorithms* $\mathsf{VBF} = (\mathsf{Gen}, \mathsf{Rep}, \mathsf{Qry}, \mathsf{Vfy})$ *defined as follows:*

- $\mathsf{Setup}(1^\lambda)$ *on input security parameter* λ*, outputs a common reference string* crs*,*
- $\mathsf{Gen}(1^\lambda, \mathsf{crs})$ *on input security parameter* λ *and CRS* crs*, outputs a verification key* vk *and an evaluation key* ek*,*
- $\mathsf{Rep}(\mathsf{ek}, \mathsf{D}, \mathsf{crs})$ *on input evaluation key* ek*, data* D *and CRS* crs*, outputs a representation* repr *and a commitment* com*,*
- $\mathsf{Qry}(\mathsf{ek}, \mathsf{repr}, x, \mathsf{crs})$ *on input evaluation key* ek*, representation* repr*, element* x*, and CRS* crs*, outputs a bit* b *indicating whether* x *is in the set represented by* repr*, and a proof* π*,*
- $\mathsf{Vfy}(\mathsf{vk}, \mathsf{com}, x, b, \pi, \mathsf{crs})$ *on input verification key* vk*, commitment* com*, element* x*, bit* b*, proof* π*, and CRS* crs*, outputs a bit indicating whether the proof is valid,*
- $\mathsf{SimSetup}(1^\lambda)$ *on input security parameter* λ*, outputs a simulated CRS* crs *and a trapdoor* τ*,*
- $\mathsf{SimQryKey}(\mathsf{com}, x, b, \mathsf{crs}, \tau)$ *on input commitment* com*, element* x*, bit* b*, CRS* crs*, and trapdoor* τ*, outputs a proof* π*.*

Rep *and* Qry *are deterministic. The algorithms are defined in Fig. 4.*

In our definition, the VBF commitment is simply the bit vector representation of the underlying BF. Moreover, for ease of exposition, we take it to be the case that the VRF outputs a k length vector of elements in $[m]$. That is the VRF output is equivalent to the normal output of the hash functions in a standard BF.

We omit a full formalization of the security properties of the VBF in the interest of readability and space. Instead, we will state the properties that a VBF achieves and provide high level arguments for why they hold.

Completeness. Given some query, any honestly generated answer produced by the VBF will verify. This follows from

```
$\mathsf{Setup}(1^\lambda)$
1 : $\mathbf{return}\ \mathsf{crs} := \mathsf{VRF.Setup}(1^\lambda)$

$\mathsf{Gen}(1^\lambda, \mathsf{crs})$
1 : $\mathbf{return}\ (\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{VRF.Gen}(1^\lambda, \mathsf{crs})$

$\mathsf{Rep}(\mathsf{ek}, \mathsf{D}, \mathsf{crs})$
1 : $\mathsf{repr} := 0^m$
2 : $\mathbf{for}\ x \in \mathsf{D}$
3 :     $y := \mathsf{VRF.Eval}(\mathsf{ek}, x, \mathsf{crs})$
4 :     $\mathbf{for}\ j \in [k] : \mathsf{repr}[y[j]] := 1$
5 : $\mathsf{com} := \mathsf{repr}$
6 : $\mathbf{return}\ (\mathsf{repr}, \mathsf{com})$

$\mathsf{Qry}(\mathsf{sk}, \mathsf{repr}, x, \mathsf{crs})$
1 : $(y, \pi_{\mathsf{VRF}}) := \mathsf{VRF.Eval}(\mathsf{sk}, x, \mathsf{crs})$
2 : $b := \bigwedge_{j \in [k]} \mathsf{repr}[y[j]]$
3 : $\pi := (y, \pi_{\mathsf{VRF}})$
4 : $\mathbf{return}\ (b, \pi)$

$\mathsf{Vfy}(\mathsf{vk}, \mathsf{com}, x, b, \pi, \mathsf{crs})$
1 : $\mathsf{vk}_{\mathsf{VRF}} := \mathsf{vk}; (y, \pi_{\mathsf{VRF}}) := \pi$
2 : $\mathbf{req}\ \ \mathsf{VRF.Vfy}(\mathsf{vk}_{\mathsf{VRF}}, x, \pi_{\mathsf{VRF}}, \mathsf{crs}) = 1$
3 : $\mathbf{if}\ b = 1\ \mathbf{req}\ \ \forall j \in [k] : b_j^1 = 1$
4 : $\mathbf{else\ req}\ \ \exists j \in [k] : b_j^0 = 1$
5 : $\mathbf{return}\ 1$

$\mathsf{SimSetup}(1^\lambda)$
1 : $\mathbf{return}\ (\mathsf{crs}, \tau) \leftarrow \mathsf{VRF.SimSetup}(1^\lambda)$

$\mathsf{SimQryKey}(\mathsf{com}, x, y, \mathsf{crs}, \tau)$
1 : $\mathbf{return}\ \pi \leftarrow \mathsf{VRF.SimEval}(\mathsf{com}, x, y, \mathsf{crs}, \tau)$
```

Figure 4: Algorithms for a verifiable Bloom filter (VBF). VRF is a VRF.

the perfect correctness of the underlying VRF (as is typical for traditional VRF constructions).

Correctness. The VBF (like a standard BF) does not permit false negatives and has a false positive rate essentially equivalent to that of the underlying BF. This follows from the fact that the VRF used in the construction essentially provides a *random mapping* of the represented collection to set bits in the filter.

Verifiability. The VBF guarantees that (for a binding CRS) there exists only one (deterministically generated) answer for a given query that will verify. This follows from the perfect unique provability of the underlying VRF.

Privacy. The leakage of a VBF is equivalent to at most the count of the unique elements that the underlying BF represents. This result follows from the analysis of the secretly-keyed BF construction in [16, 11].

Simulatability. The VBF is simulatable, meaning that arbitrary proofs can be generated for any query response using the trapdoor of a CRS in simulation mode. This follows from the simulatability of the underlying VRF.

5. Key Transparency Protocol

In this section we describe our scalable KT protocol in Fig. 5 realizing the functionality $\mathcal{F}_{\mathsf{KT}}$. To model real-world application scenarios as closely as possible we make several assumptions:

- Each user has a unique (permanent) identifier id (typically a phone number or username).
- There exists an (asymmetric) public-key infrastructure (PKI). More concretely, the service provider 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 service provider. This would typically be realized by a key-exchange via two-factor authentication such as SMS or an authenticator app.
- The service provider can (infrequently) broadcast messages (small epoch commitments) to all users. The mechanism for this would typically entail the service provider 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 [4, 38], gossip protocols [32], and a novel light-weight "consensusless" consistency protocol described in [28].

We employ our VBF to reduce the need to use the full base KT scheme to resolve queries. Under two conditions a user can use the VBF as their primary mechanism for resolving a query. The first being that a user has remained online between successive epochs. This means they are connected to the network and allowing the KT service to run in the background of their device. Secondly, they are querying for a key for a user they have previously retrieved some version of from the KT scheme. That is, they are checking for an updated key for a user in their contacts or checking on an update (or lack there of) of their own key. Then, when a user U_{id} performs a key query for an identified id', it first queries the VBF in line 2. Only, if the response $b = 1$ is positive—the user $\mathsf{U}_{\mathsf{id}'}$ has updated their key in the last epoch (or a rare false positive occurs)—the user U_{id} queries the full aZKS in line 6. As a key update or a false positive (one can make this an arbitrarily small probability by tuning the VBF parameters) will be sufficiently rare events we contend that the large majority of queries will be resolved solely using the VBF.

Again, in the case a user has been offline for a period they will need to catch up by directly querying the aZKS. We consider ways in which this requirement can be softened in Section 5.1. Further, if a user is requesting the key of a user they have never interacted with (i.e., they are getting the key of some for the first time), then this query needs to be resolved with the underlying KT scheme. For readability, we do not specify these cases in Fig. 5. Moreover, to prevent additional notational overhead, in our protocol the VBF queries are technically answered by the service provider. We emphasize that in practice the VBF queries could be answered by an edge device that holds the service provider's VBF secret key, however in the most common case the

```
$\mathsf{S}(1^\lambda)$
    $(\mathsf{vk_{KT}}, \mathsf{st_{KT}}) \leftarrow \mathsf{KT.Setup}(1^\lambda)$
    $(\mathsf{S.vk_{VBF}}, \mathsf{S.sk_{VBF}}) \leftarrow \mathsf{VBF.Gen}(1^\lambda)$
    $\mathsf{S.st} := [\mathsf{st_{KT}}]; \mathsf{S}.\boldsymbol{\tau} := 0; \mathsf{S.L}_0 := []; \mathsf{S.v} := []$
    broadcast $\mathsf{S.vk_{VBF}}$

$\mathsf{U_{id}}(\mathsf{vk_{VBF}} \leftarrow \mathsf{S})$
    $\mathsf{U_{id}.vk_{VBF}} := \mathsf{vk_{VBF}}; \mathsf{U_{id}.C} := [];$

$\mathsf{S}(\textbf{EpochUpdate} \leftarrow \mathcal{Z})$
    $\tau := \mathsf{S}.\boldsymbol{\tau}; \mathsf{S}.\boldsymbol{\tau} := \tau + 1; \mathcal{I} := (\mathsf{id})_{(\mathsf{id},\mathsf{k}) \in \mathsf{S.L}_\tau}; \mathsf{S.L}_{\tau+1} := []$
    $(\mathsf{S.st_{KT}}[\tau+1], \mathsf{S.D}_{\tau+1}, \pi^{\mathsf{upd}}_{\mathsf{KT}})$
        $\leftarrow \mathsf{KT.UpdateEpoch}(\mathsf{S.st_{KT}}[\tau], \mathsf{S.D}_\tau, \mathsf{S.L}_\tau)$
    $\mathsf{com}^{\tau+1}_{\mathsf{KT}} := \mathsf{KT.Commit}(\mathsf{S.st_{KT}}[\tau+1], \mathsf{S.D}_{\tau+1})$
    for $(\mathsf{id}, \mathsf{k}) \in \mathsf{S.L}_\tau$ do $\mathsf{S.v}[\mathsf{id}] := \mathsf{S.v}[\mathsf{id}] + 1$
    $(\mathsf{S.repr}^{\tau+1}_{\mathsf{VBF}}, \mathsf{S.com}^{\tau+1}_{\mathsf{VBF}}) := \mathsf{VBF.Rep}(\mathsf{S.sk_{VBF}}, \mathcal{I})$
    broadcast $(\textbf{EpochUpdate}, \mathsf{com}^{\tau+1}_{\mathsf{KT}}, \pi^{\mathsf{upd}}_{\mathsf{KT}})$

$\mathsf{U_{id}}(\textbf{EpochUpdate}, \mathsf{com}^{\tau+1}_{\mathsf{KT}}, \pi^{\mathsf{upd}}_{\mathsf{KT}} \leftarrow \mathsf{S})$
    $\tau' := \mathsf{U_{id}}.\boldsymbol{\tau}$
    req $\mathsf{KT.Audit}(\mathsf{U_{id}.com}^{\tau'}_{\mathsf{KT}}, \mathsf{com}^{\tau+1}_{\mathsf{KT}}, \pi_{\tau+1}) = 1$
    $\mathsf{U_{id}}.\boldsymbol{\tau} := \tau' + 1; \mathsf{U_{id}.com}^{\tau'+1}_{\mathsf{KT}} := \mathsf{com}^{\tau+1}_{\mathsf{KT}}$

$\mathsf{U_{id}}(\textbf{RegisterKey}, \mathsf{k} \leftarrow \mathcal{Z})$
    send $(\textbf{RegisterKey}, \mathsf{id}, \mathsf{k}) \rightarrow \mathsf{S}$

$\mathsf{S}(\textbf{RegisterKey}, \mathsf{k} \leftarrow \mathsf{U_{id}})$
    $\tau := \mathsf{S}.\boldsymbol{\tau}; \mathsf{S.L}_\tau[\mathsf{id}] := \mathsf{k}$

$\mathsf{U_{id}}(\textbf{QueryKey}, \mathsf{id}' \leftarrow \mathcal{Z})$
    $(\widetilde{\mathsf{k}}, \widetilde{\mathsf{v}}) := \mathsf{U_{id}.L}[\mathsf{id}']; \tau := \mathsf{U_{id}}.\boldsymbol{\tau}; (\tau_{\mathsf{last}}, \mathsf{k_{last}}) := \mathsf{U_{id}.C}[\mathsf{id}']$
    if $\tau_{\mathsf{last}} = \tau - 1$ send $(\textbf{QueryVBF}, \mathsf{id}', \tau) \rightarrow \mathsf{S}$
        receive $(\textbf{VBFResponse}, \mathsf{id}', b, \pi_{\mathsf{VBF}}) \leftarrow \mathsf{S}$
        if $b = 0 \land \mathsf{VBF.Vfy}(\mathsf{U_{id}.vk_{VBF}}, \mathsf{id}', b, \pi_{\mathsf{VBF}}) = 1$
            return $(\textbf{QueryResponse}, \tau, \mathsf{id}', \mathsf{k_{last}})$
    send $(\textbf{QueryKey}, \mathsf{id}') \rightarrow \mathsf{S}$
    receive $(\textbf{QueryResponse}, \mathsf{id}', \mathsf{k}, \mathsf{v}, \pi_{\mathsf{v}}, \pi_{\mathsf{v}+1}) \leftarrow \mathsf{S}$
    req $\mathsf{k} \neq \bot \lor \mathsf{v} = 0$   / ensure most recent key
    req $\mathsf{KT.VfyQry}(\mathsf{U_{id}.com}^{\tau}_{\mathsf{KT}}, \mathsf{id}', \mathsf{v}, \mathsf{k}, \pi_{\mathsf{v}}) = 1$
    req $\mathsf{KT.VfyQry}(\mathsf{U_{id}.com}^{\tau}_{\mathsf{KT}}, \mathsf{id}', \mathsf{v}, \bot, \pi_{\mathsf{v}+1}) = 1$
    $\mathsf{U_{id}.C}[\mathsf{id}'] := (\tau, \mathsf{k})$
    return $(\textbf{QueryResponse}, \tau, \mathsf{id}', \mathsf{k})$

$\mathsf{S}(\textbf{QueryVBF}, \mathsf{id}', \tau \leftarrow \mathsf{U_{id}})$
    $(\mathsf{k}, \pi_{\mathsf{VBF}}) := \mathsf{VBF.Qry}(\mathsf{S.sk_{VBF}}, \mathsf{S.repr}^{\tau}_{\mathsf{VBF}}, \mathsf{id}')$
    send $(\textbf{VBFResponse}, \mathsf{id}', \mathsf{k}, \pi_{\mathsf{VBF}}) \rightarrow \mathsf{U_{id}}$

$\mathsf{S}(\textbf{QueryKey}, \mathsf{id}', \tau \leftarrow \mathsf{U_{id}})$
    $\mathsf{v} := \mathsf{S.v}[\mathsf{id}']$
    $(\mathsf{k_v}, \pi_{\mathsf{v}}) := \mathsf{KT.QryKey}(\mathsf{S.st_{KT}}[\tau], \mathsf{id}', \mathsf{v})$
    $(\mathsf{k}_{\mathsf{v}+1}, \pi_{\mathsf{v}+1}) := \mathsf{KT.QryKey}(\mathsf{S.st_{KT}}[\tau], \mathsf{id}', \mathsf{v}+1)$
    send $(\textbf{QueryResponse}, \mathsf{id}', \mathsf{k_v}, \mathsf{v}, \pi_{\mathsf{v}}, \pi_{\mathsf{v}+1}) \rightarrow \mathsf{U_{id}}$
```

Figure 5: Our KT protocol π_{KT}.

VBFs will reside on light-weight nodes operated by the service provider themselves. To be clear about the security guarantees, we remark that there is no privacy guarantee against a semi-honest edge device, since the edge device holds the VBF secret key (the same guarantees the service provider has).

Theorem 1. *The protocol π_{KT} securely realizes the functionality $\mathcal{F}_{\mathsf{KT}}$ in the common reference string model.*[8]

Proof Sketch: We omit a full proof as it is analogous to the proof in [9]. Formally, we can proceed by a sequence of hybrid games that are consecutively indistinguishable. Starting from the real game we first switch the CRS to simulation mode, so that soundness of all proof no longer holds. Then we can replace honestly generated proof for VRF evaluations (in VBF and aZKS) with simulated proofs. Now, with simulated proofs, we can switch the VRF images from honestly generated to random. Now, we are in the ideal game.

The only difference is that in our protocol we also give our VBF evaluations. However, these can be simulated in an analogous fashion as the aZKS proofs—given that both are built on the same VRF. □

5.1. Practical Considerations

Observe that our protocol π_{KT} is highly modular in the way that it integrates our VBF to provide scalability. That is, the algorithms (and subroutines of algorithms) specifying all VBF operations could be omitted, and we would still have a complete KT protocol that satisfies the functionality $\mathcal{F}_{\mathsf{KT}}$. This is intentional. Recall, that in Section 3.1 we said the scheme instantiation in Fig. 3 represents the minimal components necessary to satisfy Definition 1 and its corresponding correctness and security problems. We then remarked that the scheme instantiations put forth in [9, 28, 25, 24] are super-sets of the one we provide. That is, one could use our VBF summary mechanism on top of *any* KT protocol. In case it uses a VRF-based aZKS-like scheme instantiation[9] our summary mechanism is essentially free. This is because the inputs to the VBF summary are simply the VRF images of the user's identifier and the key version (id||v), which already needs to be computed as part of the base KT scheme (for all schemes we consider). Therefore, with minimal additional computation one can produce our VBF summary. For instance if a service provider was especially concerned with efficiency in the multi-device setting they could implement ELEKTRA [24] with our VBF summary mechanism on top of it to achieve bleeding-edge scalability. We explore this exact scenario in Section 6 where we use Meta's AKD library [22] as our base KT scheme to implement our protocol and explore the expected efficiency gains achieved by using a VBF on top of it.

8. In this model, we also assume that the service provider can publish short messages, i.e., technically it has access to a broadcast functionality.

9. As is the case for all protocols we refer to in this work.

Further, recall the requirement that a user remain online between epochs to be able to query the VBF as their primary query resolver in our KT protocol. In practice this is not a big hinderance, as most users will have the KT-enabled E2EE messaging app installed with permissions to run in the background on an always-on mobile device with a near constant network connection. However, it would be beneficial if a momentary network disconnection did not necessitate that a user resynochronize with the full base KT protocol before being able to use the VBF again. To solve this, one could easily modify our KT protocol to have the light-weight edge device store the VBFs that correspond to the past (say) t epochs. That is in addition to the current VBF for epoch τ, the edge-device can additionally store the VBF summaries for epochs $\tau-1, \tau-2, \ldots, \tau-t-1$. Then, if a user experiences a short duration of network disconnection and comes back online (a disconnection period that occurs strictly between epochs τ and $\tau - t - 1$), they could linearly query every VBF during the period in which they were disconnected. This allows users to efficiently check for updates from users in their contacts and to check to ensure that no unauthorized modification of their key took place primarily at the level of the VBF. In this way, this extended mechanism allows for users to "catch up" with the current state of the KT protocol without needing to resolve the high majority of queries using the full base KT scheme.

Lastly, we explore an additional efficiency gain by using a cache along with our VBF to speed up query responses from the light-weight edge devices. The most expensive subroutine when a VBF responds to a query is computing the VRF evaluation. Therefore, at instantiation time the service provider could also build a cache for the VBF containing $(\mathsf{id}||\mathsf{v})$ inputs mapped to their corresponding $(y, \pi_{\mathsf{VRF}}) \coloneqq \mathsf{VRF.Eval}(\mathsf{sk}, x, \mathsf{crs})$ output. This cache stores pre-computations of the VRF evaluations for all keys that have been updated during a given epoch. As resources allow, an edge device could add to this cache for queries that are made to the device that were not part of this initial set during a given epoch. Thus, if a query that appears outside of the update set is also repeated its VRF evaluation can be retrieved from the cache. While this modification requires a one-time additional communication overhead with the edge devices, it greatly speeds up query responses to the clients. We explore this concretely in Section 6.

To summarize, our VBF powered KT protocol is beneficial for both the service provider and users. From the service provider's perspective it enables inexpensive and flexible scalability. Unlike current distributed KT protocols, like that of WhatsApp [23], it does not require that the distributed nodes be able to maintain the full state of the service provider. Instead, each light-weight node maintains a summary of the most recent changes to state of the service provider. This empowers service providers to build a comparatively low cost and geographically dense network of edge devices, that, in turn, greatly reduce the number of queries that need to be resolved using a high-cost and resource-intensive server storing the full state of the service provider. Likewise, for users most responses can be handled primarily at the summary level rather than at the base KT scheme level. Moreover, as summaries can be deployed on light-weight edge devices service providers can put more of them online, and have them be physically closer to users on the network. This results in an improved user experience where KT queries are resolved much faster than the current state-of-the-art solutions. We provide experimental evidence for this in the following section.

6. Implementation and Experiments

We implement our scalable KT protocol in Rust using Meta's AKD library [22] for our base KT scheme. Their implementation is based on [9] and its used to implement the WhatsApp KT protocol [23]. We also provide an implementation of our VBF summary mechanism. Our code is available on GitHub.[10] We provide standalone performance numbers comparing our VBF versus the standalone base KT scheme. We also simulate a large-scale KT deployment to demonstrate how our verifiable summary mechanism greatly aids in scalability of KT protocols. All experiments were run a server with a 10-core Intel Xeon E5-2630 CPU clocked at 2.20 GHz with 128 GB of RAM.

6.1. VBF and Base KT Scheme Performance Comparison

In Table 1 we compare the time to process updates between the VBF summary mechanism versus the base KT scheme (i.e., the underlying aZKS implemented by [22]). Specifically, we compare the time to process $2^{13}, 2^{15}$, and 2^{17} updates between the VBF with cache, the standard VBF, an empty aZKS, and a pre-populated aZKS (with a pre-existing 2^{19} entries). Entries were created by generating a random user identifier and a corresponding random public key value. For each insertion set size we ran 100 trials and report the average. As can be seen from Table 1, the VBF with cache takes about 1.5 times longer to insert the same number of entries as compared to the standard VBF as it also builds the cache at insertion time. Inserting into an empty aZKS is 2 to 3 times faster than inserting into a pre-populated aZKS. We note that our test implementation does not have the ability to use an external storage layer, so our aZKS exist solely in RAM. This is an unrealistic assumption for large scale, real-world deployments with massive user bases. Therefore, we expect the insertion time on the aZKS to *substantially* increase when they are holding hundreds of millions or billions of entries, as they must retrieve information off hard disk storage. As approximately 2^{17} entries per epoch are the current needs of WhatsApp, we argue that building a VBF is a negligible cost (≈ 2 seconds to fully build from scratch) as compared to updating the full aZKS. Further, recall that the VRF output that needs to be computed on the concatenation of each user identifier and key version to be added to the aZKS is the same input to the

10. Our code will be made publicly available after the anonymized review process.

Number of Updates	VBF with cache (s)	VBF (s)	Empty aZKS (s)	Pre-populated aZKS (s)
2^{13}	0.15	0.10	0.10	0.33
2^{15}	0.61	0.40	0.45	1.13
2^{17}	2.44	1.66	2.04	5.16

Table 1: Benchmarks for VBF with cache, VBF, (both with $\epsilon = 0.01$), empty aZKS, and pre-populated aZKS (with 2^{19} entries) insertion times measured in seconds.

Number of Queries	VBF with cache (s)	VBF (s)	aZKS (s)
2^{13}	0.31	1.82	14.93
2^{15}	1.21	7.29	60.75
2^{17}	4.89	29.19	245.53

Table 2: Benchmarks for VBF with cache, VBF, (both with $\epsilon = 0.01$) and aZKS query times measured in seconds.

underlying Bloom filter of the VBF. As computing the VRF is the majority of the computation cost of building the VBF the additional expense becomes nearly zero—in practice, just a relatively small number of hash computations and writes to a bit-array.

In Table 2 we compare the time to process queries between the VBF summary mechanism versus the base KT scheme. Specifically, we compare the time to process $2^{13}, 2^{15}$, and 2^{17} simultaneous queries between the VBF with cache, the standard VBF, and the aZKS. The VBF with cache and the standard VBF were pre-populated with 2^{17} entries and the aZKS was pre-populated with 2^{20} entries (replicating a realistic deployment large-scale deployment scenario). Queries were randomly selected user identifiers from the space of randomly generated user identifiers. For each query set size we ran 10 trials and report the average. As can be seen from the table, the VBF with cache is about 6 times faster to answer the same number of queries as compared to the standard VBF. The aZKS is a full order of magnitude (or more) slower than both variants of the VBF when handling the same number of queries. Again, we stress that this is for an aZKS that exists entirely in RAM. When external storage layers must be used (as the size of the aZKS grows) this difference in performance will become even more drastic. We also measure the average verification time for a query to the VBF and aZKS (both containing 2^{17} pre-populated entries). These values were 218.74 μs and 979.77 μs for the VBF and aZKS, respectively—an almost 5 times computation reduction for the user verifying a VBF query response. The practical implication of this being that our mechanism enhances user experience by providing much faster resolution to queries and verification of those queries.

6.2. Simulation of Large Scale Deployment

We start by comparing the relative size of a per-epoch VBF and the full aZKS that needs to be stored for a large scale KT deployment. Taking the average number of updates made to WhatsApp at $150,000$ per epoch and a targeted false positive rate of $\epsilon = 0.01$ we arrive at a VBF representation (and commitment) that is approx. 180 KB using the analysis in Appendix B. Using results from [28] the aZKS construction in SEEMless [9] would require 27 TB of storage, while the more optimized construction of Parakeet [28] would require 2.2 TB of storage to facilitate the current size of WhatsApp's user base. Moreover, per epoch our protocol only requires transmitting the VBF to each edge device, versus the full update list if one to distribute the entire state of the service provider.[11] Lastly, note that the additional data a user needs to download is small—only the VBF bit-array representation, less than 200 KB.

Recall that the additional overhead for producing a per-epoch VBF is negligible from the viewpoint of the service provider. Therefore, we focus our simulation on the concrete speed up of resolving users queries. We will say that the computation time to evaluate a VBF query is q. Extrapolating from the results in Section 6.1, we will *conservatively* say the computation time to evaluate a query using the full aZKS is $10q$. We say conservatively, because, again, our results do not reflect the real-world condition that the aZKS is stored on hard-disk storage. We will now compare the total query time Q_b for the simulated base KT scheme and Q_s for our simulated scalable KT scheme with u total users, fraction o of which have remained online between epochs, each making n queries per epoch, where fraction a of those queries are for users who have updated their key and fraction b is for queries of users they have not previously connected with. For our experiment we vary u while setting $o = 0.9$, $n = 100$, $a = b = 0.01$. Moreover, we fix the false positive rate for the VBF to be $\epsilon = 0.01$. We include our simulate script with our code, so that others can vary these parameters and obtain results that match the conditions of their particular KT deployment.

In Fig. 6 we compare the total query times Q_b when using only the base aZKS KT scheme and Q_s when using our scalable scheme (the VBF in conjunction with an aZKS), across a range of users u from 50 million to 2 billion. The results suggest that our scalable protocol can significantly decrease the query computation time in large-scale KT deployments. Our combined VBF and aZKS approach leverages the strengths of both structures: the VBF efficiently handles the bulk of queries with minimal overhead, while the aZKS provides accurate responses for cases where the VBF may be insufficient (e.g., false positives, offline users, updated keys). As shown in the plot, the use of a VBF reduces the total query time by approximately 64%

11. If one were to use the VBF with cache then the required transmission size to the distributed nodes per-epoch is about equal.

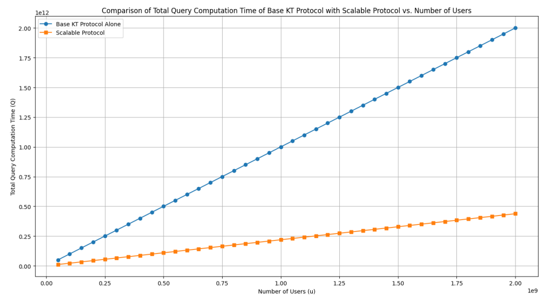

Figure 6: Comparison of total query computation time Q_b of using the base KT scheme alone and total query computation time Q_s of our scalable scheme versus the number of users u in the system. Note the x-axis scale is in billions and the y-axis scale is in trillions.

compared to using an aZKS alone when the user base reaches 2 billion. This substantial reduction can lead to improved user experience and reduced infrastructure costs. Note that these results do take into consideration transport time on the network. However, one can imagine that users will be geographically closer to VBF edge nodes, and thus have a faster connection when interacting with them versus the full aZKS maintained by the service provider. Further, recall that verifying a VBF query response is five times less total computation time for the user. In sum, our scalable KT protocol reduces the total query computation cost for the service provider, while also improving the speed at which queries are resolved and verified for users.

6.3. A Note On Meta's AKD Library

VRFs that do not assume a trusted setup (e.g. *not* in the CRS model) are typically unconditionally uniquely provable, and thus unconditionally non-simulatable. We observe that the VRF implementation in Meta's AKD library [22] does not have a CRS for setup and thus is *not simulatable*. However, both our proof and the proof of [9] rely crucially on the simulatability of the VRF. Thus, the formal security of protocols that rely on uniquely provable VRF (in particular, Meta's AKD library) is unclear.[12] Given that WhatsApp's KT protocol is based on Meta's AKD library, this means that WhatsApp does not enjoy formal privacy guarantees. To be clear, this does not constitute a security vulnerability, but rather a lack of formal security guarantees. We also note that as our experimental protocol implementation was built on top of the same library (and in particular the same VRF), it also lacks these formal guarantees.

7. Conclusion

This work advances the field of key transparency (KT) by addressing two fundamental challenges: the lack of a formal security model and the scalability limitations in current KT solutions. We present the first cryptographically sound formalization of KT as an ideal functionality, defining essential security properties and assumptions that provide a foundation for analyzing and improving KT protocols. Through this formalization, we also identify a potential security vulnerability – specifically, an impersonation risk posed by malicious service providers – and propose a backward-compatible mitigation strategy to protect against such threats.

To tackle the scalability issue, we introduce a novel cryptographic primitive, the Verifiable Bloom Filter (VBF), which enables distributed, privacy-preserving query handling without compromising the security guarantees required by KT systems. By offloading the bulk of query processing to light-weight edge devices, the VBF not only reduces the computational load on central servers but also ensures real-time response capabilities that can support large-scale deployments. Our VBF-based approach is modular, allowing for integration with existing KT protocols to enhance flexibility and performance without extensive architectural changes.

Our experimental results demonstrate the practical effectiveness of the VBF, highlighting its potential to significantly reduce query processing time and improve system throughput. By providing a scalable, secure, and adaptable solution for KT, this work lays the groundwork for future advancements in decentralized key management and aligns with ongoing efforts to standardize KT systems within the IETF. Future work will focus on further formalizing the VBF and a family of related primitives, in addition to exploring additional optimizations for real-world deployment. With this foundation, KT systems are poised to become more resilient, scalable, and adaptable to the demands of modern, secure communication platforms.

12. Even worse, security cannot be shown using known simulation-based techniques.

References

[1] Apple Security Engineering and Architecture (SEAR). *iMessage Contact Key Verification*. https://security.apple.com/blog/imessage-contact-key-verification. Accessed: 2023-04-01. 2023.

[2] B. H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[3] J. Blum et al. "Zoom Cryptography Whitepaper". In: (2022).

[4] J. Bonneau. "EthIKS: Using Ethereum to audit a CONIKS key transparency log". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 95–105.

[5] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, October 2001, pp. 136–145.

[6] R. Canetti and M. Fischlin. "Universally Composable Commitments". In: *CRYPTO 2001*. Ed. by J. Kilian. Vol. 2139. LNCS. Springer, Berlin, Heidelberg, August 2001, pp. 19–40.

[7] M. Chase. *Key Transparency: Introduction, Recent Results, and Open Problems*. Real World Cryptography 2024. 2024.

[8] M. Chase and A. Lysyanskaya. “Simulatable VRFs with Applications to Multi-theorem NIZK”. In: *CRYPTO 2007*. Ed. by A. Menezes. Vol. 4622. LNCS. Springer, Berlin, Heidelberg, August 2007, pp. 303–322.

[9] M. Chase et al. “Seemless: Secure end-to-end encrypted messaging with less trust”. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 1639–1656.

[10] B. Chen et al. “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*. Springer. 2022, pp. 547–580.

[11] D. Clayton, C. Patton, and T. Shrimpton. “Probabilistic Data Structures in Adversarial Environments”. In: *ACM SIGSAC CCS*. 2019.

[12] W. Diffie and M. E. Hellman. “New Directions in Cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[13] D. Evans, V. Kolesnikov, and M. Rosulek. “A Pragmatic Introduction to Secure Multi-Party Computation”. In: *Foundations and Trends® in Privacy and Security* 2.2-3 (2018), pp. 70–246.

[14] L. Fan et al. “Summary cache: a scalable wide-area web cache sharing protocol”. In: *IEEE/ACM transactions on networking* 8.3 (2000), pp. 281–293.

[15] M. Filić et al. “Adversarial Correctness and Privacy for Probabilistic Data Structures”. en. In: *CCS ’22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, November 2022, pp. 1037–1050.

[16] M. Filić et al. “Adversarial correctness and privacy for probabilistic data structures”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 1037–1050.

[17] T. Göbel and D. Huigens. “Proton Key Transparency Whitepaper”. In: (2024).

[18] O. Goldreich, S. Goldwasser, and S. Micali. “How to Construct Random Functions”. In: *Journal of the ACM* 33.4 (October 1986), pp. 792–807.

[19] O. Goldreich, S. Goldwasser, and S. Micali. “How to construct random functions”. In: *Journal of the ACM (JACM)* 33.4 (1986), pp. 792–807.

[20] Y. Hu et al. “Merkle 2: A low-latency transparency log system”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 285–303.

[21] R. Hurst and G. Belvin. *google/keytransparency*. https://github.com/google/keytransparency/. 2020.

[22] S. Lawlor and K. Lewi. *akd*. https://github.com/facebook/akd. Version v0.11.0. 2024.

[23] S. Lawlor and K. Lewi. *WhatsApp Key Transparency*. Real World Cryptography 2024. 2024.

[24] J. Len et al. “ELEKTRA: Efficient Lightweight multi-dEvice Key TRAnsparency”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 2915–2929.

[25] J. Len et al. “OPTIKS: An Optimized Key Transparency System”. In: *Cryptology ePrint Archive* (2023).

[26] K. Lewi. “WhatsApp Key Transparency”. In: *Proceedings of the 2023 USENIX Conference on Privacy Engineering Practice and Respect (PEPR ’23)*. Santa Clara, CA, 2023.

[27] F. Linker and D. A. Basin. “SOAP: A Social Authentication Protocol”. In: *USENIX Security 2024*. Ed. by D. Balzarotti and W. Xu. USENIX Association, August 2024.

[28] H. Malvai et al. “Parakeet: Practical key transparency for end-to-end encrypted messaging”. In: *NDSS Symposium 2023* (2023).

[29] A. Marcedone. *Key Transparency at Keybase and Zoom*. Presentation at IETF 116 Meeting. https://datatracker.ietf.org/meeting/116/materials/slides-116-keytrans-keybase-and-zoom-00.pdf. March 2023.

[30] B. McMillion. *Key Transparency Architecture*. https://www.ietf.org/archive/id/draft-ietf-keytrans-architecture-01.html. 2024.

[31] B. McMillion. *Key Transparency Architecture*. Active Internet-Draft. https://datatracker.ietf.org/doc/draft-ietf-keytrans-architecture/. August 2024.

[32] S. Meiklejohn et al. “Think global, act local: Gossip and client audits in verifiable data structures”. In: *arXiv preprint arXiv:2011.04551* (2020).

[33] M. S. Melara et al. “{CONIKS}: Bringing key transparency to end users”. In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 383–398.

[34] S. Micali, M. O. Rabin, and J. Kilian. “Zero-Knowledge Sets”. In: *44th FOCS*. IEEE Computer Society Press, October 2003, pp. 80–91.

[35] S. Micali, M. O. Rabin, and S. P. Vadhan. “Verifiable Random Functions”. In: *40th FOCS*. IEEE Computer Society Press, October 1999, pp. 120–130.

[36] Redis. *What is a Bloom Filter?* July 2022.

[37] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. “Theory and practice of bloom filters for distributed systems”. In: *IEEE Communications Surveys & Tutorials* 14.1 (2011), pp. 131–155.

[38] A. Tomescu and S. Devadas. “Catena: Efficient non-equivocation via bitcoin”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 393–409.

[39] A. Tomescu et al. “Transparency logs via append-only authenticated dictionaries”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1299–1316.

[40] N. Tyagi et al. “VeRSA: Verifiable registries with efficient client audits from RSA authenticated dictionaries”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 2793–2807.

[41] I. Tzialla et al. “Transparency dictionaries with succinct proofs of correct operation”. In: *Cryptology ePrint Archive* (2021).

[42] M. Wright. *Bloom Filters and Cuckoo Filters for Cache Summarization*. January 2023.

[43] A. C.-C. Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: *27th FOCS*. IEEE Computer Society Press, October 1986, pp. 162–167.

[44] A. C.-C. Yao. “Protocols for Secure Computations (Extended Abstract)”. In: *23rd FOCS*. IEEE Computer Society Press, November 1982, pp. 160–164.

Appendix

1. Append-Only Zero-Knowledge Set

Definition 4 (Append-Only Zero-Knowledge Set [34, 9]). *An append-only zero-knowledge set (aZKS)* aZKS *is a tuple of algorithms* $\mathsf{aZKS} = (\mathsf{CommitDS}, \mathsf{Qry}, \mathsf{Vfy}, \mathsf{UpdateDS}, \mathsf{VerifyUpd})$ *defined as follows:*

- $\mathsf{Setup}(1^\lambda)$ *on input security parameter* λ*, outputs a common reference string* crs,

- $\mathsf{CommitDS}(1^\lambda, \mathsf{D}, \mathsf{crs})$ *on input security parameter* λ*, database* D*, and CRS* crs*, outputs a commitment* com *and a state* st*,*
- $\mathsf{Qry}(\mathsf{st}, \mathsf{D}, \mathsf{l}, \mathsf{crs})$ *on input state* st*, database* D*, label* l *and CRS* crs*, outputs a value* y *and a proof* π*,*
- $\mathsf{Vfy}(\mathsf{com}, \mathsf{l}, y, \pi, \mathsf{crs})$ *on input commitment* com*, label* l*, value* y*, proof* π*, and CRS* crs*, outputs a bit indicating the validity of the proof,*
- $\mathsf{UpdateDS}(\mathsf{st}, \mathsf{D}, \mathsf{L}, \mathsf{crs})$ *on input state* st*, database* D*, update list* L*, and CRS* crs*, outputs a commitment* com'*, a state* st'*, a database* D'*, and a proof* π*,*
- $\mathsf{VerifyUpd}(\mathsf{com}, \mathsf{com}', \pi, \mathsf{crs})$ *on input commitments* $\mathsf{com}, \mathsf{com}'$*, proof* π*, and CRS* crs*, outputs a bit indicating the validity of the proof.*
- $\mathsf{SimSetup}(1^\lambda)$ *on input security parameter* λ*, outputs a simulated CRS* crs *and a trapdoor* τ*,*
- $\mathsf{SimQryKey}(\mathsf{com}, \mathsf{l}, y, \mathsf{crs}, \tau)$ *on input commitment* com*, label* l*, value* y*, CRS* crs*, and trapdoor* τ*, outputs a proof* π*,*
- $\mathsf{SimUpdateDS}(\mathsf{com}, \mathsf{com}', \mathsf{L}, \mathsf{crs}, \tau)$ *on input commitments* $\mathsf{com}, \mathsf{com}'$*, update list* L*, CRS* crs*, and trapdoor* τ*, outputs a proof* π*.*

To avoid the random oracle model, we modify the aZKS definition of [9] to explicitly use a CRS. We refer to [9] for the security guarantees of an aZKS. Chase et al. [9] 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.

2. Bloom Filter

We give an algorithmic description of the Bloom filter in Fig. 7. Recall, that the false-positive rate of a BF is a function of the bit-array size (m), the number of hash functions (k), and the size of the underlying set. Moreover, given a targeted false-positive rate ϵ one can determine the optimal BF parameters. From the analysis in [37], we have that for targeted false-positive rate ϵ and set size n: $m = \lceil -\frac{n \ln(\epsilon)}{\ln(2)^2} \rceil$ and $k = \lceil -\frac{\ln(\epsilon)}{\ln(2)} \rceil$. That is, for fixed ϵ (we set $\epsilon = 0.01$ as a default for our protocol), as the size of the underlying set n grows the bit-array used to represent the BF grows. Further, as you set ϵ closer to 0 the bit-array also grows and the number of hash functions used internally by the filter increase.

3. Verifiable Random Function Definition

As Chase et al. [9]—we require a *simulatable* VRF [8]. This is crucial for the formal security proof of our KT protocol (as well as the protocol of [9]).

Definition 5 (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* $y \in \{0,1\}^{\ell_{\mathsf{y}}(\lambda)}$ *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* τ*,*
- $\mathsf{SimEval}(\mathsf{vk}, x, y, \mathsf{crs}, \tau)$ *on input verification key* vk*, preimage* x*, image* y*, CRS* crs*, and trapdoor* τ*, outputs a proof* π*.*

Setup *and* Gen *are necessarily probabilistic,* Eval *and* Vfy *are deterministic. A simulatable VRF has two modes, a binding mode and a simulation mode. In the binding mode, the sVRF is a normal VRF with unique provability, i.e., only a single image can be proven correct relative to any given preimage. In the simulation mode, given the trapdoor, one can generate valid proofs for any preimage-image pair. Moreover, both modes are computationally indistinguishable. For the formal security guarantees we refer the reader to [8].*

```
Initalize($\mathcal{S}$)
repr $\leftarrow$ zeros$(1, m)$
for $x \in \mathcal{S}$
    $(y_1, \ldots, y_{k(\lambda)}) \leftarrow R(x)$
    for $i \in [k(\lambda)]$ :
        repr$[y_i] \leftarrow 1$
return repr
```

```
Query(repr, $x$)
$(y_1, \ldots, y_k) \leftarrow R(x)$
return $\bigwedge_{i \in [k(\lambda)]}$ [repr$[y_i] = 1$]
```

Figure 7: Bloom filter structure $\mathrm{BF}[R, m, k]$ with parameters integers $m, k \geq 0$, and a function $R : \{0,1\}^* \to [m]^k$. Function R maps data-object elements to a vector of positions in the array repr. $\mathrm{BF}[R, m, k]$ admits set-membership queries for elements $x \in \{0,1\}^*$. A concrete scheme is given by a particular choice of parameters.