Published version available at https://doi.org/10.1007/978-3-319-26059-4_10

Efficient Key Authentication Service for Secure End-to-end Communications

Mohammad Etemad Alptekin Küpçü

Crypto Group, Koç University, İstanbul, Turkey
{metemad,akupcu}@ku.edu.tr

Abstract. After four decades of public key cryptography, both the industry and academia seek better solutions for the public key infrastructure. A recent proposal, the *certificate transparency* concept, tries to enable *untrusted* servers act as public key servers, such that any key owner can verify that her key is kept properly at those servers. Unfortunately, due to high computation and communication requirements, existing certificate transparency proposals fail to address the problem as a whole. We propose a new efficient *key authentication service* (KAS). It uses server-side gossiping as the source of trust, and assumes servers are not all colluding. KAS stores all keys of each user in a separate hash chain, and always shares the last ring of the chain among the servers, ensuring the users that all servers provide the same view about them (i.e., no equivocation takes place). Storing users' keys separately reduces the server and client computation and communication dramatically, making our KAS a very efficient way of public key authentication. The KAS handles a key registration/change operation in $O(1)$ time using only $O(1)$ proof size; *independent of the number of users*. While the previous best proposal, CONIKS, requires the client to download 100 KB of proof per day, our proposal needs less than 1 KB of proof per key lifetime, while obtaining the same probabilistic guarantees as CONIKS.

Keywords: Certificate transparency, End-to-end encryption.

1 Introduction

Many online services, ranging from financial services to health and social services, process sensitive data. Most of the sensitive information is transferred through the network, unencrypted or point-to-point encrypted [8]. The former reveals the data to everyone while the latter reveals the data only to those who perform decryption and re-encryption on the path to the destination. One of the breakthroughs to this regard was the invention of public key cryptography [2, 10]. It is employed in end-to-end secure email and end-to-end encryption (E2EE). To provide security and prevent misuse, the data should be encrypted by the sender, transmitted encrypted, and decrypted by the receiver.

End-to-end secure email. Email is one of the oldest IT services for fast communication. Gmail initially launched with HTTPS support and uses an encrypted connection for sending and receiving emails, but the encryption is not end-to-end, i.e., the Gmail server has access to the plaintext of all emails that are not encrypted explicitly by the sender.

End-to-end encryption (E2EE) is required to solve the above-mentioned problem. Having easy access to the target's key, a party can encrypt a message directly with it. E2EE is being considered as an important building block in many recent schemes. Weber [12] used attribute-based encryption for secure end-to-end one-to-many mobile communication. Snake [1] is proposed as an end-to-end encrypted online social networking scheme, relying on establishment of shared keys between users at the start phase of friendship. E2EE is also used in several messaging service such as Apple iMessage and BlackBerry protected messenger [8]. The main problem with all these schemes is *key management*.

One caveat of public key cryptography is that it requires the communicating parties to obtain *genuine* keys of each other. If they had a secure physical channel (i.e., they could physically meet), they could have exchanged their public keys. Unfortunately, in the online world, a *trusted* party called *certificate authority* (CA) is required. A CA ensures the parties via issuing unforgeable cryptographic evidences (certificates) that the public key contained in the certificate does indeed belong to the party stated in the certificate. However, the CAs must be trusted, and sometimes this trust is broken [11]. Moreover, there are other applicability and usage problems such as storage, revocation, and distribution [9]. CAs are also valuable targets for adversaries through which they can attack the whole system. We need a new source of trust that cannot easily be compromised legally or under pressure, without being detected and revealed.

Certificate transparency is recently proposed to substitute the existing certificate authorities [7]. There is no single source of trust and all users and servers contribute to provide a web of trust among themselves. Following (efficient) schemes [11, 8] added key revocation support, and used it as a general key management scheme for different applications such as end-to-end encrypted email. However, all these schemes impose a heavy computation and communication burden on both clients and servers, preventing its mass application. We propose an efficient key authentication service that performs key registration/change operations optimally (i.e., with $O(1)$ computation and communication).

Our contributions are as follows:

- We give the **first security definition** of a key authentication service, and prove our construction's security formally.
- Our KAS scheme performs key registration/change operations **optimally** with $O(1)$ server and client computation and $O(1)$ communication. Hence, we reduce the costs of the existing schemes [7, 11, 8] for each key registration/change operation by a factor of $\log n$, where n is the number of users.
- The total cost of the server in the previous schemes is $O(n \log n)$, since they compute the proof for all users on each key registration/change [7, 11] or on each epoch [8]. A similar audit operation costs $O(n)$ in our KAS.
- Our scheme is **perfectly privacy-preserving**, i.e., proof of a key (of a user) does not depend on any other key (of another user), and hence, does not reveal any information about other users.
- For the first time, we consider **non-repudiation** for both client and server. In our scheme, a malicious client cannot frame an honest server.

1.1 Related Work

Laurie *et al.* [7] proposed **certificate transparency** for preventing certificate misuse by publishing the issued TLS certificates through publicly-auditable logs. The clients and some special entities (called *monitors* and *auditors*) will report any misbehavior through regularly verifying the logs. The logs are append-only Merkle hash trees. The tree has the property that any previous version is a subset of all following versions, through which the consistency of logs can be proven. On receipt of a new certificate, the log server appends it into the existing tree, commits to its new root, and returns both the new root and its commitment. The issuing user verifies the result and if it is correct, spreads it through *global gossiping*. All later verifications should be made against this new root.

Later, Laurie and Kasper [6] proposed **revocation transparency**, similar to certificate transparency, as a transparent mechanism for providing the list of revoked certificates and showing that all clients see the same list of revoked certificates. However, it is not efficient due to its space and time complexity that is *linear* in the number of revocations.

Ryan [11] proposed **enhanced certificate transparency** (ECT) that can also be used for secure end-to-end email or messaging without requiring a trusted certificate authority. ECT also solves the certificate revocation problem efficiently. There is an append-only Merkle tree as in [7] for handling certificate transparency, but in addition, there is another lexicographically ordered hash tree storing a constant number of revoked keys for each client, reducing the revocation complexity from $O(n)$ to $O(\log n)$, where n is the number of users.

Melara *et al.* [8] proposed **CONIKS** as an automated key management system. It consists of a number of key providers storing user-to-key bindings of users. The users can detect equivocations or unexpected key changes made by malicious providers. CONIKS provides the following security properties: *non-equivocation, binding consistency, and privacy-preserving bindings*.

Laurie *et al.* [7] and ECT [11] use **client-side gossiping**, which requires all clients to gossip with each other. This is very inefficient as it is required on each key change. Moreover, it requires a fully connected graph over all users during the gossiping. If the user are partitioned during gossiping, the server can *equivocate* since the others has no idea about the change that happened. CONIKS [8] uses **server-side gossiping**, which is much more efficient than the client-side gossiping, especially assuming a constant number of servers.

A problem with these schemes is the lack of **formal security definitions and proofs**. Moreover, organizing the keys in a tree data structure ties them altogether, which means that even if only one key changes, all users (except the one whose key has been changed) need to check the resulting new tree to make sure they are not affected. We store the user-to-key bindings separately, thus decreasing provider and client computation while increasing the privacy-preservation level. Another problem is that the **non-repudiation** property for all parties was not considered previously: one cannot show the origin of any potential inconsistency. A malicious client can try to frame an honest server, and neither party could provide evidence of honest behavior. We also address this issue. A comparison of these schemes is given in Table 1.

Table 1: A comparison of certificate transparency schemes. NR and PP stand for non-repudiation and privacy-preserving, respectively, and '×' means not fully supported.

Scheme	Update	Audit	Revocation	Gossiping	NR	PP
Laurie *et al.* [7]	$O(\log n)$	$O(n \log n)$	$O(n)$	Client-side	-	-
ECT [11]	$O(\log n)$	$O(n \log n)$	$O(\log n)$	Client-side	-	-
CONIKS [8]	$O(\log n)$	$O(n \log n)$	$O(\log n)$	Server-side	×	×
Our KAS	$O(1)$	$O(n)$	$O(1)$	Server-side	+	+

On a separate line of work, Wendlandt *et al.* [13] proposed Perspectives for authenticating a server's public key. It uses a number of (semi-)trusted hosts named *network notaries* that probe most of network services (e.g., SSH) and keep a history of their public keys over time. On receipt of an unauthenticated public key from a service, the client asks the notaries the history of keys used by that service to help her decide on whether to accept or reject it. Perspectives differs from the certificate transparency as it requires all notaries to be trusted.

1.2 Model

There are two parties in our model. The **providers** are the publicly available servers storing name-to-key bindings of the users. There are multiple providers each running in a different domain. Each provider manages a distinct set of clients, and shares information about them with other providers by gossiping, through a specific PKI. $\mathcal{P}$ represents the set of all providers. We use the terms *provider* and *server* interchangeably. For easier presentation, *home provider* is the provider a user registers her keys with.

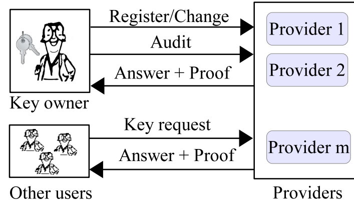

Fig. 1: Our model.

The **client** (or her device on her behalf) registers her (public) key in a provider and can access the keys of other clients through the (same or other) provider(s). We use the terms *client* and *user* interchangeably. We also differentiate between two kinds of users when they execute our protocols: The *key owner* registers, updates, and audits her own key, while the *other users* request the key of some key owner from the providers. In our scenarios, hereafter, we assume that **Alice** is the **key owner**, and **Bob** is some **other user** who wants to communicate with Alice, hence, requests her key from the providers. Our model is depicted in Figure 1.

Adversarial model. We assume that the providers may be malicious, but not all providers collude (though some may). They may try to attack the integrity of the name-to-key bindings they store, or to equivocate and show different results for the same query coming from different users, while trying to be undetected. We assume honest behavior of the users *only* during the first key registration. But, they are *not* trusted for later key changes in the sense that they may later try to frame an honest provider. Our scheme provides non-repudiation for both the user and the provider, i.e., they cannot deny what they have already done, and helps find the origin of any inconsistency or misbehavior and provides cryptographic proofs to be used as evidence.

1.3 Overview

Public logs are proposed as an alternative to CAs that are not required to be trusted. Either client-side or server-side gossiping is used as the source of trust.

The high level idea is that instead of trusting on only one entity, it is better to rely of a group of entities, hoping that not all of them misbehave simultaneously.

The existing solutions store keys of all users in an authenticated data structure, and hence, tie all keys together. This requires all users to check for their keys on any key registration/change. Instead, we store all keys of each user in a separate hash chain and remove the dependence between keys of different users. Each time a user registers a new key, the new key and the previous hash chain digest is hashed to obtain an new hash chain digest. Then, the home provider signs and shares this new digest with all other providers, i.e., we follow the server-side gossiping. This helps all providers keep the same view about the users.

To request a key, a user contacts a number of providers, and accepts the obtained key if they all provide the same view. She cannot rely on only one provider due to the possibility of equivocation.

2 Key Authentication Service

2.1 Preliminaries

Notation. We use || for concatenation, and |X| to show the number of items in X. PPT means probabilistic polynomial time, and λ is the security parameter. A function $\nu(\lambda) : Z^+ \to [0,1]$ is called *negligible* if $\forall$ positive polynomials $p, \exists$ a constant c such that $\forall\ \lambda > c,\ \nu(\lambda) < 1/p(\lambda)$. *Overwhelming* probability is greater than or equal to $1 - \nu(\lambda)$ for some negligible function $\nu(\lambda)$. By *efficient algorithms*, we mean those with expected running time polynomial in λ.

Interactive protocols. We present our key authentication scheme as a set of interactive protocols between stateful clients and stateful servers, instead of separately-executed algorithms. Each protocol may receive inputs from both parties and may output some data at both parties. Hence, we represent a protocol as $\texttt{protocolName}(output_{client})(output_{server}) \leftarrow (input_{client})(input_{server})$.

Hash functions are functions that take arbitrary-length strings, and output strings of some fixed length. Let $h : \mathcal{K} * \mathcal{M} \to \mathcal{C}$ be a family of hash functions, whose members are identified by $k \in \mathcal{K}$. A hash function family is collision resistant if $\forall\ PPT$ adversaries $\mathcal{A}, \exists$ a negligible function $\nu(\lambda)$ such that: $Pr[k \leftarrow \mathcal{K}; (x, x') \leftarrow \mathcal{A}(h, k) : (x' \neq x) \wedge (h_k(x) = h_k(x'))] \leq \nu(\lambda)$ for security parameter λ (which is correlated to $|\mathcal{K}|$ and $|\mathcal{C}|$).

A **signature scheme** is a scheme for preserving message integrity, consisting of the following PPT algorithms [5]:
— $(sk, vk) \leftarrow \texttt{Gen}(1^\lambda)$: Generates a signing and verification key pair (sk, vk) using the security parameter λ.
— $\sigma \leftarrow \texttt{Sign}(sk, m)$: Generates a signature σ on a message m using the key sk.
— $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{Verify}(vk, m, \sigma)$: Checks whether σ is a correct signature on a message m, using the verification key vk, and outputs an acceptance or a rejection signal, accordingly.

2.2 Key Authentication Service Scheme

Definition 1 (KAS scheme). *A KAS scheme is composed of the following interactive protocols between a stateful user and a stateful provider:*

- $(vk_p)(sk_p, vk_p) \leftarrow \texttt{Setup}(1^\lambda)(1^\lambda)$. The provider starts up this protocol to generate the signing and verification keys (sk_p, vk_p), given as input the security parameter λ, and shares the verification key with all users and providers. Essentially, we assume the providers' verification keys are part of some other well-established PKI.
- $(\sigma_i^u, \texttt{accept/reject})(rvk_{i-1}^u, vk_i^u, \sigma_i^u) \leftarrow \texttt{Register}(id_u, sk_{i-1}^u, sk_i^u, vk_i^u, vk_p)$ (sk_p, vk_p). The user registers/changes her key via a provider, given her ID id_u, her previous key sk_{i-1}^u (empty for the first time), and her new key pair (sk_i^u, vk_i^u) as input. For a key change, she should provide a revocation statement rvk_{i-1}^u signed with her previous signing key sk_{i-1}^u. The provider receives his key pair (sk_p, vk_p) as input, stores user's key, computes a signature σ_i^u on the result, and transfers it to the user. The user verifies it and outputs an acceptance or a rejection signal based on the result.
- $(vk_i^u, \texttt{accept/reject})() \leftarrow \texttt{Audit}(id_u, \{vk_p\}_{p \in \mathcal{P}})(info_u)$: A user initiates this protocol to retrieve the (latest) key of a user id_u. She specifies a subset of providers $P \subseteq \mathcal{P}$ at random and challenges them about the user id_u. The providers are given the user information $info_u$ as input and send back their latest view about the user id_u. If all challenged providers gave the *same* answer, she accepts and outputs the received key vk_i^u as the correct (latest) key of id_u, and `accept`. Otherwise, she outputs $\perp$ and `reject`.

Both users and providers keep their own local states during execution of the protocols. Each user stores her latest key and the respective provider's signature (and the previous ones if she has enough resources). The providers also store all keys received from the users along with the signatures they have computed. The key owner runs `Audit` to see if the providers keep a correct view about her. If `Audit` returns `accept` and the obtained key matches the locally-stored key, she outputs `accept`. Otherwise, she outputs `reject`.

2.3 Security Definitions of the Key Authentication Service

Definition 2 (Correctness). *considers the honest behavior of providers and users in a protocol execution. A key authentication scheme is correct if the following is satisfied with probability* 1 *over the randomness of the users and providers:*

Each execution of the `Audit` *protocol results in a key vk_u, which is the latest key registered by the key owner through the* `Register` *protocol. If such a user does not exist, it returns $\perp$.*

Security game. If the providers deviate from honest behavior by providing different answers to the same query for different users, the users should detect this with high probability. The security game $\texttt{AuthGame}_\mathcal{A}(\lambda)$ between a challenger $\mathcal{C}$ who plays the role of all users (key owners and others) and all non-adversarial providers[1], and the adversary $\mathcal{A}$ acting as the malicious providers, is defined as:

- **Initialization**. $\mathcal{C}$ runs the `Setup` protocol to initialize the environment on behalf of all non-adversarial providers, and shares their verification keys with $\mathcal{A}$. He stores the verification keys of malicious providers given by $\mathcal{A}$.

[1]By our assumption that not all providers are colluding, there is at least one non-adversarial provider that the challenger will simulate.

– **Adaptively Chosen Queries**. $\mathcal{A}$ asks the challenger to start a `Register` or an `Audit` protocol by providing the required information. For the `Register` protocol, he specifies a user and a provider that the user will register her key with. If the specified provider is a malicious one, $\mathcal{C}$ acts as the user in the corresponding registration protocol with $\mathcal{A}$. He also stores the signatures coming from $\mathcal{A}$. Otherwise, $\mathcal{C}$ registers the key locally (acting as both the user and a non-adversarial provider) and shares the resulting signature with $\mathcal{A}$. For the `Audit` protocol, $\mathcal{A}$ requests a user ID, and $\mathcal{C}$ performs the associated audit and verifies the answer, and notifies $\mathcal{A}$ about the result. The queries can be repeated polynomially-many times, in any order, adaptively.
– **Challenge**. $\mathcal{A}$ specifies a user and sends an audit request to $\mathcal{C}$, who initiates an `Audit` protocol, giving the specified user ID as input. He outputs the obtained key and `accept` if `Audit` accepts, and $\perp$ and `reject` otherwise.

$\mathcal{A}$ wins the game if $\mathcal{C}$ accepts while the obtained key differs from the latest key $\mathcal{C}$ successfully registered on behalf of that user. The output of the game is defined to be 1 in this case.

Definition 3 (ϵ-Security). *A key authentication service scheme is ϵ-secure if $Pr[\texttt{AuthGame}_{\mathcal{A}}(\lambda) = 1] \leq \epsilon$ for any efficient adversary $\mathcal{A}$.*

3 Construction

Problems with Previous Proposals. Laurie *et al.* [7] builds an append-only Merkle hash tree and appends the new keys as they arrive. To be secure, after each new key registration/change, they distribute the root of the updated tree to the users who will gossip it among themselves to ensure that they all keep the *same* latest view of the keys registered in the log. This requires all users to connect and contribute in gossiping. In case of any partitioning among the users, the server can easily equivocate. In addition, all users need to verify the new tree. The key owner wants to check that her key is correctly registered/changed, while the other users want to make sure that they are not affected by this key registration/change. This poses an unacceptably heavy burden on users.

To alleviate the problem, CONIKS [8] uses server-side gossiping. Although this relieves the users from gossiping, they still need heavy and frequent verifications. CONIKS goes further and divides the time into *epochs*, accumulates all key registration/change requests during each epoch, and applies them all at the beginning of the next epoch. If t key registration/change requests arrive in an epoch, instead of performing t consistency checks, a client performs only one consistency check. However, this comes at the cost of giving outdated keys for users whose keys are changed in an epoch, until the beginning of the next epoch.

Our Observations and Solution. A problem common to all existing solutions [6–8, 11] is that they put all user-to-key bindings in a single data structure (i.e., a Merkle tree) on each provider. This ties all users together, and hence, forces all users to check whether or not any change in the data structure affects them, regardless of who initiated the change, which is a costly process. Instead, we store each user's data separately, preventing unwanted costly checks due to

changes on other users. Even though on the surface it seems that we are gossiping more than just a root, the number of gossiping operations in our strategy is the same as in [6, 7, 11], since one value is gossiped on every key change (though we can also gossip per epoch as in CONIKS, leaving some vulnerability window). However, they need to update the Merkle tree with $O(\log n)$ cost, while we only add a new value into the corresponding hash chain with only $O(1)$ cost.

Storing users' data separately brings other important advantages: the resulting scheme is **perfectly privacy-preserving**, since the proof for the a user's key does not contain any information about other owners. Moreover, there is no need for consistency checks, which are very costly operations in [6, 11]. Even though CONIKS reduces the cost of this operation using epochs, the consistency check problem is inherent in tree-based approaches: on each key registration/change (or a group of them in CONIKS), the provider updates the tree and distributes a new commitment. Then, *all* other owners should check the tree and ensure that their keys have not been changed.

Our KAS provides verifiable user-to-key binding service and consists of a number of providers each managing a subset of users. Each provider stores all registered keys of each user in its namespace in a separate hash chain. The provider, after adding the key into the respective chain, commits to (signs) the result, and sends it to the user for verification. Besides, he distributes (gossips) the last hash value in the chain and its signature to all other providers. To acknowledge the receipt, the other providers sign the received hash value, and return the signature to the initiating provider. Figure 2 shows our construction.

3.1 Description of the Operations

`Register`. When a user registers her key for the first time, she provides the home provider with her id, id_u, and her key, vk_1^u. (The user is assumed to be trusted for this operation.) The provider computes hash of this key as $h_1^u = h(id_u||vk_1^u)$, signs it as $\sigma_1^u = \texttt{Sign}(sk_p, h_1^u)$, and returns back both values. When the user is registering a key change (i.e., the previous key, the $(i-1)^{th}$ key, is revoked and she is providing a new one, the ith key), she should also prepare a revocation statement rvk_{i-1}^u signed with her previous key and give it to the provider along with id_u and vk_i^u. On receipt, the provider verifies rvk_{i-1}^u, and if it is accepted, computes $h_i^u = h(id_u||h_{i-1}^u||rvk_{i-1}^u||vk_i^u)$ and $\sigma_i^u = \texttt{Sign}(sk_p, h_i^u)$. In both cases, the provider returns h_i^u and σ_i^u to the user for verification. In addition, he distributes both values to all other providers, who will verify correctness of the signature and acknowledge by signing it if it is accepted. The home provider keeps the acknowledgments for detecting the source of later inconsistencies. The data structure for a key owner, Alice, on her home provider is a table as shown in Table 2. It contains all keys registered by Alice, all her key revocation statements, the home provider's signatures, and the other providers' acknowledgments.

Each provider stores the hash values and signatures he receives about users of the other providers in a similar table, as presented in Table 3. The information in these tables are used for answering the audit queries, which if passed, ensure the auditing user that all providers have the same view about the queried user, meaning that there is no equivocation, with high probability. In other words, the

Let S=(Gen,Sign,Verify) be a secure signature scheme and $h : \{0,1\}^* \to \{0,1\}^l$ be a collision resistant hash function modeled as a random oracle. Construct a key authentication service scheme KAS=(Setup,Register,Audit) as:

- Setup$(1^\lambda)(1^\lambda)$:
  - Each provider runs $(sk_p, vk_p) = $ S.Gen(1^λ) and shares vk_p with his users and the other providers.
  - Each user stores vk_p of her home provider.
- Register$(id_u, sk^u_{i-1}, sk^u_i, vk^u_i, vk_p)(sk_p, vk_p)$:
  The key owner:
  - If $i > 1$, sets $rvk^u_{i-1} = $ S.Sign$(sk^u_{i-1}$,'revocation statement')
  - Sends vk^u_i and rvk^u_{i-1} to her home provider.

  The home provider:
  - Finds the last registered key, vk^u_{i-1}, and the last computed hash, h^u_{i-1}, for user id_u.
  - If S.Verify$(vk^u_{i-1}$, 'revocation statement', $rvk^u_{i-1}) ==$ reject, outputs reject and exits.
  - Otherwise, computes $h^u_i = h(id_u||h^u_{i-1}||rvk^u_{i-1}||vk^u_i)$ and $\sigma^u_i = $ S.Sign(sk_p, h^u_i).
  - Sends $(id_u, h^u_i, \sigma^u_i)$ to the respective user and all other providers.

  Other providers:
  - Each other provider p_t first checks if S.Verify$(vk_p, h^u_i, \sigma^u_i) ==$ accept. If so, he computes $\sigma^{u,t}_i = $ S.Sign(sk_{p_t}, h^u_i), stores h^u_i and $\sigma^{u,t}_i$ locally, and sends $\sigma^{u,t}_i$ to the home provider.

  The home provider:
  - On receipt the acknowledgments $\sigma^{u,t}_i$, the home provider checks if S.Verify$(vk_{p_t}, h^u_i, \sigma^{u,t}_i) ==$ accept for all other providers p_t. He stores the correct ones, and reveals the incorrect ones.

  The key owner:
  - The key owner checks if $h^u_i == h(id_u||h^u_{i-1}||rvk^u_i||vk^u_i)$ and S.Verify$(vk_p, h^u_i, \sigma^u_i) ==$ accept. If so, she stores h^u_i and σ^u_i locally and outputs accept. Otherwise, she outputs reject and publishes misbehavior of the provider together with the corresponding h^u_i and σ^u_i as the proof, through public social networks.
- Audit$(id_u, \{vk_p\}_{p\in\mathcal{P}})(info_u)$:
  - The user selects a random subset of providers $P \subseteq \mathcal{P}$, and sends them the Audit(id_u) request.
  - The home provider replies with the latest information about the user being queried: $(h^u_{i-1}, rvk^u_{i-1}, vk^u_i, h^u_i)$.
  - The other providers reply with the latest hash value, h^{u,i_t}_i, received from the home provider about that user.
  - On receipt $(h^u_{i-1}, rvk^u_{i-1}, vk^u_i, h^u_i)$ from the home provider and h^{u,i_j}_i from other providers, the user checks if:
    * $h(id_u||h^u_{i-1}||rvk^u_{i-1}||vk^u_i) == h^u_i$. If not, misbehavior of the home provider is detected. She outputs $\perp$ and reject, publishes the misbehavior, and exits.
    * $h^u_i == h^{u,i_1}_i == h^{u,i_2}_i == ... == h^{u,i_t}_i$, $\forall\, i_j \in P$. If so, she outputs vk^u_i and accept. Otherwise, she outputs $\perp$ and reject, publishes the misbehavior, and starts the misbehaving detection process.
  - (Owner Only) If the user audits for the key she owns, she also checks if the received key vk^u_i is the one she last registered. If so, outputs accept. Otherwise, she uses it with the signature she received during registration, publishes the misbehavior case, outputs reject, and exits.

Fig. 2: Construction of our key authentication service.

providers help to make a web of trust through ensuring the auditing user that they all confirm binding of the provided key (hash value) to the queried user.

On receipt the answer, the key owner verifies the received signature to see if the home provider has correctly registered her key. She can compute either $h^{'u}_1 = h(id_u||vk^u_1)$ or $h^{'u}_i = h(id_u||h^u_{i-1}||rvk^u_{i-1}||vk^u_i)$ (she knows all required information) and compare it with what received from the home provider: $h^{'u}_1 {==} h^u_1$ for the first key registration or $h^{'u}_i {==} h^u_i$ for the subsequent key changes. If the test passed, she verifies the signature. If it is verified successfully too, she ensures that her key is correctly registered. Then, she can run an Audit protocol to see if the key registration is correctly reflected on the other providers.

Audit. From time to time, the key owner, Alice, checks if the providers keep storing her key and hash value intact. This is because other users cannot check authenticity of her keys. Hence, they rely on regular checks of Alice. If Alice detects equivocation or any other misbehavior, in addition to going to arbitra-

Table 2: The data structure for a key owner, Alice, on her home provider.

Alice's keys	vk_1^{Alice}	$vk_2^{Alice},\ rvk_1$	...	$vk_i^{Alice},\ rvk_{i-1}$
Provider's signatures	$h_1^{Alice}, \sigma_1^{Alice}$	$h_2^{Alice}, \sigma_2^{Alice}$	...	$h_i^{Alice}, \sigma_i^{Alice}$
P_2**'s acknowledgments**	$\sigma_1^{Alice,2}$	$\sigma_2^{Alice,2}$	...	$\sigma_i^{Alice,2}$
...	...	...	...	...
P_m**'s acknowledgments**	$\sigma_1^{Alice,m}$	$\sigma_2^{Alice,m}$	...	$\sigma_i^{Alice,m}$

Table 3: The information provider P_t keeps about Alice, a user of provider P_j.

P_j**'s signatures**	$h_1^{Alice}, \sigma_1^{Alice}$	$h_2^{Alice}, \sigma_2^{Alice}$	...	$h_i^{Alice}, \sigma_i^{Alice}$
P_t**'s acknowledgments**	$\sigma_1^{Alice,t}$	$\sigma_2^{Alice,t}$	...	$\sigma_i^{Alice,t}$

tion, she should somehow inform the other users about the misbehavior (e.g., via publishing in social networks [8]) and prevent them from using the fake or outdated keys registered in her name.

When another user, Bob, wants to communicate with Alice, he asks for Alice's key from the providers. If Bob requests Alice's key only from her home provider, the home provider may equivocate and give a fake or an outdated key to him. He needs to make sure, with high probability, that he retrieves a genuine key. Moreover, before obtaining the key, he should first check and ensure that there is no equivocation report by Alice.

Bob selects a random subset P of providers $\mathcal{P}$, and sends them the audit request. They return the last hash value they have received from the home provider. Bob also contacts Alice's home provider, who returns the last registered key, as well as information required to recompute the last hash in the chain. Bob then checks whether all received hash values are the same (otherwise an equivocation is detected). He also re-hashes the values provided by Alice's home provider. If everything verifies, he returns the obtained key and `accept`. Otherwise, he outputs $\perp$ and `reject`, and starts the misbehavior detection process.

The misbehavior detection probabilities are discussed in more detail in our full version [3]. Essentially, we obtain similar probabilities as in CONIKS [8].

Misbehavior detection. When the key owner or another user discovers a misbehavior, she can trace and find its source. This is due to the non-repudiation property of our scheme. Since the key is distributed by the home provider only, and the other providers distribute the hash values, if the given key does not match the given hash value (all users can detect it), this can be used as the witness of home provider's misbehavior. If the given key matches the given hash value but it is not the genuine key (only the key owner can detect it), the home provider is misbehaving, and the key owner can use these values together with the home provider's signature received during the registration as the cryptographic proof of the misbehavior. However, she cannot claim that a given key is fake if she has already registered the key. The home provider can show this case using the signed revocation statement he received from the key owner.

Whenever the received hash values does not match each other (all users can detect it), a subset of providers are equivocating. The user (or any authority) asks the home provider acknowledgments of the other challenged providers, and asks the other challenged providers the signature they have received from the home provider. Verifying the signatures will reveal the misbehaving provider(s).

4 Analysis

4.1 KAS Security Proof

Correctness follows from the correctness of the hash function used by the providers. If the key owner has already been registered, the hash function guarantees that whenever her key is requested, the requesting party receives the correct key. If not, $\perp$ together with non-membership proof will be returned.

Theorem 1. *Our KAS scheme is secure according to Definition 3 provided that the underlying hash function is collision-resistant.*

Proof. We reduce the security of our KAS scheme to that of the underlying hash function. If a PPT adversary $\mathcal{A}$ wins the security game of our KAS scheme with non-negligible probability, we use it to construct another PPT algorithm $\mathcal{B}$ who breaks collision-resistance of the hash function, with non-negligible probability. $\mathcal{B}$ acts as the adversary in the hash function security game with the challenger $\mathcal{C}$. In parallel, $\mathcal{B}$ plays the role of the challenger in the KAS game with $\mathcal{A}$.

Initialization. $\mathcal{B}$ receives a hash function $h(.)$ from $\mathcal{C}$, and shares it with $\mathcal{A}$. The signature keys are generated and shared.

Adaptively Chosen Queries. $\mathcal{A}$ sends $\mathcal{B}$ a request to start a protocol, with the required information:

- For a `Register` protocol, a user ID, id_u, and a provider ID are given. $\mathcal{B}$ generates key pair (sk_i^u, vk_i^u) and the revocation statement rvk_{i-1} for the given user and stores them locally. If the given provider is non-adversarial, $\mathcal{B}$ computes $h_i^u = h(id_u||h_{i-1}^u||rvk_{i-1}^u||vk_i^u)$, signs it as σ_i^u, stores h_i^u and σ_i^u locally, sends them to $\mathcal{A}$, and keeps the acknowledgments coming back. If the given provider is malicious, $\mathcal{B}$ asks $\mathcal{A}$ to register the key for the user, and stores and acknowledges the potential hashes and signatures coming back.
- For an `Audit` protocol about id_u, $\mathcal{B}$ selects a random subset of providers $P \subseteq \mathcal{P}$, as well as the user's home provider, and challenges them about id_u. For those challenged providers who are non-adversarial, $\mathcal{B}$ himself provides the answer, while the answer of the malicious ones are given by $\mathcal{A}$. On receipt all answers, $\mathcal{B}$ performs the audit verification as in the protocol. He also informs $\mathcal{A}$ about the result.

Challenge. The adversary specifies a user id_u and asks $\mathcal{B}$ to challenge him about this user. $\mathcal{B}$ runs the `Audit` protocol as described. If $\mathcal{B}$ accepts $\mathcal{A}$'s answer while the obtained key disagrees with his local knowledge, we consider two cases:

- The returned hash value(s) is the same as the locally-stored one. In this case, a collision is found. If $\mathcal{B}$ accepts $\mathcal{A}$'s answer with probability ϵ_1, we can use it to break security of the hash function with probability ϵ_1. Since the hash function is collision-resistant by assumption, ϵ_1 must be negligible, which means that $\mathcal{A}$ has negligible chance to pass $\mathcal{B}$'s verification successfully and win the game. Hence, our KAS scheme is secure in this regard assuming that the underlying hash function is collision-resistant.
- The returned hash values differ from the locally-stored one: that is an indication of an equivocation. The equivocation of providers is discussed in more detail in our full version [3]. It states that we can approximate the provider equivocation probability with $(1-f)^k$ against the key owner and f^k against

the other users, where f is the fraction of providers distributing fake hash values, and $k = |P|$ is the size of the challenged subset of providers. Moreover, they will mutually detect the equivocation with probability $1 - (f - f^2)^k$. Hence, our scheme is $(f - f^2)^k$-*equivocal*, which means that the equivocation detection probability increases very fast as we increase k. This is maximized when $f = 1/2$. Interestingly, increasing f (from 1/2) decreases the detection probability (except if $f = 1$ where all providers are maliciously colluding).

- When f is very small, the probability of misbehavior detection by Alice is not very high (e.g., when f=0.01, k=50 leads to detection probability $1 - (1 - 0.01)^{50}$=0.395). But, this means that the `Audit` protocol will select all honest providers for audit with high probability, and hence, the obtained key will be the correct key with high probability.
- When f is close to 1, the key owner's probability of misbehavior detection is very high (e.g., when f=0.99, k=50 leads to detection probability $1 - (1 - 0.99)^{50}$, which is almost 1).
- These computations show only the results of one audit. But, the key owner audits regularly over the time. If the `Audit` protocol selects the providers uniformly, it is expected that after $|\mathcal{P}|/|P|$ audits, all providers will be challenged at least once. This means that the key owner will eventually detect the misbehavior.

To sum up, our KAS scheme is $\epsilon = (\epsilon_1, \epsilon_2)$-secure, i.e., the adversary can break it with probability ϵ_1, which is negligible in the security parameter if the underlying hash function is secure, and it is (ϵ_2=$(f - f^2)^k$)-equivocal meaning that the adversary can equivocate with a very small probability.

4.2 Asymptotic Comparison to Previous Work

Key registration/change. Since all users' data are tied together in the existing schemes, a key registration/change (or an epoch in CONIKS) forces the provider to generate proofs of size $O(\log n)$ for all users and send them to the respective users (consistency proofs). This requires total $O(n \log n)$ computation and communication cost at the provider, which are not taken into account in the previous work. This is an $O(1)$ operation in our KAS since the provider performs a constant number of operations and distributes constant-size proofs over the network on each update. However, the key owners of KAS audit the providers once in a time, roughly like an epoch in CONIKS. The providers reply only with the latest keys and hash values without any computation, which accounts for $O(n)$ server computation and communication, in total.

Gossiping. In CONIKS and our scheme, there is a gossiping at the provider side, whereas in the other two schemes gossiping is done at the client side. Note that in practice, we expect maybe only hundreds of providers whereas many millions of clients will take place in the system. Table 4 presents the comparison, showing an $O(\log n)$ decrease in all operations in our KAS.

Key revocation. The scheme given in [6] has no an efficient way for key revocation, hence, it needs to traverse the whole ADS with cost $O(n)$. But, the two others [11, 8] can do it with $O(\log n)$ time and space complexity, i.e., the server should generate an $O(\log n)$ proof for each user (summing up to $O(n \log n)$

Table 4: A comparison of key registration/change costs. n is the number of users.

Scheme	**Provider**		**User**		**Gossiping**
	Key Reg.	**Proof Gen.**	**Comp.**	**Proof Size**	
Laurie *et al.* [6]	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(\log n)$	Client-side
ECT [11]	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(\log n)$	Client-side
CONIKS [8]	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(\log n)$	Server-side
Our KAS	$O(1)$	$O(1)$ ($O(n)$ audits)	$O(1)$	$O(1)$	Server-side

communication) in $O(n \log n)$ time. This is also an $O(1)$ operation in our scheme since there is no correlation between data of different users.

Audit. This operation is only needed in CONIKS and our scheme. Similarly, it requires $O(n \log n)$ and $O(n)$ server computation and communication in CONIKS and KAS, respectively. Moreover, the client computation and proof size are $O(\log n)$ and $O(1)$ in CONIKS and KAS, respectively.

4.3 Performance Analysis

Setup. To evaluate our KAS scheme, we implemented a prototype using the Cashlib library. All experiments were performed on a 2.50 GHz machine with 24 cores (using a single core), with 16 GB RAM and Ubuntu 12.04 LTS operating system, hosting all our providers. Hence, the inter-provider communications were performed as loopback. The clients were deployed on a dualcore 2.5GHz laptop machine with 4 GB RAM, running Ubuntu 14.04 LTS, connected to the same LAN. We used settings similar to CONIKS: there are n=10M users registered at each provider, and each user changes her key once a year, which accounts for $\sim$ 27,400 changes per day. The security parameter $\lambda = 128$, and we use SHA-256 as the hash function and the DSA signature scheme with key pair size (2048,256) bits according to [4]. The numbers are averages of 50 runs.

Key registration/change. The client sends the home provider her new key (and revocation statement of the previous key), which has a fixed size. The provider computes hash of the key (32 B), signs the result (64 B), and shares the hash values and the signature with other providers, as well as the key owner. Therefore, the inter-provider communication is 96 KB in total when $|\mathcal{P}|$=1000, and the provider-user communication is only 96 B. *This happens for each client once per key lifetime, not per epoch as in CONIKS.*

The previous schemes require the provider to generate and send a consistency proof to each user on each key registration/change [6, 11] (each epoch in CONIKS [8]). Considering that the proof is of size $O(\log n)$, where n is the number of users, and there are r key registrations/changes per day, each client receives and verifies $O(r \log n)$ proofs per day. This is ~800 MB (for 10^9 users) in ECT and 100 KB (for 10^7 users) using the compressed proofs in CONIKS, per day.

The $O(n \log n)$ proof that the provider prepares and distributes over the network each time is more than 3.11 GB in total in CONIKS. Assuming 27,400 key changes and 288 epochs per day, the providers prepare and distribute more than 895 GB proof per day in CONIKS. The others [6, 11] perform even worse. In our scheme, the providers perform a constant number of operations and distributes constant-size proofs, leading to a total of ~2.5 MB of proof per day.

Computation time. It takes ~ 0.5 *ms* for the provider in KAS to perform the key registration and send back the proof, while it takes ~ 2.6 *s* in CONIKS.

Audit. Assuming the key owners perform audit operations once in a time equal to a CONIKS epoch (i.e., 5 minutes), the providers return the already-computed hash values (and signatures in previous schemes), without any computation. During the normal operation, our providers respond with the latest hash values only. Assuming the challenge size k=50, each user receives a response of size ∼1.56 KB in our scheme, and ∼4.68 KB in CONIKS, for each audit. These sum up to ∼450 KB in our scheme, and ∼1.31 MB in CONIKS, per day. For the providers, this is an $O(n)$ operation, which sums to ∼305 MB of proof for one round of audit, and ∼85 GB per day, in our scheme. While CONIKS will transfer ∼915 MB of proof for one round of audit, and ∼257.5 GB per day. Our KAS saves more than 67% of the daily communication compared to CONIKS. The results are shown in Table 5.

Table 5: A comparison of audit proof sizes in KAS and CONIKS, k=50.

Scheme	**Provider**			**User**		
	Complexity	**One epoch**	**Per day**	**Complexity**	**One Audit**	**Per day**
CONIKS	$O(n \log n)$	915 MB	257.5 GB	$O(1)$	4.68 KB	1.31 MB
Our KAS	$O(n)$	305 MB	85 GB	$O(1)$	1.56 KB	450 KB

Acknowledgement. We acknowledge support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project number 114E487.

References

1. A. Barenghi, M. Beretta, A. D. Federico, and G. Pelosi. Snake: An end-to-end encrypted online social network. In *HPCC*, pages 763–770. IEEE, 2014.
2. W. Diffie and M. E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.
3. M. Etemad and A. Küpçü. Efficient key authentication service for secure end-to-end communications. Cryptology ePrint Archive, Report 2015/833, 2015.
4. P. Gallagher and C. Kerry. Digital signature standard (dss). NIST, 2013. FIPS PUB 186-4.
5. S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM*, 17(2):281–308, 1988.
6. B. Laurie and E. Kasper. Revocation transparency. `http://www.links.org/files/RevocationTransparency.pdf`. Accessed: 20/04/2015.
7. B. Laurie, A. Langley, and E. Kasper. Rfc 6962: Certificate transparency, 2013.
8. M. S. Melara, A. Blankstein, J. Bonneau, M. J. Freedman, and E. W. Felten. Coniks: A privacy-preserving consistent key service for secure end-to-end communication. 2014.
9. M. Naor and K. Nissim. Certificate revocation and certificate update. *Selected Areas in Communications, IEEE Journal on*, 18(4):561–570, 2000.
10. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
11. M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. *Proceedings of NDSS. The Internet Society*, 2014.
12. S. G. Weber. Enabling end-to-end secure communication with anonymous and mobile receivers - an attribute-based messaging approach. Cryptology ePrint Archive, Report 2013/478, 2013.
13. D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving ssh-style host authentication with multi-path probing. In *USENIX*, pages 321–334, 2008.