# CONIKS: Bringing Key Transparency to End Users

Marcela S. Melara, Aaron Blankstein, Joseph Bonneau[†], Edward W. Felten, Michael J. Freedman

*Princeton University, [†]Stanford University/Electronic Frontier Foundation*

## Abstract

We present CONIKS, an end-user key verification service capable of integration in end-to-end encrypted communication systems. CONIKS builds on transparency log proposals for web server certificates but solves several new challenges specific to key verification for end users. CONIKS obviates the need for global third-party monitors and enables users to efficiently monitor their own key bindings for consistency, downloading less than 20 kB per day to do so even for a provider with billions of users. CONIKS users and providers can collectively audit providers for non-equivocation, and this requires downloading a constant 2.5 kB per provider per day. Additionally, CONIKS preserves the level of privacy offered by today's major communication services, hiding the list of usernames present and even allowing providers to conceal the total number of users in the system.

## 1 Introduction

Billions of users now depend on online services for sensitive communication. While much of this traffic is transmitted encrypted via SSL/TLS, the vast majority is not end-to-end encrypted meaning service providers still have access to the plaintext in transit or storage. Not only are users exposed to the well-documented insecurity of certificate authorities managing TLS certificates [10, 11, 65], they also face data collection by communication providers for improved personalization and advertising [26] and government surveillance or censorship [25, 58].

Spurred by these security threats and users' desire for stronger security [44], several large services including Apple iMessage and WhatsApp have recently deployed end-to-end encryption [20, 63]. However, while these services have limited users' exposure to TLS failures and demonstrated that end-to-end encryption can be deployed with an excellent user experience, they still rely on a centralized directory of public keys maintained by the service provider. These key servers remain vulnerable to technical compromise [18, 49], and legal or extralegal pressure for access by surveillance agencies or others.

Despite its critical importance, secure key verification for end users remains an unsolved problem. Over two decades of experience with PGP email encryption [12, 56, 71] suggests that manual key verification is error-prone and irritating [23, 70]. The EFF's recent Secure Messaging Scorecard reported that none of 40 secure messaging apps which were evaluated have a practical and secure system for contact verification [51]. Similar conclusions were reached by a recent academic survey on key verification mechanisms [67].

To address this essential problem, we present CONIKS, a deployable and privacy-preserving system for end-user key verification.

**Key directories with consistency.** We retain the basic model of service providers issuing authoritative name-to-key bindings within their namespaces, but ensure that users can automatically verify *consistency* of their bindings. That is, given an *authenticated binding* issued by *foo.com* from the name *alice@foo.com* to one or more public keys, anybody can verify that this is the same binding for *alice@foo.com* that every other party observed.

Ensuring a stronger *correctness* property of bindings is impractical to automate as it would require users to verify that keys bound to the name *alice@foo.com* are genuinely controlled by an individual named Alice. Instead, with CONIKS, Bob can confidently use an authenticated binding for the name *alice@foo.com* because he knows Alice's software will monitor this binding and detect if it does not represent the key (or keys) Alice actually controls.

These bindings function somewhat like certificates in that users can present them to other users to set up a secure communication channel. However, unlike certificates, which present only an authoritative signature as a proof of validity, CONIKS bindings contain a cryptographic proof of consistency. To enable consistency checking, CONIKS servers periodically sign and publish an authenticated data structure encapsulating all bindings issued within their namespace, which all clients automatically verify is consistent with their expectations. If a CONIKS server ever tries to *equivocate* by issuing multiple bindings for a single username, this would require publishing distinct data structures which would provide irrefutable proof of the server's equivocation. CONIKS clients will detect the equivocation promptly with high probability.

**Transparency solutions for web PKI.** Several proposals seek to make the complete set of valid PKIX (SSL/TLS) certificates visible by use of public authenticated data

structures often called *transparency logs* [4, 35, 39, 40, 54, 61]. The security model is similar to CONIKS in that publication does not ensure a certificate is correct, but users can accept it knowing the valid domain owner will promptly detect any certificate issued maliciously.

Follow-up proposals have incorporated more advanced features such as revocation [4, 35, 39, 61] and finer-grained limitations on certificate issuance [4, 35], but all have made several basic assumptions which make sense for web PKI but not for end-user key verification. Specifically, all of these systems make the set of names and keys/certificates completely public and rely to varying degrees on third-party monitors interested in ensuring the security of web PKI on the whole. End-user key verification has stricter requirements: there are hundreds of thousands of email providers and communication applications, most of which are too small to be monitored by independent parties and many of which would like to keep their users' names and public keys private.

CONIKS solves these two problems:

**1. Efficient monitoring.** All previous schemes include third-party monitors since monitoring the certificates/bindings issued for a single domain or user requires tracking the entire log. Webmasters might be willing to pay for this service or have their certificate authority provide it as an add-on benefit. For individual users, it is not clear who might provide this service free of charge or how users would choose such a monitoring service, which must be independent of their service provider itself.

CONIKS obviates this problem by using an efficient data structure, a *Merkle prefix tree*, which allows a single small proof (logarithmic in the total number of users) to guarantee the consistency of a user's entry in the directory. This allows users to monitor only their own entry without needing to rely on third parties to perform expensive monitoring of the entire tree. A user's device can automatically monitor the user's key binding and alert the user if unexpected keys are ever bound to their username.

**2. Privacy-preserving key directories.** In prior systems, third-party monitors must view the entire system log, which reveals the set of users who have been issued keys [35, 40, 54, 61]. CONIKS, on the contrary, is privacy-preserving. CONIKS clients may only query for individual usernames (which can be rate-limited and/or authenticated) and the response for any individual queries leaks no information about which other users exist or what key data is mapped to their username. CONIKS also naturally supports obfuscating the number of users and updates in a given directory.

**CONIKS in Practice.** We have built a prototype CONIKS system, which includes both the application-agnostic CONIKS server and an example CONIKS Chat application integrated into the OTR plug-in [8, 27, 66] for Pidgin [1]. Our CONIKS clients automatically monitor their directory entry by regularly downloading consistency proofs from the CONIKS server in the background, avoiding any explicit user action except in the case of notifications that a new key binding has been issued.

In addition to the strong security and privacy features, CONIKS is also efficient in terms of bandwidth, computation, and storage for clients and servers. Clients need to download about 17.6 kB per day from the CONIKS server and verifying key bindings can be done in milliseconds. Our prototype server implementation is able to easily support 10 million users (with 1% changing keys per day) on a commodity machine.

## 2 System Model and Design Goals

The goal of CONIKS is to provide a key verification system that facilitates practical, seamless, and secure communication for virtually all of today's users.

### 2.1 Participants and Assumptions

CONIKS's security model includes four main types of principals: identity providers, clients (specifically client software), auditors and users.

**Identity Providers.** Identity providers run CONIKS servers and manage disjoint namespaces, each of which has its own set of name-to-key bindings.[1] We assume a separate PKI exists for distributing providers' public keys, which they use to sign authenticated bindings and to transform users' names for privacy purposes.

While we assume that CONIKS providers may be malicious, we assume they have a reputation to protect and do not wish to attack their users in a public manner. Because CONIKS primarily provides transparency and enables reactive security in case of provider attacks, CONIKS cannot deter a service provider which is willing to attack its users openly (although it will expose the attacks).

**Clients.** Users run CONIKS client software on one or more trusted devices; CONIKS does not address the problem of compromised client endpoints. Clients *monitor* the consistency of their user's own bindings. To support monitoring, we assume that at least one of a user's clients has access to a reasonably accurate clock as well as access to secure local storage in which the client can save the results of prior checks.

We also assume clients have network access which cannot be reliably blocked by their communication provider. This is necessary for *whistleblowing* if a client detects

[1] Existing communication service providers can act as identity providers, although CONIKS also enables dedicated "stand-alone" identity providers to become part of the system.

misbehavior by an identity provider (more details in §4.2). CONIKS cannot ensure security if clients have no means of communication that is not under their communication provider's control.[2]

**Auditors.** To verify that identity providers are not equivocating, *auditors* track the chain of signed "snapshots" of the key directory. Auditors publish and gossip with other auditors to ensure global consistency. Indeed, CONIKS clients all serve as auditors for their own identity provider and providers audit each other. Third-party auditors are also able to participate if they desire.

**Users.** An important design strategy is to provide good baseline security which is accessible to nearly all users, necessarily requiring some security tradeoffs, with the opportunity for upgraded security for advanced users within the same system to avoid fragmenting the communication network. While there are many gradations possible, we draw a recurring distinction between *default users* and *strict users* to illustrate the differing security properties and usability challenges of the system.

We discuss the security tradeoffs between these two user security policies in §4.3.

## 2.2 Design Goals

The design goals of CONIKS are divided into security, privacy and deployability goals.

**Security goals.**
**G1: Non-equivocation.** An identity provider may attempt to equivocate by presenting diverging views of the name-to-key bindings in its namespace to different users. Because CONIKS providers issue signed, chained "snapshots" of each version of the key directory, any equivocation to two distinct parties must be maintained forever or else it will be detected by auditors who can then broadcast non-repudiable cryptographic evidence, ensuring that equivocation will be detected with high probability (see Appendix B for a detailed analysis).
**G2: No spurious keys.** If an identity provider inserts a malicious key binding for a given user, her client software will rapidly detect this and alert the user. For default users, this will not produce non-repudiable evidence as key changes are not necessarily cryptographically signed with a key controlled by the user. However, the user will still see evidence of the attack and can report it publicly. For strict users, all key changes must be signed by the user's previous key and therefore malicious bindings will not be accepted by other users.

[2] Even given a communication provider who also controls all network access, it may be possible for users to whistleblow manually by reading information from their device and using a channel such as physical mail or sneakernet, but we will not model this in detail.

**Privacy goals.**
**G3: Privacy-preserving consistency proofs.** CONIKS servers do not need to make any information about their bindings public in order to allow consistency verification. Specifically, an adversary who has obtained an arbitrary number of consistency proofs at a given time, even for adversarially chosen usernames, cannot learn any information about which other users exist in the namespace or what data is bound to their usernames.
**G4: Concealed number of users.** Identity providers may not wish to reveal their exact number of users. CONIKS allows providers to insert an arbitrary number of *dummy entries* into their key directory which are indistinguishable from real users (assuming goal G3 is met), exposing only an upper bound on the number of users.

**Deployability goals.**
**G5: Strong security with human-readable names.** With CONIKS, users of the system only need to learn their contacts' usernames in order to communicate with end-to-end encryption. They need not explicitly reason about keys. This enables seamless integration in end-to-end encrypted communication systems and requires no effort from users in normal operation.
**G6: Efficiency.** Computational and communication overhead should be minimized so that CONIKS is feasible to implement for identity providers using commodity servers and for clients on mobile devices. All overhead should scale at most logarithmically in the number of total users.

# 3 Core Data Structure Design

At a high level, CONIKS identity providers manage a directory of verifiable bindings of usernames to public keys. This directory is constructed as a Merkle prefix tree of all registered bindings in the provider's namespace.

At regular time intervals, or epochs, the identity provider generates a non-repudiable "snapshot" of the directory by digitally signing the root of the Merkle tree. We call this snapshot a *signed tree root* (STR) (see §3.3). Clients can use these STRs to check the consistency of key bindings in an efficient manner, obviating the need for clients to have access to the entire contents of the key directory. Each STR includes the hash of the previous STR, committing to a linear history of the directory.

To make the directory privacy-preserving, CONIKS employs two cryptographic primitives. First, a *private index* is computed for each username via a verifiable unpredictable function (described in §3.4). Each user's keys are stored at the associated private index rather than his or her username (or a hash of it). This prevents the data structure from leaking information about usernames. Second, to ensure that it is not possible to test if a users'

key data is equal to some known value even given this user's lookup index, a cryptographic *commitment*[3] to each user's key data is stored at the private index, rather than the public keys themselves.

## 3.1 Merkle Prefix Tree

CONIKS directories are constructed as Merkle binary prefix trees. Each node in the tree represents a unique prefix $i$. Each branch of the tree adds either a 0 or a 1 to the prefix of the parent node. There are three types of nodes, each of which is hashed slightly differently into a representative value using a collision-resistant hash $H()$:

**Interior nodes** exist for any prefix which is shared by more than one index present in the tree. An interior node is hashed as follows, committing to its two children:

$$h_{\text{interior}} = \mathrm{H}\left(h_{\text{child.0}}||h_{\text{child.1}}\right)$$

**Empty nodes** represent a prefix $i$ of length $\ell$ (depth $\ell$ in the tree) which is not a prefix of any index included in the tree. Empty nodes are hashed as:

$$\mathrm{h}_{\text{empty}} = \mathrm{H}\left(k_{\text{empty}}||k_n||i||\ell\right)$$

**Leaf nodes** represent exactly one complete index $i$ present in the tree at depth $\ell$ (meaning its first $\ell$ bits form a unique prefix). Leaf nodes are hashed as follows:

$$\mathrm{h}_{\text{leaf}} = \mathrm{H}\left(k_{\text{leaf}}||k_n||i||\ell||\text{commit}(\text{name}_i||\text{keys}_i)\right)$$

where $\text{commit}(\text{name}_i||\text{keys}_i)$ is a cryptographic commitment to the name and the associated key data. Committing to the name, rather than the index $i$, protects against collisions in the VRF used to generate $i$ (see §3.4).

**Collision attacks.** While arbitrary collisions in the hash function are not useful, a malicious provider can mount a birthday attack to try to find two nodes with the same hash (for example by varying the randomness used in the key data commitment). Therefore, for $t$-bit security our hash function must produce at least $2t$ bits of output.

The inclusion of depths $\ell$ and prefixes $i$ in leaf and empty nodes (as well as constants $k_{\text{empty}}$ and $k_{\text{leaf}}$ to distinguish the two) ensures that no node's pre-image can be valid at more than one location in the tree (including interior nodes, whose location is implicit given the embedded locations of all of its descendants). The use of a tree-wide nonce $k_n$ ensures that no node's pre-image can be valid at the same location between two distinct trees which have chosen different nonces. Both are countermeasures for the *multi-instance setting* of an attacker attempting to find

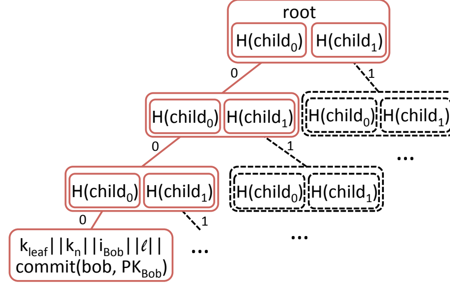


**Figure 1: An authentication path from Bob's key entry to the root node of the Merkle prefix tree. Bob's index, $i_{\mathbf{Bob}}$, has the prefix "000". Dotted nodes are not included in the proof's authentication path.**

a collision at more than one location simultaneously.[4]. Uniquely encoding the location requires the attacker to target a specific epoch and location in the tree and ensures full $t$-bit security.

If the tree-wide nonce $k_n$ is re-used between epochs, a parallel birthday attack is possible against each version of the tree. However, choosing a new $k_n$ each epoch means that every node in the tree will change.

## 3.2 Proofs of Inclusion

Since clients no longer have a direct view on the contents of the key directory, CONIKS needs to be able to prove that a particular index exists in the tree. This is done by providing a proof of inclusion which consists of the complete *authentication path* between the corresponding leaf node and the root. This is a pruned tree containing the prefix path to the requested index, as shown in Figure 1. By itself, this path only reveals that an index exists in the directory, because the commitment hides the key data mapped to an index. To prove inclusion of the full binding, the server provides an opening of the commitment in addition to the authentication path.

**Proofs of Absence.** To prove that a given index $j$ has no key data mapped to it, an authentication path is provided to the longest prefix match of $j$ currently in the directory. That node will either be a leaf node at depth $\ell$ with an index $i \neq j$ which matches $j$ in the first $\ell$ bits, or an empty node whose index $i$ is a prefix of $j$.

## 3.3 Signed Tree Roots

At each epoch, the provider signs the root of the directory tree, as well as some metadata, using their directory-signing key $SK_d$. Specifically, an STR consists of

$$\text{STR} = \text{Sign}_{SK_d}\left(t||t_{prev}||\text{root}_t||\text{H}(\text{STR}_{prev})||P\right)$$

[3] Commitments are a basic cryptographic primitive. A simple implementation computes a collision-resistant hash of the input data and a random nonce.

[4] This is inspired by Katz' analysis [34] of hash-based signature trees

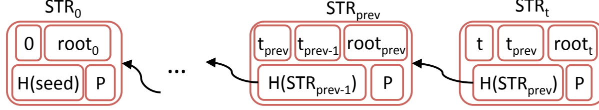


**Figure 2: The directory's history is published as a linear hash chain of signed tree roots.**

where $t$ is the epoch number and $P$ is a summary of this provider's current security policies. $P$ may include, for example, the key $K_{\mathrm{VRF}}$ used to generate private indices, an expected time the next epoch will be published, as well as the cryptographic algorithms in use, protocol version numbers, and so forth. The previous epoch number $t_{prev}$ must be included because epoch numbers need not be sequential (only increasing). In practice, our implementation uses UNIX timestamps.

By including the hash of the previous epoch's STR, the STRs form a *hash chain* committing to the entire history, as shown in Figure 2. This hash chain is used to ensure that if an identity provider ever equivocates by creating a fork in its history, the provider must maintain these forked hash chains for the rest of time (i.e. it must maintain *fork consistency* [42]). Otherwise, clients will immediately detect the equivocation when presented with an STR belonging to a different branch of the hash chain.

## 3.4 Private Index Calculation

A key design goal is to ensure that each authentication path reveals no information about whether any *other* names are present in the directory. If indices were computed using any publicly computable function of the username (such as a simple hash), each user's authentication path would reveal information about the presence of other users with prefixes "close" to that user.

For example, if a user *alice@foo.com*'s shortest unique prefix in the tree is $i$ and her immediate neighbor in the tree is a non-empty node, this reveals that at least one users exists with the same prefix $i$. An attacker could hash a large number of potential usernames offline, searching for a potential username whose index shares this prefix $i$.

**Private Indices.** To prevent such leakage, we compute private indices using a *verifiable random function*, a pseudorandom function that requires a private key to compute. VRF computations can be verified with a short proof, but without the proof the result is indistinguishable from random.[5] Given such a function VRF(), we generate the index $i$ for a user $u$ as:

$$i = \mathrm{H}\left(\mathrm{VRF}_{K_{\mathrm{VRF}}}(u)\right)$$

[5] Related to VRFs are verifiable unpredictable functions [48], whose output is self-verifying without a distinct proof and therefore not indistinguishable from random. This is insufficient for privacy because a username might be recovered given some bits of a VUF result.

$K_{\mathrm{VRF}}$ is a public key belonging to the provider, and it is specified in the policy field of each STR. A full proof of inclusion for user $u$ therefore requires the value of $\mathrm{VRF}(u)$ in addition to the authentication path and an opening of the commitment to the user's key data.

We can implement a VRF by applying a hash function (modeled as a random oracle) to the result of any deterministic, existentially unforgeable signature scheme [48]. The signature scheme must be deterministic or else the identity provider could insert multiple bindings for a user at different locations each with a valid authentication path. We discuss our choice for this primitive in §5.2.

Note that we might like our VRF to be collision-resistant to ensure that a malicious provider cannot produce two usernames $u, u'$ which map to the same index. However, VRFs are not guaranteed to be collision-resistant given knowledge of the private key (and the ability to pick this key maliciously). To prevent any potential problems we commit to the username $u$ in each leaf node. This ensures that only one of $u$ or $u'$ can be validly included in the tree even if the provider has crafted them to share an index.

# 4 CONIKS Operation

With the properties of key directories outlined in §3, CONIKS provides four efficient protocols that together allow end users to verify each other's keys to communicate securely: registration, lookup, monitoring and auditing. In these protocols, providers, clients and auditors collaborate to ensure that identity providers do not publish spurious keys, and maintain a single linear history of STRs.

## 4.1 Protocols

### 4.1.1 Registration and Temporary Bindings

CONIKS provides a registration protocol, which clients use to register a new name-to-key binding with an identity provider on behalf of its user, or to update the public key of the user's existing binding when revoking her key. An important deployability goal is for users to be able to communicate immediately after enrollment. This means users must be able to use new keys *before* they can be added to the key directory. An alternate approach would be to reduce the epoch time to a very short interval (on the order of seconds). However, we consider this undesirable both on the server end and in terms of client overhead.

CONIKS providers may issue *temporary bindings* without writing any data to the Merkle prefix tree. A temporary binding consists of:

$$\mathrm{TB} = \mathrm{Sign}_{K_d}\left(STR_t, i, k\right)$$

The binding includes the most recent signed tree root $STR_t$, the index $i$ for the user's binding, and the user's new key information $k$. The binding is signed by the identity provider, creating a non-repudiable promise to add this data to the next version of the tree.

To register a user's key binding with a CONIKS identity provider, her client now participates in the following protocol. First, the client generates a key pair for the user and stores it in some secure storage on the device. Next, the client sends a registration request to the provider to the bind the public key to the user's online name, and if this name is not already taken in the provider's namespace, it returns a temporary binding for this key. The client then needs to wait for the next epoch and ensure that the provider has kept its promise of inserting Alice's binding into its key directory by the next epoch.

#### 4.1.2 Key Lookups

Since CONIKS clients only regularly check directory roots for consistency, they need to ensure that public keys retrieved from the provider are contained in the most recently validated directory. Thus, whenever a CONIKS client looks up a user's public key to contact her client, the provider also returns a proof of inclusion showing that the retrieved binding is consistent with a specific STR. This way, if a malicious identity provider attempts to distribute a spurious key for a user, it is not able to do so without leaving evidence of the misbehavior. Any client that looks up this user's key and verifies that the binding is included in the presented STR will then promptly detect the attack.

In more detail, CONIKS's lookup protocol achieves this goal in three steps (summarized in Fig. 3). When a user wants to send a secure message to another user, her client first requests the recipient's public key at her provider. To allow the client to check whether the recipient's binding is included in the STR for the current epoch, the identity provider returns the full authentication path for the recipient's binding in the Merkle prefix tree along with the current STR. In the final step, the client recomputes the root of the tree using the authentication path and checks that this root is consistent with the presented STR. Note that, if the recipient has not registered a binding with the identity provider, it returns an authentication path as a proof of absence allowing the client to verify that the binding is indeed absent in the tree and consistent with the current STR.

#### 4.1.3 Monitoring for Spurious Keys

CONIKS depends on the fact that each client monitors its own user's binding every epoch to ensure that her key binding has not changed unexpectedly. This prevents a malicious identity provider from inserting spurious keys

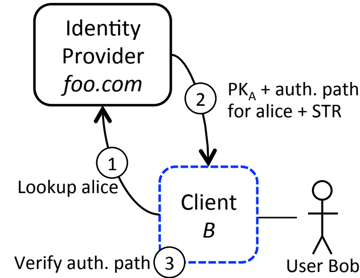


**Figure 3: Steps taken when a client looks up a user's public key at her identity provider.**

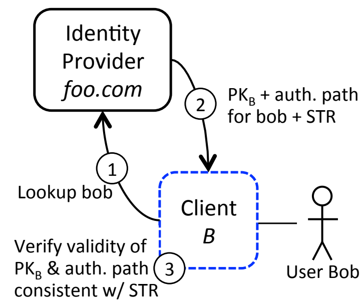


**Figure 4: Steps taken when a client monitors its own user's binding for spurious keys every epoch.**

that are properly included in the STR. Clients do not monitor other user's bindings as they may not have enough information to determine when another user's binding has changed unexpectedly.

Fig. 4 summarizes the steps taken during the monitoring protocol. The client begins monitoring by performing a key lookup for its own user's name to obtain a proof of inclusion for the user's binding. Next, the client checks the binding to ensure it represents the public key data the user believes is correct. In the simplest case, this is done by checking that a user's key is consistent between epochs. If the keys have not changed, or the client detects an authorized key change, the user need not be notified.

In the case of an unexpected key change, by default the user chooses what course of action to take as this change may reflect, for example, having recently enrolled a new device with a new key. Alternatively, security-conscious users may request a stricter key change policy which can be automatically enforced, and which we discuss further in §4.3. After checking the binding for spurious keys, the client verifies the authentication path as described in §3, including verifying the user's private index.

#### 4.1.4 Auditing for Non-Equivocation

Even if a client monitors its own user's binding, it still needs to ensure that its user's identity provider is presenting consistent versions of its key directory to all participants in the system. Similarly, clients need to check

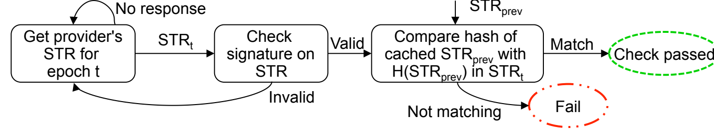


**Figure 5: Steps taken when verifying if a provider's STR history is linear in the auditing protocol.**

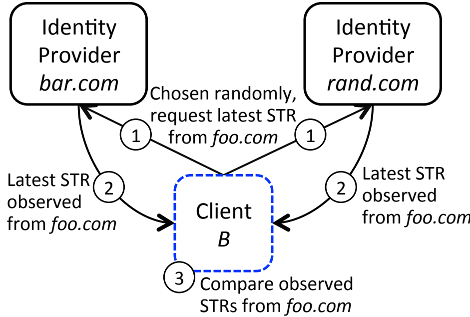


**Figure 6: Steps taken when comparing STRs in the auditing protocol.**

that the identity provider of any user they contact is not equivocating about its directory. In other words, clients need to verify that any provider of interest is maintaining a linear STR history. Comparing each observed STR with every single other client with which a given client communicates would be a significant performance burden.

Therefore, CONIKS allows identity providers to facilitate auditing for their clients by acting as auditors of all CONIKS providers with which their users have been in communication (although it is also possible for any other entity to act as an auditor). Providers achieve this by distributing their most recent STR to other identity providers in the system at the beginning of every epoch.[6]

The auditing protocol in CONIKS checks whether an identity provider is maintaining a linear STR history. Identity providers perform the history verification whenever they observe a new STR from any other provider, while clients do so whenever they request the most recent STR from a specific identity provider directly. We summarize the steps required for an auditor to verify an STR history in Fig. 5. The auditor first ensures that the provider correctly signed the STR before checking whether the embedded hash of the previous epoch's STR matches what the auditor saw previously. If they do not match, the provider has generated a fork in its STR history.

Because each auditor has independently verified a provider's history, each has its own view of a provider's STR, so clients must perform an STR comparison to check for possible equivocation between these views (summarized in Fig. 6). Once a client has verified the provider's STR history is linear, the client queries one or more CONIKS identity providers at random.[7] The client asks the auditor for the most recent STR it observed from the provider in question. Because the auditor has already verified the provider's history, the client need not verify the STR received from the auditor. The client then compares the auditor's observed STR with the STR which the provider directly presented it. The client may repeat this process with different auditors as desired to increase confidence. For an analysis of the number of checks necessary to detect equivocation with high probability, see App. B.

CONIKS auditors store the current STRs of CONIKS providers; since the STRs are chained, maintaining the current STR commits to the entire history. Because this is a small, constant amount of data (less than 1 kB) it is efficient for a single machine to act as an auditor for thousands of CONIKS providers.

## 4.2 Secure Communication with CONIKS

When a user Bob wants to communicate with a user Alice via their CONIKS-backed secure messaging service *foo.com*, his client *client B* performs the following steps. We assume both Alice's and Bob's clients have registered their respective name-to-key bindings with *foo.com* as described in §4.1.1.

1. Periodically, *client B* checks the consistency of Bob's binding. To do so, the client first performs the monitoring protocol (per §4.1.3), and then it audits *foo.com* (per §4.1.4).
2. Before sending Bob's message to *client A*, *client B* looks up the public key for the username *alice* at *foo.com* (§4.1.2). It verifies the proof of inclusion for *alice* and performs the auditing protocol (§4.1.4) for *foo.com* if the STR received as part of the lookup is different or newer than the STR it observed for *foo.com* in its latest run of step 1.
3. If *client B* determines that Alice's binding is consistent, it encrypts Bob's message using *alice*'s public key and signs it using Bob's key. It then sends the message.

**Performing checks after missed epochs.** Because STRs are associated with each other across epochs, clients can "catch up" to the most recent epoch if they have not veri-

[6] CONIKS could support an auditing protocol in which clients directly exchange observed STRs, obviating the need of providers to act as auditors. The design of such a protocol is left as future work.

[7] We assume the client maintains a list of CONIKS providers acting as auditors from which it can choose any provider with equal probability. The larger this list, the harder it is for an adversary to guess which providers a client will query.

fied the consistency of a binding for several epochs. They do so by performing a series of the appropriate checks until they are sure that the proofs of inclusion and STRs they last verified are consistent with the more recent proofs. This is the only way a client can be sure that the security of its communication has not been compromised during the missed epochs.

**Liveness.** CONIKS servers may attempt to hide malicious behavior by ceasing to respond to queries. We provide flexible defense against this, as servers may also simply go down. Servers may publish an expected next epoch number with each STR in the policy section $P$. Clients must decide whether they will accept STRs published at a later time than previously indicated.

**Whistleblowing.** If a client ever discovers two inconsistent STRs (for example, two distinct versions signed for the same epoch time), they should notify the user and *whistleblow* by publishing them to all auditors they are able to contact. For example, clients could include them in messages sent to other clients, or they could explicitly send whistleblowing messages to other identity providers. We also envision out-of-band whistleblowing in which users publish inconsistent STRs via social media or other high-traffic sites. We leave the complete specification of a whistleblowing protocol for future work.

## 4.3 Multiple Security Options

CONIKS gives users the flexibility to choose the level of security they want to enforce with respect to key lookups and key change. For each functionality, we propose two security policies: a default policy and a strict policy, which have different tradeoffs of security and privacy against usability. All security policies are denoted by flags that are set as part of a user's directory entry, and the consistency checks allow users to verify that the flags do not change unexpectedly.

### 4.3.1 Visibility of Public Keys

Our goal is to enable the same level of privacy SMTP servers employ today,[8] in which usernames can be queried (subject to rate-limiting) but it is difficult to enumerate the entire list of names.

Users need to decide whether their public key(s) in the directory should be publicly visible. The difference between the default and the strict lookup policies is whether the user's public keys are encrypted with a secret symmetric key known only to the binding's owner and any other user of her choosing. For example, if the user Alice follows the default lookup policy, her public keys are not encrypted. Thus, anyone who knows Alice's name *alice@foo.com* can look up and obtain her keys from her *foo.com*'s directory. On the other hand, if Alice follows the strict lookup policy, her public keys are encrypted with a symmetric key only known to Alice and the users of her choosing.

Under both lookup policies, any user can verify the consistency of Alice's binding as described in §4, but if she enforces the strict policy, only those users with the symmetric key learn her public keys. The main advantage of the default policy is that it matches users' intuition about interacting with any user whose username they know without requiring explicit "permission". On the other hand, the strict lookup policy provides stronger privacy, but it requires additional action to distribute the symmetric key which protects her public keys.

[8] The SMTP protocol defines a *VRFY* command to query the existence of an email address at a given server. To protect user's privacy, however, it has long been recommended to ignore this command (reporting that any usernames exists if asked) [43].

### 4.3.2 Key Change

Dealing with key loss is a difficult quandary for any security system. Automatic key recovery is an indispensable option for the vast majority of users who cannot perpetually maintain a private key. Using password authentication or some other fallback method, users can request that identity providers change a user's public key in the event that the user's previous device was lost or destroyed. If Alice chooses the default key change policy, her identity provider *foo.com* accepts any key change statement in which the new key is signed by the previous key, as well as unsigned key change requests. Thus, *foo.com* should change the public key bound to *alice@foo.com* only upon her request, and it should reflect the update to Alice's binding by including a key change statement in her directory entry. The strict key change policy *requires* that Alice's client sign all of her key change statements with the key that is being changed. Thus, Alice's client only considers a new key to be valid if the key change statement has been authenticated by one of her public keys.

While the default key change policy makes it easy for users to recover from key loss and reclaim their username, it allows an identity provider to maliciously change a user's key and falsely claim that the user requested the operation. Only Alice can determine with certainty that she has not requested the new key (and password-based authentication means the server cannot prove Alice requested it). Still, her client will detect these updates and can notify Alice, making surreptitious key changes risky for identity providers to attempt. Requiring authenticated key changes, on the other hand, does sacrifice the ability for Alice to regain control of her username if her key is

ever lost. We discuss some implications for key loss for strict users in §6.

# 5 Implementation and Evaluation

CONIKS provides a framework for integrating key verification into communications services that support end-to-end encryption. To demonstrate the practicality of CONIKS and how it interacts with existing secure communications services, we implemented a prototype CONIKS Chat, a secure chat service based on the Off-the-Record Messaging [8] (OTR) plug-in for the Pidgin instant messaging client [1, 27]. We implemented a stand-alone CONIKS server in Java (∼2.5k sloc), and modified the OTR plug-in (∼2.2k sloc diff) to communicate with our server for key management. We have released a basic reference implementation of our prototype on Github.[9]

## 5.1 Implementation Details

CONIKS Chat consists of an enhanced OTR plug-in for the Pidgin chat client and a stand-alone CONIKS server which runs alongside an unmodified Tigase XMPP server. Clients and servers communicate using Google Protocol Buffers [2], allowing us to define specific message formats. We use our client and server implementations for our performance evaluation of CONIKS.

Our implementation of the CONIKS server provides the basic functionality of an identity provider. Every version of the directory (implemented as a Merkle prefix tree) as well as every generated STR are persisted in a MySQL database. The server supports key registration in the namespace of the XMPP service, and the directory efficiently generates the authentication path for proofs of inclusion and proofs of absence, both of which implicitly prove the proper construction of the directory. Our server implementation additionally supports STR exchanges between identity providers.

The CONIKS-OTR plug-in automatically registers a user's public key with the server upon the generation of a new key pair and automatically stores information about the user's binding locally on the client to facilitate future consistency checks. To facilitate CONIKS integration, we leave the DH-based key exchange protocol in OTR unchanged, but replace the socialist millionaires protocol used for key verification with a public key lookup at the CONIKS server. If two users, Alice and Bob, both having already registered their keys with the *coniks.org* identity provider, want to chat, Alice's client will automatically request a proof of inclusion for Bob's binding in *coniks.org*'s most recent version of the directory. Upon receipt of this proof, Alice's client automatically verifies the authentication path for Bob's name-to-key binding (as described in §4.1.2), and caches the newest information about Bob's binding if the consistency checks pass. If Bob has not registered his key with *coniks.org*, the client falls back to the original key verification mechanism. Additionally, Alice's client and Bob's clients automatically perform all monitoring and auditing checks for their respective bindings upon every login and cache the most recent proofs.

CONIKS Chat currently does not support key changes. Furthermore, our prototype only supports the default lookup policy for name-to-key bindings. Fully implementing these features is planned for the near future.

## 5.2 Choice of Cryptographic Primitives

To provide a 128-bit security level, we use SHA-256 as our hash function and EC-Schnorr signatures [22, 64].

Unfortunately Schnorr signatures (and related discrete-log based signature schemes like DSA [37]) are not immediately applicable as a VRF as they are not deterministic, requiring a random nonce which the server can choose arbitrarily.[10] In Appendix A we describe a discrete-log based scheme for producing a VRF in the random-oracle model. Note that discrete-log based VRFs are longer than basic signatures: at a 128-bit security level using elliptic curves, we expect signatures of size 512 bits and 768 bits for the VRF result and proof.

Alternately, we could build a VRF from a deterministic signature scheme like classic RSA signatures [60] (using a deterministic padding scheme such as PKCS v. 1.5 [32]) plus an additional hash, although RSA is not particularly space-efficient at a 128-bit security level. Using RSA-2048 provides approximately 112 bits of security [3] with proofs of size 2048 bits. [11]

Using elliptic curve groups equipped with a pairing, we can use BLS "short signatures" [7] plus the VRF construction of Dodis et al. [15] to achieve 256-bit signatures and VRFs of 512 bits (including the result and a 256-bit proof). BLS signatures also support *aggregation*, that is, multiple signatures can be compressed into a single signature, meaning the server can combine the signatures on $n$ consecutive roots. However support for pairing calculations required for BLS is not as widespread, making it more difficult to standardize and deploy.

We evaluate performance in Table 1 in the next section for all three potential choices of primtives to build the signature and VRF schemes.

[9] `https://github.com/coniks-sys/coniks-ref-implementation`

[10] There are deterministic variants of Schnorr or DSA [5, 50] but these are not *verifiably* deterministic as they generate nonces pseudorandomly as a symmetric-key MAC of the data to be signed.

[11] We might tolerate slightly lower security in our VRF than our signature scheme, as this key only ensures privacy and not non-equivocation.

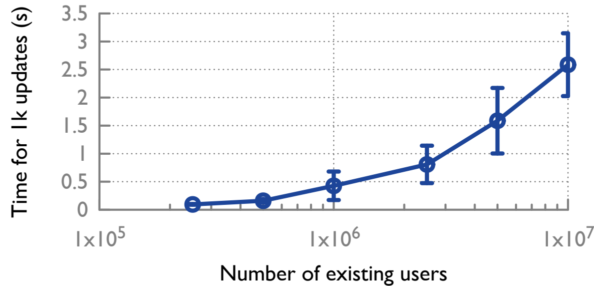


**Figure 7: Mean time to re-compute the tree for a new epoch with 1K updated nodes. The x-axis is logarithmic and each data point is the mean of 10 executions. Error bars indicate standard deviation.**

## 5.3 Performance Evaluation

To estimate the performance of CONIKS, we collect both theoretical and real performance characteristics of our prototype implementation. We evaluate client and server overheads with the following parameters:

- A single provider might support $N \approx 2^{32}$ users.
- Epochs occur roughly once per hour.
- Up to 1% of users change or add keys per day, meaning $n \approx 2^{21}$ directory updates in an average epoch.
- Servers use a 128-bit cryptographic security level.

**Server Overheads.** To measure how long it takes for a server to compute the changes for an epoch, we evaluated our server prototype on a 2.4 GHz Intel Xeon E5620 machine with 64 GB of RAM allotted to the OpenJDK 1.7 JVM. We executed batches of 1000 insertions (roughly 3 times the expected number of directory updates per epoch) into a Merkle prefix with 10 M users, and measured the time it took for the server to compute the next epoch.

Figure 7 shows the time to compute a version of the directory with 1000 new entries as the size of the original namespace varies. For a server with 10 M users, computing a new Merkle tree with 1000 insertions takes on average 2.6 $s$. As epochs only need to be computed every hour, this is not cumbersome for a large service provider. These numbers indicate that even with a relatively unoptimized implementation, a single machine is able to handle the additional overhead imposed by CONIKS for workloads similar in scale to a medium-sized communication providers (e.g., TextSecure) today.

While our prototype server implementation on a commodity machine comfortably supports 10M users, we note that due to the statistically random allocation of users to indices and the recursive nature of the tree structure, the task parallelizes near-perfectly and it would be trivial to scale horizontally with additional identical servers to compute a directory with billions of users.

**Lookup Cost.** Every time a client looks up a user's binding, it needs to download the current STR, a proof of of inclusion consisting of about $\lg_2(N)+1$ hashes plus one 96-byte VRF proof (proving the validity of the binding's private index). This will require downloading $32 \cdot (\lg_2(N)+1)+96 \approx 1216$ bytes. Verifying the proof will require up to $\lg_2(N)+1$ hash verifications on the authentication path as well as one VRF verification. On a 2 GHz Intel Core i7 laptop, verifying the authentication path returned by a server with 10 million users, required on average 159 $\mu s$ (sampled over 1000 runs, with $\sigma = 30$). Verifying the signature takes approximately 400 $\mu s$, dominating the cost of verifying the authentication path. While mobile-phone clients would require more computation time, we do not believe this overhead presents a significant barrier to adoption.

**Monitoring Cost.** In order for any client to monitor the consistency of its own binding, it needs fetch proof that this binding is validly included in the epoch's STR. Each epoch's STR signature (64 bytes) must be downloaded and the client must fetch its new authentication path. However, the server can significantly compress the length of this path by only sending the hashes on the user's path which have changed since the last epoch. If $n$ changes are made to the tree, a given authentication path will have $\lg_2(n)$ expected changed nodes. (This is the expected longest prefix match between the $n$ changed indices and the terminating index of the given authentication path.) Therefore each epoch requires downloading an average of $64+\lg_2(n) \cdot 32 \approx 736$ bytes. Verification time will be similar to verifying another user's proof, dominated by the cost of signature verification. While clients need to fetch each STR from the server, they are only required to *store* the most recent STR (see §5.3).

To monitor a binding for a day, the client must download a total of about 19.1 kB. Note that we have assumed users update randomly throughout the day, but for a fixed number of updates this is actually the worst-case scenario for bandwidth consumption; bursty updates will actually lead to a lower amount of bandwidth as each epoch's proof is $\lg_2(n)$ for $n$ changes. These numbers indicate that neither bandwidth nor computational overheads pose a significant burden for CONIKS clients.

**Auditing cost.** For a client or other auditor tracking all of a provider's STRs, assuming the policy field changes rarely, the only new data in an STR is the new timestamp, the new tree root and signature (the previous STR and epoch number can be inferred and need not be transmitted). The total size of each STR in minimal form is just 104 bytes (64 for the signature, 32 for the root and 8 for a timestamp), or 2.5 kB per day to audit a specific provider.

| | # VRFs | # sigs. | # hashes | approximate bandwidth RSA | | traditional EC | | EC w/pairings | |
|---|---|---|---|---|---|---|---|---|---|
| lookup (per binding) | 1 | 1 | $\lg N + 1$ | 1600 | B | 1216 | B | 1152 | B |
| monitor (epoch) | 0 | 1 | $\lg n$ | 928 | B | 736 | B | 704 | B |
| monitor (day) | 1 | $k^{\dagger}$ | $k \lg n$ | 22.6 | kB | 17.6 | kB | 16.9 | kB |
| audit (epoch, per STR) | 0 | 1 | 1 | 288 | B | 96 | B | 64 | B |
| audit (day, per STR) | 0 | $k^{\dagger}$ | $k$ | 6.9 | kB | 2.3 | kB | 0.8 | kB |

**Table 1: Client bandwidth requirements for three underlying public-key cryptographic primitive: RSA, elliptic curves using EC-Schnorr plus the VRF defined in Appendix A, and an elliptic curve group equipped with a bilinear pairing using BLS signatures [7] and the Dodis et al. VRF [15]. Results are given for routine lookups, monitoring, and auditing in terms of the number of signatures, VRFs and hashes downloaded assuming $N \approx 2^{32}$ total users, $n \approx 2^{21}$ changes per epoch, and $k \approx 24$ epochs per day. Signatures that can be aggregated into a single signature to transmit in the BLS signature scheme are denoted by †.**

# 6 Discussion

## 6.1 Coercion of Identity Providers

Government agencies or other powerful adversaries may attempt to coerce identity providers into malicious behavior. Recent revelations about government surveillance and collection of user communications data world-wide have revealed that governments use mandatory legal process to demand access to information providers' data about users' private communications and Internet activity [9, 24, 25, 52, 53]. A government might demand that an identity provider equivocate about some or all name-to-key bindings. Since the identity provider is the entity actually mounting the attack, a user of CONIKS has no way of technologically differentiating between a malicious insider attack mounted by the provider itself and this coerced attack [19]. Nevertheless, because of the consistency and non-equivocation checks CONIKS provides, users could expose such attacks, and thereby mitigate their effect.

Furthermore, running a CONIKS server may provide some legal protection for service providers under U.S. law for providers attempting to fight legal orders, because complying with such a demand will produce public evidence that may harm the provider's reputation. (Legal experts disagree about whether and when this type of argument shelters a provider[46].)

## 6.2 Key Loss and Account Protection

CONIKS clients are responsible for managing their private keys. However, CONIKS can provide account protection for users who enforce the paranoid key change policy and have forfeit their username due to key loss. Even if Alice's key is lost, her identity remains secure; she can continue performing consistency checks on her old binding. Unfortunately, if a future attacker manages to obtain her private key, that attacker may be able to assume her "lost identity".

In practice, this could be prevented by allowing the provider to place a "tombstone" on a name with its own signature, regardless of the user's key policy. The provider would use some specific out-of-band authorization steps to authorize such an action. Unlike allowing providers to issue key change operations, though, a permanent account deactivation does not require much additional trust in the provider, because a malicious provider could already render an account unusable through denial of service.

## 6.3 Protocol Extensions

**Speeding up auditing with version history.** By default, clients who go offline for an extended period of time, say $\ell$ epochs, face $\Theta(\ell)$ monitoring costs to verify that their data was not changed in any of the epochs they missed. It is not sufficient to simply check that their data in the most recent STR is the same as it was at the point they went offline, as the server might have changed their binding at some point but changed it back prior to the client coming back online.

These checks could be skipped if each leaf contained trustworthy version history information. In the simplest implementation, this could be an *update counter* in each leaf representing the number of times the data bound to that leaf's index has been changed. A more powerful version would store a complete *binding history* in each leaf (e.g. in an append-only binary Merkle tree), containing commitments to all prior values for that binding. In either case, clients could then skip monitoring their binding in every epoch they missed as long as the most recent STR indicates that their data hasn't been changed since the last STR they observed before going offline. This could enable the server to use much shorter epoch without overwhelming clients with limited network bandwidth.

However, clients would need to trust a third-party auditor to check that in each epoch the server correctly updates the version history information for all leaves which have changed. This would require $\Theta(n)$ work by the auditor if $n$ changes are made in a typical epoch, so it may not be realistic in all deployments to assume an auditor exists that will do this checking for free. However, there is little cost to adding version history information—clients which don't want to trust any third-party auditors can continue to monitor their binding for changes in every epoch.

**Limiting the effects of denied service.** Sufficiently powerful identity providers may refuse to distribute STRs to providers with which they do not collude. In these cases, clients who query these honest providers will be unable to obtain explicit proof of equivocation. Fortunately, clients may help circumvent this by submitting observed STRs to these honest identity providers. The honest identity providers can verify the other identity provider's signature, and then store and redistribute the STR.

Similarly, any identity provider might ignore requests about individual bindings in order to prevent clients from performing consistency checks or key changes. In these cases, clients may be able to circumvent this attack by using other providers to proxy their requests, with the caveat that a malicious provider may ignore all requests for a name. This renders this binding unusable for as long as the provider denies service. However, this only allows the provider to deny service, any modification to the binding during this attack would become evident as soon as the service is restored.

**Obfuscating the social graph.** As an additional privacy requirement, users may want to conceal with whom they are in communication, or providers may want to offer anonymized communication. In principle, users could use Tor to anonymize their communications. However, if only few users in CONIKS use Tor, it is possible for providers to distinguish clients connecting through Tor from those connecting to the directly.

CONIKS could leverage the proxying mechanism described in §6.3 for obfuscating the social graph. If Alice would like to conceal with whom she communicates, she could require her client to use other providers to proxy any requests for her contacts' bindings or consistency proofs. Clients could choose these proxying providers uniformly at random to minimize the amount of information any single provider has about a particular user's contacts. This can be further improved the more providers agree to act as proxies. Thus, the only way for providers to gain information about whom a given user is contacting would be to aggregate collected requests. For system-wide Tor-like anonymization, CONIKS providers could form a mixnet [13], which would provide much higher privacy guarantees but would likely hamper the deployability of the system.

**Randomizing the order of directory entries.** Once a user learns the lookup index of a name, this position in the tree is known for the rest of time because the index is a deterministic value. If a user has an authentication path for two users *bob@foo.com* and *alice@foo.com* which share a common prefix in the tree, the Bob's authentication path will leak any changes to Alice's binding if his key has not changed, and vice-versa. *foo.com* can prevent this information leakage by randomizing the ordering of entries periodically by including additional data when computing their lookup indices. However, such randomized reordering of all directory entries would require a complete reconstruction of the tree. Thus, if done every epoch, the identity provider would be able to provide enhanced privacy guarantees at the expense of efficiency. The shorter the epochs, the greater the tradeoff between efficiency and privacy. An alternative would be to reorder all entries every $n$ epochs to obtain better efficiency.

**Key Expiration.** To reduce the time frame during which a compromised key can be used by an attacker, users may want to enforce key expiration. This would entail including the epoch in which the public key is to expire as part of the directory entry, and clients would need to ensure that such keys are not expired when checking the consistency of bindings. Furthermore, CONIKS could allow users to choose whether to enforce key expiration on their binding, and provide multiple security options allowing users to set shorter or longer expiration periods. When the key expires, clients can automatically change the expired key and specify the new expiration date according to the user's policies.

**Support for Multiple Devices.** Any modern communication system must support users communicating from multiple devices. CONIKS easily allows users to bind multiple keys to their username. Unfortunately, device pairing has proved cumbersome and error-prone for users in practice [33, 68]. As a result, most widely-deployed chat applications allow users to simply install software to a new device which will automatically create a new key and add it to the directory via password authentication.

The tradeoffs for supporting multiple devices are the same as for key change. Following this easy enrollment procedure requires that Alice enforce the cautious key change policy, and her client will no longer be able to automatically determine if a newly observed key has been maliciously inserted by the server or represents the addition of a new device. Users can deal with this issue by requiring that any new device key is authenticated with a previously-registered key for a different device. This means that clients can automatically detect if new bind-

ings are inconsistent, but will require users to execute a manual pairing procedure to sign the new keys as part of the paranoid key change policy discussed above.

## 7 Related Work

**Certificate validation systems.** Several proposals for validating SSL/TLS certificates seek to detect fraudulent certificates via transparency logs [4, 35, 39, 40, 54], or observatories from different points in the network [4, 35, 55, 59, 69]. Certificate Transparency (CT) [40] publicly logs all certificates as they are issued in a signed append-only log. This log is implemented as a chronologically-ordered Merkle binary search tree. Auditors check that each signed tree head represents an extension of the previous version of the log and gossip to ensure that the log server is not equivocating.

This design only maintains a set of issued certificates, so domain administrators must scan the entire list of issued certificates (or use a third-party monitor) in order to detect any newly-logged, suspicious certificates issued for their domain. We consider this a major limitation for user communication as independent, trustworthy monitors may not exist for small identity providers. CT is also not privacy-preserving; indeed it was designed with the opposite goal of making all certificates publicly visible.

Enhanced Certificate Transparency (ECT) [61], which was developed concurrently [47] extends the basic CT design to support efficient queries of the current set of valid certificates for a domain, enabling built-in revocation. Since ECT adds a second Merkle tree of currently valid certificates organized as a binary search tree sorted lexicographically by domain name, third-party auditors must verify that no certificate appears in only one of the trees by mirroring the entire structure and verifying all insertions and deletions.

Because of this additional consistency check, auditing in ECT requires effort linear in the total number of changes to the logs, unlike in CT or CONIKS, which only require auditors to verify a small number of signed tree roots. ECT also does not provide privacy: the proposal suggests storing users in the lexicographic tree by a hash of their name, but this provides only weak privacy as most usernames are predictable and their hash can easily be determined by a dictionary attack.

Other proposals include public certificate observatories such as Perspectives [55, 59, 69], and more complex designs such as Sovereign Keys [54] and AKI/ARPKI [4, 35] which combine append-only logs with policy specifications to require multiple parties to sign key changes and revocations to provide proactive as well as reactive security.

All of these systems are designed for TLS certificates, which differ from CONIKS in a few important ways. First, TLS has many certificate authorities sharing a single, global namespace. It is not required that the different CAs offer only certificates that are consistent or non-overlapping. Second, there is no notion of certificate or name privacy in the TLS setting,[12] and as a result, they use data structures making the entire name-space public. Finally, stronger assumptions, such as maintaining a private key forever or designating multiple parties to authorize key changes, might be feasible for web administrators but are not practical for end users.

**Key pinning.** An alternative to auditable certificate systems are schemes which limit the set of certificate authorities capable of signing for a given name, such as certificate pinning [17] or TACK [45]. These approaches are brittle, with the possibility of losing access to a domain if an overly strict pinning policy is set. Deployment of pinning has been limited due to this fear and most web administrators have set very loose policies [36]. This difficulty of managing keys, experienced even by technically savvy administrators, highlights how important it is to require no key management by end users.

**Identity and key services.** As end users are accustomed to interacting with a multitude of identities at various online services, recent proposals for online identity verification have focused on providing a secure means for consolidating these identities, including encryption keys.

Keybase [38] allows users to consolidate their online account information while also providing semi-automated consistency checking of name-to-key bindings by verifying control of third-party accounts. This system's primary function is to provide an easy means to consolidate online identity information in a publicly auditable log. It is not designed for automated key verification and it does not integrate seamlessly into existing applications.

Nicknym [57] is designed to be purely an end-user key verification service, which allows users to register existing third-party usernames with public keys. These bindings are publicly auditable by allowing clients to query any Nicknym provider for *individual* bindings they observe. While equivocation about bindings can be detected in this manner in principle, Nicknym does not maintain an authenticated history of published bindings which would provide more robust consistency checking as in CONIKS.

**Cryptographically accountable authorities.** Identity-based encryption inherently requires a trusted private-key generator (PKG). Goyal [29] proposed the accountable-authority model, in which the PKG and a user cooperate

[12] Some organizations use "private CAs" which members manually install in their browsers. Certificate transparency specifically exempts these certificates and cannot detect if private CAs misbehave.

to generate the user's private key in such a way that the PKG does not know what private key the user has chosen. If the PKG ever runs this protocol with another party to generate a second private key, the existence of two private keys would be proof of misbehavior. This concept was later extended to the black-box accountable-authority model [30, 62], in which even issuing a black-box decoder algorithm is enough to prove misbehavior. These schemes have somewhat different security goals than CONIKS in that they require discovering two *private* keys to prove misbehavior (and provide no built-in mechanism for such discovery). By contrast, CONIKS is designed to provide a mechanism to discover if two distinct *public* keys have been issued for a single name.

**VRFs and dictionary attacks.** DNSSEC [16] provides a hierarchical mapping between domains and signing keys via an authenticated linked list. Because each domain references its immediate neighbors lexicographically in this design, it is possible for an adversary to enumerate the entire set of domains in a given zone via *zone walking* (repeatedly querying neighboring domains). In response, the NSEC3 extension [41] was added; while it prevents trivial enumeration, it suffers a similar vulnerability to ECT in that likely domain names can be found via a dictionary attack because records are sorted by the hash of their domain name. Concurrent with our work on CONIKS, [28] proposed NSEC5, effectively also using a verifiable random function (also in the form of a deterministic RSA signature) to prevent zone enumeration.

## 8 Conclusion

We have presented CONIKS, a key verification system for end users that provides consistency and privacy for users' name-to-key bindings, all without requiring explicit key management by users. CONIKS allows clients to efficiently monitor their own bindings and quickly detect equivocation with high probability. CONIKS is highly scalable and is backward compatible with existing secure communication protocols. We have built a prototype CONIKS system which is application-agnostic and supports millions of users on a single commodity server.

As of this writing, several major providers are implementing CONIKS-based key servers to bolster their end-to-end encrypted communications tools. While automatic, decentralized key management without least a semi-trusted key directory remains an open challenge, we believe CONIKS provides a reasonable baseline of security that any key directory should support to reduce user's exposure to mass surveillance.


## Acknowledgments

We thank Gary Belvin, Yan Zhu, Arpit Gupta, Josh Kroll, David Gil, Ian Miers, Henry Corrigan-Gibbs, Trevor Perrin, and the anonymous USENIX reviewers for their feedback. This research was supported by NSF Award TC-1111734. Joseph Bonneau is supported by a Secure Usability Fellowship from OTF and Simply Secure.


## References


[1] Pidgin. `http://pidgin.im`, Retr. Apr. 2014.

[2] Protocol Buffers. `https://code.google.com/p/protobuf`, Retr. Apr. 2014.

[3] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Special Publication 800-57 rev. 3. *NIST*, 2012.

[4] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski. ARPKI: attack resilient public-key infrastructure. *ACM CCS*, 2014.

[5] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2), 2012.

[6] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. *ACM CCS*, 2013.

[7] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *ASIACRYPT*, 2001.

[8] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. *WPES*, 2004.

[9] S. Braun, A. Flaherty, J. Gillum, and M. Apuzzo. Secret to Prism program: Even bigger data seizure. Associated Press, Jun. 2013.

[10] P. Bright. Another fraudulent certificate raises the same old questions about certificate authorities. Ars Technica, Aug. 2011.

[11] P. Bright. Independent Iranian hacker claims responsibility for Comodo hack. Ars Technica, Mar. 2011.

[12] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. RFC 2440 OpenPGP Message Format, Nov. 1998.

[13] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

[14] D. Chaum and T. P. Pedersen. Wallet databases with observers. *CRYPTO*, 1993.

[15] Y. Dodis and A. Yampolskiy. A verifiable random function with short proofs and keys. *Public Key Cryptography*, 2005.

[16] D. Eastlake. RFC 2535: Domain Name System Security Extensions. 1999.

[17] C. Evans, C. Palmer, and R. Sleevi. Internet-Draft: Public Key Pinning Extension for HTTP. 2012.

[18] P. Everton. Google's Gmail Hacked This Weekend? Tips To Beef Up Your Security. Huffington Post, Jul. 2013.

[19] E. Felten. A Court Order is an Insider Attack, Oct. 2013.

[20] T. Fox-Brewster. WhatsApp adds end-to-end encryption using TextSecure. The Guardian, Nov. 2014.

[21] M. Franklin and H. Zhang. Unique ring signatures: A practical construction. *Financial Cryptography*, 2013.

[22] P. Gallagher and C. Kerry. FIPS Pub 186-4: Digital signature standard, DSS. NIST, 2013.

[23] S. Gaw, E. W. Felten, and P. Fernandez-Kelly. Secrecy, flagging, and paranoia: Adoption criteria in encrypted email. *CHI*, 2006.

[24] B. Gellman. The FBI's Secret Scrutiny. The Wasington Post, Nov. 2005.

[25] B. Gellman and L. Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. The Washington Post, Jun. 2013.

[26] S. Gibbs. Gmail does scan all emails, new Google terms clarify. The Guardian, Apr. 2014.

[27] I. Goldberg, K. Hanna, and N. Borisov. pidgin-otr. `http://sourceforge.net/p/otr/pidgin-otr/ci/master/tree/`, Retr. Apr. 2014.

[28] S. Goldberg, M. Naor, D. Papadopoulos, L. Reyzin, S. Vasant, and A. Ziv. NSEC5: Provably Preventing DNSSEC Zone Enumeration. *NDSS*, 2015.

[29] V. Goyal. Reducing trust in the pkg in identity based cryptosystems. *CRYPTO*, 2007.

[30] V. Goyal, S. Lu, A. Sahai, and B. Waters. Black-box accountable authority identity-based encryption. *ACM CCS*, 2008.

[31] T. Icart. How to hash into elliptic curves. *CRYPTO*, 2009.

[32] J. Jonsson and B. Kaliski. RFC 3447 Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, Feb. 2003.

[33] R. Kainda, I. Flechais, and A. W. Roscoe. Usability and Security of Out-of-band Channels in Secure Device Pairing Protocols. *SOUPS*, 2009.

[34] J. Katz. Analysis of a proposed hash-based signature standard. `https://www.cs.umd.edu/~jkatz/papers/HashBasedSigs.pdf`, 2014.

[35] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. *WWW*, 2013.

[36] M. Kranch and J. Bonneau. Upgrading HTTPS in midair: HSTS and key pinning in practice. *NDSS*, 2015.

[37] D. W. Kravitz. Digital signature algorithm, 1993. US Patent 5,231,668.

[38] M. Krohn and C. Coyne. Keybase. `https://keybase.io`, Retr. Feb. 2014.

[39] B. Laurie and E. Kasper. Revocation Transparency. `http://sump2.links.org/files/RevocationTransparency.pdf`, Retr. Feb. 2014.

[40] B. Laurie, A. Langley, E. Kasper, and G. Inc. RFC 6962 Certificate Transparency, Jun. 2013.

[41] B. Laurie, G. Sisson, R. Arends, and D. Black. RFC 5155: DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. 2008.

[42] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). *OSDI*, 2004.

[43] G. Lindberg. RFC 2505 Anti-Spam Recommendations for SMTP MTAs, Feb. 1999.

[44] M. Madden. Public Perceptions of Privacy and Security in the Post-Snowden Era. *Pew Research Internet Project*, Nov. 2014.

[45] M. Marlinspike and T. Perrin. Internet-Draft: Trust Assertions for Certificate Keys. 2012.

[46] J. Mayer. Surveillance law. Available at `https://class.coursera.org/surveillance-001`.

[47] M. S. Melara. CONIKS: Preserving Secure Communication with Untrusted Identity Providers. Master's thesis, Princeton University, Jun 2014.

[48] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. *FOCS*, 1999.

[49] N. Perloth. Yahoo Breach Extends Beyond Yahoo to Gmail, Hotmail, AOL Users. New York Times Bits Blog, Jul. 2012.

[50] T. Pornin. RFC 6979: Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). 2013.

[51] Electronic Frontier Foundation. Secure Messaging Scorecard. `https://www.eff.org/secure-messaging-scorecard`, Retr. 2014.

[52] Electronic Frontier Foundation. National Security Letters - EFF Surveillance Self-Defense Project. `https://ssd.eff.org/foreign/nsl`, Retr. Aug. 2013.

[53] Electronic Frontier Foundation. National Security Letters. `https://www.eff.org/issues/national-security-letters`, Retr. Nov. 2013.

[54] Electronic Frontier Foundation. Sovereign Keys. `https://www.eff.org/sovereign-keys`, Retr. Nov. 2013.

[55] Electronic Frontier Foundation. SSL Observatory. `https://www.eff.org/observatory`, Retr. Nov. 2013.

[56] Internet Mail Consortium. S/MIME and OpenPGP. `http://www.imc.org/smime-pgpmime.html`, Retr. Aug. 2013.

[57] LEAP Encryption Access Project. Nicknym. `https://leap.se/en/docs/design/nicknym`, Retr. Feb. 2015.

[58] Reuters. At Sina Weibo's Censorship Hub, 'Little Brothers' Cleanse Online Chatter, Nov. 2013.

[59] Thoughtcrime Labs Production. Convergence. `http://convergence.io`, Retr. Aug. 2013.

[60] 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.

[61] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted email. *NDSS*, Feb. 2014.

[62] A. Sahai and H. Seyalioglu. Fully secure accountable-authority identity-based encryption. In *Public Key Cryptography–PKC 2011*, pages 296–316. Springer, 2011.

[63] B. Schneier. Apple's iMessage Encryption Seems to Be Pretty Good. `https://www.schneier.com/blog/archives/2013/04/apples_imessage.html`, Retr. Feb. 2015.

[64] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3), 1991.

[65] C. Soghoian and S. Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks against SSL. *Financial Crypto'*, 2012.

[66] R. Stedman, K. Yoshida, and I. Goldberg. A User Study of Off-the-Record Messaging. *SOUPS*, Jul. 2008.

[67] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. SoK: Secure Messaging. *IEEE Symposium on Security and Privacy*, 2015.

[68] B. Warner. Pairing Problems, 2014.

[69] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: improving SSH-style host authentication with multi-path probing. In *Usenix ATC*, Jun. 2008.

[70] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. *USENIX Security*, 1999.

[71] P. R. Zimmermann. *The official PGP user's guide*. MIT Press, Cambridge, MA, USA, 1995.

# A Discrete-log Based VRF Construction

We propose a simple discrete-log based VRF in the random oracle model. This construction was described by Franklin and Zhang [21] in 2013[13] although they considered it already well-known. Following Micali et al.'s outline [48], the basic idea is to publish a commitment $c$ to the seed $k$ of a pseudo-random function, compute $y = f_k(x)$ as the VRF, and issue non-interactive zero-knowledge proofs that $y = f_k(x)$ for some $k$ to which $c$ is a commitment of. The public key and private key are $c$ and $k$.

**Parameters.** For a group[14] $\mathcal{G}$ with generator $g$ of prime order $q$, the prover chooses a random $k \xleftarrow{R} (1,q)$ as their private key and publishes $G = g^k$ as their public key. We require two hash functions which are modeled as random oracles: one which maps to curve points [6, 31] $\mathbf{H}_1 : * \to \mathcal{G}$ and one which maps to integers $\mathbf{H}_2 : * \to (1,q)$.

**VRF computation.** The VRF is defined as:

$$\mathbf{VRF}_k(m) = \mathbf{H}_1(m)^k \tag{1}$$

**Non-interactive proof** For a given output $v = \mathbf{VRF}_k(m)$, the prover must show in zero-knowledge that there is some $k$ for which $G = g^k$ and $H = h^k$ for $h = \mathbf{H}_1(m)$. The proof is a standard Sigma proof of equality for two discrete logarithms made non-interactive using the Fiat-Shamir heuristic [14]. The prover chooses $r \xleftarrow{R} (1,q)$ and transmits the values $(v,s,t)$ where

$$s = \mathbf{H}_2(g,h,G,v,g^r,h^r) \tag{2}$$

$$t = r - sk \pmod q \tag{3}$$

To verify that $v = \mathbf{VRF}_k(m) = \mathbf{H}_1(m)^k$ is a correct VRF computation given a proof $(v,s,t)$, the verifier checks that

$$s = \mathbf{H}_2\left(g,h,G,v,g^t \cdot G^s, \mathbf{H}_1(m)^t \cdot v^s\right) \tag{4}$$

We refer the reader to [14, 21] for proof that this scheme satisfies the properties of a VRF. Note that the pseudorandomness reduces to the Decisional Diffie-Hellman assumption. The tuple $(\mathbf{H}_1(m), G = g^k, v = \mathbf{H}_1(m)^k)$ is a DDH triple, therefore an attacker that could distinguish the VRF output $v$ from random could break the DDH assumption for $\mathcal{G}$.

[13] The scheme in Franklin and Zhang's original paper contained an error, reproduced in earlier versions of this paper. It has sinced been fixed in Franklin and Zhang's paper as well as this paper. Thanks to Sharon Goldberg and Leonid Reyzin for finding this error.

[14] Note that we use multiplicative group notation here, though this scheme applies equally to elliptic-curve groups.

**Efficiency.** Proofs consist two integers which are the size of the order of the group ($(s,t)$). For 256-bit elliptic curve, this leads to proofs of size 512 bits (64 bytes) plus the VRF result itself which is an additional 32 bytes.

# B Analysis of Equivocation Detection

CONIKS participants check for non-equivocation by consulting auditors to ensure that they both see an identical STR for a given provider $P$. Clients perform this cross-verification by choosing uniformly at random a small set of auditors from the set of known auditors, querying them for the observed STRs from $P$, and comparing these observed STRs to the signed tree root presented directly to the client by $P$. If any of the observed STRs differ from the STR presented to the client, the client is sure to have detected an equivocation attack.

## B.1 Single Equivocating Provider

Suppose that *foo.com* wants to allow impersonation of a user Alice to hijack all encrypted messages that a user Bob sends her. To mount this attack, *foo.com* equivocates by showing Alice STR A, which is consistent with Alice's *valid* name-to-key binding, and showing Bob STR B, which is consistent with a fraudulent binding for Alice.

If Bob is the only participant in the system to whom *foo.com* presents STR B, while all other users and auditors receive STR A, Alice will not detect the equivocation (unless she compares her STR directly with Bob's). Bob, on the other hand, will detect the equivocation immediately because performing the non-equivocation check with a single randomly chosen auditor is sufficient for him to discover a diverging STR for *foo.com*.

A more effective approach for *foo.com* is to choose a subset of auditors who will be presented STR A, and to present the remaining auditors with STR B. Suppose the first subset contains a fraction $f$ of all auditors, and the second subset contains the fraction $1 - f$. If Alice and Bob each contact $k$ randomly chosen providers to check consistency of *foo.com*'s STR, the probability that Alice fails to discover an inconsistency is $f^k$, and the probability that Bob fails to discover an inconsistency is $(1-f)^k$. The probability that both will fail is $(f - f^2)^k$, which is maximized with $f = \frac{1}{2}$. Alice and Bob therefore *fail* to discover equivocation with probability

$$\varepsilon \leq \left(\frac{1}{4}\right)^k$$

In order to discover the equivocation with probability $1 - \varepsilon$, Alice and Bob must perform $-\frac{1}{2}\log\frac{\varepsilon}{2}$ checks. After performing 5 checks each, Alice and Bob would have discovered an equivocation with 99.9% probability.

## B.2 Colluding Auditors

Now suppose that *foo.com* colludes with auditors in an attempt to better hide its equivocation about Alice's binding. The colluding auditors agree to tell Alice that *foo.com* is distributing STR A while telling Bob that *foo.com* is distributing STR B. As

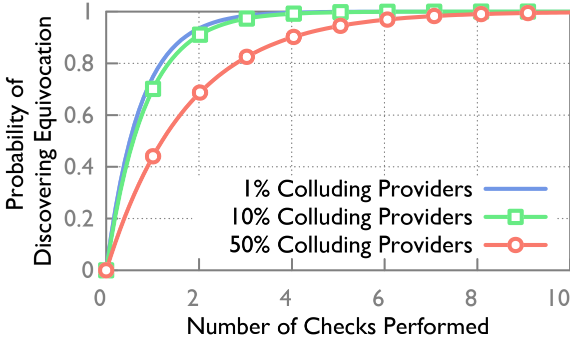


**Figure 8: This graph shows the probability that Alice and Bob will detect an equivocation after each performing $k$ checks with randomly chosen auditors.**

the size of the collusion increases, Alice and Bob become less likely to detect the equivocation. However, as the number of auditors in the system (and therefore, the number of auditors not participating in the collusion) increases, the difficulty of detecting the attack decreases.

More precisely, we assume that *foo.com* is colluding with a proportion $p$ of all auditors. The colluding auditors behave as described above, and *foo.com* presents STR A to a fraction $f$ of the non-colluding providers. Alice and Bob each contacts $k$ randomly chosen providers. The probability of Alice failing to detect equivocation within $k$ checks is therefore $(p+(1-p)f)^k$ and the probability of Bob failing to detect equivocation within $k$ checks is $(p+(1-p)(1-f))^k$. The probability that neither Alice nor Bob detects equivocation is then

$$\varepsilon = ((p+(1-p)f)(p+(1-p)(1-f)))^k$$

As before, this is maximized when $f = \frac{1}{2}$, so the probability that Alice and Bob *fail* to detect the equivocation is

$$\varepsilon \leq \left(\frac{1+p}{2}\right)^{2k}$$

If $p = 0.1$, then by doing 5 checks each, Alice and Bob will discover equivocation with 99.7% probability.

Figure 8 plots the probability of discovery as $p$ and $k$ vary. If fewer than 50% of auditors are colluding, Alice and Bob will detect an equivocation within 5 checks with over 94% probability. In practice, large-scale collusion is unexpected, as today's secure messaging services have many providers operating with different business models and under many different legal and regulatory regimes. In any case, if Alice and Bob can agree on a single auditor whom they both trust to be honest, then they can detect equivocation with certainty if they both check with that trusted auditor.