Bringing Deployable Key Transparency to End Users

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

Princeton University

Abstract

We present CONIKS, an end-user key verification service capable of integration in end-to-end encrypted communication systems. CONIKS builds on related designs for transparency of web server certificates but solves several new challenges specific to key verification for end users. In comparison to prior designs, CONIKS enables more efficient monitoring and auditing of keys, allowing small organizations to effectively audit even very large key servers. CONIKS users can 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 day regardless of server size. Unlike any previous proposal, CONIKS also 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. Given both the well-documented insecurity of certificate authorities protecting TLS certificates [9, 10, 55] and the aggressive level of government surveillance of communication providers revealed by recent leaks [8, 22, 25], this status quo is unacceptable.

Spurred by these security threats and users' desire for stronger security [39], several large services including Apple iMessage and WhatsApp have recently deployed end-to-end encryption [18, 54]. 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 [15, 30, 44], 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 [11, 49, 62] suggest that manual key verification is error-prone and irritating [20, 61]. 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 [17]. Similar conclusions were reached by a recent academic survey on key verification mechanisms [58].

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 the *foo.com* server, binding the name *alice@foo.com* to one or more public keys, any party can verify that this is the same (and only) binding for *alice@foo.com* that any other user 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 would 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* [5, 29, 34, 35, 47, 53]. 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 be able to promptly detect any certificate issued maliciously. In Certificate Transparency [35], one of the earlier proposals and the closest to practical deployment today, this data structure is simply an append-only log (implemented as a Merkle tree) of all certificates in chronological order.

Follow-up proposals have incorporated more advanced features such as revocation [5, 29, 34, 53] and finer-grained limitations on certificate issuance [5, 29], 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 all 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.

Our primary contributions solve these two problems:

1. Efficient self-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, receiving alerts whenever a new certificate was issued for one of their domains. 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 computationally-intensive monitoring. A user's device can automatically monitor the user's key directory entry and alert the user if unexpected keys are ever bound to their username.

2. Privacy-preserving key directories. Since prior systems to date [29, 35, 47, 53] require third-party monitors to view the entire system log, monitoring leaks the set of users who have been issued keys. CONIKS, on the contrary, is privacy-preserving. CONIKS servers can choose when to respond to queries 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 data is mapped to their username. As a result, CONIKS can be used to provide authenticated bindings for sensitive application-specific data (e.g., contact lists, profile photos) in addition to public keys. 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 [7, 23, 56] 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 very efficient in terms of bandwidth, computation, and storage overheads for clients and servers. Clients need to download about 19.1 kB per day from the CONIKS server, and verifying key bindings is trivially fast. Our prototype server implementation is able to easily support 10 million users (with 1% changing keys per day) on a commodity machine with 64 GB of RAM. Feedback from industrial developers at major communication service companies has confirmed that CONIKS is a viable solution for a key verification system for their users.

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 public keys for each identity provider to sign its bindings.

[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.

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 openly willing to attack its users (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* their identity provider for consistency of their own bindings, unlike previous systems in which this functionality is provided by third parties. To support consistency 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 misbehavior by an identity provider, the threat of which we assume will prevent the identity provider from certain attacks including equivocation. CONIKS cannot ensure security if clients have no means of communication that the communication provider does not control.[2]

Auditors. To verify that identity providers are not equivocating, *auditors* track the chain of signed "snapshots" of the Merkle prefix tree that they publish and gossip with other auditors to ensure global consistency. Being a CONIKS auditor is very lightweight, with less than 20 kB per day of data to fetch and a trivial number of cryptographic calculations. 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 *cautious users* and *paranoid users* to illustrate the differing security properties and usability challenges of the system.

An essential difference is that cautious users permit unauthenticated (but publicly visible) key changes by the server in case of lost keys, while paranoid users commit to maintaining a private key forever and will lose access to their username if they lose private key material. We will discuss further security tradeoffs, such as whether usernames are publicly visible in §4.4.

[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.

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 (called STRs), 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. This built-in auditing provides high assurance that equivocation will be detected with high probability (see Appendix A for a detailed analysis). Because any equivocation may produce non-repudiable evidence, we assume that identity providers will avoid this attack strategy for fear of damaging their reputation.

G2: Key binding consistency. 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 cautious 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 paranoid users, all key changes must be signed by the user's previous key and therefore malicious bindings can be proved publicly.

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. To our knowledge, CONIKS is the first system which enables this level of privacy while still allowing auditability for consistency.

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 they manage.

	CT [35]	ECT [53]	CONIKS
Auditor cost	$O(\lg N)$	$O(n)$	$O(1)$
Monitor cost	$O(n)$	$O(\lg N)$	$O(\lg N)$
Privacy	—	—	✓

Table 1: Comparison of CONIKS with similar proposals in terms of efficiency and privacy. N is the total number of log entries and n is the total number of updates to the log.

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 and 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: Efficient data structure for key directories. Computational and communication overhead should be minimized so that CONIKS is feasible to implement for identity providers using commodity servers and for clients on common mobile devices. All overhead should scale at most logarithmically in the number of total users.

2.3 Related proposals

We will briefly discuss two related proposals: Certificate Transparency [35] and Enhanced Certificate Transparency [53], which was developed concurrently [42]. A summary is provided in Table 1. CT introduced the terminology of *auditors* to check for non-equivocation and *monitors* to check for consistency/validity of data, which we adapt as well, although we emphasize that users' own clients perform both auditing and monitoring in CONIKS.

Certificate Transparency (CT) [35] is designed to mitigate the problem of mis-issued TLS certificates by publicly logging all certificates as they are issued in a signed append-only log which is implemented as a chronologically-ordered Merkle binary search tree. Auditors check that each signed tree root[3] represents an extension of the previous version of the log and gossip to ensure that the log server is not equivocating. A domain administrator can monitor the set of newly logged-certificates (or use a third-party monitor) to detect suspicious certificates issued for their domain.

This design only maintains a set of issued certificates and not a mapping from domains to certificates and does not allow for efficient queries of the form "what is the most recent certificate issued for domain X." Because of this, in practice it requires monitors to scan the entire list of issued certificates on behalf of small domain owners, so acting as a CT monitor requires an amount of work linear in the total number of issued certificates. We consider this a major limitation for user communication because independent and 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 [53] extends the basic CT design to support efficient queries of the current set of valid certificates for a domain, enabling built-in revocation. ECT servers maintain an append-only log of certificates in chronological order, as in CT, and 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 the two trees are maintained consistently with each other, i.e., that no certificate appears in only one of the trees, by mirroring the entire structure and verifying all insertions and deletions. Auditors must also gossip to detect equivocation.

Assuming third-party auditors perform these checks correctly, ECT enables users to efficiently query the current state of the system for all certificates considered valid for their domain or username. However, unlike in CT or CONIKS, auditing in ECT is requires effort linear in the total number of changes to the logs, which we do not consider practical for end-user communication.

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.

3 Core Data Structure Design

At a high level, CONIKS identity providers manage a directory of verifiable bindings of usernames to data. In a key verification service, a username is mapped to a user's public key. This directory is constructed as a Merkle prefix tree of all registered bindings in the provider's namespace.

In order to make the directory privacy-preserving, a *private index* is computed for each username via a verifiable unpredictable function. Each user's data is stored at the associated private index rather than his or her username (or a hash of it). This prevents the data structure from leaking data about usernames. To ensure that users' data can be verified without revealing any information about other users' data, a cryptographic *commitment*[4] to each user's data is stored at the private index, rather than the data itself.

[3] CT uses the term *signed tree head* roughly equivalently to our signed tree *root*.

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

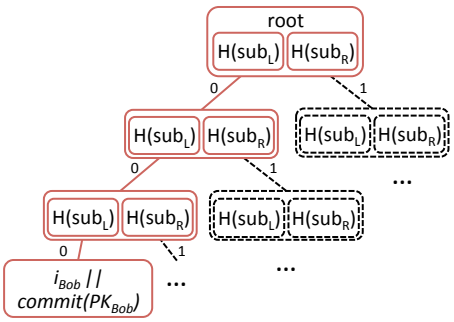

Figure 1: An authentication path from Bob's data to the root node of the Merkle prefix tree. Bob's index, i_{Bob}, has the prefix "000". Dotted nodes are not included in the proof's authentication path.

At regular time intervals, or epochs, the identity provider publicly commits to a "snapshot" of the directory by digitally signing the root of the Merkle tree. We call this signature a *signed tree root* (STR). Each STR includes the hash of the previous version to associate it with past versions of the directory.

3.1 Merkle Prefix Tree

CONIKS directories are constructed as a Merkle binary prefix tree. In a binary prefix tree, each branch of the tree represents the next bit in the binary representation of a user's index. Interior nodes have a left and right subtree. The left subtree contains all indices whose next bit is 0, and the right subtree contains all indices whose next bit is 1. The hash of an interior node is computed as:

$$\text{H}(\text{“interior”}||\text{H}(\text{substree}_{\text{left}})||\text{H}(\text{substree}_{\text{right}}))$$

where H() is a one-way hash function. As a convenience, empty subtrees can be represented with a null string of all zeroes.

A leaf node includes the complete lookup index i and a cryptographic commitment to the data mapped to that index:

$$\text{H}(\text{“leaf”}||i||\text{commit}(\text{data}_i))$$

Data Binding Proofs. To prove that a particular index exists in the tree, a data binding proof 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 data mapped to an index. To prove the full index-to-data binding, the server provides an opening of the commitment in addition to the authentication path.

Proofs of Absence. To prove that a given index i has no data mapped to it, an authentication path is provided to the longest prefix match of i currently in the directory. That node will be a leaf node with a different private index, or it will have an empty subtree where i's data would reside.

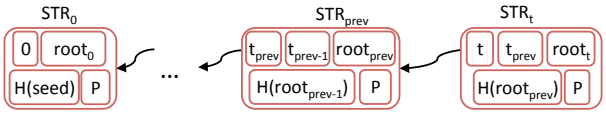

Figure 2: The directory's history is published as a linear hash chain of signed tree roots.

3.2 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

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

where t is the epoch number and P is a summary of this provider's current security policies. P may include, for example, a key K_i 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 root, the STRs form a *hash chain* of STRs, as shown in Figure 2. This hash chain is used to ensure that providers maintain a linear history of STRs. 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. Otherwise, clients will immediately detect the equivocation when presented with an STR belonging to a different branch of the hash chain [37].

3.3 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. Without further consideration, or by using a simple hash function of the username as the index (as is proposed in Extended Certificate Transparency), a user's authentication path would reveal some information about whether there are other users with prefixes "close" to that user.

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

An active attacker, seeking to test if a user *bob@foo.com* has registered a key, could instead hash many strings until they find one whose index shares a prefix of length ℓ with the target user and then register this string as their name. As ℓ increases, the existence of another user sharing this prefix provides increasingly strong evidence that *bob@foo.com* has registered.

Private Indices. To prevent such leakage, we compute private indices using a *verifiable unpredictable function*, which is a function that requires a private key to compute but can then be publicly verified. VUFs are a simpler form of a stronger cryptographic construction called verifiable random functions [43]. In our application, we only need to ensure that a user's location in the tree is not predictable and do not need strong randomness (although randomness is helpful to produce a balanced tree).

We can implement a VUF using any deterministic, existentially unforgeable signature scheme [43]. The signature scheme must be deterministic or else (in our application) the identity provider could insert multiple bindings for a user at different locations each with a valid authentication path. In practice, this could be a classic RSA signature [52] (using a deterministic padding scheme such as PKCS v. 1.5 [27]) or BLS "short signatures" [6], which provide the best space efficiency. Discrete-log based signature schemes such as DSA [32] are not immediately applicable as they require a random nonce.

We generate the index for a user u as:

$$i = \mathrm{H}(\mathrm{VUF}_{K_i}(u))$$

where VUF() is a deterministic, existentially unforgeable signature scheme. The key K_i is specified in each STR. A hash function is required because some bits of $\mathrm{VUF}_{K_i}(u)$ might otherwise leak through data binding proofs, and signature schemes are generally not unforgeable given some bits of the valid signature. A full index-to-data binding proof for user u therefore requires the authentication path, the value of $\mathrm{VUF}(u)$ and the user u's data.

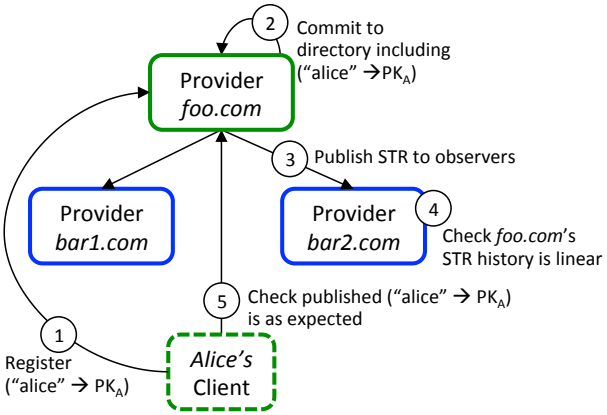

Figure 3: Steps required for a user Alice to register a key PK_A with her provider *foo.com*. After registration, *foo.com* publishes a new STR to any auditors, and Alice's client verifies that *foo.com* is publishing her expected binding.

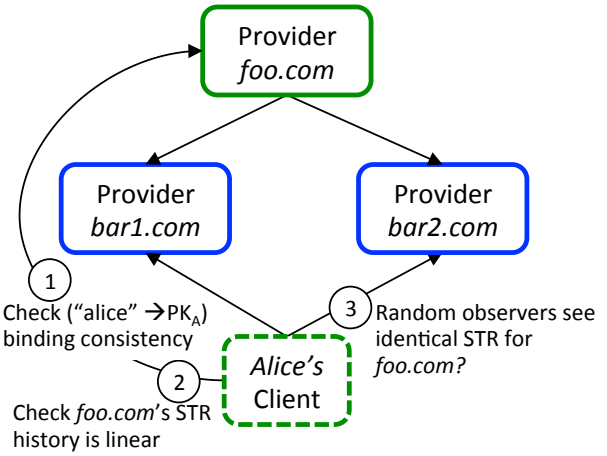

Figure 4: Steps required for self-monitoring in CONIKS. At the beginning of every epoch, Alice's client verifies the consistency of her binding and checks that *foo.com* is publishing a linear STR history. To check for non-equivocation, Alice's client queries randomly chosen auditors and compares the received STRs to the one *foo.com* presented her.

4 CONIKS Operation

CONIKS providers, clients and auditors collaborate in ensuring that identity providers maintain a single linear history of STRs. This enables clients to verify the consistency of bindings and check for non-equivocation. Fig. 3 and Fig. 4 summarize the basic protocols for registration and verification of bindings.

4.1 Auditing Provider History

CONIKS auditors maintain the current STRs of CONIKS providers. Note that because 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.

In typical deployment, we expect most CONIKS providers will act 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. Indeed, CONIKS clients are themselves auditors of their own provider since they will maintain their provider's current STR in the course of self-monitoring.

Ensuring a Linear STR History. The steps for an auditor (including client software) to verify a new STR are summarized in Figure 5. The auditor first ensures that the provider correctly signed the STR before checking whether the embedded previous root hash matches what the auditor saw previously. If they do not match, the provider has generated a fork in its STR history.

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 commitments published at a later time than previously indicated.

4.2 Verifying Bindings

Clients ensure the validity of users' bindings before communication. CONIKS depends on the fact that each client will monitor its own user's binding for consistency. Given that, for one client to contact another client, it need only ensure that both clients see the same STR. This is done by querying the user's provider for the data binding proof, verifying that the binding is included in the presented STR, and checking with auditors for equivocation.

Self-Monitoring Checks. When clients check a user's own bindings, the client begins by querying the user's provider for a data binding proof. Next, the client verifies the validity of the binding to ensure the binding represents the data the user believes is correct. In the simplest case, this is done by checking the consistency of a user's key between epochs. If the keys have not changed, or the client knows the changes were requested, then the user does not need to 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. Alternately, security-conscious users may request a paranoid policy for key changes which can be automatically enforced, and which we will discuss further in §4.4. After checking the validity of the binding, the client verifies the authentication path as described in § 3, including verifying the user's private index. Fig. 6 summarizes the steps taken during the self-monitoring checks.

Auditing Other Users' Bindings. CONIKS assumes that a binding's owner is responsible for self-monitoring, so clients need only audit another user's binding to ensure her binding is consistent with her provider's STR for a given epoch. Thus, if a binding is invalid, the binding's owner will detect it during self-monitoring.

The client begins by querying a user's provider for the data binding proof (or obtains it from their communication partner directly), and verifies the authentication path and private index. After verifying that the binding is consistent, no further checks need be performed for auditing other users' bindings.

Checking for Non-Equivocation. After checking the consistency of any user's binding (its own or another's), the client queries one or more CONIKS auditors at random.[5] The client asks the auditor for the signed tree root STR'_t it observed from the provider in question at epoch t. The client then compares STR'_t with the signed tree root STR_t 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. A.

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 verified the consistency of a their own or a contact's bindings for several epochs. They do so by performing a series of checks until they are sure that the data binding proofs they last verified are consistent with the more recent proofs.

Whistleblowing. If a client ever discovers two inconsistent STRs (for example, two distinct versions signed for the same epoch time), they will *whistleblow* by publishing them to all auditors they are able to contact. Any honest auditor noting that a provider has equivocated should permanently consider that provider untrustworthy.

4.3 Temporary bindings

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) but we consider this undesirable both on the server end and in terms of client overhead. Enhanced Certificate Transparency [53] appears to take this approach, relying on third-party monitors to check the frequent updates to the tree.

Certificate Transparency [35] issues "signed certificate timestamps" upon receipt of new certificates for logging before they are added to the main transparency log. These can be used as proof of logging, with failure to add a signed certificate to the log within a "maximum merge delay" being considered a sign of misbehavior.

[5] 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.

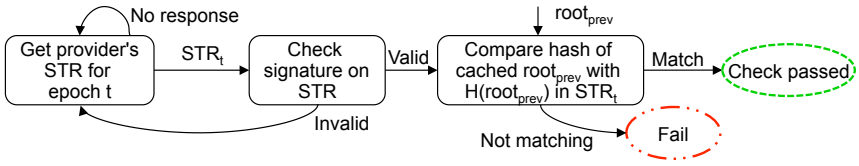

Figure 5: Steps taken when auditors check that a provider's STR history is linear. Auditors verify the hash chain formed by the STRs.

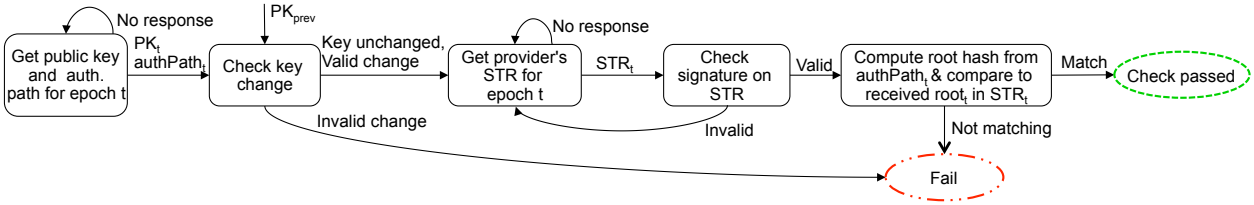

Figure 6: Steps taken during the self-monitoring checks a client performs every epoch. Clients check whether the public key has changed unexpectedly, and verify the authentication path in the Merkle prefix tree.

With CONIKS, we take a similar approach to CT. CONIKS servers may issue *temporary bindings* without writing any data to the Merkle prefix tree. A temporary binding consists of:

$$\text{TB} = \text{Sign}_{K_d}(STR_t, i, \text{commit}(k))$$

The binding includes the most recent signed tree root STR_t, the index i for the user's binding, and a commitment to 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.

Accepting and Verifying a Temporary Binding. If Bob's client accepts a temporary binding for a user Alice, Bob has the obligation to check the next version of the tree to ensure that the data is added as promised. If it has not been added to a version STR_{t+1} which references STR_t as its predecessor, Bob's client should whistleblow with the temporary binding TB and signed commitment STR_{t+1}, along with an authentication path for Alice's data. Client software may want to indicate that a communication has not been fully confirmed when a temporary binding is used. However, we expect that temporary bindings are not a particularly appealing attack vector for a malicious identity provider, as they provide non-repudiable evidence of misbehavior.

4.4 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 cautious policy and a paranoid 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.

Visibility of Public Keys. Our goal is to enable the same level of privacy SMTP servers employ today, [6] 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 cautious and the paranoid 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 cautious 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 paranoid 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 §4.2, but if she enforces the paranoid policy, only those users with the symmetric key learn her public keys. The main advantage of the cautious 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 paranoid lookup policy provides stronger privacy, but it requires additional action to distribute the symmetric key which protects her public keys.

Key Change. Dealing with key loss is a difficult quandary for any security system. Automatic key recovery is an

[6] 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) [38].

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 cautious key change policy, her identity provider *foo.com* accepts any key change statement which is signed with the key Alice's client is changing, as well as unsigned key change requests. Thus, *foo.com* should change the public key bound to *alice@foo.com* only upon her request. If Alice wishes to permanently deactivate her binding, she can send a key change request including a special null-marker telling the server to place a *tombstone* on the directory entry. In both cases, the server should reflect the update to Alice's binding by including a key change statement in her directory entry. The paranoid key change policy *requires* that the Alice's client sign all of her key change statements or a tombstone with the key that is being changed. Thus, Alice's client only considers a new key or tombstone to be valid if the key change statement has been authenticated by one of her public keys.

While the cautious 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 a Alice to regain control of her username if her key is ever lost. We discuss some implications for key loss for paranoid 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 [7] (OTR) plug-in for the Pidgin instant messaging client [1, 23]. 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.

OTR. The OTR secure messaging protocol provides a suitable foundation for a first prototype of a CONIKS-based end-to-end secure communications application. OTR makes strong security guarantees and is designed to provide perfect forward secrecy and repudiability [56]. The vanilla OTR Pidgin plug-in already provides some key management functionalities which we enhance in our CONIKS Chat client to provide consistency:

Key generation: The plug-in automatically generates a DSA key pair and stores it locally on the client.

Key exchange: When a user begins a private conversation with another user, OTR exchanges both users' public keys to facilitate authentication and verification of message signatures. *We do not consider this mechanism to be secure since it is not coupled with key verification.*

Key verification: The plug-in implements the socialist millionaires protocol [26], allowing users to securely verify each others' identities remotely through a shared secret without having to manually compare public key fingerprints. *We consider this mechanism to be out-of-band since it requires explicit user intervention.*

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 binding consistency and proofs of absence, both of which implicitly prove the proper construction of the directory. Our server implementation additionally supports commitment 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 key exchange protocol in OTR unchanged, but replace the explicit 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 consistency 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 §4.2, and caches the

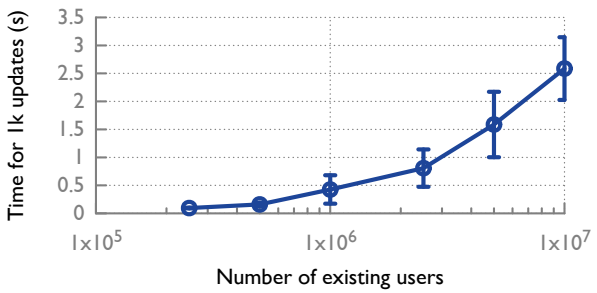

Figure 7: Mean time to re-compute the tree for a new epoch with 1K updated nodes. The x-axis is logarithmic and each data point is the mean of 10 executions. Error bars indicate standard deviation.

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 via the socialist millionaires protocol. Additionally, Alice's client and Bob's clients automatically perform all consistency 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 cautious lookup policy for name-to-key bindings, meaning that clients cannot exchange proofs through messages to enforce the paranoid lookup policy. Fully implementing these features is planned for the near future.

5.2 Performance Evaluation

To understand the expected performance of CONIKS in practice, we collect both theoretical and real performance characteristics of our prototype implementation. We evaluate 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.

We simulated this scenario to measure both client and server overheads. To provide a 128-bit security level, we use SHA-256 as our hash function and ECDSA signatures on the tree root using the P-256 curve [19]. We use RSA-2048 as our signature scheme for determining user's indices in the tree; while this key size is estimated to provide approximately 112 bits of security [4], this key only ensures privacy and not non-equivocation.

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 new epoch with 1000 new entries as the size of the original namespace varies. For a server with 10 M users, computing a new epoch with 1000 insertions takes on average 2.6 seconds. 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 quickly 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.

	# VUFs	# sigs.	# hashes	approx. size
data binding proof	1	1	$\lg N + 1$	1376 B
monitor (epoch)	0	1	$\lg n$	736 B
monitor (day)	1	k	$k \lg n$	19.1 kB
audit (epoch)	0	1	1	104 B
audit (day)	0	k	k	2.5 kB

Table 2: Signatures, VUFs and hashes needed for a full binding proof, daily and per-epoch monitoring, and daily and per-epoch auditing. Sizes are given assuming a $N \approx 2^{32}$ total users, $n \approx 2^{21}$ changes per epoch, and $k \approx 24$ epochs per day.

Auditing cost. For an auditor tracking all of a provider's STRs, assuming the policy field in changes rarely, the only new data in an STR is the new timestamp, the new tree root and one ECDSA signature (the previous root 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.

Monitoring cost. In order for any client to completely monitor the consistency of a binding, it needs to download the current STR, a proof consisting of about $\lg_2(N)+1$ hashes plus one 256-byte RSA signature (proving the validity of the binding's private index). This will require downloading $32 \cdot (\lg_2(N)+1)+256 \approx 1376$ bytes. Verifying the proof will require up to $\lg_2(N)+1$ hash verifications on the authentication path as well as one RSA 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 microseconds (sampled

over 1000 runs, with $\sigma = 30$). Verifying the signature takes approximately 400 μs, 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.

In addition to the computational overhead of verifying other user's key bindings, each client needs to fetch proof that its own binding is validly included in each epoch. 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. Assuming n changes are made to the tree, only $\lg_2(n)$ nodes will change in each epoch. 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 (104 bytes).

To check every STR 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.

Further bandwidth optimization. Bandwidth can be reduced further using BLS short signatures [6], which can replace both the RSA signatures used for index verification and the ECDSA signatures used for directory STRs. BLS signatures require only 32 bytes at a 128-bit security level. They also support *aggregation*, that is, the server can combine the signatures on n consecutive roots into a single signature for transmission. Together, these optimizations reduce the size of an full data binding proof from 1376 bytes to 1088 bytes, and can reduce the proofs required to verify the consistency of bindings from 19.1 kB to 16.1 kB per day (transmitting only one aggregated signature per day), a savings of about 19%. We use traditional RSA and ECDSA in our current implementation primarily because these schemes have more mature library support, though they also offer slightly faster signature generation and verification time.

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 the U.S. government uses mandatory legal process to demand access to information providers' data about American citizens' private communications and Internet activity [8, 21, 22, 25, 45, 46]. 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 [16]. 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[41].)

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

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 [12], 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 [28, 59]. 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 bindings 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 [5, 29, 34, 35, 47], or observatories from different points in the network [5, 29, 48, 51, 60]. Beyond CT and ECT, discussed in § 2.3, other proposals include public certificate observatories such as Perspectives [48, 51, 60], and more complex designs such

as Sovereign Keys [47] and AKI/ARPKI [5, 29] 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,[7] 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 [14] or TACK [40]. 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 [31]. These problems with properly managing keys even experienced by advanced users like web admins highlight 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 [33] 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. Openname (formerly known as OneName) [57] is a decentralized identity and naming system which stores mappings of usernames to online identity information (e.g., Bitcoin wallet address, PGP key, Twitter username) in the Namecoin [3] blockchain ensuring that names are globally unique. Before registering a new binding on the blockchain, users link their Openname username to accounts with third-party services and may also include public key information in their profile. Once in the blockchain, anyone can look up a username in the Openname directory and obtain the associated profile information. 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 communication applications.

Nicknym [50] 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.

DNSSEC zone enumeration. DNSSEC [13] provides an authenticated hierarchical mapping between domains and signing keys. This is implemented via an authenticated linked list, with each domain referencing its immediate neighbors lexicographically, and two neighbors being returned to provide a proof of non-existence for any record lexicographically in between them. This initial design was criticized for enabling 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 [36] was added which sorted records by hash of their domain name, preventing trivial enumeration. However, this suffers a similar vulnerability to ECT in that likely domain names can be found via a dictionary attack. Concurrent with our work on CONIKS, Goldberg et al. [24] proposed NSEC5, effectively using a verifiable unpredictable function (also in the form of a deterministic RSA signature) to prevent zone enumeration.

8 Conclusion

We have presented the design of CONIKS, a key verification service for end users that provides consistency and protects the privacy of users' name-to-key bindings, all without requiring explicit key management by the users. CONIKS's consistency checks allow clients to efficiently self-monitor their bindings and quickly detect equivocation with high probability. CONIKS is highly scalable incurring minimal overheads in terms of bandwidth, computation and storage, and is backward compatible with existing secure communication protocols such as OTR messaging. We have built a prototype CONIKS system which includes a prototype CONIKS server and a prototype chat application. The client application performs consistency checks in the background, and avoids any explicit user action except in the case of notifications that a key change has occurred. Our prototype CONIKS service is application-agnostic and supports millions of users on a single commodity server.

[7] 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.

References

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

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

[3] Namecoin. http://namecoin.info, Retrieved Feb. 2015.

[4] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Nist special publication 800-57 rev. 3. *NIST Special Publication*, 800(57), 2012.

[5] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski. Arpki: attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 382–393. ACM, 2014.

[6] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *Advances in Cryptology - ASIACRYPT 2001*. Springer, 2001.

[7] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use pgp. In *Proc. WPES*, Oct. 2004.

[8] S. Braun, A. Flaherty, J. Gillum, and M. Apuzzo. Secret to Prism program: Even bigger data seizure. http://bigstory.ap.org/article/secret-prism-success-even-bigger-data-seizure, Jun. 2013.

[9] P. Bright. Another fraudulent certificate raises the same old questions about certificate authorities. http://arstechnica.com/security/2011/08/earlier-this-year-an-iranian/, Aug. 2011.

[10] P. Bright. Independent Iranian hacker claims responsibility for Comodo hack. http://arstechnica.com/security/2011/03/independent-iranian-hacker-claims-responsibility-for-comodo-hack/, Mar. 2011.

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

[12] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, Feb. 1981.

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

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

[15] P. Everton. Google's Gmail Hacked This Weekend? Tips To Beef Up Your Security. http://www.huffingtonpost.com/paul-everton/googles-gmail-hacked-this_b_3641842.html, Jul. 2013.

[16] E. Felten. A Court Order is an Insider Attack. https://freedom-to-tinker.com/blog/felten/a-court-order-is-an-insider-attack/, Oct. 2013.

[17] E. F. Foundation. Secure messaging scorecard. https://www.eff.org/secure-messaging-scorecard, 2014.

[18] T. Fox-Brewster. WhatsApp adds end-to-end encryption using TextSecure. http://www.theguardian.com/technology/2014/nov/19/whatsapp-messaging-encryption-android-ios, Nov. 2014.

[19] P. Gallagher and C. Kerry. Fips pub 186-4: Digital signature standard, dss. NIST, 2013.

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

[21] B. Gellman. The FBI's Secret Scrutiny. http://washingtonpost.com/wp-dyn/content/article/2005/11/05/AR2005110505366.html, Nov. 2005.

[22] B. Gellman and L. Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. http://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html, Jun. 2013.

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

[24] S. Goldberg, M. Naor, D. Papadopoulos, L. Reyzin, S. Vasant, and A. Ziv. NSEC5: Provably Preventing DNSSEC Zone Enumeration. In *NDSS '15: The 2015 Network and Distributed System Security Symposium*, 2015.

[25] G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others. http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data, Jun. 2013.

[26] M. Jakobsson and M. Yung. Proving without knowing: On oblivious, agnostic and blindfolded provers. In *CRYPTO*. 1996.

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

[28] R. Kainda, I. Flechais, and A. W. Roscoe. Usability and Security of Out-of-band Channels in Secure Device Pairing Protocols. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, SOUPS, 2009.

[29] T. H.-J. Kim, L.-S. Huang, A. Perring, C. Jackson, and V. Gligor. Accountable key infrastructure (AKI):

a proposal for a public-key validation infrastructure. In *Proc. WWW*, 2013.

[30] J. Kirk. Researchers challenge Apple's claim of unbreakable iMessage encryption. http://www.macworld.com/article/2055640/researchers-challenge-apples-claim-of-unbreakable-imessage-encryption.html, Oct. 2013.

[31] M. Kranch and J. Bonneau. Upgrading HTTPS in midair: HSTS and key pinning in practice. In *NDSS '15: The 2015 Network and Distributed System Security Symposium*, February 2015.

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

[33] M. Krohn and C. Coyne. Keybase. https://keybase.io, Retrieved Feb. 2014.

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

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

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

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

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

[39] M. Madden. Public perceptions of privacy and security in the post-snowden era. *Pew Research Internet Project*, Nov. 2014.

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

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

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

[43] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

[44] N. Perloth. Yahoo Breach Extends Beyond Yahoo to Gmail, Hotmail, AOL Users. http://bits.blogs.nytimes.com/2012/07/12/yahoo-breach-extends-beyond-yahoo-to-gmail-hotmail-aol-users/, Jul. 2012.

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

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

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

[48] Electronic Frontier Foundation. SSL Observatory. https://www.eff.org/observatory, Retrieved Nov. 2013.

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

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

[51] Thoughtcrime Labs Production. Convergence. http://convergence.io, Retrieved Aug. 2013.

[52] 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.

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

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

[55] C. Soghoian and S. Stamm. Certified lies: Detecting and defeating government interception attacks against ssl (short paper). In *Financial Cryptography and Data Security*, pages 250–259. 2012.

[56] R. Stedman, K. Yoshida, and I. Goldberg. A user study of off-the-record messaging. In *Proc. SOUPS*, Jul. 2008.

[57] the openname system. Opename. https://openname.org, Retrieved Feb. 2015.

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

[59] B. Warner. Pairing Problems. https://blog.mozilla.org/warner/2014/04/02/pairing-problems/, 2014.

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

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

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

A 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.

A.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.

A.2 Colluding Auditors

Now suppose that *foo.com* colludes with auditors in an attempt to better hide its equivocation about Alice's binding.

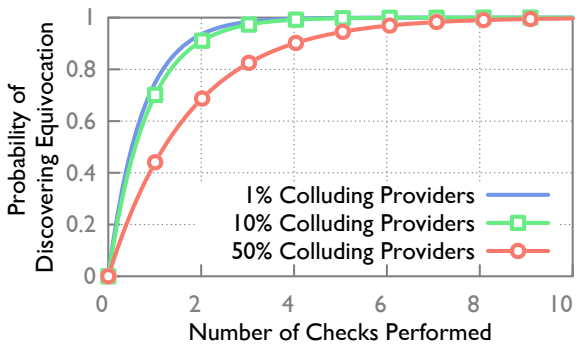

Figure 8: This graph shows the probability that Alice and Bob will detect an equivocation after each performing k checks with randomly chosen auditors.

The colluding auditors agree to tell Alice that *foo.com* is distributing STR A while telling Bob that *foo.com* is distributing STR B. As 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.