CONIKS: A Privacy-Preserving Consistent Key Service for Secure End-to-End Communication

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

ABSTRACT

Recent revelations about government surveillance have significantly increased the demand for end-to-end secure communications. However, key management remains a major barrier to adoption. Current systems are often either vulnerable to a malicious or coerced key directory or they make unrealistic assumptions about user behavior, for example, that users will verify key fingerprints out of band.

We present CONIKS, a system that provides automated key management for end users capable of seamless integration into existing secure messaging applications. In CONIKS, key servers maintain consistent directories of username-to-public key bindings that allow participants to detect any equivocation or unexpected key changes by malicious key servers. CONIKS also preserves user's privacy by ensuring that adversaries cannot harvest large numbers of usernames from the directories. Our prototype chat application extends the Off-the-Record Messaging plug-in for Pidgin. A single commodity server can support up to 10 million users and clients need only download less than 100 kB per day of additional data.

1. INTRODUCTION

Billions of users now depend on online services for sensitive communication. The vast majority of traffic is still transmitted unencrypted or, at best, "point-to-point" encrypted to a central server (e.g., over TLS) which relays it to the destination. Such a server still has access to plaintext in transit or storage. Recent revelations about mass data collection and surveillance [11, 22, 24] confirm that the insecure status quo is no longer acceptable. 80% of Americans now say they are concerned about government surveillance of electronic communication. Yet the majority of respondents in the same poll (68% and 57%, respectively) reported feeling "not very secure" or "not at all secure" when using online chat and email [38].

The solution requires practical *end-to-end* encryption for the masses. Encryption keys should be generated and stored on end-user devices, with content encrypted directly to the target user's key from the source. While keys may still be compromised on users' devices, this is (hopefully) a much more difficult and less scalable means of surveillance. Furthermore, past communication can be protected by forward secret protocols.

The dream of mass deployment of end-to-end encryption has inspired a flowering of new security-focused communication tools in the past year. However, most of these tools require users to manage keys and verify identities explicitly. Over two decades of experience with PGP email encryption [12, 49, 58] suggest that manual key verification is error-prone and irritating [20, 57]. The EFF's recent Secure Messaging Scorecard report considered over 40 popular secure messaging apps and assessed that none have a practical system for contact verification [18]. While there has been some research on streamlining manual key verification procedures [16], it is unreasonable to expect the majority of users to perform explicit actions for their communication security. Countering mass surveillance requires a system that provides at least some security for users who exert zero effort.

In contrast to new security-specific tools, some large existing services, including Apple iMessage and BlackBerry Protected Messenger, have successfully deployed end-to-end encryption for all users with no changes to the user experience. Similarly, WhatsApp is now integrating end-to-end encryption by default. These services put complete trust in a centralized directory of public keys maintained by the service provider. In light of the aggressive surveillance revealed by recent leaks [11, 22, 24], we cannot accept the risk of an intelligence agency tampering with these key directories through technical compromise, legal or extralegal pressure. Nor can we accept the risk of compromise by malicious insiders, organized crime, or other attackers [15, 44]. Further, these communication systems are largely walled gardens, so they need not worry about key discovery and management across service providers.

As a result, secure key management remains the main barrier to adoption of end-to-end secure messaging. To address this, we present CONIKS (the CONsistent Identity and Key Service), a new key management system.

Key directories with consistency. We retain the basic model of service providers issuing name-to-key bindings within their namespaces, but ensure that users can verify *consistency* of their bindings. We believe that assuring *correctness* 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 servers may issue an *authenticated binding* for the name *alice@foo.com*, and client software can automatically verify the issued bindings are consistent with those shown to all other users.

These bindings function somewhat like certificates in that users can present them to other users to set up a secure communication channel. However, we avoid the term "certificate" as CONIKS bindings go beyond simply verifying validity via a signature. To enable consistency checking, Alice's CONIKS server generates efficient periodic proofs of the set of all valid bindings issued for her username *alice@foo.com*, which Alice uses to verify that the server's bindings are consistent with her expectations. In other words, if a spurious key binding is created to impersonate Alice and intercept her communication, this will quickly become evident to Alice's software, which can alert her of the security threat.

Similar approaches have been proposed for improving the security of HTTPS (secure web browsing) by ensuring all PKIX certificates are included in a public log [28, 34, 35, 47, 52] or having users compare observed certificates for a given domain to what others observe from a different point in the network [4, 55]. While our

approach is similar, it achieves two additional properties that may not apply to TLS server certificates.

1. Efficient self-auditing. All previous schemes require a third-party auditor to track the entire log to check for consistency and detect new certificates/bindings. 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 auditing service free of charge or how users would choose such a service. CONIKS obviates this challenge by using an efficient data structure, a Merkle prefix tree, which enables users to receive a small proof (logarithmic in the total number of users) from their CONIKS server containing all valid key bindings which have been issued on their behalf. This allows users to *self-audit*, with their device automatically downloading consistency proofs periodically and alerting the user if new unexpected keys are ever bound to their username.

2. Privacy-preserving key directories. Prior systems to date [33, 52] required auditors to download the entire system log and learn information about every user. CONIKS, on the contrary, is *privacy-preserving*. Servers respond to queries for individual usernames (which can be rate-limited and/or authenticated) and the information returned to such queries is sufficient to verify the consistency of the prefix tree, without leaking any information about other users or even the existence of their names. In addition, communication services providing CONIKS may allow users to store sensitive application-specific data (e.g., contact lists, profile photos) as part of their bindings and ensure the privacy (and authenticity) of this data through CONIKS.

CONIKS in Practice. We have built a prototype CONIKS system, which includes both the application-agnostic CONIKS server and CONIKS Chat. Our chat application is integrated into the OTR plug-in [10, 23, 53] for Pidgin [1], providing XMPP-based instant messaging via an unmodified Jabber server. Our CONIKS client automatically performs self-audits 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 bindings has been issued.

In addition to the strong security and privacy features, CONIKS is backward compatible. It can seamlessly handle other deployment scenarios such as PGP email using legacy key management for users who are not enrolled with a CONIKS server. CONIKS is also very efficient in terms of bandwidth, computation, and storage overheads for clients and servers. Clients need to download about 96.3 kB per day from the CONIKS server, and verifying key bindings is trivially fast. Our prototype server implementation is able to support 10 million users (with 1% changing keys per day) on a commodity machine with 64 GB of RAM.

2. SECURITY GOALS AND ASSUMPTIONS

The goal of CONIKS is to provide a key management and verification system that facilitates practical, seamless, and secure communication for virtually all of today's users. We seek to provide good baseline security and the opportunity for higher security for advanced users within the same system.

2.1 Threat Model

CONIKS's security model includes two types of principals: identity providers and clients. Note that unlike many other proposals, CONIKS has no need for third-party auditors or monitors.

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 with.

CONIKS must prevent a malicious identity provider from violating the confidentiality or integrity of user communications. Identity providers might do this by disseminating a binding that maps a user's name to a key controlled by the identity provider.

We expect that CONIKS will deter identity providers from equivocating about their users' bindings since consistency checks will detect equivocation with high probability and provide non-repudiable evidence of misbehavior, threatening the identity provider's reputation (see §4.2). The BAR and covert adversary models consider how malicious behavior interacts with probabilistic detection [5, 6].

A malicious provider may refuse to respond to specific client queries to try to delay detection of misbehavior in an *availability* attack. We detail the consequences of this behavior in §4.3.

Clients. Users run CONIKS client software on one or more trusted devices. To support consistency checks, 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 timestamped information about prior checks.

We assume that any client which detects misbehavior by an identity provider can report it by publishing their cryptographic evidence. Therefore we assume that if a violation is detected, a majority of participants will learn about it promptly.

2.2 Desired Security Properties

CONIKS is designed to provide four security properties:

P1: Non-equivocation. If an identity provider equivocates by presenting diverging views of the name-to-key bindings in its namespace to different users, CONIKS is able to guarantee that clients will rapidly detect any inconsistency with high probability and obtain non-repudiable cryptographic evidence of this misbehavior.

P2: Binding consistency. If an identity provider maliciously inserts a new key binding for a user Alice, her client software will rapidly detect this and alert Alice that the binding has changed. CONIKS is designed to make auditing name-to-key bindings efficient enough that clients can perform consistency checks without deteriorating the user's experience.

P3: Privacy-preserving bindings. CONIKS servers do not need to make any information about their bindings public in order to allow consistency verification. Formally, an adversary who has obtained an arbitrary number of consistency proofs at a given time, even for adversarially chosen usernames, should not learn any information about which other users exist in the namespace. To our knowledge, CONIKS is the first system which enables this level of privacy while still allowing auditability for consistency.

P4: 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.

2.3 Usability Considerations

Providing these security properties in a practical system requires accepting several difficult real-world design constraints to enable mass deployment with no changes to the current user experience. In several areas, this requires offering weaker security properties by default but allowing security-conscious users to enable additional

[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.

security properties. We consider it important to enable such security options so that all users can use a common platform and the most security-conscious users do not abandon the system for one with stronger security guarantees. Users may choose the security policy they want to enforce, and CONIKS allows them to verify that the security policy remains consistent.

CONIKS needs to make tradeoffs between security and usability, taking users' expectations into consideration: Users may or may not want their usernames to be publicly queryable, or they may not be willing to lose the ability to regain control of their username if their private key is ever lost. Furthermore, most users today communicate via multiple devices, but many secure messaging applications do not require that these devices be synchronized to propagate the information about newly generated public keys on the other devices. As with key recovery, if Alice does not authenticate her additional public key bindings, how can the client software on her unpaired devices ensure that newly observed bindings for Alice are not a malicious insertion by the provider? We discuss how CONIKS's design deals with these tradeoffs in §3.3.

3. SYSTEM DESIGN

CONIKS identity providers manage a directory of verifiable name-to-public key bindings. This directory is implemented as a Merkle prefix tree of all registered bindings in the provider's namespace. In order to make the directory privacy-preserving, CONIKS uses a verifiable unpredictable function to insert usernames rather than a conventional hash function when constructing the Merkle prefix tree. At regular time intervals, or *epochs*, the identity provider includes all new binding registrations and updates received up to that point, and commits to the updated version of the directory by digitally signing the root of the Merkle tree. Additionally, the provider maintains a history of all commitments it has previously published.

CONIKS Usage. To allow clients and providers to check the consistency of bindings and commitments, CONIKS clients perform checks for each epoch. To demonstrate the normal operation of a client, we outline the series of steps required for a user Alice and her client to register with an identity provider and begin using CONIKS to communicate with Bob (Fig. 1):

1. Alice registers the name *alice* with her identity provider *foo.com*. Her client later verifies that *foo.com* is publishing the registered key for the name.
2. At the beginning of the next epoch t, *foo.com* generates the signed commitment for the new version of its directory.
3. Provider *foo.com* publishes this new commitment to all other providers in the system.
4. Upon receiving *foo.com*'s new commitment, provider *bar.com* checks that *foo.com*'s commitment history is consistent with what *bar.com* has observed so far.
5. Alice checks the consistency of her binding in three steps:
   (a) Her client requests *foo.com*'s newest commitment to check *foo.com*'s commitment history.
   (b) Her client randomly chooses one or more CONIKS providers to query (*bar.com* in this example) for the commitment it observed from *foo.com* at the beginning of the epoch. This check looks for evidence that *foo.com* has equivocated about its new commitment.
   (c) Her client looks up her binding from *foo.com* requesting a proof of consistency for her binding in the new version of *foo.com*'s directory to check that her binding has not changed unexpectedly.

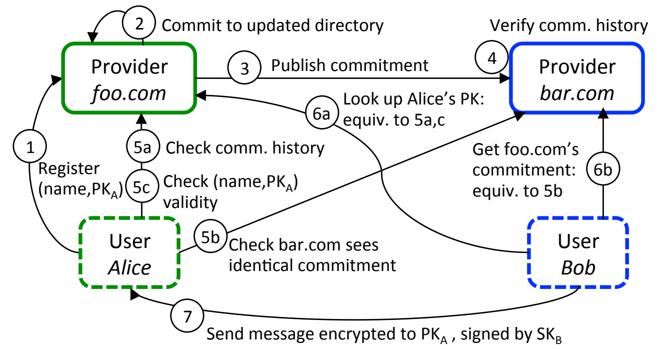

Figure 1: Interactions between users and identity providers when performing the consistency checks in CONIKS.

6. Now, Bob, as user of *bar.com*, wants to communicate with *alice@foo.com*. His client looks up Alice's public key at *foo.com* and performs the same checks as Alice in step 5.
7. If Bob's client detects no misbehavior, it encrypts his message with Alice's public key, and signs it with his private key. We assume Bob has also verified his own binding in this epoch per step 5.
8. Once Alice receives the message from *bob@bar.com*, her client looks up Bob's public key at *bar.com* and checks the binding's consistency. If the checks pass, Alice's client verifies his signature and decrypts the message.[2]

If any of the continuity checks fail, the proofs supplied by the identity provider serve as evidence of misbehavior by the offending provider. Clients can publish this evidence on a heavily visited site or through social media to ensure quick propagation.

3.1 Making Directories Efficiently Verifiable

To provide our security properties, clients and servers perform various consistency checks on the directory. Thus, the infrastructure for an identity provider's namespace must support the following consistency proofs and their efficient verification:

1. **Proofs of binding consistency:** These must show that a binding is valid, i.e., the public key has not changed, or it has been revoked. These proofs must also show that a binding is unique in the directory, serving as a proof of inclusion.
2. **Proofs of absence:** These must show that a given name is not in the directory.
3. **Checks for non-equivocation:** These must allow the client to verify that the identity provider is presenting all entities in the system that same view of its directory which implies creating a single commitment history.

The Merkle prefix tree structure. To support consistency checks that can be efficiently performed by clients, identity providers construct a Merkle tree whose ordering is verifiable along every path and digitally signs the root node of the tree. Before verifying that a key is valid, users must be able to tell whether the reported binding is uniquely present in the tree. Thus, CONIKS employs a Merkle tree structured as a binary prefix tree where the lookup index corresponds to an unpredictable function of a given username. More specifically, left subtrees correspond to a 0 and right subtrees correspond to a 1 in the next bit of the prefix. Subtrees for prefixes that have not been mapped are replaced by null nodes.

When looking up a particular binding, providers respond with a proof of binding consistency or proof of absence depending on whether the requested binding is present in the tree. For these

[2]We have omitted this last step from Figure 1 for simplicity.

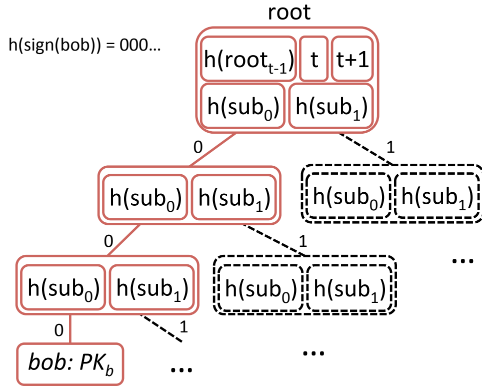

Figure 2: An authentication path from Bob's mapping to the root node of the Merkle prefix tree. Dotted nodes are not included in the proof's authentication path. Internal nodes only need to store the hash of each of their children since the prefix of a leaf node's lookup index determines the path from the root to a given leaf node.

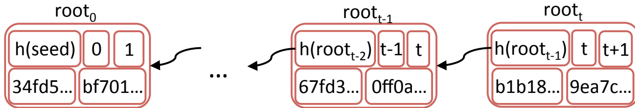

Figure 3: The identity provider's commitment history is represented as a linear hash chain of Merkle prefix tree root nodes. Initial epochs and epoch intervals are shown as single digits for simplicity. In practice, CONIKS denotes epoch numbers using Unix timestamps.

proofs, providers return an *authentication path* (Figure 2), i.e., a pruned tree containing the prefix path to the requested name-to-key binding, or the path to the null subtree that would contain the requested binding if it were not absent in the tree. Because the position of each leaf node in the tree is defined by the prefix of its lookup index, the proof of binding consistency shows that the requested name-to key binding is included in the tree, and that there does not exist another valid authentication path to the same binding.

An identity provider's commitments are computed by digitally signing the hash of the root node. Since *any* change of a binding results in a change in the value of the commitment, clients verifying a proof of binding consistency or absence can directly compare two commitments and can be sure that the authentication path is consistent with the received commitment.

Maintaining consistent commitments. To enable efficient checks for non-equivocation, CONIKS requires that identity providers maintain a consistent history of commitments across time epochs. Since different providers may publish commitments at different time intervals, epochs are denoted by Unix timestamps and providers commit to their epoch duration by specifying the epoch at which clients should *expect* the next commitment.[3] More specifically, CONIKS providers do this by including a hash of the *previous* commitment in the current root of their Merkle prefix tree, along with the current epoch number and the next expected epoch, creating a hash chain of root nodes (see Figure 3).

[3]This accounts for server outages that last longer than one epoch. Clients can decide whether they accept commitments published at a later time than the indicated expected next epoch.

Preserving Users' Privacy. One of our main goals is to provide a privacy-preserving key service that will not allow a user who already knows a given username in a directory and who has obtained a valid authentication path for this binding to gain any information on whether any *other* usernames' bindings are present or absent in the directory. The only way that an adversary can learn which usernames are present is by performing a name registration, which CONIKS strictly rate-limits. Generating a user's lookup index using a cryptographic hash function is not sufficient for this purpose, as any user can compute this index for any other username and the neighboring leaf nodes in each proof will leak the existence (or absence) of bindings for indices with a common prefix.

For example, if an adversary wanted to determine which username shares a prefix with *alice@foo.com* given the authentication path to her binding, they could find (by brute-force) a username x such that h(*alice@example.com*) and h(x) share a common prefix of length ℓ.

If the binding consistency proof for Alice does not include any neighbors with a common prefix of length ℓ, this will show that x is not included in the directory. If such a neighbor does exist, this will means that x may be in the tree, and for longer values of ℓ will provide increasingly strong probabilistic evidence that x is in the tree and not some other user with a common hash prefix. To ensure this, the attacker would want to find x such that $\ell \gg N$, the total number of users in the directory.

Instead, CONIKS uses a *verifiable unpredictable function* to ensure that each user's index in the directory cannot be predicted by attackers. VUF are a simpler form of a stronger cryptographic construction called *verifiable random functions* [42]. In our application, we only need to ensure that a user's location in the tree is not predictable; we do not need strong randomness although statistical randomness is helpful to produce a balanced tree.

We can implement a VUF using any deterministic, existentially unforgeable signature scheme [42]. 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 [51] (using a deterministic padding scheme such as PKCS v. 1.5 [26]) or BLS "short signatures" [9], which provide the best space efficiency. Many common discrete-log based signature schemes such as DSA [30] are not immediately applicable as they require a random nonce.

In CONIKS, we generate the index for a user u as:

$$i = \mathrm{h}\left(\mathrm{sign}_K(u)\right)$$

where h() is a one-way hash function and sign() is a deterministic, existentially unforgeable signature scheme. The extra hash function is added because some bits of $\mathrm{sign}_K(u)$ might otherwise leak through binding validity proofs, and signature schemes are generally not unforgeable given some bits of the valid signature. The hash function prevents this by ensuring the final lookup index leaks no information about the value of $\mathrm{sign}_K(u)$.

Therefore, absence and binding consistency proofs for an index i will leak no information about the usernames involved. The provider could even make public the entire list of empty and non-empty indices in the tree. This would, of course, reveal the number of users in the system, but this can already be approximated by the average length of binding validity and absence proofs. Users can verify the consistency of the binding by requesting $\mathrm{sign}_K(u)$ from the server, from which they can compute their index i.

The only drawback of using this method is that once Alice learns the lookup index of *bob@example.com*, this position in the tree is known for the rest of time. Thus, if any new bindings are inserted

in the vicinity of *bob@example.com*'s binding, Alice may gain information about those new neighboring bindings. In §6, we discuss possible enhancements to the Merkle prefix tree structure that make stronger guarantees about privacy.

3.2 Consistency Checks

To enable users to communicate securely, CONIKS ensures that name-to-key bindings are consistent via three types of proofs discussed in §3.1: proofs of binding consistency, proofs of absence, and checks for non-equivocation. CONIKS provides three protocols in which clients and identity providers collaborate to verify and detect any violation.

Verifying Binding Consistency. To verify that the identity provider returns a consistency name-to-key binding for *alice@foo.com* (see Figure 4), the client begins by looking up her binding for the current epoch PK^t along with the associated proof $authPath^t$. If it has not already done so, the client also downloads *foo.com*'s commitment for the current epoch cmt^t (including the root node $root^t$ for signature verification). The client additionally retrieves Alice's public key for the previous epoch PK^{t-1} from local storage, or requests it from the provider, to check that the current key meets one of these conditions:

1. the key is unchanged: $PK^t = PK^{t-1}$; or
2. the key has been revoked (with a valid revocation statement).

The client can check the first condition directly. For the second condition, the verification performed depends on whether Alice is a security-conscious user enforcing a stricter security policy at epoch $t-1$.[4] If she is, her binding at epoch t must include a revocation statement which *must* be signed by Alice's key from epoch $t-1$ and must contain the new key (or a "tombstone" marker denoting the permanent deactivation of her binding). Otherwise, the client simply checks that the identity provider has included a standard revocation statement. Alice's binding is inconsistent if it does not meet either of these conditions, or worse has transitioned from a *tombstoned* state to any other state. Next, the client verifies the authentication path to ensure that PK^t is uniquely present in the Merkle prefix tree. This includes checking that the reconstructed tree root is consistent with the cmt^t.

Verifying Absence of a Binding. There are two scenarios in which a client may want a proof of absence for a specific binding: (1) When registering a key, and (2) In response to a public key lookup. For instance, when registering a new name-to-key binding, Alice may want to check that the directory does not already contain a binding for *alice@foo.com*.[5] Alternatively, Bob wants to look up Alice's binding at *foo.com* without knowing if she has registered a binding in the directory. Thus, the identity provider returns an authentication path similar to the one returned as part of the proof of binding validity. This path shows that the lookup index for *alice@foo.com*, which the client can verify via the verifiable unpredictable function, leads to a null subtree along the prefix path. In the same fashion as in the binding validity check, the client checks the consistency of this absence proof by reconstructing the root hash and verifying *foo.com*'s commitment.

Checking for Non-Equivocation. To check that an identity provider is not presenting different commitments to different users and providers, clients and providers verify the hash chain and compare the results of these verifications. Clients perform this check whenever the user's identity provider issues a new commitment. Identity providers, however, perform this check for every commitment they receive from other providers so that clients can later compare their version of a commitment with what other providers are observing to rapidly detect equivocation.

The hash chain verification is summarized in Figure 5. To verify the hash chain, the verifier first ensures that the provider correctly signed the commitment cmt^t for the current epoch before checking whether the *previous* hash in the current root node $root^t$ matches the hash of the root node $root^{t-1}$ observed in the previous epoch. If these two hashes do not match, the hash chain is invalid and the provider is equivocating about its bindings.

In order to detect whether an identity provider is equivocating, the client performs the protocol detailed in Figure 6. The client begins by querying one or more CONIKS identity providers at random.[6] The client asks this assisting provider *bar.com* for the signed commitment $obsCmt^t$ it observed from provider *foo.com*. The client then compares $obsCmt^t$ with the commitment cmt^t which *foo.com* directly presented it.

Performing Checks after Missing Epochs. Because commitments are associated with each other across epochs, clients can "catch up" to the most recent epoch if they have not verified the consistency of their bindings for several epochs. They do so by performing a series of checks until they are sure that the consistency proofs they last verified are consistent with the more recent proofs.

3.3 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 revocation. For each functionality, we propose two security policies: a *relaxed* 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.

Queryable bindings. Our goal is to enable the same level of privacy which SMTP servers employ today, [7] in which usernames can be queried (subject to rate-limiting) but it is not easy to enumerate the entire list of names.

Users need to decide whether their binding in the directory should be publicly queryable. In the relaxed lookup policy, the binding is publicly discoverable. For example, anyone who knows Alice's name *alice@foo.com* can look up her key in *foo.com*'s directory and check the consistency of her binding. On the other hand, the strict lookup policy allows only Alice to look up her binding at *foo.com* requiring the server to deny any requests for *alice@foo.com* that do not originate from Alice. To allow Bob to communicate with her, Alice sends him an invitation which contains Alice's binding. To allow Bob to verify the consistency of her binding, Alice can send the proof of consistency (which she has successfully verified) for her binding along with her messages to Bob. Upon receipt of any message from Alice, Bob's client automatically obtains the appropriate commitment from *foo.com* and

[4] We discuss this and other security options in §3.3.

[5] Even if Alice receives a proof of absence, she cannot be sure that nobody else has registered *alice@foo.com* during the same epoch before the identity provider has updated the directory.

[6] We assume the client maintains a list of CONIKS providers from which it can choose any provider with equal probability. Evidently, the longer this list, the harder it is for an adversary to guess which providers a client will query.

[7] The SMTP protocol defines a *VRFY* command to query the existence of an email address at a given server. However, it has long been recommended to ignore this command (reporting that any user names exists if asked) to protect the privacy of user's accounts [37].

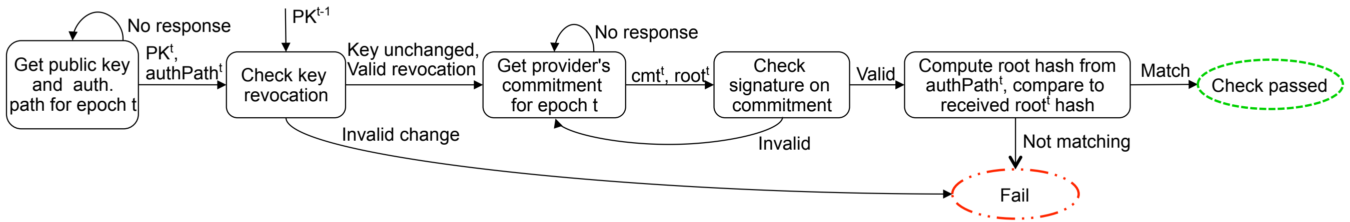

Figure 4: Flow chart of the binding consistency check a client performs every epoch. Clients first verify whether public key has changed unexpectedly, and then verify the authentication path in the Merkle prefix tree.

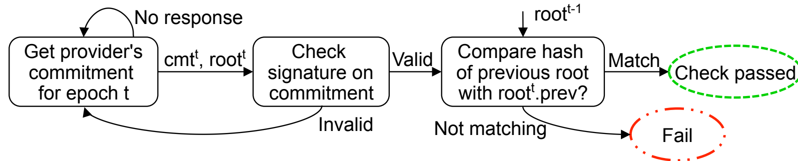

Figure 5: Flow chart of hash chain verification that clients and providers perform every epoch when checking for non-equivocation.

checks that it is consistent with the authentication path included in her message.

The main advantage of the relaxed policy is that it matches users' intuition about being able to interact with any user whose username they know without requiring explicit "permission" or an invitation to do so. On the other hand, the strict lookup policy provides stronger privacy guarantees since it allows Alice to control who knows her binding, but it requires additional action by her to send chat or email invitations to all of her contacts, or it requires Bob to wait for Alice to approve his contact request before the two can exchange messages.

Key Revocation. 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 *unilaterally* revoke a user's public key after such a recovery process in the event that the user's previous device was lost or destroyed. If Alice chooses the relaxed revocation policy, her identity provider *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 revocation 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 revocation statement in her directory entry.

However, an identity provider could maliciously change a user's key and falsely claim that the user requested the operation. In this case, CONIKS only guarantees that clients interacting with Alice will observe the key change. However, CONIKS *cannot* guarantee that these users will be able to distinguish automatically between malicious and benevolent key changes because the client does not obtain non-repudiable cryptographic evidence that a new key binding is not genuine. 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, the client will detect these updates and can notify Alice, making surreptitious revocations risky for identity providers to attempt.

Security-conscious users may choose to require that *foo.com* include a revocation statement signed by the key that is being revoked. More specifically, if Alice has revoked her public key, her client sends this statement along with her new public key to her identity provider. Similarly, her client can generate a signed revocation statement containing a null-marker (instead of the new public key) denoting that Alice wishes to place a tombstone on her binding. These users can be sure immediately that *any* unauthorized key change or tombstone is a sign of malicious server behavior. But requiring authenticated key changes does sacrifice the ability for a user to regain control of their username if her key is ever lost. We discuss some implications for key loss for these users in §6.

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 [27, 54]. 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 revocation. Following this easy enrollment procedure requires that Alice enforce the relaxed revocation policy, and her client will no longer be able to automatically determine if a newly observed binding has been maliciously inserted by the server or simply represents Alice installing a second client on 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 strict revocation policy discussed above.

4. SECURITY ANALYSIS

CONIKS's infrastructure and protocols are designed to protect against and detect attacks from identity providers. In doing so, CONIKS provides the following guarantees:

- Users will immediately detect any inconsistent bindings.
- With high probability, users attempting to communicate will rapidly detect any provider equivocation.

4.1 Ensuring Binding Consistency

An inconsistent name-to-key binding has either had its key revoked without a valid revocation statement; or two bindings exist in the same directory for the same name. Because of the structure of the Merkle prefix tree, there cannot be two distinct binding proofs for the same name that are both valid. CONIKS provides a

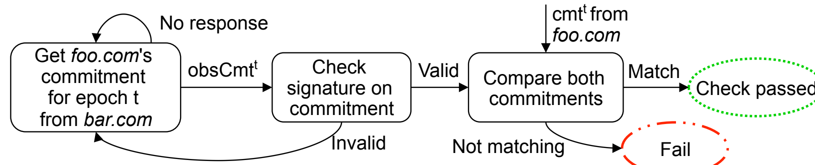

Figure 6: Flow chart of the comparison of observed commitments that clients perform every epoch when checking for non-equivocation. Identity providers and clients collaborate in verifying that the commitments they are observing for a particular identity provider are consistent.

choice of security policies for revocation, but can only guarantee the *automatic* detection of consistency violations for the bindings that require owner-signed revocation statements. In this case, if the binding has changed without such a signed revocation statement, when the client requests a proof of binding consistency for the updated binding, the misbehaving provider will be unable to respond without revealing the attack. If the user enforces the relaxed revocation policy, all the client can do is report any suspicious key changes to the user, but the user is then guaranteed to detect any consistency violations. As long as the owner of the binding performs the consistency check at every epoch, such attacks will be promptly detected.

4.2 Detecting Equivocation

Without violating the consistency of name-to-key bindings, malicious identity providers might instead present two different, but independently valid, name-to-key binding histories, choosing which users and providers will be presented which set of bindings. Even if both histories are otherwise consistent, this is equivocation.

To detect equivocation, providers exchange commitments, and clients check with multiple randomly chosen providers to ensure they are observing the same commitments from any provider of interest. In Appendix A we show that if a fraction p of providers are colluding, and the colluding providers coordinate their behavior, then if Alice and Bob each perform k checks, the probability that both of their clients fail to detect equivocation is

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

Figure 7 plots the probability of discovery as p and k vary. If fewer than 50% of providers 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 provider whom they both trust to be honest, then they can detect equivocation with certainty if they both check with that trusted provider.

4.3 Limiting the Effects of Denied Service

Sufficiently powerful identity providers may refuse to distribute commitments 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 commitments from identity providers to these honest identity providers. The honest identity provider can verify the other identity provider's signature, and then store and redistribute the root commitment.

Similarly, any identity provider might ignore requests about individual bindings in order to prevent clients from performing binding validity checks. In these cases, clients may be able to circumvent

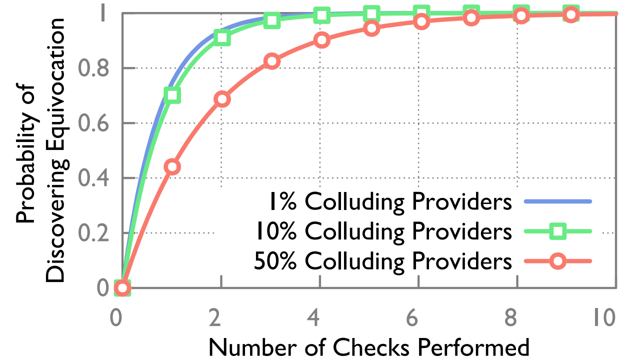

Figure 7: Two users, Alice and Bob, will be able to, with high probability, detect any equivocation attacks with a few verification checks. This graph shows the probability that Alice and Bob will detect an equivocation after each performing k checks with random providers.

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.

5. IMPLEMENTATION AND EVALUATION

CONIKS provides a framework for integrating key management 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 [10] (OTR) plug-in for the Pidgin instant messaging client [1, 23]. We implemented a stand-alone CONIKS server in Java (∼2.5k sloc). We 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 [53]. 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 signa-

tures. *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 [25], 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 as well as every generated commitment 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 §3.2, 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 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 revocation. Furthermore, our prototype only supports public lookups for name-to-key bindings, meaning that clients cannot exchange proofs through messages to enforce the strict lookup policy. Fully implementing these features is planned for the near future.

5.2 Performance Evaluation

In order to understand the expected performance of CONIKS in practice, we collect both theoretical performance and real performance characteristics of our prototype implementation. We evaluate with the following assumptions:

- A single server instance supports $N \approx$10M users.
- Epochs occur every 5 minutes (roughly 2^8 per day).
- Up to 1% of users change or add keys per day, meaning $k \approx$ 350 changes in an average epoch.
- Servers use a 128-bit cryptographic security level.

We have simulated this scenario from the point of view of both client and server overhead. 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 [7], this key only ensures privacy and not non-equivocation. A summary of bandwidth requirements under alternate scenarios is given in Table 1.

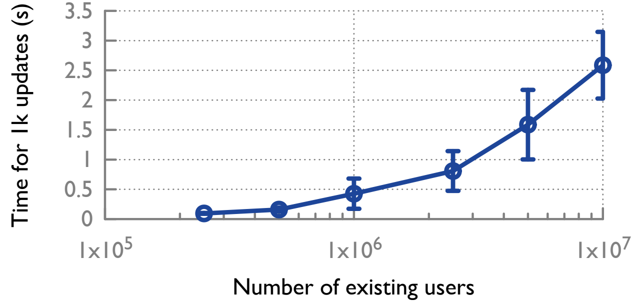

Figure 8: Mean time to re-compute the tree for a new epoch with 1K updated nodes. The x-axis is logarithmic and each data point is the mean of 10 executions. Error bars indicate standard deviation.

sig. scheme	sec. level	root commitment	binding proof	full update	incremental update	daily update
RSA/ECDSA	80	80	721	801	209	58.7k
RSA/ECDSA	128	128	1000	1128	334	94.0k
BLS	80	60	485	545	189	53.1k
BLS	128	96	776	872	302	85.0k

Table 1: Estimated bandwidth consumption for an 80- and 128-bit security level using traditional RSA and ECDSA primitives or BLS short signatures.

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 average expected epoch updates) into the Merkle prefix tree, and measured the time it took for the server to compute the next epoch.

Figure 8 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 million users, computing a new epoch with 1000 updates takes on average 2.6 seconds. As epochs only need to be computed every 5 minutes, this amount of time is not cumbersome for a large service provider.

In addition to the computational cost of epoch commitments, identity providers must exchange and *transmit* these commitments to one another. Each commitment consists of the root node of the tree and one ECDSA signature (the previous root and epoch number can be inferred and need not be transmitted). The total size of each commitment in minimal form is just 96 bytes (64 for the signature and 32 for the root). If a network of 1 million providers exchanges commitments between each other, the total upstream and downstream bandwidth requirements per epoch are 96 MB.

These numbers indicate that even with a relatively unoptimized implementation, a single machine is able to quickly handle the ad-

ditional overhead imposed by CONIKS for workloads similar in scale to large communication providers today.

Client Overheads. In order for any client to completely check the consistency of a binding, it needs to download a proof consisting of about $\lg_2(N)+1$ hashes plus one 256-byte RSA signature (proving the validity of the binding's index). This will require downloading $32 \cdot (\lg_2(N)+1)+256 \approx 1000$ 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 commitment signature (64 bytes) must be downloaded, however, the server can significantly compress the length of each proof by only sending the hashes on the user's path to the root 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 $64+\lg_2(n)\cdot 32 = 334$ bytes. Verification time will be similar to verifying another user's proof, dominated by the cost of signature verification.

To check every commitment for a day, the client must download a total of about 96.3 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 [9], which can replace both the RSA signatures used for index verification and the ECDSA signatures used for tree commitments. 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 limit the size of an arbitrary binding from 1000 bytes to 776 bytes, and can reduce the proofs required to verify binding consistency from 96.3 kB to 77.9 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 as these schemes have more mature library support and faster signature generation time, minimizing computational load on the server.

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 [11, 21, 22, 24, 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 [17]. Nevertheless, because of the consistency 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[40].)

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 strict revocation 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 revocation policy. The provider would use some specific out-of-band authorization steps to authorize such an action. Unlike allowing providers to issue key revocation 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

Returning Updated Bindings between Epochs. Users of identity providers which have longer epochs may desire the ability to use updated bindings *before* the next epoch. Identity providers could support this by maintaining a second "staging" tree, which contains the updates which have been made during the current epoch. As in the main Merkle prefix tree, the bindings in this staging tree are guaranteed to be consistent. However, normal consistency checks cannot be performed until the following epoch. When an updated binding is requested, the provider returns the binding in the staging tree, as well as an authentication path in the main tree demonstrating that the current epoch's binding is consistent with the updated binding. The provider signs this message to make it irrefutable.

During the next epoch, the identity provider incorporates the new updates and adds a link from the new epoch's root node to the last staging tree. Then, any clients which observed an updated binding on the previous day must perform a check to ensure that the observed update was correctly incorporated into the epoch. They will check that the binding they received exists in the current version of the directory and that no conflicting bindings existed in the same staging tree. Any future requests for the binding will be checked as in the normal CONIKS checks.

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. To prevent Alice from gaining information about the new neighboring bindings inserted into the directory in the vicinity of *bob@foo.com*'s binding, the identity provider can choose to randomize the ordering of entries periodically by including additional data when computing their lookup indices. For example, `foo.com` could include the epoch number in every lookup index calculation so as to change the ordering every epoch. 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 further automate key management and 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 revoke the expired key and specify the new expiration date according to the user's policies.

7. RELATED WORK

Certificate Validation Systems. A number of projects have proposed systems and protocols for validating TLS certificates and for making certificate authorities (CAs) accountable for their activity [4, 28, 35, 47, 48, 52, 56]. We briefly discuss three of these projects that have been particularly influential on CONIKS, and describe how we are adapting techniques from these approaches and improving upon them to provide our desired security properties.

Certificate Transparency (CT) [33, 35, 36] is a log-server based system in which CAs register issued TLS certificates with log servers which maintain these in append-only Merkle trees. Log servers make signed tree roots available for third-party auditors and clients. To detect equivocation about a log, auditors and clients can gossip about observed signed tree roots. Revocation Transparency [34] attempts to extend CT to support revocation, but is much more costly in terms of performance. Enhanced Certificate Transparency (ECT) [52] seeks to improve upon CT to efficiently handle certificate revocation. It does so by requiring log servers to maintain two types of Merkle trees, one ordered chronologically, the second one lexicographically. Each tree is meant to provide efficient proofs of inclusion or proofs of revocation, respectively. The authors Enhanced Certificate Transparency discuss how ECT can allow end users to rely on CAs for key management without having to trust them, which in turn facilitates end-to-end encrypted email. However, ECT is not privacy-preserving and does not provide multiple security options. The Accountable Key Infrastructure (AKI) combines the techniques used in log servers [35, 47] and in certificate observatories [4, 48, 56], to create a certificate validation system which focuses on providing key revocation and accountability.

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 name space. 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,[8] and as a result, they use data structures making the entire name-space public for third-party auditors to check.

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 [39]. 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 [29]. These problems with properly managing keys that even advanced users like web admins have highlight how important it is to require no key management by end users.

Alternative Public-Key Infrastructures. We are far from the first to recognize that neither the traditional CA model nor the decentralized "Web of Trust" model [8] are perfect solutions to binding certification. Towards this end, PKIs alternative to the traditional X.509 model have been proposed. As a key example, the SPKI/SDSI model [13] emphasizes naming, groups, and flexible authorization. Yet, while CONIKS and SPKI/SDSI both support disjoint per-provider namespaces and do not give specific semantics to the name bindings, SPKI/SDSI exposes the management question of choosing trusted authorities to end-users. And perhaps more importantly, SPKI/SDSI does not address the security threat of malicious or coerced providers.

SFS [41] introduced the concept of *self-certifying* names: by containing the public key itself (or a hash thereof), a name can separate the issue of key management from the system security. Self-certifying names do not require an external PKI to certify name-to-key bindings, which is ideal in the face of untrusted communication providers. However, self-certifying names are not human-readable, and do not solve the problem of (securely) discovering and disseminating names in the first place.

Identity and Key Services. Keybase [31] provides a public directory of publicly-auditable name-to-key bindings. It records a name-to-key binding, as well as registered account information about the same user from one or more third-party service providers (e.g., Twitter). When requesting a name, a Keybase client checks that these accounts belong to the same principal by verifying that the same key was used to generate signatures across third-party accounts. Users additionally form a web of trust by "tracking" each other's bindings. The directory is also maintained in a Merkle tree, whose signed root is published at various third-party websites and the Bitcoin blockchain [32].

OneName [3] stores each username in the Namecoin [43] blockchain ensuring that names are globally unique. Once in the blockchain, anyone can look up a OneName username and obtain the associated public key. Similar to Keybase, OneName users can link their name-to-key bindings to accounts with third-party services.

The main drawback of both Keybase and OneName is that they require explicit user action to add keys and verify them with various third-party services. Recovery from key loss or adding new keys also requires manual user intervention.

Nicknym [50] is a protocol for mapping user nicknames to public keys, and makes these bindings publicly auditable by allowing clients to query any provider for *individual bindings* they observe. While equivocation about bindings can be detected in principle in this manner, users do not have the option of stronger security guarantees by preventing malicious key changes via user-authenticated key revocation as is possible in CONIKS. This becomes a greater concern given that Nicknym does not maintain an authenticated history of published bindings which would provide more robust consistency checking and give clients better evidence that the provider has maliciously changed keys. Thus, Nicknym requires users to place more trust in providers to detect inconsistencies, and does not consider provider collusion.

[8] 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.

8. CONCLUSION

We have presented the design of CONIKS, a key 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-audit 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 messaging protocols such as OTR chat and PGP email. We have built a prototype CONIKS system which includes a prototype CONIKS service and a prototype chat application CONIKS Chat. The client application performs consistency checks in the background, and avoids any explicit user action except in the case of notifications that a new key binding has been issued. Our prototype CONIKS service is application-agnostic and supports millions of users on a single commodity server.

References

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

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

[3] OneName. `https://onename.io`, Retrieved May 2014.

[4] Convergence. `https://convergence.io`, Retrieved Nov. 2014.

[5] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. SOSP*, Oct. 2005.

[6] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptol.*, 23(2):281–343, Apr. 2010.

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

[8] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. SP*, May 1996.

[9] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *Advances in CryptologyâĂŤASIACRYPT 2001*, pages 514–532. Springer, 2001.

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

[11] 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.

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

[13] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, Dec. 2001.

[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] M. Farb, Y.-H. Lin, T. H.-J. Kim, J. McCune, and A. Perrig. Safeslinger: easy-to-use and secure public-key exchange. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 417–428. ACM, 2013.

[17] 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.

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

[25] M. Jakobsson and M. Yung. Proving without knowing: On oblivious, agnostic and blindfolded provers. In N. Koblitz, editor, *Advances in Cryptology âĂŤ CRYPTO âĂŹ96*, volume 1109 of *Lecture Notes in Computer Science*, pages 186–200. Springer Berlin Heidelberg, 1996.

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

[27] 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 '09, pages 11:1–11:12, New York, NY, USA, 2009. ACM.

[28] 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.

[29] 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.

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

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

[32] M. Krohn and C. Coyne. Keybase. `https://keybase.io/docs/server_security/merkle_root_in_bitcoin_blockchain`, Retrieved Nov. 2014.

[33] B. Laurie. Certificate Transparency. *Queue*, 12(8):10:10–10:19, Aug. 2014.

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

[35] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. `http://www.certificate-transparency.org`, Retrieved Aug. 2013.

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

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

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

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

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

[41] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. SOSP*, Dec. 1999.

[42] 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.

[43] namecoin1. Namecoin. `http://namecoin.info/`, Retrieved Nov. 2013.

[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 Nov. 2014.

[51] 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.

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

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

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

[55] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008.

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

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

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

APPENDIX

A. DETAILED ANALYSIS OF EQUIVOCATION DETECTION

A.1 Single Equivocating Provider

Suppose that *foo.com* equivocates by showing commitment A to Alice and commitment B to Bob. *foo.com* chooses a fraction f of providers who will be shown commitment A, and the remaining fraction $1-f$ see commitment B. If Alice and Bob each contact k randomly chosen providers to check consistency of *foo.com*'s commitment, 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, Alice and Bob would have discovered an equivocation with 99.9% probability.

A.2 Multiple Colluding Providers

Now suppose that *foo.com* colludes with other providers, and the colluding providers agree to tell Alice that *foo.com* is distributing commitment A while telling Bob that *foo.com* is distributing commitment B. As the size of the collusion increases, Alice and Bob become less likely to detect the equivocation. However, as the number of identity providers in the system (and therefore, the number of identity providers not participating in the collusion) increases, the difficulty of verification decreases.

More precisely, we assume that *foo.com* is colluding with a proportion p of all providers. The colluding providers behave as described above, and *foo.com* shows commitment A to a fraction f of the non-colluding 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.