Dename: A Decentralized Egalitarian Name System

ABSTRACT

This paper presents dename, a system for public key distribution that provides strong security guarantees in the face of server compromises. In the X.509 certificate authority system used by the web, a single compromised certificate authority server allows an adversary to break all security guarantees. By contrast, dename guarantees security as long as just one of the servers remains honest. To achieve these goals, dename provides a simple design built around three key ideas: (1) Require all servers to reach consensus on the assignment of names to keys, using a *blinded consensus protocol*, which prevents a subset of compromised servers from violating security guarantees; (2) Provide first-come-first-served name registration, which makes it easy to audit the correctness of a sequence of name operations; and (3) Maintain both a log of operations and a tree-based summary of the current state, which enables efficient client name lookups as well as third-party auditing. Using a prototype implementation of dename, we demonstrate that it is easy to incorporate dename into existing applications, including OpenSSH, GPG, and Pond. Experimental results demonstrate that dename achieves good performance, even with a geographically distributed set of servers.

1 INTRODUCTION

A critical aspect of most cryptographic systems is the association of users' public keys to application-level usernames, and many systems rely on a trusted directory service that associates public keys with user identities. For example, web browsers rely on a set of certificate authorities to sign X.509 certificates that bind a web site's hostname to that web site's public key and allow users to know what web site they are visiting. An important benefit of such a design lies in its simplicity for application developers and end users. The directory service takes care of mapping keys to names; the application can assume the existence of a function to lookup the key for a given name or verify a name-to-key mapping, as in X.509 certificates; and users can likewise assume a perfect oracle that associates appropriate keys with user-visible identities.

Unfortunately, most directory-based systems, including the X.509 certificate infrastructure used on the web, suffer from several problems, as follows.

First, compromises of trusted directory servers, such as X.509 certificate authorities, enable an adversary to completely break the assumptions that applications and end users have about name-to-key mappings [2][11][6]. For instance, recent compromises of the DigiNotar and Comodo certificate authorities enabled adversaries to impersonate well-known web sites such as Google. As the directory service grows, adding more servers is likely to decrease overall security, since it suffices for an adversary to compromise any one of the servers in order to break the system's security.

Second, most directory-based systems are based on the premise that the directory service will verify whether a user really owns a particular name (such as a domain name, a person's full name, or an email address). This is appealing because it associates public keys with existing real-world identities. Unfortunately, it is exceedingly difficult to formalize what it means for a user to own a non-cryptographic name, or to audit whether a directory service properly verified the ownership of any given name. Not surprisingly, this has led to high-profile mistakes, such as Verisign signing certificates containing Microsoft's name for a party that fraudulently claimed to be Microsoft.

Third, clients must be able to obtain and verify name-to-key mappings from the directory service. Most directory services rely on long-lived certificates issued to name owners to address this problem, which makes it difficult for a client to determine if a name-to-key mapping is still valid, and makes it difficult for a name owner to revoke a mapping in a timely manner. As a result, even if a name owner knows that the mapping is no longer valid (e.g., because the key has been compromised), clients may still accept the old mapping and thus the old key.

One workaround for these problems, used by many security-critical systems, is to avoid the assumption of a single directory service, and instead ask for help from the user. For example, OpenSSH, OpenPGP, OTR, and Pond all require users to manually communicate critical bits of authenticating information to each other. While this improves security for expert users, it is unfortunately tedious and error-prone, and can be difficult to use if the users are not online at the same time [14][12][3].

This paper presents dename, a public key distribution system that addresses the above challenges and provides a practical design that integrates easily into existing systems such as OpenSSH, GPG, and Pond, without requiring any additional user effort for key management. At a high level, dename works by having a group of well-known, independently administered servers maintain identical copies of the directory. Clients can contact any server to register, update, or lookup name-to-key mappings; updates must be signed by the user's key. As long as just one of the servers remains honest, dename guarantees that the name-to-key mapping is correct. An additional set of third-party *verifiers* audits the work of servers and can be used by clients to increase the level of assurance.

The design of dename is purposely simple, and dename builds on several key ideas:

First, instead of allowing *any* server to assign a key to a name, dename requires *all* servers to reach consensus on the assignment of names to keys, and the order in which these assignments occurred. This means that an adversary that wants to change the key for a name, or pretend to have already registered an existing name, would have to compromise all of the servers responsible for registering names, instead of just one server. dename introduces a round-based *blinded consensus protocol* that allows servers to reach consensus on the set of registered names without allowing a compromised server to subvert new names by introducing its own concurrent registrations.

Second, dename eliminates the notion of a user owning any name a priori. Instead, dename provides first-come-first-served name registration. The key advantage of this design choice lies in the fact that it is easy to audit algorithmically: if a name was not previously registered, it can be registered, and if a name was already registered, it cannot be registered again. Since no human input or real-world checks are required to determine if a name registration was performed properly, all name registration servers can mechanically check the work of all other servers, and even third-party verifiers can make sure that all name registration servers did their work correctly.

Third, dename servers construct two authenticated data structures: a signed log of all name registration operations, and a Merkle

tree summarizing the current state of all registered names. The tree allows clients to efficiently lookup and verify name-to-key mappings, and ensures freshness with the help of periodic timestamps. The log enables servers to cross-check each others' registrations, and also enables third-party verifiers to make sure all name operations in the log are legitimate and correspond to the summary in the tree.

To demonstrate the practicality of `dename`, we implemented a prototype in Go, and integrated `dename` with OpenSSH, GPG, and Pond. These systems previously relied on users to manually transfer and authenticate keys. With `dename`, users achieve strong security guarantees with the convenience of a global PKI name-to-key mapping, without error-prone manual steps. Experimental results show that integrating `dename` into an existing system requires little effort, and that `dename` servers can achieve good performance even in a large, geographically distributed configuration.

2 Related work

There are multiple systems that map human-meaningful names to security-critical public information such as public keys. Subtle differences in the semantics of how the names are assigned can have a huge impact on a system's security properties.

Single-assigner: The most widely used way of associating public keys with names is the X.509 Public Key Infrastructure. Any one of the globally known and trusted set of certificate authorities can handle a certificate signing request (usually for a fee) and produce a digitally signed certificate stating that a certain public key belongs to the entity bearing the specified name. The compromise of any one of the certificate authorities can and has enabled attackers to impersonate arbitrary names[2][11][6]. Other systems that permit a single party to change the mapping, such as DNS and Keybase, are subject to similar issues. Secure *lookup* protocols such as DNSSEC or DNSCurve do not protect against authority compromise either.

In **DANE**[5], the manager of a DNS domain assigns public keys to its subdomains. This limits the effects of the compromise of an assigner to its subtree, but the root is still a central point of failure.

Certificate Transparency[8] provides a means to detect certificate authority misbehavior. Its creators argue that the threat of public scrutiny would deter intentional violations of the certification practices and provide additional motivation for the certificate authorities to keep their systems secure. However, the effectiveness of these indirect measures has not been proven, so they currently insufficient to give a strong security guarantee.

Availability-based: With Convergence[10] or Perspectives[13], public "network notary" servers regularly monitor the public keys used by public websites. Anyone can run a network notary server; anyone can choose the set of notaries to trust. These systems are inherently limited to servers that are always online when their public keys are looked up. The mechanism the notaries use to contact the servers is also a potential point of attack; currently unauthenticated DNS is used. Furthermore, no consistency guarantees are provided: two honest notaries may report a different public key for the same site.

Cross-certification systems such as Crypto-Book[9] and Keybase's account ownership proofs do not even aim to provide better security than the social network websites they rely on for user authentication. In general, designs that strictly follow the naming conventions of an existing non-cryptographic system provide weaker security guarantees because the name assignment process is a single point of failure.

NameCoin aims for very similar semantics to `dename` but uses a completely different construction. The mapping is determined by the longest log of name assignments available; anybody can extend any log, but doing so is computationally intensive. If most of the computational power is spent on NameCoin is controlled by parties who adhere to the protocol, a correct log will be the longest with high probability. As extensive computation is required to register a name, the entities who perform this computation charge for registration. This provides an incentive for nodes to invest computational power into NameCoin, but there is no guarantee that good nodes are computationally more powerful than bad nodes. Either way, this competitive use of resources results in high operating costs[15].

3 Overview

A deployment of `dename` has a fixed namespace and set of servers. Each server (and each verifier) has a signing key and knows the public key of every other server. To modify a name-profile mapping, all core servers have synchronize with each other, so if one of them is down, no progress can be made. Verifiers are the same as core servers, except that all core servers are not required to know about a verifier and the core servers will continue to operate even when a verifier is not available. We assume that the clients obtain the public keys of the core servers and any verifiers whose confirmation they require out of band. To register a name, modify a profile, or look up the profile associated with an existing name, a client can contact a server of its choice. The servers prevent clients from modifying each others' profiles by requiring the change request to be digitally signed with a key designated in that profile.

A `dename` server exposes the following API:

- `modify(name, newProfile, newSignature, oldSignature)`: Make the name point to the new profile. `newSignature` is a signature on the request with the key contained in the new profile. If an old profile exists, `oldSignature` is a signature on the request with the old key.
- `getRoot()`: Return the hash of the current directory state signed by all servers.
- `lookup(name) -> (profile, proof)`: Return the profile that the name points to, along with a proof that the name-profile pair is present in the directory and up to date. Clients can check the proof against the latest root hash.

We envision that `dename`'s first-come-first-served registration policy can be easily incorporated into existing systems, by simply changing the order of name registration. For example, instead of first registering for an account with an email client, and then creating a mapping for that name, the email client should first register an appropriate name for the user in `dename`, and if that succeeds, create an email account under that username.

The discussion of the operation of this system is organized as follows: Section 2 contains a short review of several systems that seek to provide similar properties. In Section 4 we describe how the servers communicate with each other to apply changes to the directory while ensuring that they end up with identical results. In general, this is the problem of replicating a state machine in the presence of malicious faults, but the case we tackle is simpler: we require all parties to participate, sacrificing availability for consistency. We describe the procedure of looking up users' profiles in Section 5: We start with a trivial but inefficient protocol and end up storing the directory in a Merkle hashed radix tree and serving its branches. We argue that if a lookup succeeds, then the result must have been accepted by all servers. In Section 6 we tackle the issue of freshness; that is, we provide a system for ensuring that the result represents an up-to-date state of the system. In Section 7 we show how independent verifiers can be added to this system in the spirit of Certificate Transparency[8]. Given the Merkle tree data structure, this addition

is relatively straightforward and, as a side effect, enables efficient coherent caching of lookup results. We describe the specifics of our implementation of the protocol described in this paper in Section 8 and its performance in Section 9.2. Finally, we evaluate the impact of our system on the usability of three security-critical applications: digital signature management and public key encryption (OpenPGP), server administration (OpenSSH), and asynchronous messaging (Pond); see Section 9.1.

4 MAINTAINING CONSENSUS

Changes to the user directory happen in discrete rounds: every Δt (currently 3 seconds), each server proposes a set of changes and all servers apply them in lockstep. We use a verified broadcast primitive (described below) to ensure that all servers receive the same set of requested changes, and we handle the changes in a deterministic order. Additionally, we describe some malicious behavior that servers could engage in which would not directly violate the security claim, but is nevertheless undesirable, and *blind* the protocol to counteract that behavior.

The physical analogy of verified broadcast is a public announcement: everybody learns what the announcer has to say and can be sure that others heard the same thing. In computer networks allowing only point-to-point communication, we can emulate this using a two-phase protocol: first the announcer broadcasts the message, then every server broadcasts an acknowledgment of what they received from the announcer. In `dename`, all n servers announce exactly one set of changes $\Delta_1 \ldots \Delta_n$ during each round, so we can group each server's acknowledgments of all messages it received into one message. Furthermore, as only the equality of the sets of announcements received by different servers is important, rather than the actual contents, we can sign a cryptographic hash $h(\Delta_1 \parallel \ldots \parallel \Delta_n)$ of all received announcements in an acknowledgment instead of the announcements themselves. The verified broadcast protocol can be seen in Figure 1.

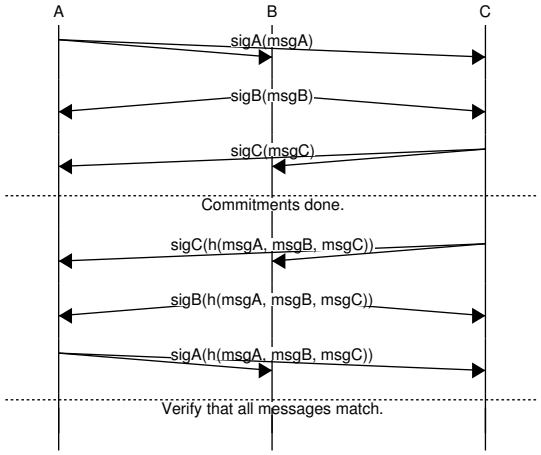

Figure 1: Verified broadcast

In the description above, all messages are assumed to be authenticated. If one server were able to impersonate another, it could fool other servers into thinking that a different set of changes has been announced. We use digital signatures for authentication because unlike faster symmetric authentication mechanisms, signatures can be used to construct an audit trail in case one of the servers sends out different announcements or acknowledgments during the same round.

The semantics of what kind of changes are allowed are in some sense a detail, but they are important. For example, if one user were able to edit another's profile without their consent, the directory would be of little use. Out implementation sets the following constraints:

1. All proposed profiles must contain a public key that can be used to verify digital signatures.
2. If a name is not currently in use (does not map to a profile), all requests to make it map to a valid profile are accepted.
3. If a name already maps to a profile, requests to change the mapping are only accepted if they are also signed with the key in the current profile in addition to that in the new one.

These rules ensure that if a user keeps their signing key secret, nobody else can modify their profile. To free up names for which the corresponding secret key has been lost, we also allow expiration:

4. If the profile a name maps to has not been modified in the last T_e rounds, the profile is automatically deleted.

Table 1 shows the fields stored by `dename` with example data.

name	pubkey	profile	last change
alice	pk_a	$pk_{\texttt{ssh}}$, pk_{x509}	18926 (2014-01-11)
bob	pk_b	`bob@mit.edu`, pk_{gpg}	47707 (2014-01-12)

Table 1: `dename` directory schema

This requires users to regularly confirm that they still use that profile by requesting a null change to it. The possibility of name-profile mapping expiration complicates the situation because somebody other than the original owner may later claim the used name, while the old profile continues to fit the criteria of being accepted by all servers – this is the main motivation for freshness assertions (Section 6). It is, of course, possible not to have names expire, but doing so would seriously hamper the usability of the system owing to the proliferation of names that map to lost keys.

The described rules of changing the directory are sensitive to the order in which changes are processed: if two servers propose two valid requests to modify the same name in different ways, it is crucial to ensure that all servers choose to apply them in the same order because applying one of them may make the other invalid. We use a standard commit-and-combine protocol to establish shared randomness between servers and use it to pick a random permutation of the list of servers that determines the order in which the requests they introduced are handled.

However, a malicious server could observe the announcements other servers make and deliberately introduce requests that conflict with a particular user's requests. To prevent this, a blinded protocol is required: the requests are hashed before they are broadcast using the verified broadcast protocol, and actual requests are only revealed after every server has announced the hash of their proposed changes. Therefore, all changes a server proposes must be independent of the ones proposed by other servers because it only gets to observe the other proposals after broadcasting its own. To spread out network load, the current implementation actually sends encrypted requests to other servers before having received hashes from them and reveals the encryption key to reveal the requests. The final protocol is displayed in Figure 2.

5 LOOKUPS

Simplistically, looking up a profile could be implemented by having the client download the entire directory from each server and consider it correct if all copies are equal. This is impractical if there are millions of users. As an improvement on this, the client could

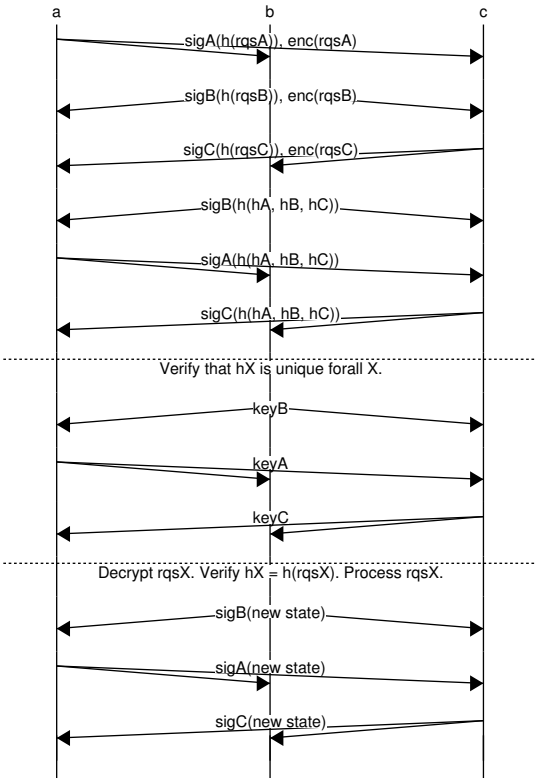

Figure 2: `dename` consensus protocol

instead download the hash of the directory from all servers and the whole directory from one server. If the hashes the servers reported are all equal to the hash of the downloaded directory, the directory must be correct. This scheme is slightly better, but still insufficient.

We need a mechanism to prove that a single name-profile pair is a part of a larger directory with the given hash without downloading the whole directory. Assume that the directory is implemented as a binary prefix tree with profiles in the leaves. Now, every node in the tree is augmented with a hash of its children as shown in Figure 3. If the hash function is collision resistant, each node uniquely determines the state of all names (and the respective profiles) that start with the prefix this node corresponds to. The root hash summarizes the whole directory.

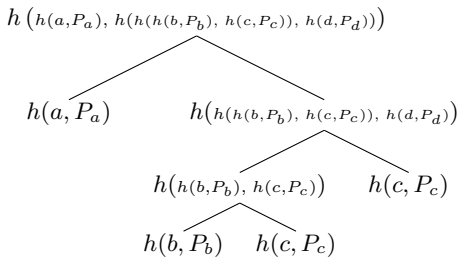

Figure 3: Merkle-hashed prefix tree of the following mapping: $\{a{:}P_a,\ b{:}P_b\ \ c{:}P_c,\ d{:}P_d\}$ where $h(a) = \texttt{0}_b...$, $h(b) = \texttt{100}_b...$, $h(c) = \texttt{101}_b...$ and $h(d) = 11_b...$

To prove that a name-profile pair is a part of a directory with a known root hash, a server supplies the client with the list of hashes stored in the siblings of all the nodes along the path from the leaf to the root – the *Merkle hash path*. To verify the proof, the client first hashes together the name and the profile. This hash is then hashed together with the server-provided hash stored in the sibling of the leaf, resulting in the hash of their parent. This is repeated recursively, hashing in the rest of the siblings all the way up to the root. If the resulting root hash matches, the client knows the name indeed maps to the given profile in the directory with the known root hash. As all servers vouched for the whole directory and we are assuming that at least one of them is honest, the profile must have been registered adhering to the requirements of this system.

As an optimization, the servers can sign the root hash after each round and send the signature to all other servers. This way, a client only has to talk to one server to do a lookup but can still be assured that all servers agree about the result after verifying the signatures. The current implementation also uses the hash of a name instead of the name itself in the prefix tree. This serves to keep the tree balanced and simplify the implementation. As we assume hash collisions do not happen, it does not change any other properties of the system.

5.1 Name absence proofs

If a requested name is not in the tree, the server can prove its absence by returning the name-identity pairs right before and right after the missing name, as defined by the lexicographical order of the hashes of the names, along with the associated Merkle hash paths. The client has to verify that both hash paths result give the correct root hash and that there are indeed no nodes in between: up to their common ancestor, the former only has left siblings and the latter only has right siblings.

6 FRESHNESS

The protocol as described guarantees that if a client looks up a profile, the mapping between a name and this profile must have been approved by all servers at some point in time. However, nothing so far prevents this mapping from being superseded by a later change to the same profile. In this section we describe two mechanisms for ensuring that the lookup result is *fresh* (not superseded). The more efficient one requires the client and the servers to have a reasonably accurate clock; without that there is an option to get a confirmation of freshness from each server individually.

Each server will regularly (every Δt seconds) sign a *freshness assertion* with the contents "As of time t, the most recent root hash is H". The most recent assertion from each server will be distributed together with the root hash. Before accepting a root hash as valid, the client will verify the signatures on the freshness assertions and check that the timestamps are within $\Delta t + \lambda$ of its current time. This accounts for the client's clock being at most λ ahead and it taking up to Δt to get the change timestamped by a server.

Requiring that all timestamps be up to date causes an availability problem: if any one of the servers is down, all lookups will fail. However, this requirement can be easily relaxed: for any $f \geq 0$ of the client's choice, the client can check that no more than f of the timestamps are outside the allowed range and thus continue operating even when f servers are down.

Note that unlike in Spanner[1], the time uncertainty can be quite large (on the order of minutes) even in a correctly operating system. In particular, the client will accept any mapping bearing a timestamp less than $\Delta t + \lambda$ before its own observed time, but its clock may also be at most λ behind of the server time, in which case it may accept a mapping that was timestamped 2λ ago. All changes made more than $\Delta t + 2\lambda$ ago must be reflected in the mapping. However, we do not see it as a problem because we expect security-critical profile changes to be rare and thus consider waiting for them acceptable. If strict security guarantees are not required and a server can be trusted to return the latest mapping – which is at most Δt old – we can

assume that all changes made at least Δt (rather than $\Delta t + 2\lambda$) ago are observed.

If a reasonably accurate clock source is not available, a client can still look up the current profile for a username by contacting a set of servers, at least one of which can be considered honest, and requiring a unanimous answer.

7 VERIFIERS

We view having a fixed set of servers as a necessary evil: it is inherently a central point of compromise, but the only alternative we know is having the evolution of the directory state determined by entities that score highest by some arbitrary metric, such as the hashing power they control, as in NameCoin. To mitigate this weakness, we provide an additional accountability mechanism: everyone can observe how the central servers change the state of the directory, detect deviations from the rules and, in case of invalid changes being applied, have proof of wrongdoing on the servers' part. We describe a *verifier* design that is significantly simpler (and therefore more likely to be implemented correctly) than the servers themselves. We also show how to utilize the Merkle tree structure already used for lookups to audit the changes made during an interval of time without ever having to download the whole directory.

7.1 The simple offline verifier

The purpose of the simple verifier design is to check that the core servers have been enforcing the rules of changing the directory. The design we describe here does not aim to provide optimal throughput or responsiveness: instead, we focus on keeping the implementation as simple as possible with the hope that it can therefore be widely audited and gain public confidence.

The program takes as input a range of rounds starting with the very first one (in the beginning of which the directory was empty) and for each round the ordered sequence of changes applied by the core servers. It processes the change requests in order, validating each one against the current state of the directory and then updating the directory to reflect this change. At the end of each round, it prints out the current hash of the Merkle tree. To verify that a sequence of observed root hashes corresponds to a consistent history of valid changes to the directory, one would download a copy of the purported changes applied to the directory from any one server and then use the simple verifier to compute the root hashes for all past rounds. If all observed root hashes are present in the output of the simple verifier in the right order, the server must have adhered to the rules of the directory.

We implemented this verifier using 40 lines of readable Python code and 20 lines of `protobuf`[4] format specification. No custom libraries were used; an in-memory implementation of the Merkle tree is included in these 40 lines.

7.2 Incremental verification

The verification system described in the previous paragraph may be simple, but it will become more and more costly to use as the total number of handled requests increases. We wish to provide a mechanism through which independent parties can participate in the verification of new changes made to the directory without having to pay the up-front cost of downloading all past changes. Naively omitting the old changes from the inputs of the simple verifier would not yield a solution: it would have no way of determining whether a name has been already registered or not. Instead, the core servers will supply the verifiers with Merkle tree proofs about the relevant directory state in addition to the requested changes. Specifically, each request to transfer a name will be annotated with the old profile, its Merkle path (or an absence proof if there is none), and all siblings used to calculate the hashes for the new Merkle path. The verifier will then use the lookup procedure to verify the old mapping and calculate the new root hash using the server-provided values instead of storing a local copy of the whole tree.

7.3 Coherent caching

A continuously running incremental verifier will observe all changes to the directory. Therefore, if the verifier ever learns a name and the corresponding profile, it will also receive all future updates to that name. Servicing a lookup for a name requires having the branch of the Merkle tree that corresponds to that name, not the whole tree, so a verifier can serve the contents of its possibly incomplete but up-to-date directory. Furthermore, when presented with a lookup request for a name that it does not have a profile for, it can just look up the name from a server that has a more complete version of the directory. A server operating this way is effectively a cache and can be useful to reduce lookup latency in a local network and reduce the load on the core servers. Unlike with DNS and Namecoin, a client using a cache achieves the same security guarantees as a client that interacts with any one core server.

8 IMPLEMENTATION

We implemented a prototype `dename` server and the client libraries in less than 4000 lines of `go` using `postgresql` for storage. The current implementation is a compromise between performance and understandability. For example, independent tasks are done in parallel and in-process state is kept to eliminate redundant database accesses and server signature verifications, but client signatures are verified twice in some scenarios, batch signature verification is not used at all, and some invariants are enforced using expensive database byte array indices even though doing it manually is possible and has shown better performance. Our implementation also detects and reports various kinds of deviations from the specified protocol by other servers, even if ignoring them would be completely harmless – this is intended to assist with the debugging and validation of potential alternative implementations.

8.1 The consensus protocol

The protocol we use to maintain verifiable consensus in a group of peers, some of which may be malicious, is not specific to `dename` and can potentially be of interest for other projects. We preserve this separation in the implementation: roughly a quarter of the codebase is made up of a reusable consensus library. The library waits for the application to submit operations to be handled and calls an application-specified state transition function with the inputs chosen for a single round whenever one is processed. Similarly, network communication is implemented by the application and exposed to the library using a simple `send(peer, data)` interface. However, persistence is currently handled by the consensus library itself because the crash recovery procedure involves fairly complicated queries to the consensus-specific state.

We sought to preserve the semantic separation between rounds in the implementation by making each round managed by its own thread and control structures, but in order to simplify the crash recovery procedure, we added an explicit dependency between adjacent rounds. If a server has not seen all other servers' signatures for the shared state at the end of round i, it is not allowed to push (encrypted) requests to rounds after $i + 1$. Therefore, if a server has published its final signature for round $i - 1$ but not i, it must have finished processing rounds $\leq i - 2$ and does not need to do anything for rounds $\geq i + 2$: there can be at most 3 rounds in progress at the same time. The code also makes use of similar constraints implied by this rule and the explicit requirements of the protocol to bound

the window during which each type of message can arrive from another correct server during one round.

When a server crashes, it loses its in-memory state. To have a complete overview of the exact behavior of other servers, our implementation stores all received messages on disk. This requirement could be loosened in a performance-oriented implementation, but it is absolutely critical to synchronize the following pieces of information to disk before acting upon them:

- The symmetric key used for pushed requests
- Whether a commitment has been made
- Any requests pushed or committed to
- The acknowledged commitments' signatures

Currently, the in-memory data structures are reconstructed after a crash by replaying the stored messages and using the saved encryption key instead of generating a new one. This approach is very robust and allows for relatively straightforward code, but it depends on having a full log of all messages received during the last 3 rounds. Synchronously writing these messages to disk is a performance bottleneck of the current implementation.

8.2 Cryptography

No exotic cryptographic primitives are required for the operation of `dename`, but because the choice of specific algorithms dictates performance and log size, will describe our choices and the reasoning behind them. For all algorithms, we required a security level of 128 bits and existing adoption in real-world systems.

- `ed25519` for digital signatures. Fast signature verification and small signature and public key size are essential for the performance of `dename`. Unlike other common digital signature schemes, `ed25519` supports even faster batch verification.
- `sha256` for collision-resistant hashing and entropy extraction. Widely used, fast enough.
- `salsa20poly1305` encryption for concealing messages from servers during the commitment phase of a round. Any authenticated encryption scheme suffices. Chosen for simplicity.
- `salsa20` keystream for pseudo-random number generation to break ties between requested changes. Chosen for simplicity.

9 EVALUATION

To see whether `dename` is suitable for large-scale adoption, we evaluated the two aspects of `dename` which are the most different from the present standards. First, it is important to check that the name assignment policy is not overly limiting and can be used in real-world applications. To show that it is practical to replace manual public key distribution with `dename`, we integrated a `dename` client with the Pond asynchronous messaging system, OpenSSH, and OpenPGP and compared the resulting user experience to the original application. Second, it is important that the servers are as independent as possible, so a geographically distributed `dename` setup must be able to handle we reasonable throughput of change requests. As profile lookups can be performed against caches in addition to the core servers, lookup performance is not critical. We also expect it to be much faster – a name's profile and the Merkle tree proof can be assembled during a single B-tree traversal.

9.1 Applicability

A usability evaluation of PGP[14] pointed out the need for a conceptual model of security simpler and smaller than manual handling of public keys as used in PGP. Pond[7] and OTR do not require users to learn fundamentally new ways of reasoning about security, but they do not scale as well to broader use: while proficient OpenPGP users can use the web of trust to verify each other's public keys non-interactively, Pond and OTR require each pair of users to communicate with each other through some trusted channel. Furthermore, while substantially simpler that PGP's, the model used by Pond and OTR is still alien to most users – we are not aware of any widely adopted programs that require users to establish shared secrets. Using `dename`, we can get the best of both worlds: assuming the user trusts that one of the `dename` servers will treat them honestly, the only piece of information a user needs to give to the software about the user they wish to communicate with is the recipient's hand-picked username. All security-specific details can be handled behind the scenes. This model is even simpler than Pond's and OTR's, and is likely to be already familiar to a large fraction of users, for example from email or Twitter.

Modifying Pond to work with `dename` required changing 50 lines of logic code and 200 lines of user interface declarations. The two `ssh` wrapper scripts are 2 lines each and the `gpg` wrapper is 15 lines. Pond requires each pair of users to establish a shared secret before they can use Pond to communicate with each other. The Pond User Guide gives a detailed explanation of several acceptable ways that may be used to establish a shared secret, but despite the presence of instructions, using Pond requires effort; it is designed for users with a strong commitment to privacy. We believe that exchanging public keys between contacts is the limiting factor of Pond's usability. Our variant of Pond includes a user interface for associating a Pond account with a `dename` profile and adding contacts using their `dename` names instead of shared secrets, thus enabling a user experience similar to email.

We can utilize `dename` to improve the usability of `ssh` in two separate ways. As with Pond, we can use `dename` to look up users' public keys. As `ssh` reads the keys with which a user is allowed to log in from a file, we can simply get the newly authorized user's public key from `dename` and append it to that file. A less obvious but arguably more useful enhancement is the verification of `ssh` *host* keys against the maintainer's `dename` profile. When connecting to a machine for the first time, `ssh` usually presents the user with the server's public key hash and asks them to check its authenticity. Instead, if we know the server maintainer's `dename` name, we can preemptively look up the corresponding profile and get the host key from there without having to prompt the user at all. As in our experience many users tend to neglect the `ssh` host key validation step, this modification will not only increase convenience but also improve security.

OpenPGP is notorious for being hard to use, largely due to the complexity of manual public key distribution[14][12][3]. We wrote a wrapper script for the `gpg` implementation of OpenPGP that uses `dename` to look up the public keys and then lets `gpg` continue as if the user had acquired and verified the key manually. In particular, this composition provides a very simple interface for verifying digitally signed messages and encrypting messages and files: there is no per-contact setup overhead and the user only has to specify the operation they wish to perform and the `dename` username of the other party.

9.2 Performance

We ran our `dename` prototype on 7 AWS `i2.2xlarge` instances in 7 different datacenters on 4 continents. The servers achieved 157 profile writes per second. This is not high compared to non-cryptographic databases, but since it equals 400 million profile changes per month, it is unlikely to become a limiting factor in any realistic deployment scenario. As the number of servers increases,

the performance does not degrade significantly: the throughput bottleneck in the profile modification process is synchronizing the individual changes to each server's disk and not synchronizing the changes between the servers.

10 Limitations and Future Work

`dename` does not handle the revocation of keys that have already been downloaded. While a user can make their name point to a different key, they lack a guarantee that other users who have already downloaded the old key will stop using it. As the appropriate moments for revocation-checking are application-dependent, this needs to be tackled separately. Interacting with usernames can also be surprisingly tricky in an adversarial environment: the `dename` infrastructure does not prevent typosquatting and homograph attacks; that is another responsibility of the application writer.

We have shown that it is practical to maintain a globally replicated identical copied of the user directory. For a large-scale deployment, better-optimized implementations are probably desirable. In case of a single machine being unable to handle the churn of updates, the directory can be easily sharded by the hash of the name. Similarly, a single `dename` server can be implemented using a replicated state machine to improve availability.

If it were possible to configure `dename` servers to make progress even if some number of them are not available, a larger set of servers could be admitted. While not requiring the approval of all servers would obviously weaken the security guarantee, we believe that this loss is offset by the security gained from having a more diverse set of parties operating the servers. We are not aware of any state machine replication protocol that could be used to implement this.

11 Conclusion

We present `dename`, a public key distribution mechanism that provides a very strong security guarantee: as long as at least one of the core servers is secure and honest, a client that successfully looks up a profile corresponding to a name will receive a fresh, correct response. Thus, `dename` can replace other key distribution mechanisms in a variety of security-critical applications, improving usability and decreasing the potential for user errors.

References

[1] James C. Corbett et al. "Spanner: Google's Globally-Distributed Database" (2012).

[2] Carl Ellison and Bruce Schneier. "Ten Risks of PKI: What You're not Being Told about Public Key" (2004).

[3] Dan Goodin. "Guardian reporter delayed e-mailing NSA source because crypto is a pain". *Ars Technica* (June 11, 2013).

[4] Google. "Protocol Buffers: Developer Guide" (Apr. 2, 2012).

[5] P. Hoffman and J. Schlyter. "RFC 6698: The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA" (Apr. 16, 2013).

[6] Mozilla Johnathan Nightingale. "Comodo Certificate Issue – Follow Up" (Mar. 25, 2011).

[7] Adam Langley. "Pond" (2013).

[8] Ben Laurie and Adam Langley. "Certificate Authority Transparency and Auditability" (Nov. 22, 2011).

[9] John Maheswaran, David Isaac Wolinsky, and Bryan Ford. "Crypto-Book: An Architecture for Privacy Preserving Online Identities". *Hotnets* (Nov. 21, 2013).

[10] Moxie Marlinspike. "Convergence" (2011).

[11] Bruce Schneier. "VeriSign Hacked, Successfully and Repeatedly, in 2010" (Feb. 3, 2012).

[12] Jan Sousedek. "Why Johnny Can't Encrypt: A Usability Study of PGP". *Technische Universität Berlin Internet Security Seminar* (2008).

[13] Dan Wendlandt. "Perspectives Project" (2014).

[14] Alma Whitten and J. D. Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0". *Proceedings of the 8th USENIX Security Symposium* (1999).

[15] Tim Worstall. "Bitcoin Mining Uses $15 Million's Worth Of Electricity Every Day". *Forbes* (2013-12-03).