KeyNet: a Secure, Byzantine Fault Tolerant Distributed Public Key Directory

Colin Man
colinman@stanford.edu

Jean-Luc Watson
jlwatson@cs.stanford.edu

Sydney Li
sydli@stanford.edu

***Abstract*—KeyNet is a distributed directory for secure public key storage, modification, and lookup. Local consistency, and therefore trust, is maintained at scale through the use of byzantine fault tolerance. We discuss an emerging federated byzantine consensus protocol, Stellar, that can enable open participation without sacrificing safety, and present KeyNet's design in that context. Our initial implementation includes a PGP key store that reaches consensus through the Practical Byzantine Fault Tolerance protocol and can successfully be used to send encrypted messages over a modern email client.**

I. Introduction

Public interest in communication privacy has grown significantly as a reaction to extensive industry data collection practices, government surveillance, and high-profile cybersecurity breaches. As a result, privacy efforts have focused enabling secure data exchange through the use of end-to-end encryption. Companies such as WhatsApp and Signal have implemented instant messaging clients on that principle [1], [2]. Yet no matter the sophistication of the eventual communication channel, asymmetric encryption requires the exchange of cryptographic public keys upon initialization. While the public nature of the key protects the security of the eventually transmitted data, their direct exchange by the clients is problematic because an adversary may capture and modify the key en route. Fundamentally, we lack the method to securely link an identity with the authorized use of a key on a network.

Current approaches to key exchange security have added mechanisms to ensure that, with high confidence, keys sent from the expected source and not maliciously modified are received by a client. The most common Public Key Infrastructure (PKI) is comprised of a core hierarchy of recognized Certificate Authorities (CAs) who can sign certificates attesting to the validity of a given public key. Commonly, some source of off-line information is used to confirm a correspondence between the key and the purported identity [3]. However, CAs present an additional set of challenges in that they form a centralized root of trust and a single point of failure. Since an application must continually trust a CA's public key to accept certificates signed by it, any misbehaving CA can propagate malicious public keys [4]. Worse, such a CA may revoke a key with no prior notice, censoring encrypted communication or replacing the key while continuing to appear legitimate.

For those unwilling to cede control over their public keys, the Web of Trust has been constructed alongside the OpenPGP standard as a more decentralized PKI model. Trust is distributed among each key, and other users can attest to a key's identity by signing it. As more trusted users sign a new key, it is itself increasingly trusted. While clearly more fault-tolerant than a CA-based PKI, web of trust schemes fall prey to other similar issues. Key verification must now take place entirely off line, most likely requiring in-person communication at a high cost. The lack of a single authority places the onus for validating keys onto the clients when signing or requesting a key and at any time, they may choose to unilaterally weaken verification or abandon it entirely for ease-of-use.

On the whole, current PKIs suffer from not effectively enforcing the security and consistency of their key directories, which allows unauthorized updates and unsynchronized state to undermine the legitimacy of the public keys they store. With that in mind, this paper presents *KeyNet*, a distributed public key store that achieves security through distributed consensus, even in the face of byzantine failures. We allow for open membership on an Internet scale, public auditing, and the easy detection of malicious behavior with incontrovertible proof. KeyNet relies on recognized roots of trust during key creation only; after a key is signed, authority for all modifications reverts solely to the key owner. Finally, the consistent directory state guaranteed under consensus allows KeyNet to enforce a unique identity-key mapping and serve local key requests directly from

the trusted state.

A. Byzantine Fault Tolerance (BFT)

Since we rely on our local key store to provide the information required to process requests, our project must maintain a consistent state relative to the other nodes in the cluster. This is critical when one or more nodes become compromised and begin to submit faulty key creation or update operations. Therefore, KeyNet clusters must not only survive system crashes or power loss but byzantine failure cases. Practical Byzantine Fault Tolerance (PBFT) is a well-known consensus algorithm that achieves these safety and fault tolerance guarantees [5]. In summary, the protocol serializes updates at a round-robin primary node before committing; each node communicates with cryptographically signed messages to detect byzantine failures, and commits an operation once a quorum of nodes agree to apply it. One drawback to PBFT is its requirement for static membership, since all nodes must be known before hand to guarantee lasting consensus. However, we found PBFT to be useful in developing our proof of concept for KeyNet, discussed in Section V.

B. Federated Byzantine Agreement

An additional point of concern is that PBFT, given its rigid membership, is not designed to scale, which makes the protocol ill-suited for an Internet level system. In order to enable open membership for any node willing to join the network, we can run KeyNet on a *Federated* Byzantine Agreement System (FBAS) while safely maintaining consistent state and byzantine fault tolerance [6]. Each node is allowed to decide on its own dynamic *quorum slices* representing other cluster participants it trusts. If every well-behaved node has at least one quorum slice that transitively intersects with all other well-behaved participants, then the KeyNet cluster will remain consistent. This property allows nodes to reach consensus even when they have no direct knowledge of all other nodes.

II. Design Goals

KeyNet's potential use as a backbone for large-scale public key infrastructure entails consideration of both security and system architecture design goals.

A. Security

Non-equivocation No KeyNet node should externalize conflicting key values for any entity, nor should the key directory state ever diverge. Any failed set of key server nodes exhibiting arbitrary or malicious behavior should not affect the remaining nodes' ability to safely replicate the key directory. Note that such a (byzantine fault tolerant) guarantee only holds under protocol-specific conditions: PBFT can maintain safety when up to f failures occur in a cluster of $3f + 1$ nodes, while an FBAS is tolerant to any number of faults under which a quorum intersection of well-behaved nodes exists. The point of externalization is also subject to protocol-specific conditions: in the case of PBFT, the client must receive $f + 1$ of the same response for the value to be externalized.

Continuity Every directory update must be verified by each node as a deterministic function of the current key server state. For example, every key change must be correctly signed by its owner, verified by the current public key. Newly created keys must be signed first by the domain authority for the proposed identity, then by the owner. As a result, nodes may maintain trust in the integrity of each key in the directory.

Auditability The system must allow any entity to monitor updates as a full member of the cluster. In particular, should any KeyNet node attempt to submit a modification or deletion of a public key binding that it does not own, every node will receive indisputable proof of the malicious behavior: each operation that attempts to reach consensus is cryptographically signed by its creator and is authorized by the very key being modified.

B. Architecture

Open Membership Any key directory cluster with closed membership would *implicitly* limit scalability and flexibility in terms of defining trust on the client side. Appropriately-verified entities should be allowed to replicate the key directory, propose modifications or updates, and participate directly in consensus. Further, the system should scale as a decentralized network, with no *explicit* controlling root of trust.

Compatibility KeyNet should be compatible with, and develop off of, existing public key infrastructure. Rather than begin with a clean directory, the system should be able to initially accept strongly authenticated domain administrators or certificate authorities. Additionally, KeyNet nodes must support common PKI standards, such as PGP or X.509 certificates.

III. System Design

Many applications implemented over consensus protocols become inevitably tied to each other. As an example,

when first creating the KeyNet codebase, we cloned etcd-raft [7], a popular key-value store implemented over a full-featured version of the RAFT consensus protocol [8]. We observed a large amount of coupling (both in message types and bridging function calls); given that we would expect to run KeyNet on a vastly different consensus algorithm (FBAS) in practice than what was used during this quarter's development (PBFT), we viewed maintaining a *separation of concerns* between different components as an important requirement during development.

A. Interface

Our primary client-side interface is a key-value store, in which an arbitrary identity token maps to a matching public key. Commonly, the identity token takes the form of an email address, and in Section V we store a committed PGP key for each individual email address. In addition, we allow for four operations on a KeyNet directory: *Create*, *Lookup*, *Update*, and *Recover*. Our implementation currently implements every operation except for *Recover*.

Create Submits a key creation request to the KeyNet cluster, which must consist of an identity-public key pair. The request should be signed by the authority controlling the relevant email domain: the authority has the ability to bootstrap its own users. While completing the consensus protocol, each node should first verify that there are no current public keys already assigned to the same identity. If a duplicate exists, the operation can be aborted and restarted as an *Update*. A KeyNet node should then verify that the creation was authorized by the controlling party for the email domain. If verification and the remainder of the consensus completes successfully, the key is added to the directory and available for reads. Finally, it is important to note that during *Create* operations, we must leverage existing PKI at each KeyNet node in order to acquire the correct public keys corresponding to domain authorities not present in the key directory (more discussion can be found in subsection III.E).

Lookup Retrieves the matching public key from the director for a given identity token (i.e. email address). Since the BFT consensus algorithms ensure that every node sees the same, eventually consistent key directory state, reads can be performed entirely from the local copy by a KeyNet node. Remote client behavior to ensure that it received the correct public key on *Lookup*s is discussed further in Section V.

Update Changes the public key bound to a specific identity. Such requests **must** be signed by the owner's old private key and verified by each node with the soon-to-be replaced public key. Critically, this does not give the domain authority the ability to change a public key binding it previously signed off on, as a CA would in a classic PKI scheme. While this also prevents revoking all keys for a particular domain in the case of a compromised domain authority, even a malicious node could do no worse than create spurious additional keys for emails that are not already in the directory.

Recover A major downside to the semantics of the *Update* call is that if a node or client were to lose its private key, they would be unable to update their public key in the directory and in the case of a node, unable to communicate with the remainder of the KeyNet cluster. The *Recover* operation allows for preemptive preparation in the event of key loss: a KeyNet node or client could encrypt their private key before backing it up in a secure location and spreading the decryption shares among a set of trusted KeyNet nodes. In the event of a key loss, the node can contact each node out-of-band, prove its identity, and receive enough decryption shares to recover its private key.

B. KeyNet Node

The KeyNet node is the core of our distributed key directory and implements the application-specific logic. It orchestrates three components:

1) *Client Handler*: handles incoming client requests to the interface described above, either locally if a full node in an FBAS, or over an HTTP(S) handler in a setup like PBFT with remote clients.
2) *Key Store*: the local, raw key-value storage for KeyNet's public key bindings.
3) *Consensus Node*: the implementation of the directory's distributed consensus protocol, which provides the KeyNet node with committed operations to apply.

Each component is independent and can be replaced without harming the raw functionality of the system. However, replacing the FBAS or PBFT implementation with RAFT, for example, would create critical flaws in the system when experiencing byzantine failure.

C. Consensus Node

The consensus node separates the high-level functionality of the key directory from the implementation of the BFT protocol using a simple *Propose*/*Committed* interface on opaque string operations. This allows the

consensus client to embed information such as operation type, request signature, etc. without bleeding types into the consensus implementation or vice versa.

D. Client

The KeyNet client performs key lookup, creation, and update. In the PBFT protocol, the client must receive $f + 1$ of the same response from the cluster in order to confirm the result of any operation. Key creation must be signed by a domain authority and key updates must be signed by the previous key; if these conditions are not met, the update is rejected by all non-faulty nodes. The leader of the PBFT cluster must assign a sequence number and submit the request; all non-leader nodes forward the client request to the leader. In order to deduplicate requests, the client sends a unique timestamp with each request submitted.

E. Bootstrapping Authority

In certain cases, it is not possible to verify key creation based solely on the contents of KeyNet. For example, a *Create* request with a public key for `paul@example.com` when the domain authority for `example.com` is not registered with the system is difficult for the KeyNet nodes to verify. Thus, we fall back onto existing PKI like a CA certificate chain to bootstrap authorities. Theoretically, this means that we may always be dependent on an exterior root of trust during the *Create* phase. We believe this is not a fatal scenario: even in the case of a compromised CA chain, modifications to the key directory are limited to adding new keys, as previously-issued keys cannot be modified by the domain authority; thus, only emails that have not previously inserted into KeyNet are vulnerable to a compromised domain authority. The amount of participation in the KeyNet network trades off directly with the reliance on previously-existing PKI for key verification. Finally, since key updates are heavily restricted, the basis for trust on all future updates to a public key mapping is the distributed directory itself.

IV. Stellar Consensus Protocol (SCP)

To make use of an FBAS for consensus, a protocol implementation is necessary, one of which is the Stellar Consensus Protocol [6]. The resulting architecture will look slightly different from the one presented in Section V – in Stellar, every client node can choose to participate as a full member of the consensus algorithm (the exact position in the network depends on the chosen quorum slices) and keep a copy of the directory state. Every instance of the Stellar protocol runs in two phases: a federated nomination phase where nodes vote and converge on a shared set of operations to apply, and a balloting phase where the nodes are voting on which operation set to externalize. The nomination protocol may continue to run during the balloting phase and update the operations being balloted. Invalid operations, such as attempting to update a key without holding the appropriate private key, can be detected using the local directory state and removed prior to any nomination occurring. Once a quorum has voted to externalize a set of operations, they can immediately be applied to the key directory and made visible to the user. Because operations are applied in small groups for each slot in the log during successive iterations of nomination and balloting protocols, it is easy for Stellar nodes to maintain a transaction history that can be used when malicious behavior or equivocation is detected to evaluate the root cause.

V. Implementation

Over the course of the quarter, we implemented a smaller scale version of the full KeyNet system in approximately 3000 lines of Go and 300 lines of Javascript. Among others, this included an extensive PBFT implementation, a key directory store, and a modified Chrome extension for PGP email encryption integration. We hosted a cluster of up to 10 PBFT nodes, as well as a mock authority server, on separate Google Cloud instances to better examine performance across network links.

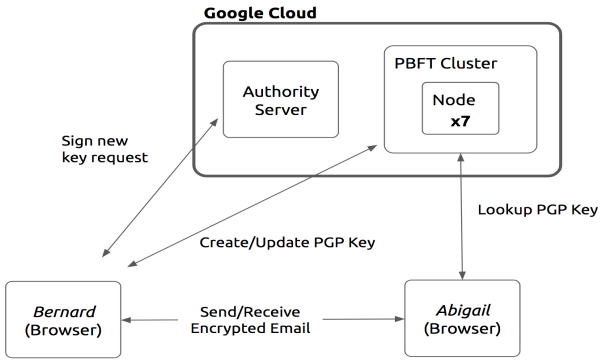

Fig. 1. Our KeyNet implementation.

A. Practical Byzantine Fault Tolerant Protocol

The implementation of the underlying protocol follows the design from the original PBFT paper closely, with a couple of practical additions inspired by Raft that do

not compromise the safety and consistency guarantees of PBFT's design.

During regular operation, the cluster follows a standard three-phase commit protocol. To perform an update, the client broadcasts a signed request to the entire cluster. The leader, or primary, of the cluster determines the ordering of all client requests and begins the protocol. If a node notices that the leader does not begin the three-phase commit within a given timeout, it initiates a "view-change", which round-robins the leader. A new leader is deterministically chosen and the nodes consolidate their state at the beginning of the new view. After a certain number of commits, a checkpoint is initiated, during which all the nodes snapshot their state.

We implemented two additions to protect against regular faults: 1) heartbeats and 2) recovery. (1) If the primary simply goes down, the network does not discover it until the next client request. To pro-actively catch these faults, we have the primary send periodic empty pre-prepare messages, to the cluster. If a node misses several heartbeats, it initiates a view change. (2) The original PBFT paper is vague in how it conducts node recovery, punting it until the next view change. Our nodes piggyback state information onto heartbeats and heartbeat acks. Non-primary nodes piggyback their most recent commit timestamp onto the ack. If they are behind, the primary, on the next heartbeat, responds with information about the next closest sequence number, either the last checkpoint (and a proof of $2f+1$ uniquely signed checkpoint messages), or a committed entry (and a proof of $2f+1$ uniquely signed commit messages). If the proof is correct, then the node applies it to its log.

Finally, to perform a lookup, the client broadcasts their query to the cluster and waits for $f+1$ matching replies.

B. Mock Authority Server

Since KeyNet relies on existing PKI to bootstrap its root of trust, we have a "mock" authority with valid certificates. The authority signs off on all public key bindings for names under their domain.

C. Encrypted Email

Our current implementation of the KeyNet client is a modification of the open source Javascript extension, Mailvelope. Since HTTP requests must be initiated by the client, KeyNet PBFT requests from the extension must be broadcast to every node.

1) *Key Management*: The KeyNet browser extension provides the ability to manage multiple keys tied to different email addresses. Each email is currently limited to one key, but KeyNet can be extended to support multiple keys per email in the future. The keys are password protected, though unlocked keys can be kept in memory for a short period for convenience.
2) *Key Creation*: A key creation request is constructed with the public key, sent to the domain authority for signing, then to KeyNet for insertion.
3) *Key Update*: Key update follows the same process as creation with a signature from the previous key instead of from the domain authority.
4) *Key Lookup and Email Encryption*: The PGP public keys of the email recipient are looked up by email address on KeyNet and the email encrypted for those keys.
5) *Email Decryption*: A received email can be decrypted by the extension if the private key it was encrypted for is present in the browser.

VI. Evaluation

A. Performance

We performed several experiments to show the viability of the KeyNet system. KeyNet was deployed onto 10 Google Cloud instances, 4 in `us-west`, 3 in `us-east`, and 3 in `us-central` zones. The performance of our system is reasonable for an unoptimized, vanilla PBFT implementation (Fig 2), and performs at-par with existing key server pools. This is sufficient for our use case since key databases are very read-heavy; most receive only a couple thousand updates per day.

Cluster size	Write (ms)	Read (ms)
4	448.51	46.42
7	450.84	47.34
10	625.06	47.21

Fig. 2. Average read/write request execution time over 10000 requests with 10 clients, on clusters of $3f+1$ for $f \in \{1,2,3\}$. f + 1 of the nodes were located in zone us-west, and f in us-central and us-east. The cluster also checkpoints every 100 updates.

The bottleneck for write request execution is correlated with the diameter of the largest 2f+1 cluster, and similarly, the largest f+1 cluster for read requests. After increasing the number of nodes located in the same zone to 2f+1 from f+1, write request execution averaged 274.76 ms.

The local experiment (Fig 3) shows the performance of computing and verifying cryptographic signatures is not bottlenecking throughput at this scale. Finally, a brief experiment over intermittent failures (which exceed the heartbeat timeout and triggers a view change) shows that KeyNet remains performant.

Mode	Write (ms)	Read (ms)
Regular operation	49.12	0.33
Intermittent failures	63.92	0.34

Fig. 3. Average read/write request execution time over 10000 requests with 10 clients, performed on local clusters. In the second case, every 2 seconds, f machines were brought down (and brought back up on the subsequent timeout). The cluster also checkpoints every 100 updates.

B. Design

Deploying KeyNet on an FBAS like Stellar allows the system to attain our design goals. From a security perspective, the safety and consistency guarantees of the protocol fundamentally prevent well-behaving nodes from externalizing more than one public key for a given identity; equivocation can be easily detected when clients compare data received from different nodes. We've illustrated how distributed directory operations can succeed or abort based entirely on local directory contents and a signature from a domain authority, which prevents malicious key changes without owner consent. The nature of an FBAS allows for an arbitrary number of participants - since all nodes would keep a full copy of the state under SCP, the ability for 3rd parties to monitor changes in the directory is unhindered. And as every update must be signed before replication, any suspicious activity can be automatically flagged and traced to the source. We can then use each node's dynamic quorum slices to exclude malicious nodes from consensus. Finally, we've shown how KeyNet can support the use of existing PKIs and domain authorities to bootstrap trust from the ground up and implemented a PGP key store that can interface with a real-world mail client. For comparison, in our PBFT-based implementation, adherence to the design goals as listed is weaker. We still implement a byzantine fault tolerant protocol, so non-faulty nodes will never equivocate public keys, and the separation of concerns in our system design ensures that directory modifications are handled in the same way as with an FBAS. However, PBFT has a closed membership, which necessarily excludes potential auditing or the addition of KeyNet nodes. Finally, we demonstrated that compatibility with existing PKIs remains achievable regardless of the consensus backing.

VII. Future Work

Given the scope and time frame of this quarter's project, we made a simplifying decision to first implement KeyNet on top of PBFT as a proof of concept. However, as hinted at in the previous section, a glaring next step would be to begin porting our existing code for use with a Stellar implementation as it would allow the network to scale and open membership.

In Section III.A, we discussed our strict key update policy and the resulting need for a key recovery mechanism. Implementing a threshold cryptography scheme that could be used at the last resort, especially when coupled with the trust preferences inherent in Stellar quorum slices, would be quite useful.

VIII. Conclusion

KeyNet is a distributed public key directory that provides strong safety guarantees and allows clients to securely access and trust public key records. Combined with an emerging class of byzantine consensus algorithms, Federated Byzantine Agreement Systems, KeyNet has the ability to retain those guarantees while scaling to a decentralized, internet-wide network with open membership. Trust in our directory can be bootstrapped off of existing infrastructure, with minimal reliance on single points of failure. Our work on KeyNet can be viewed at the following locations:

The main KeyNet node and key directory implementation. `https://github.com/sydneyli/keynet`
Custom PBFT implementation. `https://github.com/sydneyli/keynet/tree/master/src/pbft`
Our fork of the *Mailvelope* Chrome Extension backed by a KeyNet cluster. `https://github.com/colinman/mailvelope-pbft`

References

[1] "Advanced cryptographic ratcheting." [Online]. Available: https://signal.org/blog/advanced-ratcheting/

[2] "Whatsapp faq - end-to-end encryption." [Online]. Available: https://faq.whatsapp.com/en/general/28030015

[3] [Online]. Available: http://www.digicert.com/wp-content/uploads/2017/01/DigiCert_CPS_v401.pdf

[4] "The turktrust ssl certificate fiasco what really happened, and what happens next?" Feb 2013. [Online]. Available: https://nakedsecurity.sophos.com/2013/01/08/the-turktrust-ssl-certificate-fiasco-what-happened-and-what-happens-next/

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance," *Proceedings of the Third Symposium on Operating Systems Designs and Implementation*, 1999.

[6] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus." [Online]. Available: https://www.stellar.org/papers/stellar-consensus-protocol.pdf

[7] "coreos/raftexample." [Online]. Available: https://github.com/coreos/etcd/tree/master/contrib/raftexample

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *USENIX Annual Technical Conference*, 2014.