A Byzantine Fault-tolerant KV Store for Decentralized PKI and Blockchain

Ryuji Ishiguro

September 10, 2018

https://github.com/yahoo/bftkv

Abstract

Most distributed key-value stores are tolerant to only benign failure, which makes it difficult to run the system in an insecure environment (i.e., the Internet) as a single point of failure could compromise the whole system. Such systems not only are vulnerable to malicious attacks, but also need to rely on centralized authorities to protect data integrity. We developed a distributed key-value store that is tolerant to Byzantine faults, keeping it in mind to run the system over the public Internet without any central authorities. While data integrity is secured among distributed nodes, anyone can join / leave the network freely and can see all transactions. The system provides not only a robust key-value store but also authentication, encryption and signing features with a threshold cryptosystem. Integrating encryption and signature schemes with a Byzantine fault-tolerant protocol is much more robust than using separated KMS and PKI with an ordinary distributed key-value store, as it maximizes a benefit of distributed systems which fit very well with threshold cryptography. There is no single point of failure in the system. The system by itself is useful as a secure key-value store, but those properties such as Byzantine fault-tolerance, transparency, distributed global storage and threshold cryptosystem can make the system an ideal building block for blockchain technologies.

1 Introduction

How to reach a consensus with Byzantine type failure is the main problem of blockchain technologies. Bitcoin blockchain uses PoW (Proof of Work) with incentives [1]. The majority of public blockchain (or permission-less blockchain) systems have the same type of consensus mechanism. Another way to establish a consensus is to use BFT (Byzantine Fault-Tolerant) protocols, which is a major mechanism for local/enterprise blockchain (or permissioned system). Some PoS (Proof of Stake) type blockchain technologies use BFT as well to avoid the mining process.

The proposed system is based on the Byzantine quorum system introduced by Malhi and Reiter [2], and adopts a BFT protocol proposed by the same authors [3]. We construct a b-masking quorum system based on the WoT (Web of Trust) graph. We also introduce quorum certificates combined with a threshold authentication scheme to protect data from unauthorized mutation. Unlike a centralized PKI such as X.509, quorum certificates are verified collectively with other quorum members that are chosen with a dynamic graph constructed independently at each node. With the strong authentication scheme backed by a threshold cryptosystem and flexible certificate mechanism, the system allows anyone to join / leave the network freely without any authorization process while it is immune from sybil attacks.

Another key aspect of blockchain technologies is the global distributed ledger. Bitcoin blockchain uses hash chain – transactions in a specific period are all hashed together with the hash value of the previous block. All bitcoin nodes maintain the same view of the hash chain. The proposed system uses a distributed key-value store with the TOFU (Trust on First Use) policy, that is, the first use of a key locks out others from mutating the value associated with the key. Only the user who has written the key-value first will posses the right to update the value. Also it supports WRITE ONCE permission which guarantees that once a value is written it will never be modified in any way. This special permission will be preferable for applications

like the global ledger which must be tamper-proof. The system does not guarantee absolute consistency. Also the order of transactions for different keys does not matter. The system does not even provide eventual consistency among individual nodes. Instead, it has a property such that: $READ(Q_1, x) = READ(Q_2, x), \forall Q_1, Q_2 \in QS$, that is, the consistency is established collectively with a quorum system (QS).

Smart contracts play an important role in some blockchain technologies such as Ethereum and Hyperledger Fabric. As the proposed system is a simple key-value store, it does not provide a platform for smart contracts at the moment. Also, the system itself does not provide any incentive mechanism to discourage nodes to do malicious actions, or to encourage to participate the consensus process. The system has a robust revocation scheme but it does not answer for a question like why would one want to run a node? Although all transactions keep a proof of good or bad actions for each node therefore it will be straightforward to map it to economic incenstives and penalties, we defer fintech discussions to an application layer.

2 Background

This section describes some existing technologies used in the system.

2.1 Quorum Systems

A variety of quorum systems have been used to manage replicated data / storage in distributed systems. We briefly describe the original quorum systems and its extension called Byzantine quorum systems. Later, we construct a byzantine quorum system based on Web of Trust. The system defines two operations, read and write, between a client and a set of servers called a quorum. A quorum system ($QS \subseteq 2^U$) is a subset of the powerset of all servers (U), and it satisfies a property:

$$\forall Q_1, Q_2 \in QS, Q_1 \cap Q_2 \neq \emptyset$$

(intersection property)

With this property, it is guaranteed that the client always retrieves the latest value from at least one server.

Melhi and Reiter [2] extend the property to handle Byzantine failure:

$$\forall Q_1, Q_2 \in QS, \forall B \in BF, Q_1 \cap Q_2 \nsubseteq B$$

where $BF \subseteq 2^U$ and $\cup_{B \in BF} B$ is all Byzantine failure nodes. Especially when $|Q_1 \cap Q_2| \geq 2b + 1$, QS is called a b-masking quorum system. When the client can be dishonet, signed messages are no longer trustworthy therefore we rely on the quorum system to avoid equivocation (aka double spending in blockchain terms).

2.2 Web of Trust

A web of trust is a directed graph $G = (V, E)$, where V is a set of nodes (each of which is a pair of unique ID and public key) and E is a set of trust relationship: when $(v_1, v_2) \in E$, $v1$ *trusts* v_2, i.e., the certificate of $v2$ includes a signature over its public key with the private key of v_1. WoT was introduced by PGP to authenticate certificates of peers without relying on central authorities. We use the same mechanism to authenticate not only end-users' certificates but quorum members' as well.

2.3 Threshold Cryptosystems

Threshold cryptosystems play important roles in this system. We use it not just for fault-tolerance but for improving security.

Shamir's Shared Secret (SSS) [7] is a major tool to construct threshold cryptographic schemes. The system uses SSS for both password authentication and DSA / ECDSA signature schemes. Especially for the latter, the system uses a threshold siganture scheme introduced by Gennaro et al. [4]. For the threshold password authentication, the system uses a simple DH key exchange protocol with a similar setting to [5]. See 3.3 for the details. SSS uses a $t-1$ degree random polynomial on $\mathbb{Z}_q$

$$f(x) = \sum_{i=0}^{t-1} a_i x^i \bmod q$$

Each *share* is $(i, f(i))_{i=1..n}$. To reconstruct the shared secret $f(0)$, calculate lagrange interpolate from t responses out of n

$$\lambda_j = \prod_{l \in \mathcal{T} \setminus \{j\}} i_l/(i_l - i_j) \bmod q$$

then we get

$$f(0) = \sum_{j \in \mathcal{T}} f(j)\lambda_j$$

where $\mathcal{T}$ is a subset of $\{1..n\}$ and $|\mathcal{T}| = t$.

To construct a (t, n) threshold scheme, we follow the quorum threshold, i.e., $n = |Q|, t = n - b$.

For RSA signatures, it may seem straightforward to apply SSS to the RSA signing process such as:

$$S = M^{f(0)} = \prod_{i \in \mathcal{T}} M^{f(i)\lambda_i} \bmod N$$

however, since to calculate λ_i we need the multiplicative inverse in $\varphi(N)$ which must be kept as secret as the private key, we cannot simply appy SSS to RSA signatures. Shoup [6] solved this problem by getting rid of multiplicative inverse all together but it needs a special construction of the RSA parameters, which makes it difficult to apply existing keys to the method. Therefore we use a simple key hierachy to address this issue. See 3.4 for the details.

For DSA and ECDSA, SSS has preferable properties due to its linearity:

$$f(0) + g(0) = \sum_{i \in \mathcal{T}} (f(i) + g(i))\lambda_i$$
$$f(0) \cdot g(0) = \sum_{i \in \mathcal{T}'} (f(i) \cdot g(i))\lambda_i$$

where

$$f(0) = \sum_{i \in \mathcal{T}} f(i)\lambda_i$$
$$g(0) = \sum_{i \in \mathcal{T}} g(i)\lambda_i$$

$\mathcal{T}'$ is a subset of $\{1..n\}$ and $|\mathcal{T}'| = 2t$. These properties make it possible to calculate the DSA / ECDSA signature (r, s) which invovles additions and multiplications of shared variables, such that:

$$r = g^{k^{-1}} \bmod p \bmod q$$
$$s = k(x + mr) \bmod q$$

from partial signatures (r_i, v_i, s_i):

$$r_i = g^{a_i} \bmod p$$
$$v_i = k_i a_i \bmod q$$
$$s_i = k_i(x_i + mr) \bmod q$$

where x_i is the share of the private key for each node. (a_i, k_i) is generated randomly by each node. See [4] for how to construct r from (r_i, v_i).

3 Methodology

This section describes quorum systems based on the WoT graph, which are the main building blocks for our BFT protocols. Protocols are always between a client and a quorum. Every message must be signed by a set of special nodes called *Quorum Cliques*, and the signed messages are stored in another quorum called *KV quorum*. To write a key-value pair the client performs the three step process:

1. Collect timestamps from a quorum and choose the latest one,
2. Request to sign the message to a quorum in the *Quorum Cliques*,
3. Write the signed message to a *KV quorum*.

To read a key-value pair, the client performs the following process:

1. Collect key-value pairs from a *KV quorum*,
2. Choose one that has the latest timestamp.

See the section 4 below for details.

In order to complete the system, authentication and signature schemes based on a threshold cryptosystem are introduced in this section as well.

3.1 Quorum Cliques

We follow the faulty clients (or dishonest writers) scenario described in "Byzantine Quorum Systems" [2, 3], which uses signed messages with b-masking quorum systems to avoid equivocation. [1] In our system, such quorum system is constructed from maximal cliques in the WoT graph.

If the graph G contains some cliques and the starting node s has outdegree edges to the cliques within the L distrance, i.e.,

$$\exists (s, u) \in G.E, s.t., u \in QC \text{ and } dist(s, u) \leq L$$

where QC is quorum cliques obtained by the algorithm GetQC 1, it will construct a quorum to sign messages. Note that the quorum depends on only the graph G. There are no other configuration data or anything. Such graph is constructed from the quorum certificates.

Quorum Certificates

Quorum certificates represent the proof of trustworthy. Each node keeps its own quorum certificate along with the private key. A quorum certifiate consists of:

- a unique ID
- a public key
- a self signed signature over the ID and public key

[1] Aka double-spending in the blockchain world.

- a set of signatures signed by Quorum Cliques

When a node (n_1) is signed by another node (n_2) an edge is added to the WoT graph, i.e., $G.E = G.E \cup \{(n_2, n_1)\}$.

The quorum certificate not only constructs the graph but also gives permissions to clients to mutate the value. Every write request includes the client's quorum certificate along with the self signed signature over the variable, timestamp and value which we denote $\langle x, t, v \rangle$. Each member of QC verifies the signature and the quorum certificate before it sends back the signed message. See algorithms VerifyCollectiveSignatures 3 and CheckQuorumCert 4.

Sybil Attack

When a node is compromised the node can try to make its own cliques with made-up colluding nodes to outnumber the honest nodes. By the algorithm GetQC 1 which is used to verify the quorum certificates and the collective signatures, a node cannot be a member of more than one cliques, which means the compromised node has to sever the trust links to other honest nodes to make links with the colluding nodes, otherwise the clique can no longer be a member of a quorum. If such a faulty clique disagrees with other cliques a consensus will be no longer established until clients exclude the compromised nodes.

3.2 Key-value Store

Once data is signed by quorum cliques it can be sent to other nodes that are not necessarily members of the cliques. Such nodes can form another quorum system and we call it *KV quorum*. The main purpose of this quorum system is to make sure that clients can retrieve the latest key-value. We no longer need the b-masking quorum as all messages are signed by quorum cliques which have already handled the faulty client case. *KV quorum* handles only f benign faulty nodes. The read process writes back the latest message to nodes that keeps old values:

1. The client collects $f + 1$ responses and chooses one which has the latest timestamp.

2. If some servers return an old value or *nil* the client will write back the latest value to those servers.

See 4.1 for the actual protocols. For load balancing, *KV quorum* is typically chosen from $U \setminus QC$.

Each member of *KV quorum* must check equivocation and the permission of mutation (TOFU), when it receives the signed transaction.

Equivocation Check / Revocation

Revocation is the only way to keep the system sound in the long run. Keeping the number of faulty nodes within the quorum threshold is the key to the soundness of the system.

Each node severs the trust link independently without consulting others when it detects a node that has signed both $\langle x, t, v \rangle$ and $\langle x, t, v' \rangle$ s.t. $v \neq v'$. Also servers revoke clients when it detects a client signing different values with the same timestamp as well. Once a node is revoked, it will be excluded from the WoT graph forever and there is no way to restore it. See algorithm CheckEquivocation 5 for the equivocation check.

TOFU Policy

The system enforces the TOFU policy on every write request. If the slot is empty each node in the KV quorum will simply store the data. If the slot already has data, the node first retrieves the latest data and check if the signer is the same as the one of the requested data. See CheckTOFU 6 for the algorithm to check the TOFU policy.

3.3 Threshold Password Authentication ($\mathcal{TPA}$)

The quorum system based on the WoT graph guarantees data integrity. The TOFU policy with the quorum certificate prevents unauthorized mutations. The collective signatures make it possible to check equivocation. All those functions rely on the digital signature scheme therefore managing the signing key is significantly important for this system. We use a threshold password authentication ($\mathcal{TPA}$) for:

- enrolling nodes to the system
- recovering from a key-loss situation
- sharing an ID with multiple devices
- data secrecy (roaming encryption)

$\mathcal{TPA}$ is immune from offline dictionary attacks (as long as the number of compromised servers is less than the threshold). The authentication protocol is similar to [5] by Ford and Kaliski, but the shared secret is calculated by Shamir's Secret Sharing

(SSS) [7].

To set up the *shares* the client generates a random polynomial for (t, n) SSS over a prime field $\mathbb{Z}_q$, s.t.,

$$f(x) = \sum_{i=0}^{t-1} a_i x^i \bmod q$$

then calculates $n = |Q|$ pairs $(i, f(i)), i = 1..n$. The shared secret is $S = f(0)$. Each *share* will be $\langle i, y_i, v_i, u_i \rangle$, where:

$$\begin{aligned} y_i &= f(i) \\ v_i &= g_\pi^{S s_i} \bmod p \\ s_i &= \mathsf{OS2I}(H(pw, u_i)) \\ g_\pi &= \pi^2 \bmod p \\ \pi &= \mathsf{OS2I}(H(pw)) \end{aligned}$$

p and q are prime numbers such that $p = 2q+1$ (i.e., p is a safe prime). u_i is a salt. The random polynomial must be generated randomly for each password.

To get a password authenticated:

1. The client generates a random number $a \xleftarrow{\mathcal{R}} \mathbb{Z}_q$ and calculates X from the password:

   $$X = g_\pi^a \bmod p$$

   then broadcast it to a quorum Q.

2. Each quorum member returns:

   $$Y_i = X^{y_i} \bmod p$$

   with the salt u_i.

3. The client collects t responses from the quorum ($\mathcal{T} \subseteq Q$), and calculates:

   $$G_S = \prod_{j \in \mathcal{T}} Y_i^{\lambda_j} \bmod p$$

   where each λ_j is the lagrange interporate:

   $$\lambda_j = \prod_{l \in \mathcal{T} \setminus \{j\}} i_l / (i_l - i_j) \bmod q.$$

   Then, generate another random number $a' \xleftarrow{\mathcal{R}} \mathbb{Z}_q$ and for each server $i \in \mathcal{T}$, calculate and send back:

   $$X_i = G_S^{s_i a'} \bmod p$$

   where

   $$s_i = H(pw, u_i).$$

4. Each quorum member($\in \mathcal{T}$) generates a random exponent $b_i \xleftarrow{\mathcal{R}} \mathbb{Z}_q$ and sends back:

   $$B_i = v_i^{b_i} \bmod p.$$

   The DH session key is:

   $$K_i = X_i^{b_i} \bmod p.$$

5. The client shares the session key with Q_i:

   $$K_i = B_i^{a a'} \bmod p$$

   then requests to get the proof with MAC:

   $$N_i := MAC(H(K_i), X_i || B_i).$$

6. Each quorum member sends back encrypted proof ($P_i = Sig_{Q_i}(\centerdot)$) if and only if the MAC is correct:

   $$Z_i = E_{H(K_i)}(P_i, N_i).$$

7. The client decrypts Z_i with K_i and gets:

   $$(P_i, A) = D_{H(K_i)}(Z_i).$$

   Check if $A = N_i$ to make sure the proof is fresh.

We use $\{P_i\}_{\{i \in \mathcal{T}\}}$ as the proof of authentication which will be attached to protocol messages and checked at each node during the read / write / register protocols. The signed data for each $P_i = Sig_{Q_i}(\centerdot)$ depends on those protocols.

See Appendix A for correctness and security analysis of $\mathcal{TPA}$.

$\mathcal{TPA}$ for Enrollment

$\mathcal{TPA}$ is used to enroll new nodes to the system so it can recover from a key-loss situation and support multiple devices per user.

Enrollment:

1. Generate a unique ID and a key pair,
2. Send the certificate signing request to a quorum with the $\mathcal{TPA}$ *shares* to get a quorum certificate.

Key recovery / secondary key generation:

1. Generate a new key pair with the existing ID,
2. Do the password authentication and get a proof,
3. With the proof, send the new certificate to a quorum to get a new quorum certificate.

See 4.3 for the actual protocols.

$\mathcal{TPA}$ for Data Secrecy

The system provides a roaming service to store and access encrypted data. Users encrypt a value at a node then send the encrypted value to a quorum. They can decrypt the value at any node with the password. The password is stored along with $\langle x, t, v\rangle$. The encryption key is $H(g_\pi^S)$. S (and the random polynomial as well) has to be randomly generated everytime the data is encrypted at a client.

To decrypt the data the user specifies the variable x along with the password then the client starts the threshold password authentication process first to get the proof. With the proof the client proceeds the `read` process and decrypts the value with the above key.

With this roaming scheme, BFTKV can be a secure vault that is not only fault-tolerant for high availability but immune to single point of failures. The same kv storage can be used to keep both public / private keys without relying on a separated KMS.

3.4 Distributed Signing

Some crypto-currency platforms still use simple digital signature schemes to sign transactions. If private keys are stolen all assets can be transferred to someone's address without authorization from the original users. One of ways to mitigate such situation is to use *multi-sig*. With the multi-sig scheme multiple parties need to be involved in the signning process. Most systems adopt multi-sig as an add-on security measure to the blockchain technologies. BFTKV natively supports distributed signing schemes based on threshold cryptography, which is compatible with the standard algorithms therefore the verifiers do not need to change the way to process the transactions.

As a PKI platform, protecting CA's private keys must be the most important *feature*. At the same time, certificate signing requests should be processed quickly and automatically. The distributed signing feature satisfies those conflicting requirements.

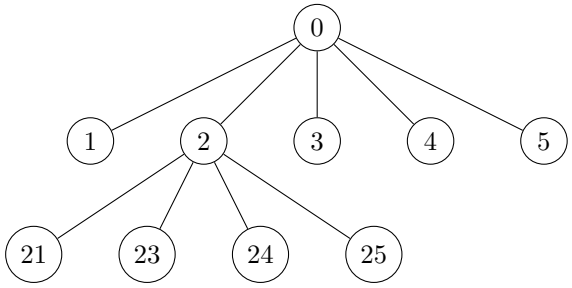
The system takes advantages of the distributed platform to make a signature without revealing the private key. Once the key is distributed among quorum members, the complete private key will never appear at any server or client. The system supports three signature algorithms: RSA (PKCS1.5), DSA and ECDSA. Our scheme takes existing keys as-is and distributes it to multiple servers, which will make it easy to migrate from existing systems.

RSA

It is straightforward to construct a (n, n) "threshold" scheme with RSA [8, 9]. Decompose the private key d into $\{d_i\}_{i=1..n}$ s.t., $d = \sum_{i=1}^{n} d_i \bmod \varphi(N)$, and the signature is calculated from partial signatures $S_i = M^{d_i} \bmod N$:

$$S = \prod_{i=1}^{n} S_i = M^d \bmod N.$$

To convert the (n, n) threshold scheme to a somewhat (t, n) threshold scheme, we construct the partial keys recursively. In the following tree, if a node (2) is faulty its key will be compensated by other nodes (1, 3, 4, 5) with the keys (21, 23, 24, 25) respectively, i.e., $S = \prod_{i\in\{1,2,3,4,5\}} S_i \bmod N$ where $S_2 = \prod_{i\in\{21,23,24,25\}} S_i \bmod N$.



The number of keys stored in each node will be increased exponentially as $n-k$ becomes big. Up to $(7, 10)$ threshold seems practical.

DSA, ECDSA

BFTKV implements a DSS threshold scheme introduced by Gennaro et al. [4]. Since the scheme has a restriction such as $n \geq 2t$ in the (t, n)-threshold scheme, we no longer be able to use the quorum threshold for (t, n), but we follow the protocols between the client and a quorum. The signing protocol consists of three phases:

1. Collect joint shared secrets generated by each quorum member: $\{(i, f_j(i))|f_j(0) = k_j\}_{\{i=1\ldots n, j\in Q\}}$.

2. Distribute the secrets to the quorum and calculate $r = g^{k^{-1}} \bmod p \bmod q$ where k is the joint Shamir's shared secret (i.e., $k = \sum k_i$).

3. Distribute r to the quorum and calculate $s = k(m + xr) \bmod q$ from each $s_i = k_i(m + x_i r) \bmod q$ returned from each quorum member.

4 Protocols

The system uses Phalanx [3] for the underlying read/write operations. To maintain a graph for the quorum system, we extend the protocols with join/leave. Also register protocol is used to get quorum certificates with the threshold password authentication.

Notations: $\langle x, t, v, s_C, S\rangle$ is an ordered set that denotes the protocol data. s_C and S can be omitted.

- x is the variable and an arbitrary length octet string. The variable will be the "key" of the key-value store.
- t is the timestamp and a 64-bit non-negative integer. $2^{64}-1$ is a special value to denote that the variable is no longer able to be updated.
- v is the value and an arbitrary length octet string.
- s_C is an object of the signature and quorum certificate of a client C. $s_C.sig$ is the signature over $\langle x, t, v\rangle$ signed by the private key of C. $s_C.cert$ is the quorum certificate of C. $s_C.cert.ID$ is the unique ID to identify each node.
- S is an unordered set of the signature object. Each $S_i \in S$ has the same structure of the s_C. The signers must be quorum members.

4.1 Read / Write

read/write protocols are done between a client (C) and a quorum (Q). A quorum is chosen from a quorum system ($QS \subseteq 2^U$) which is a subset of the powerset of all nodes (U). To write a value v into a variable x, we follow the write protocol specified in Phalanx.

[write]

C : choose a quorum from quorum cliques $Q \in QC$
$C \to Q$: get timestamp(x)
$C \leftarrow Q_i$: t_i
C : collect $2b+1$ timestamps: $\{t_i\}_{i\in\mathcal{T}}, |\mathcal{T}| = 2b+1, \mathcal{T} \subseteq Q$
C : choose the maximum timestamp $t = max(t_i) + 1$
$C \to Q$: get signature($\langle x, t, v, s_C\rangle$), where $s_C = (Sign_C(\langle x, t, v\rangle), Cert_C)$
Q_i : verify s_C with C's quorum certificates
Q_i : check the TOFU policy
$C \leftarrow Q_i$: $S_i = \{\langle x, t, v, s_C\rangle\}_{Q_i \in Q}$
C : collect signatures from $|\mathcal{T}|$ members $S = \{S_i\}_{i\in\mathcal{T}}$
C : choose a quorum from $Q' = U \setminus QC$
$C \to Q'$: write($\langle x, t, v, s_C, S\rangle$)
Q'_i : verify the signature set S
Q'_i : do the equivocation check
Q'_i : store($\langle x, t, v, S_C, S\rangle$)

[read]

C : choose a quorum $Q \in U \setminus QC$
$C \to Q$: read(x)
$C \leftarrow Q_i$: $M_i = \langle x, t, v, s_C, S\rangle$
C : collect $f+1$ responses
C : verify the signature set S
C : do the equivocation check
C : choose the latest timestamp $t_L = max(M_i.t)$
C : $Q' = \{Q_i \subset Q | M_i.t < t_L\}$
$C \to Q'$: write back($\langle x, t_L, v, s_C, S\rangle$)

4.2 Join / Leave

Any node can join / leave anytime by sending its quorum certificate to nodes it trusts. The node received the request verifies the certificate and adds it to the local graph which will be returned to the caller node. To leave the network, a node broadcasts its quorum certificate to the quorum it belongs to. The node that has received the graph constructs its own local graph from the graphs, then sends the join request to nodes that have not been connected. See algorithm Join 7.

4.3 Register

Each node generates its own public/private key pair, then sends the public key part to a quorum to get a quorum certificate. Each quorum member keeps all certificate issued to clients along with a partial password secret. If it finds a certificate request that has the same ID as one of the stored certificates it will sign it only if the password authentication is done. The client will get a "proof" of the authentication when it is finished. See algorithm Register 8

4.4 Authenticate

auth protocol is done in advance of read / write / register to get a proof and each subsequence protocol verifies it before processing the request. The follwing diagram is for a quick review of $\mathcal{TPA}$ 3.3.

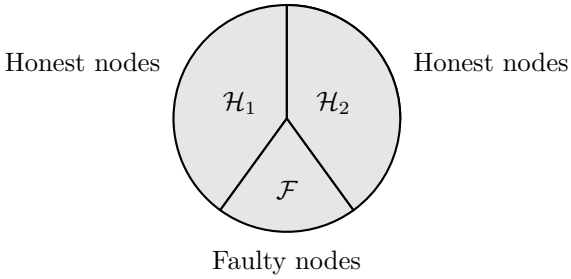
$$
\begin{aligned}
C : \pi \leftarrow H(pw),\ g_\pi \leftarrow \pi^2,\ a \xleftarrow{\mathcal{R}} \mathbb{Z}_q \\
C \rightarrow Q : X := g_\pi^a \bmod p \\
C \leftarrow Q_i : Y_i := X^{y_i} \bmod p, u_i \\
C : G_S \leftarrow \textstyle\prod_{j\in\mathcal{T}} Y_i^{\lambda_j} \bmod p, \\
s_i \leftarrow H(pw, u_i),\ a' \xleftarrow{\mathcal{R}} \mathbb{Z}_q \\
C \rightarrow Q_i : X_i := G_S^{s_i a'} \bmod p \\
Q_i : K_i \leftarrow X_i^{b_i} \bmod p,\ b_i \xleftarrow{\mathcal{R}} \mathbb{Z}_q \\
C \leftarrow Q_i : B_i := v_i^{b_i} \bmod p \\
C : K_i \leftarrow B_i^{aa'} \bmod p \\
C \rightarrow Q_i : N_i := MAC(K_i, X_i || B_i) \\
C \leftarrow Q_i : Z_i := E_{K_i}(P_i, N_i) \\
C : (P_i, N_i) = D_{K_i}(Z_i)
\end{aligned}
$$

At the end of the protocol, C obtains the proof $P = \{P_i\}_{i\in\mathcal{T}}$.

The servers keep track of the client ID, the number of attempts and the time. If the process does not go through next time the first response from each server will be slowed down to mitigate online dictionary attacks.

5 Security Analysis

We look into attacks against the fundamental property: $READ(Q_1, x) = READ(Q_2, x)$ for $\forall Q_1, Q_2 \in QS$, which is known as equivocation. The best that attackers can do is divide a clique into two sets ($\mathcal{H}_1$ and $\mathcal{H}_2$) and ask each set to sign $\langle x, t, v\rangle$ and $\langle x, t, v'\rangle$ separately. Then do the write protocol for the target nodes with collected signature sets S and S'. Honest servers will refuse the request because it does not satisfy the basic b-masking quorum condition: $|S| \geq b+1$. But with $b+1$ colluding nodes ($\mathcal{F}$), the attack will succeed.



The maximum number of signatures dishonest clients can get is $b + (n-b)/2$. Therefore, to overcome the attack we need:

$$n - b > b + (n-b)/2 \Rightarrow n > 3b.$$

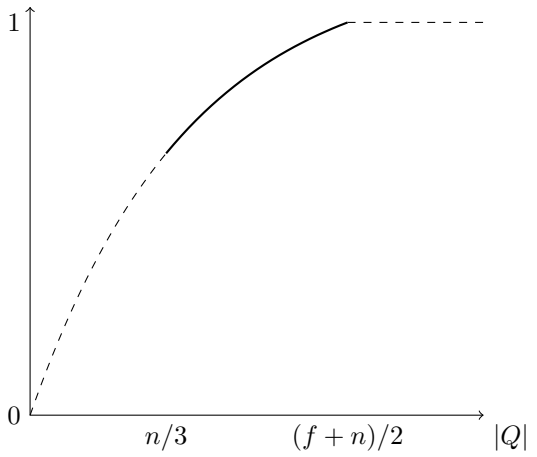
Detecting equivocation on read

Even if the number of faulty nodes exceeds the above threshold, the system can detect malicious actions with the following probability:

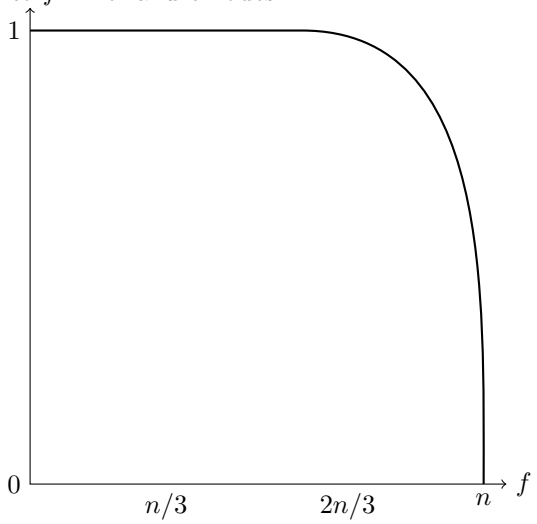
$$
\begin{aligned}
D_p &= Pr[Q \cap \mathcal{H}_1 \neq \emptyset \wedge Q \cap \mathcal{H}_2 \neq \emptyset] \\
&= 1 - Pr[Q \subseteq \mathcal{F} \cup \mathcal{H}_1] \\
&= 1 - ((n+f)/2n)^{|Q|}
\end{aligned}
$$

when $f > b$ and $|Q| < (f+n)/2$, where $f = |\mathcal{F}|$ is the number of the faulty nodes, assuming the sizes of $\mathcal{H}_1$ and $\mathcal{H}_2$ are the same.

In the case of $f \leq b$ malicious messages will never go through therefore it does not make sense to talk about the detection rate on read, however, it is possible to detect malicious messages on write if such actions are taken by clients. When the number of faulty nodes exceeds the threshold, i.e., $f > b$, it will be possible that the (honest) client cannot detect all malicious actions. Since the minimum quorum size is $(n-1)/3$ it does not need to consider the case where $|Q| < n/3$. Also, if the size exceeds $(f+n)/2$ any quorum always includes at least one node from each $\mathcal{H}_i$ which makes the detection rate 100%.



For example, assuming we choose a quorum $|Q| = 3b + 1$ out of $n = 4b + 1$, which is the default setup of the kv quorum system, the detection rate is 100% up to $f = 2b$ failure nodes.



6 Applications

We discuss some potential appliations of BFTKV.

6.1 Decentralized PKI with DKMS

User authentication is a long-standing problem for end-to-end systems. Even if we have semantically secure cryptographic protocols to exchange data between users, if it was with a wrong one, the whole security system would not make sense. On the other hand, once we have a robust user authentication scheme, we can build up many kinds of security systems on top of that, such as PGP and Signal. Our goal is to construct an infrastructure to exchange public keys that represent users' identities. Exchanging public keys can be done in person, using a QR code, confirming the fingerprint of public keys, etc. Those methods seem to be relevant for some situations, such as sending money. Also, public key infrastructures using central authorities, such as X.509 which is based on chain of trust, are widely used. A PKI like X.509, however, still have a problem when issuing a certificate to each end user. CAs issue certificates to corporates, organizations, and individuals based on trustworthiness of requesters but for end users whose authenticity is not easy to be proven, we have the same basic issues. From end users' point of view, blindly relying on a central authority based on its authenticity is no longer secure and contradict the end-to-end philosophy.

Our proposed PKI does not "strongly" rely on central authorities, yet it does not require to exchange public keys in person. Here are the high level system requirements:

- Scalability – the system can grow without affecting the current running services
- Transparency – anyone can monitor every system activity
- Quantifiability – security and efficiency can be formally analyzed
- Robustness – the system has to recover from erroneous situations by itself
- Privacy – the system should not reveal unnecessary information about users
- Non-interactivity – a client may not be able to interact peers before sending a message. This particular requirement makes it difficult to design a system that guarantees the "what I saw is what you see" concept. SMTP, for example, is not a mutual explicit authentication protocol. When an email is encrypted then sent out, if it is encrypted with a wrong key, it will be too late – someone in the middle could read the email when the recipient receives the email and notice that the encrypted email is not actually for her.

6.2 Consensus Mechanism for Blockchain Technologies

Key-value stores can be suitable to store transactions. Without knowing the key it will be difficult to access the value – in this sense it can be less transparent than other ledgers such as hash chain. To keep the transactions we can simply use the hash value of the whole transaction data. As every transaction should have a public key ("address") and

transaction ID in the sense of bitcoin Tx, the hash value should be unique. We write the value with the WRITE ONCE permission.

7 Conclusions and Future Work

We developed an efficient and secure distributed key-value store which can be a solution for a long standing problem: How to authenticate public keys without relying on central authorities. Here is a quote from "The Sesame Algorithm" [10] which is a part of the Signal protocol:

> "Sesame relies on users to authenticate identity public keys with their correspondents. For example, a pair of users might compare public key fingerprints for all their devices manually, or by scanning a QR code. The details of authentication methods are outside the scope of this document.
>
> If authentication is not performed, users receive no cryptographic guarantee as to who they are communicating with."

BFTKV is designed to be the last piece of this kind of real end-to-end encryption system.

We also integrated unique threshold password authentication and distributed signature schemes with a Byzantine fault-tolerant key-value store to solve nasty problems like unauthorized enrollment, recovering from a key-loss situation and supporting multiple devices under the same ID, which most permissioned blockchain systems will suffer when they actually try to deploy a system.

Future Work

Scalability of bitcoin blockchain is excellent – you can add as many nodes as you want and it will not affect the whole system, but the latency and throughput of the whole process is terrible – it takes one hour to settle a transaction. On the other hand, BFT type of blockchain settles transactions in a range from sub milliseconds to a few seconds, but scaling out is difficult. Specifically, with quorum systems we have theoretical lower bounds such as $n = 3f + 1$ and we cannot do anything about it.

BFTKV can separate the role of the node into two parts: signing and reading / writing, corresponding to the auth quorum and kv quorum for load balancing. Once a transaction gets a collective signature, the transaction can be stored in any way. The only concern is how to know a transaction is latest. Each transaction has the timestamp and we can easily get the latest $\langle x, t, v\rangle$ corresponding to a x in a node but we do not know if other nodes might have a newer value. BFTKV addresses this issue by a simple quorum system, i.e., as long as $\forall Q_1, Q_2 \in QS, |Q_1 \cap Q_2| > f$ holds, at least one node in a quorum has the latest value and the client will choose that one. This method is simple but it is difficult to scale out. We need to collect data from at least $f + 1$ nodes, which depends on the total number of servers.

Another concern is the number of collective signatures. Increasing the number of servers does not help improve the throughput at all, because every available server in quorum cliques needs to get invovled in the signing process. Also it will increase the number of the collective signatures so that each client and kv node needs to verify more signatures. One of the ideas to address this issue is to use a threshold signature to combine the set of collective signatures so that each node verifies the signature only once. But we need a dealer for the threshold signature scheme and it will contradict the decentralized concept and make the system less flexible.

References

[1] Nakamoto, S. (2008) Bitcoin: A Peer-to-Peer Electronic Cash System

[2] Malhi, D. and Reiter, M. (1998) Byzantine Quorum Systems

[3] Malhi, D. and Reiter, M. (1998) Secure and Scalable Replication in Phalanx

[4] Gennaro, R., Jarecki, S., Krawczyk, H. and Rabin, T. (1999) Robust Threshold DSS Signatures

[5] Ford, W. and Kaliski, B. (2000) Server-Assisted Generation of a Strong Secret from a Password, Proc. 9th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise, IEEE, June 14-16, 2000

[6] Shoup, V. Practical Threshold Signatures

[7] Shamir, A. (1979) How to Share a Secret

[8] Garay, J. A., Gennaro, R., Jutlab, C., Rabin, T. (2000) Secure distributed storage and retrieval

[9] Rabin, T. (1998) A Simplified Approach to Threshold and Proactive RSA

[10] Marlinspike, M. and Perrin, T. (2017) The Sesame Algorithm: Session Management for Asynchronous Message Encryption

[11] Boneh, D. The Decision Diffie-Hellman Problem

Acknowledgements

We would like to thank Prof. Idit Keidar for her expert advices about the quorum systems. In fact, the idea of separating the KV quorum system from the auth quorum system first appeared in her email messages. Also, Dr. Edward Bortnikov and Prof. Juan A. Garay helped us move the project forward in many ways.

A Security Analysis for $\mathcal{TPA}$

A.1 Correctness

Assume each $\{Y_i\}_{\{i\in\mathcal{T}\}}$ is correctly calculated, G_S will be:

$$G_S = \prod_{j\in\mathcal{T}} Y_i^{\lambda_j} = g_\pi^{a\sum_{j\in\mathcal{T}} f(j)\lambda_j} = g_\pi^{aS} \pmod p$$

and by raising $s_i = H(pw, u_i)$ to G_S, we get:

$$X_i = (G_S^{s_i})^{a'} = (g_\pi^{Ss_i})^{aa'} \pmod p$$

which must be the same as $v_i^{aa'} \bmod p$ for each $i \in \mathcal{T}$ iff the correct password (g_π) is given at the step 1. With each $B_i = v_i^{b_i} \bmod p$, the client and servers share DH keys $K_i = g_\pi^{Ss_iaa'b_i} \bmod p$.

Lemma 1. *From $\{Y_i\}_{i\in\mathcal{L}}$ where $\mathcal{L} \subset \{1..n\}$ and $|\mathcal{L}| < t$, correct G_S cannot be obtained.*

Proof. S cannot be reconstructed from $|\mathcal{L}|$ servers, from the SSS property, thus g_π^{aS} cannot be obtained whatever a is. □

A.2 Protocol Analysis

We look into the $\mathcal{TPA}$ protocols by dividing it into two parts: the first roundtrip (step 1 and 2) is to recover the shared secret S, and the second roundtrip (step 3 and 4) is for key exchange.

$g_\pi = \pi^2 \bmod p$, where π is a secure hash value over the password, is a safe generator in $\mathbb{Z}_q^*$ with high probability. Raise a random exponent a to mask the generator. Servers blindly raise its own share y_i to whatever $X = g_\pi^a$ is. As long as p is a safe prime $X^{y_i} \bmod p$ does not reveal y_i.

The second roundtrip can be seen as the traditional DH with the generator $G_S^{s_i}$ where $s_i := H(pw, u_i)$. u_i is a salt generated uniquely for each server. Another random exponent a' is applied for making it difficult to do offline dictionary attacks over s_i.

A.3 Resistance to MiTM

Assume we have $|\mathcal{L}| < t$ compromised servers and try to reconstruct g_π^S from $\{y_i\}_{i\in\mathcal{L}}$. From the Lemma 1 we need at least one Y_i, $i \notin \mathcal{L}$, and unless the server i is not compromised the only way to get Y_i is to issue the protocol $X = g_\pi^a \bmod p$. Unless the attacker knows π all he can do is to guess the password and to keep asking the server to calculate $Y_i = g_{\pi'}^a \bmod p$ for π'. If the guess is wrong Y_i' will not calculate g_π^S correctly with the compromised data $\{f(j)\}_{j\in\mathcal{L}}$. Unless the attacker can calculate S from $g^S \bmod p$ with a fabricated g, he cannot calculate $g_{\pi'}^S \bmod p$ in order to brute force for $g_{\pi'} \notin \{g^e | e = 1..p-1\}$.

A.4 Resistance to offline dictionary attacks

Under the same assumption as MiTM, what the client can obtain is:

$$y_i = f(i)$$
$$v_i = g_\pi^{Ss_i} \bmod p$$

for $i \in \mathcal{L}$. We also get $G_S = g_\pi^{aS} \bmod p$ from the step 2. B_i will not contribute anything to the attack as they already have v_i. To brute force on v_i there are two possible ways:

1. $\exists S \in \mathbb{Z}_q$, check if $v_i \stackrel{?}{=} (g_\pi^{s_i})^S \pmod p$ for each $g_\pi^{s_i}$

2. $\exists g_\pi \in \mathbb{Z}_q^*$, check if $v_i \stackrel{?}{=} (g_\pi^S)^{s_i} \pmod p$ for each s_i

For the former case we need S and for the latter case we need g_π^S. From the protocol 1, we can obtain either $g^S \bmod p$ for $\forall g \in \mathbb{Z}$ or $g_\pi^{aS} \bmod p$. Obtaining S is simply the DLog problem. $g_\pi^S \bmod p$ can be obtained iff g happens to be g_π or a legitimate client chooses $a = 1$ which must be excluded. If g_π^S is obtained, the dictionary attack is possible on s_i, however, it is difficult to obtain g_π^S from G_S which is calcualted by a legitimate client as a has been randomly chosen. Attackers can choose $a = 1$ at the step 1 and they can obtain $g_{\pi'}^S$, but they need to start over the protocol to get the correct g_π^S.

B Algorithms

Algorithm 1: GetQC

Input: G: the graph, s: the start node, L: the maximum distance
Output: QC: quorum cliques
queue $\leftarrow (s, 0)$;
$QC = \{\}$;
repeat
  $(v, d) \leftarrow$ queue;
  if $d \geq L$ **then**
    break
  end
  $QC = QC \cup$ `FindMaximalClique` (G, v);
  foreach $n \in v.adj$ **do**
    queue $\leftarrow (n, d + 1)$ if n has not been visited;
  end
until queue $= \emptyset$;
Check if $\forall C_1, C_2 \in QC, C_1 \cap C_2 = \emptyset$;
return QC

Algorithm 2: FindMaximalClique

Input: G: the graph, s: the start node
Output: C: a clique
$C \leftarrow \{s\}$;
foreach $v \in G.V$ **do**
  $C = C \cup \{v\}$ **if** $(c_i, v) \in G.E \wedge (v, c_i) \in G.E$ for $\forall c_i \in C$;
end
if $|C| < 4$ **then**
  return $\perp$
end
return C

Algorithm 3: VerifyCollectiveSignatures

Input: G: the graph, $S = \{S_i | S_i = Sign_{Q_i}(\langle x, t, v, s_C \rangle)\}$
$QC =$ `GetQC` $(G, self)$;
foreach $C \in QC$ **do**
  $cnt = 0$;
  foreach $s \in S \cap C$ **do**
    if `Verify` *(s)* **then**
      $cnt++$;
    end
  end
  if $cnt \leq 2f$ **then**
    return *false*
  end
end
return *true*

Algorithm 4: CheckQuorumCert

Input: G: the graph, $Cert$: quorum certificate
$C =$ `FindMaximalClique` $(self)$;
$cnt = 0$;
foreach $c \in Cert$ **do**
  if $c.Signer \in C$ **and** `Verify` *(c.Signature)* **then**
    cnt++;
  end
end
return $cnt > (2|C|)/3$

Algorithm 5: CheckEquivocation

Input: $req = \langle x, t, v, s_C, S \rangle$
$z =$ Storage $[x, t]$;
if $z \neq \perp$ **and** $req.v \neq z.v$ **then**
  `Revoke` $(req.S \cap z.S)$;
end

Algorithm 6: CheckTOFU

Input: $req = \langle x, t, v, s_C, S \rangle$
Verify $req.s_C$ with quorum certificate;
if Storage $[x, 0] = \perp$ **then**
  Storage $[x, t] = req$;
else
  $last =$ Storage $[x, t - 1]$;
  if $last.s_C.cert.ID = req.s_C.cert.ID$ **then**
    Storage $[x, t] = req$;
  else
    Error;
  end
end

Algorithm 7: Join

Input: Cert
$G.V = Cert.sigs[*].cert$;
$peers = G.V$;
for $peers = \perp$ **do**
  $newPeers = \{\}$;
  for $peer \in peers$ **do**
    `Send` $(peer.addr, Cert)$;
    $certs =$ `Receive` ();
    $newPeers = newPeers \cup (certs \setminus G.V)$;
    $G.V = G.V \cup certs$;
  end
  $peers = newPeers$;
end

Algorithm 8: Register

Input: req: a client certificate, $proof$: the proof of the password authentication
$z =$ Storage $[req.x]$;
if $z \neq \perp$ **and** $req.s_C.ID = z.s_C.ID$ **then**
  $clique =$ `FindMaximalClique` $(self)$;
  if $proof \subseteq clique$ **and**
   $|proof| \geq |clique| \cdot 2/3$ **then**
    `Sign` (req);
  end
end