DISTRIBUTED PASSWORD AUTHENTICATION

Evan Liu, Brendon Go, Kenneth Xu
Department of Computer Science
Stanford University
Stanford, CA 94305, USA
`{evanliu, bgo, kenxu95}@cs.stanford.edu`

ABSTRACT

Traditional password systems are vulnerable to attackers who steal the file containing the hashes of all user passwords. The attacker can then mount an offline dictionary attack to recover the plaintext password. To solve this problem, we introduce ThresholdDB, a key-value storage system that requires multiple cooperating nodes to service get requests. Consequently, attackers who compromise f or fewer ThresholdDB nodes cannot recover the information stored in the database. This has practical applications for storing information like social security numbers, credit card numbers, or passwords. We implement a password authentication system on top of ThresholdDB to provide distributed password-authenticated key exchange.

1 INTRODUCTION

Recently, several high profile security breaches have severely compromised important personal information. For example, a file containing information about all the passwords at LinkedIn was stolen in 2012, and more recently, up to 143 million social security numbers were stolen from Equifax. An important factor that helps enable attacks like these is that this important information is typically stored in such a way that it can be accessed from a single server. As a result, a sufficiently motivated attacker is able to compromise this information by compromising a single server.

To mitigate this issue, we introduce ThresholdDB, a database that supports a key-value interface. Like many databases, ThresholdDB provides durability guarantees through replication on many servers. However, in contrast, ThresholdDB stores information in such a way that read requests require cooperation of many servers. Consequently, in order to steal any information from a ThresholdDB instance, an attacker must compromise many servers. These servers may use different operating systems or have different permissioning systems to ensure that their failures are uncorrelated, thus preventing attackers from obtaining private information.

We implement ThresholdDB and apply it to the application of password authentication. In our application, we use password-authenticated key exchange to authenticate clients (and for clients to authenticate the application), where the password secrets are stored in ThresholdDB.

The rest of the paper is structured as follows. In section 2, we give an overview of threshold-encryption, a cryptographic primitive that we use to implement ThresholdDB. In section 3, we give concrete details about the protocols we use in ThresholdDB. In section 4, we describe the particular details necessary to implement our password authentication application. In section 5, we show that while performance is sacrificed in our password authentication application, as compared to systems that do not provide the same durability or security we do, our system can still process around 10 requests per second, which is fast enough to do authentication. Finally, we conclude in section 6.

2 THRESHOLD PUBLIC KEY ENCRYPTION

In the threshold encryption scheme that we use (Baek & Zheng, 2003), messages are encrypted using the public key. Decryption shares can be obtained by decrypting the ciphertext using any of the N private keys. The original message is then recovered by combining $f + 1$ decryption shares obtained with different private keys. Concretely, a threshold public key encryption scheme exposes the following interface:

- **generate_keys()** $\rightarrow PubKey, [PriKey_i]$: Generates a single public encryption key and N private keys
- **encrypt(**$PubKey$**,** msg**)** $\rightarrow ciphertext$: Encrypts the message msg
- **decryption_share(**$PriKey_i$**,** $ciphertext$**)** $\rightarrow \sigma_i$: gets a decryption share given a private key and ciphertext
- **combine_shares(**$PubKey$**,** $ciphertext$**,** **[**σ_i**])** $\rightarrow msg$: combine decryption shares to obtain the message. Fails if any decryption share was invalid or an insufficient number of shares are given.

Note that sharing decryption shares does not reveal any information about ones private key. We use threshold encryption to encrypt information stored in our database. To service read requests, each server provides a different decryption share for the value, requiring at least $f + 1$ servers to provide correct decryption shares for successful reads.

3 THESHOLDDB

In this section, we describe the details of ThresholdDB. In 3.1, we describe the exact attack scenarios we are interested in defending against, as well as those we leave for future work. In subsection 3.2, we give an overview of the design. In subsection 3.4, we sketch out a proof of correctness to show that our system satisfies safety, fault-tolerance, as well as liveness assuming weak synchronization.

3.1 ATTACK SETTINGS

One-Time Passive Compromise. In one-time passive compromise, the attacker gains unauthorized access to private information on a server. In this setting, we assume that the attacker downloads this private information somewhere else, and access is thereafter revoked. The attacker cannot observe any additional activity on the server after downloading the private information, and cannot use the server's permissions to do other activity.

This is a relatively common attack setting. For example, an attacker might steal the file containing the salted hashes of passwords from a server (as in the LinkedIn attack). The attacker must then perform an offline dictionary attack to recover the actual passwords. ThresholdDB is designed to address this setting. Each (key, value) is encrypted using threshold encryption, so that recovering the (key, value) requires cooperation from many servers.

One-Time Active Compromise. In one-time active compromise, the attacker can additionally send authenticated messages from the compromised server, effectively impersonating the server. ThresholdDB also handles this setting, by making our protocol Byzantine Fault Tolerant. In particular, our protocols are robust for up to f servers running arbitrary code, in the sense that these f servers can neither obtain any stored (key, value) pairs, or modify / add any new (key, value) pairs. However, in one-time active compromise, we assume that the attacker will be detected shortly, so the attacker only controls the server for a short period of time and cannot participate in protocols for a prolonged period of time.

Long-term Active Compromise. Long-term active compromise is the strongest attack setting, where the attacker remains undetected on a server, indefinitely controlling the server. Under this attack setting, an attacker can learn (key, value) pairs stored in ThresholdDB, but we do provide some guarantees. First, even under long-term active compromise, an attacker can never modify the database by deleting, changing, or adding (key, value) pairs. This is covered by the fact that the protocol is Byzantine fault tolerant. Thus, as long as the attacker compromises f or fewer servers, no data is ever lost. Second, the attacker cannot learn of arbitrary (key, value) pairs. In our protocols, the attacker can learn the (key, value) pairs associated with any get or put requests it observes, but as long as the attacker cannot issue his own get or put requests through the client, he cannot learn about arbitrary (key, value) pairs. In our password authentication application, the attacker does have the power to issue his own get or put requests by logging in arbitrary users or enrolling new users, but we can mitigate these effects by rate-limiting login requests. We leave further robustness to this powerful attack setting to future work.

3.2 Design Overview

ThresholdDB is a Byzantine fault tolerant system consisting of $N = 3f + 1$ servers (tolerating f failures) that allows clients to perform two operations:

- **put**(key**,** $value$): Stores the (key, $value$) pair in the database
- **get**(key) Retrieves the $value$ associated with key in the database if it exists

Client C performs operation O by sending <O, $timestamp$, C> to some leader server. If a non-leader server recieves a request from the client it forwards it to the leader. The leader server is chosen with a leadership election algorithm and new leaders are elected when the current one fails to respond. The timestamp is to ensure exactly once semantics. If the client does not receive responses fast enough, it broadcasts the request to all the servers.

Each server knows the threshold encryption public key and has its own threshold encryption private key. Each server also has their own persistent database.

All messages that are sent to and from the client and between servers are signed with some public key encryption (in our implimentation we use RSA) so each server also has their own private signing key and knows the public signing keys of all other servers and all clients. All the servers also log received messages so that it can survive crashes. It can truncate entries in the log that belong to completed transactions.

Due to space constraints we omit specific details of leadership election or logging in this paper, the details mirror (Castro et al., 1999).

3.3 Normal Case Operation

Put When a leader recieves a new Put Request from a client, it broadcasts a PutAccept message: $< Accept, server\_id, transaction\_id, client\_request >$ to all its peer servers, choosing $transaction\_id$ so that puts are linearized. When a server hears a PutAccept message for a new $client\_request$, it also broadcasts a PutAccept message. The $client\_request$ is included in the PutAccept message so that each server can validate that a client is in fact making this request so prevents a faulty server from issuing requests and also prevents a faulty primary from sending different Put Requests to different peer servers.

Each server then waits to hear PutAcccepts from $2f + 1$ different servers. Once it does, it encrypts the $value$ in the $client\_request$ with the threshold encryption public key to get ciphertext c and stores $key, c, transaction\_id$ in a persistent database and sends the client a PutSucess message: $< Success, server\_id, client\_id, timestamp >$.

The client waits for $f + 1$ success messages from different servers before it is sure that the operation completed.

Get When a leader recieves a new Get Request from a client, it retrieves the ciphertext associated with the key from its persistent database, calculates its decryption share using its threshold encryption private key, then broadcasts a DecryptionShare message: $< DecryptionShare\sigma_i, server\_id, client\_request >$ to all its peer servers. Again, the client request is included so that each server can indepently validate the $client\_request$.

Once each server has a total of $f + 1$ unique decryption shares, it can try to combine the shares and return a successfully decrypted ciphertext to the client (note that if any of the shares are invalid decryption will fail). Since we have a maximum of f faulty servers, each server will have to wait for at most $2f + 1$ unique decryption shares to successfully decrypt a ciphertext value.

Get Requests fail if a server is unable to get a valid decryption with $2f + 1$ unique decryption shares. This happens when a key is retrieved as the same key is being overwritten with another put. In this case we send the client a GetFailure message and they can try again. An alternative would be to serialize reads and wait behind writes but we choose to instead to have non-blocking reads that can possibly fail since a database that stores sensitive information should not recieved a high number of overwrites (password, social security number, birthday, etc are unlikely to change).

Once the client recieves $f + 1$ matching responses from $f + 1$ different servers it can be sure that the value returned is correct.

Catchup When a server boots up, it recovers from any crashes by replaying its log, but this server still needs to catchup on messages it missed all together while it was down. To catchup, it broadcasts a Catchup Request message: $< CatchupRequest, last\_transaction\_id, server\_id >$.

All the other servers simply send the catching up server all the $(key, c, transaction\_id)$ triples that they have where $transaction\_id > last\_transaction\_id$ as data in a Catchup Response message: $< CatchupResponse, server\_id, data >$. The catching up server writes a triple to its database only if $f + 1$ servers have a matching entry in theirs. While it is catching up, a server can participate in new put requests but should store all puts in an auxiliary database and merge those new entries into a caught up database after it has finished catching up.

3.4 Correctness

In this section we give a proof sketch for safety, liveness, and Byzantine Fault Tolerance of our system.

Safety ThresholdDB provides safety if all non-faulty servers agree on Puts. An entry is only every written to persistent storage of server i when $2f + 1$ other servers agree to put. Since we can have at most f faulty servers, at least f non-faulty servers sent a PutAccept and will eventually write the entry to their storage as well. Thus we will have at least $f + 1$ servers with the correct entry in the database so Gets and Catchups for this entry will be correctly satisfied.

Liveness Under weak synchronization assumptions, as long as there are $2f+1$ non-faulty servers, progress will eventually be made on Puts and Gets from a non-faulty leader. A faulty leader can be detected by other servers since they compare client requests and client requests are eventually broadcast to all servers. A detected faulty leader will thus eventually trigger a new leader election, so progress will eventually be made.

Byzantine Fault Tolerance We provide Byzantine Fault Tolerance by using signed messages and including client requests in all messages. We also are able to recover from crashes with logfile replay and the catchup protocol.

4 Password Authentication

In this section, we describe our password authentication application. First, in 4.1 we give an overview of password-authenticated key exchange (PAKE). Then we describe how we implement a distributed version of PAKE, which we call DPAKE, using ThresholdDB in 4.2.

4.1 Password-Authenticated Key Exchange (PAKE)

One key challenge that password authentication systems face is verifying the identity of the server. Concretely, the client initiates an enroll protocol to share a secret (the password) with a server. Later, the client contacts the server again in the login protocol. The client verifies his own identity by providing the password, but how can the client be sure that the server is associated with the application he enrolled with (e.g. the IP address of the enrollment server may differ from the login server)? One common method (e.g. used for verifying websites) for addressing this question is certificate authorities, trusted third-parties who provide public keys associated with servers. Servers additionally provide a certificate signed with their private key, which can be verified with the public key from the certificate authority. However, many domains lack such certificate authorities (e.g. SSH protocol). PAKE provides an alternate way to verify the server's identity even without the presence of certificate authorities. Even when certificate authorities exist, PAKE is useful for warding off phishing attacks. For example even with certificate authorities, an attacker may obtain a valid certificate for bankofthevvest.com (two v's instead of a w) and may trick users into revealing their passwords (Boneh & Shoup, 2017).

Specifically, we implement the PAKE 2+ protocol, illustrated in 1. The client P on the left and the server Q on the right are assumed to have shared a secret password through some enrollment protocol. During the enrollment phase, both P and Q calculate (π_0, π_1) by applying a hash function H' to the shared password they agree on. Instead of directly storing π_0, π_1, the server stores g^{π_1}, where g is some group element. The server also stores π_0. Additionally, group elements a, b are publicly known. To authenticate, the client recalculates (π_0, π_1) from the password and sends $u = g^{\alpha} a^{\pi_0}$ where α is a randomly drawn integer in $\mathbb{Z}_q$.

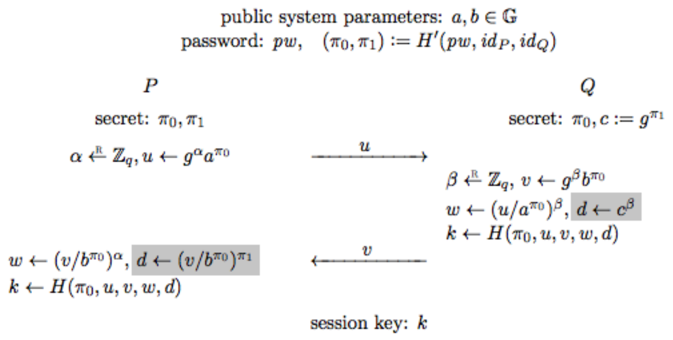
public system parameters: $a, b \in \mathbb{G}$
password: pw, $(\pi_0, \pi_1) := H'(pw, id_P, id_Q)$
P
Q
secret: π_0, π_1
secret: $\pi_0, c := g^{\pi_1}$
$\alpha \xleftarrow{R} \mathbb{Z}_q, u \leftarrow g^{\alpha} a^{\pi_0}$
u
$\beta \xleftarrow{R} \mathbb{Z}_q, v \leftarrow g^{\beta} b^{\pi_0}$
$w \leftarrow (u/a^{\pi_0})^{\beta}$, $d \leftarrow c^{\beta}$
$k \leftarrow H(\pi_0, u, v, w, d)$
v
$w \leftarrow (v/b^{\pi_0})^{\alpha}$, $d \leftarrow (v/b^{\pi_0})^{\pi_1}$
$k \leftarrow H(\pi_0, u, v, w, d)$
session key: k

Figure 1: Illustration of PAKE 2+ protocol. Figure taken from (Boneh & Shoup, 2017).

The server sends $v = g^{\beta} b^{\pi_0}$ where β is a randomly drawn integer in $\mathbb{Z}_q$. Then, both the server and client can calculate $g^{\alpha\beta}$ and $g^{\pi_1\beta}$, which an attacker cannot, without knowing π_0 and π_1 or g. The server and client both determinstically use these quantities to derive a shared session key k. Authentication can occur by the server and client communicating in a channel encrypted by k. If communication is successful, then the key k must be shared, and therefore both parties are authenticated.

Notably, PAKE 2+ provides some security against the one-time passive attack setting, where an attacker steals the (π_0, g) associated with each password (stored on the server), but is still vulnerable against this attack in general. An attacker can conduct an offline-dictionary attack by checking $H'(password)$ for many different common passwords. If the result matches π_0 then the password is likely correct. If no passwords match on π_0, PAKE 2+ does provide strong guarantees, because an attacker would need to compute π_1 from g, which would be an efficient solution to the discrete log problem. Our contribution, DPAKE further addresses this problem, so that even if a weak password is used (which would likely match on π_0 in a dictionary attack), an attacker cannot recover the password without compromising more than f servers.

4.2 Distributed Password-Authenticated Key Exchange (DPAKE)

Whereas in PAKE 2+, the server directly stores (π_0, g), vulnerable to an adversary that can compromise the server and steal this, in DPAKE, the server stores (π_0, g) in ThresholdDB. Specifically, when a new user enrolls with username u and password pw, the server makes a call to ThresholdDB $put(u, (\pi_0, g))$ and authenticates users by calling $get(u)$ and performing PAKE 2+. The server also ensures that old users are not overwritten by new enrollments by checking with the database before additional puts. Under this scheme, the one-time passive attack, which could compromise many PAKE 2+ passwords, no longer works, since each (π_0, g) is threshold encrypted in ThresholdDB, making it impossible for the client to recover this information without compromising more than f ThresholdDB servers.

5 Experiments

We report on the throughput of our system and compare against our implementation of the normal PAKE 2+ algorithm, which does not store (π_0, g) via ThresholdDB, but only writes it to a non-replicated database (vulnerable to the one-time passive attack). We also report how throughput changes as we change f and consequently N. In these experiments, we report on workloads consisting of a mix of login and enrollment requests, although we observed that logins and enrollments are serviced in approximately the same amount of time. We report how throughput changes under varying loads ranging from 100 ongoing requests to 500. These results are summarized in Figure 2.

Figure 2 shows that, unsurprisingly, non-replicated storage of (π_0, g) is approximately 4x faster than replicated secure storage in ThresholdDB. This gap is largely due to the network round-trip times and time to sign messages. However, importantly, ThresholdDB still achieves a throughput of around 10 requests / s, which is fast enough to service logins and enrollment. Additionally, we believe that this performance sacrifice is justified for building more secure systems. Figure 2 also shows that, somewhat surprisingly, our system's

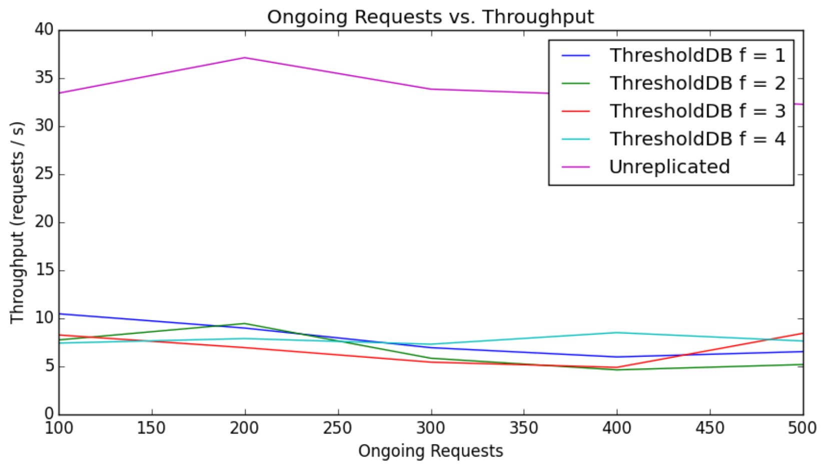

Figure 2: Experimental results comparing throughput of ThresholdDB with various values of f and unreplicated storage. The x-axis denotes the workload, e.g. when ongoing requests is 100, this reports the throughput under a sustained load of 100 login and enroll requests.

performance is relatively unaffected by changing f. This suggests that we can choose a higher f to provide greater replication and security with little performance loss.

6 Conclusion

We have implemented ThresholdDB, a key-value store that provides replication and stronger security guarantees than normal databases and use this to implement a password authentication system that uses PAKE. Our Byzantine fault-tolerant protocols closely resemble the protocols in Practical Byzantine Fault Tolerance (Castro et al., 1999) and we use thresholded PAKE similar to (MacKenzie et al., 2002; Abdalla et al., 2005). Our experiments show that our system has lower throughput than unreplicated systems, but is still fast enough to practically service login and enrollment.

While we address some previously unaddressed security concerns, our current work still is not fully robust in the long-term active compromise setting. As future work, we are interested in modifying the get and put protocols such that no single server has enough information to recover the stored data at any time, which would protect against this setting. We are also interested in applying ThresholdDB to storing other types of information.

References

Michel Abdalla, Olivier Chevassut, Pierre-Alain Fouque, and David Pointcheval. A simple threshold authenticated key exchange from short secrets. In *ASIACRYPT*, volume 3788, pp. 566–584. Springer, 2005.

Joonsang Baek and Yuliang Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. In *Global Telecommunications Conference, 2003. GLOBECOM'03. IEEE*, volume 3, pp. 1491–1495. IEEE, 2003.

D. Boneh and V. Shoup. *A Graduate Course in Applied Cryptography*. 2017.

Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pp. 173–186, 1999.

Philip MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In *Annual International Cryptology Conference*, pp. 385–400. Springer, 2002.