TRACELESS

Nikhil Buduma, Pratheek Nagaraj, Mayuri Sridhar
nkbuduma@mit.edu, pnagaraj@mit.edu, mayuri@mit.edu

May 8, 2015

6.857 Project Report

1 Introduction

Messaging is a popular form of interpersonal communication. Most modern communication systems have clients communicate through a server. The concern arises when the server is curious and can thus peek at messages. To solve this concern, end to end encryption may be used. Though this isn't enough since information is leaked about users communicating with specific users. To our knowledge, Traceless is the first system to serve as a oblivious messaging platform.

2 Overview

Traceless is a cryptographically secure and oblivious messaging system. In this paper we examine the design of Traceless and for brevity exclude any performance analysis. Section 3 discusses the design goals of the system and assumptions. Section 4 explores the cryptographic considerations in Traceless. Section 5 examines the distributed aspect of the system. Section 6 discusses some aspects of the implemented system. Section 7 the concludes.

3 Design Goals and Assumptions

This section explores some of the main design goals involved in building Traceless as well as discusses some assumptions that are made in order to achieve these goals.

3.1 Design Goals

Obliviousness - Our design goal is to ensure that the server is provably incapable of determining whether two individuals ever had a conversation on the service (using a computational secrecy model). In particular, we claim that our design ensures that 1) Any message pushed to the server cannot be traced back to a user and 2) any pull request for a message stored on the server also cannot be traced back to a user.

Responsiveness - To reduce server load, we divide server state into multiple non-overlapping shards. Because users only have to communicate with the shard containing their respective information (determined by the message slot they want to pull/push from), we can multiplex responses proportional to the number of shards. Locations of these shards are managed a master server.

Fault Tolerance - To prevent the system from crashing when a single machine, each shard is replicated (total of two copies - primary and backup). The master server manages which of these shards is the primary and which of them is the backup (state is synchronized by a two-phase commit). This means that even if a primary server for a shard fails, the secondary shard can take over for the primary.

3.2 Assumptions

1. Computational Security - We make assumptions about what kinds of computation a machine can reasonably achieve. We make no claims of perfect secrecy.

2. IP Spoofing - We assume that actors are able to (and are responsible for) keeping their IP hidden. This is a realistic assumption that is made in the unlinkable serial transactions

4 Cryptographic Design

Traceless is designed as a client server model. Clients wish to communicate to other clients and the server is used as a means for passing such messages around. The system is dependent on unlinkable serial transactions (UST) [1] as well as the RSA cryptosystem. This section of the paper explores the different aspects associated with the cryptography of the system briefly.

4.1 Subscription

When a client wants to join the service, he or she first generates a username. Behind the scenes, the client code then generates two random 4096 bit RSA keys, as well their first blinded nonce and sends this information to the server.

Assuming that subscription is successful (the username hasn't previously been taken), the server then adds this user's username and public RSA keys to its list of clients.

Existing clients (including the new member) constantly ping the server and download the server's list of clients and maintain a local copy.

This is essential because it ensures that the clients don't have to query the server to discover how to send or verify messages. That is, by maintaining a local copy of both RSA keys (one for encrypting and one for signing) for all users, a user can both send a message to a specific user and verify the sender of a message without requesting additional information from the server.

The blinded nonce that the client sends the server here initializes an unlinked serial transaction and ensures that no future commmunications between the server and this client can be easily correlated.

When a subscription transaction is successful, the server returns the user their first blinded nonce and signature pair. The user can then unblind this the next time they send the server a request.

4.2 Reservations

To ensure that the server doesn't know which users are communicating, our messaging protocol is based on client A pushing a message to a specific block on the server's message table and client B pulling from the same block. However, this idea is based on the idea that clients A and B can share a secret key - the ID of the block - without the server's knowledge. While this is easy to implement a single time (Shamir's three-pass protocol, for instance), it becomes more complex when the secret key must be updated quickly. To deal with this problem, our protocol uses the idea of reserving a block on the server.

That is, before client A sends a message to client B on a block M, client A must first send the server a request, reserving the block.
To send the server a reservation request, client A sends the server it's current (nonce, sig) pair, a new blinded nonce, a blinded nonce for deletion, and the ID of the block.

If the block is not previously reserved, then, the server checks if the (nonce, sig) pair is valid. If it is, then it marks the block M as reserved and returns the client a new blinded signature for their new nonce and a blinded signature for their random deletion nonce.
By using unlinked serial transactions here, the protocol ensures that the server has no idea who the block is being reserved by. The (nonce, sig) pair for deletion ensures that the information in block M is only deleted by either the sender or the receiver.

Now, assuming that client A and client B have started their conversation on some block, then client A transmits M as part of his or her message, as the next block that client B should read from.

Since the only unencrypted information the server has access to is the block ID, the server just knows that there are messages being transmitted through block M. However, assuming IP spoofing, it's impossible to correlate this information with a specific client or specific conversations.

4.3 Initiating Conversations

Although it's relatively simple to continue a conversation using the reservation protocol, initiating a conversation was trickier. That is, fully secure transmission requires that the two clients are able to establish the first block that each will read from, without the server gaining any new information. The solution that Traceless utilizes is based on a new-conversations data structure that's held by the server.

When client A wants to start a conversation with client B, A first reserves two blocks on the server. Then, A sends the server a new-conversation request, using unlinked serial transactions.

Essentially, this request contains a conversation object, encrypted with B's RSA public key for encryption. This conversation object is then appended to the server's new conversations table. This table is continually pinged by all the clients. When the table is updated, each client tries to decrypt the new conversation object. In this case, only B can decrypt the message.

The message that B decrypts will be divided into two parts: P and sign(P). That is, the first part of the message will be P = A's username, B's username, two slot ID's, a deletion nonce, and the signed deletion nonce. The last 1024 bits of the message will be the signature of P.

B knows that he has decrypted the message properly when he or she can recover both his username and the username of the sender. Then, B uses A's public signing key to verify that (P, sign(P)) is valid. This protocol will ideally use a signature scheme, like PKCS1, which is difficult to forge. Essentially, this ensures that B knows that the message is sent by A.

The two slot ID's at the end of P are randomly generated slots that A had previously reserved. The first slot is the slot that B will store as the first slot that A will write to. B will start continuously pinging this slot until A updates it with a message that B can decrypt. The next slot ID is the first slot that B can write to, that A will be continously be pinging from.

Assuming that client A can spoof his or her IP, the server doesn't have any way of correlating the reserve requests with the message on the new conversations table. Thus, although the server knows that there is a new conversation between two clients, it's impossible for the server to have any information about where the messages are being sent, their frequency, or their origin.

The protocol for message deletion is covered in section 4.5.

4.4 Message Push and Pull

Assuming that A and B have initiated a conversation, using the protocol covered in the previous section, we want to ensure that conversations can be continued without revealing any extra information to the server.

Without loss of generality, let client B know that the next message from client A will be written on block K.

Before sending a message on block K, client A first sends a reserve request to the server and reserves the next block it will write to, W. If the server accepts the reserve request, it sends A a nonce, signature pair for W's deletion.

Client A will then create a message P containing the text that he or she wants to send, the ID of W, and the deletion nonce, signature pair. Then, client A will sign the message P using its private signing key. Both P and sign(P) are encrypted separately using B's public key, and then concatenated into a single message M.

Then, client A contacts the shard server responsible for slot K ands sends a push request. This request contains M as well as the block ID K. Meanwhile, client B is continuously pinging the shard for block K.

When the block has been updated by client A, the shard responds successfully and returns the message M to B.

Client B can then decrypt the message using his or her secret key. First, he or she can verify that the message was truly from A. Then, client B knows the next block that A will write to and B can begin pinging block W for A's next message.

Client B also recovers the message's text that A has sent, as well as the deletion nonce and signature pair.

Our protocol ensures that all requests to the server are done using unlinked serial transaction. This ensures that client A's reserve request cannot be easily correlated to its push request, or any future reserve requests. This is similarly true for client B's reserve and pull requests. This establishes that the server cannot determine the frequency of conversation between users, nor their identities.

4.5 Message Deletion

Up until now, the server has no information about any two users which share a conversation, or even if two given users are currently in a conversation. To implement message deletion, however, requires that two users that share a conversation share extra information that can be easily updated. That is, if client A pushes a message M to slot K, then both the sender and the receiver should be able to delete the information at slot K after it has been received and decrypted. Yet, no other user should have these permissions.

To satisfy these requirements, reservation requests to the server require that the client send an additional nonce for deletion. The server responds by sending a blinded (nonce, sig) pair. When client A receives this information, he or she unblinds this information.

Now, client A can send the server a delete request, with the (nonce, sig) pair and the ID of the slot. However, due to the nature of unlinked serial transactions, the server can only verify that this (nonce, sig) pair is valid for some delete request. This would leave the protocol vulnerable to an adversary, since they could simply request random slots, generate several deletion nonce and delete different slots.

To fix this problem, it's required that the first 128 bits of the nonce that the client sends the server is exactly the ID of the block that he or she is currently reserving. Thus, when the server receives the unblinded (nonce, sig) pair, with the block ID in the deletion request, it first verifies that the nonce is valid and also belongs to the correct block, before removing information from that block.

When client A pushes a message to the server on a block K, A encrypts the deletion (nonce, sig) pair for K, using B's public key. Thus, when B decrypts the message, he or she can send the server a delete request for block K and delete the information on block K.

This ensures that only the sender and recipient can delete information from the block that they are using. It also saves space on the server, since once the server receives a delete request, that block is no longer marked as reserved and other users can use it to send messages.

This deletion process uses more serial unlinked transactions. Hence, none of the delete requests can be traced back to a specific user or conversation and the server gains no additional information.

4.6 Other Cryptographic Mechanisms

The current implementation of Traceless does not include the deletion mechanism at the moment for simplicity and performance considerations.

5 Distributed Systems Discussion

Traceless implements some key distributed systems features on the client and server side. Each side is discussed below.

5.1 Client Side

The main distributed systems feature set is the retry function in case a request is dropped or is unsuccessful. The client uses a timeout based mechanism to detect when a packet is lost. Specifically, the code uses a unreliable parameter to simulate whether a packet has been dropped going to the server or on the way back. The client locks on key objects so that with multiple threads spinning there is no inconsistent state in the system.

In addition, the client side must be responsive to the server side implementation of sharding. Specifically, the client uses a thread to communicate with the server in order to update its view on which servers are responsible for which shard.

5.2 Server Side

The server side implements the majority of the distributed systems features. Namely, implementations of master-slave models, sharding of key value pairs, and primary-backup reliability.

5.2.1 Master Slave Model

The use of a master-slave model allows the system to maintain organization and privilege levels. The master is responsible for the user table, conversation table, as well as maintaining the shard organiztion. Specifically, with regard to shard organization, the master is responsible for assignment of shards as well as upgrading backups and changing to new views appropriately. The master communicates with the client as well as other slaves to make sure that their views are updated. Note that the client only gets a partial view in that they cannot contact backup shards. Further, the master is responsible for signing of blinded nonces including the initial of the slaves.

5.2.2 Sharding and Availability

The use of sharding is a key features of the system. Since this is a messaging platform availability is a key concern. In order to make sure that clients and push and pull in a timely manner, we implement sharding of the server message table so that slots are distributed across slave servers. Specifically, we shard a contiguous range of slot ids and assign to a slave server. The clients can then communicate directly with the slave server rather than going through the master. This allows for high availability of the system improving the overall performance of message pushes and pulls.

5.2.3 Primary Backup and Fault-Tolerance

The system uses a complete replication model as part of fault-tolerance. Specifically, if sufficient servers exist then a shard will both have a slave serving as the primary and the backup for the slot range it is assigned. When a message object is sent to the primary from the client, the primary will first send it to the backup, if one is assigned, to insert, and then insert into its own message table. In the case of the backup failing, the system proceeds as normal and assigns a new slave to serve as the backup if available. In the case that the primary fails then the master will designate the backup to be the successor primary for the shard.

6 Discussion

This section explores some of the security, performance, and limitations of the system. It concludes with possible future work. Figure **??** is a screenshot of Traceless.

6.1 Security

The server cannot discern the client from the requests as seen by the cryptographic encryption of the entire message data. With IP spoofing the clients would be relatively immune to discovery by the server. Some concerns are timing attacks which the solution might be to reserve blocks in a random or batched manner instead of when a message needs to be sent.

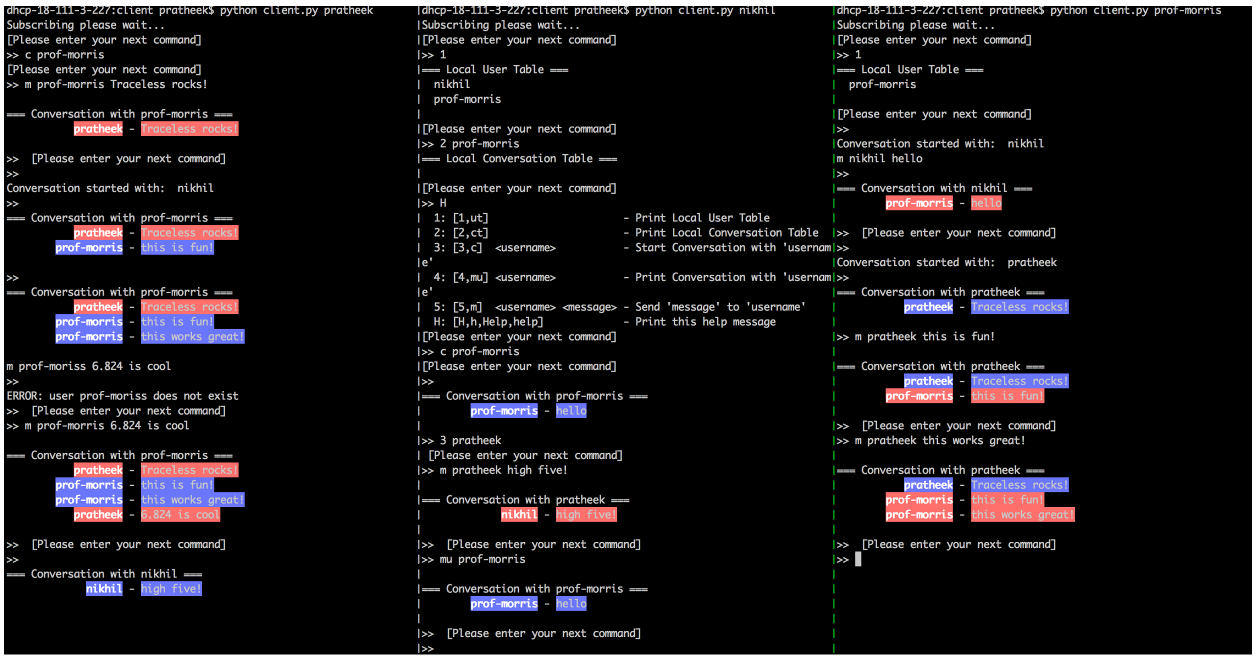
6.2 Performance

The system is perfomant in the small scale testing we provided. The use of threads allow the system to rapidly perform any command argument provided by a user while also updating user tables, conversation tables, message pulls, and server views. The parameters for waiting are simply relative weights on which threads should progress faster than others. The authors believe that we can reduce the latency and throughput of the system by both reducing the waiting time as well as optimizing the thread performance.

6.3 Limitations

Some limitations of the system include a possible master server bottleneck. In order to resolve this issue we consider splitting off some of the functions of the master into other slaves, such as the user table and conversation table. In addition, we could provide a read-only version of certain parts of the master.

Another limitation of the system is in the case of a complete shard partition or failure wherein both the primary and backup fail nearly simultaneously - or at least before the master can respond. In this case, Traceless is unable to recover from this state. The authors believe that this is more unlikely and suggest alternative sharding mechanisms such as partial sharding or distributing among more backups.



```
dhcp-18-111-3-227:client pratheek$ python client.py pratheek
Subscribing please wait...
[Please enter your next command]
>> c prof-morris
[Please enter your next command]
>> m prof-morris Traceless rocks!

=== Conversation with prof-morris ===
           pratheek - Traceless rocks!

>>  [Please enter your next command]
>>
Conversation started with:  nikhil
>>
=== Conversation with prof-morris ===
           pratheek - Traceless rocks!
        prof-morris - this is fun!

>>
=== Conversation with prof-morris ===
           pratheek - Traceless rocks!
        prof-morris - this is fun!
        prof-morris - this works great!

m prof-moriss 6.824 is cool
>>
ERROR: user prof-moriss does not exist
>>  [Please enter your next command]
>> m prof-morris 6.824 is cool

=== Conversation with prof-morris ===
           pratheek - Traceless rocks!
        prof-morris - this is fun!
        prof-morris - this works great!
           pratheek - 6.824 is cool

>>  [Please enter your next command]
>>
=== Conversation with nikhil ===
             nikhil - high five!
```

```
dhcp-18-111-3-227:client pratheek$ python client.py nikhil
Subscribing please wait...
[Please enter your next command]
>> 1
=== Local User Table ===
  nikhil
  prof-morris

[Please enter your next command]
>> 2 prof-morris
=== Local Conversation Table ===

[Please enter your next command]
>> H
  1: [1,ut]                       - Print Local User Table
  2: [2,ct]                       - Print Local Conversation Table
  3: [3,c]  <username>            - Start Conversation with 'usernam
e'
  4: [4,mu] <username>            - Print Conversation with 'usernam
e'
  5: [5,m]  <username> <message> - Send 'message' to 'username'
  H: [H,h,Help,help]              - Print this help message
[Please enter your next command]
>> c prof-morris
[Please enter your next command]
>>
=== Conversation with prof-morris ===
        prof-morris - hello

>> 3 pratheek
 [Please enter your next command]
>> m pratheek high five!

=== Conversation with pratheek ===
             nikhil - high five!

>>  [Please enter your next command]
>> mu prof-morris

=== Conversation with prof-morris ===
        prof-morris - hello

>>  [Please enter your next command]
>>
```

```
dhcp-18-111-3-227:client pratheek$ python client.py prof-morris
Subscribing please wait...
[Please enter your next command]
>> 1
=== Local User Table ===
  prof-morris

[Please enter your next command]
>>
Conversation started with:  nikhil
m nikhil hello
>>
=== Conversation with nikhil ===
        prof-morris - hello

>>  [Please enter your next command]
>>
Conversation started with:  pratheek
>>
=== Conversation with pratheek ===
           pratheek - Traceless rocks!

>> m pratheek this is fun!

=== Conversation with pratheek ===
           pratheek - Traceless rocks!
        prof-morris - this is fun!

>>  [Please enter your next command]
>> m pratheek this works great!

=== Conversation with pratheek ===
           pratheek - Traceless rocks!
        prof-morris - this is fun!
        prof-morris - this works great!

>>  [Please enter your next command]
>>
```

Figure 1: An image depicting Traceless with three clients.

6.4 Future Work

Some future work includes developing a better primary backup model so that slave count is reduced. At the moment a complete replication provides great write throughput but introducing a more distributed sharding mechanism will reduce the hardware cost of the system and serve as a smarter replication scheme.

In addition, the authors hope to add the reservation and deletion aspects into the cryptographic system. This would allow for better memory management. The theoretical derivation for cryptographic security is not presented here for brevity.

For feature set, we would like to explore the possibility of group conversations such that multiple users can communicate with one another simultaneously.

For further failure tolerance, we hope to implement a recovery system for clients such that they can retrieve their previous state from other clients and the server while not compromising the security of the system. The solution to this recovery mechanism is not presented here for brevity.

7 Conclusion

Traceless provides a mechanism for oblivious and secure messaging. The system is distributed in a manner that achieves high availability while ensuring fault tolerance.

8 Acknowledgements

The authors would like to thank Prof. Rivest for his instruction and thoughts on the project.

9 Repository

Traceless can be downloaded from the git repository provided below.

`https://github.com/pratheeknagaraj/traceless`

Several packages must be downloaded and configurations must be made so that the distributed system can work; please see the README on proper installation.

References

[1] Stuart G. Stubblebine , Paul F. Syverson , David M. Goldschlag, Unlinkable serial transactions: protocols and applications, ACM Transactions on Information and System Security (TISSEC), v.2 n.4, p.354-389, Nov. 1999