Automatic Detection and Prevention of Fake Key Attacks in Signal

Tarun Kumar Yadav

Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Physical Sciences and Mathematics Commons

BYU ScholarsArchive Citation


https://scholarsarchive.byu.edu/etd/9072

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Automatic Detection and Prevention of Fake Key Attacks in Signal

Tarun Kumar Yadav

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Master of Science

Kent Seamons, Chair
Daniel Zappala
Eric G. Mercer

Department of Computer Science
Brigham Young University

Copyright © 2020 Tarun Kumar Yadav
All Rights Reserved
ABSTRACT

Automatic Detection and Prevention of Fake Key Attacks in Signal

Tarun Kumar Yadav
Department of Computer Science, BYU
Master of Science

The Signal protocol provides end-to-end encryption for billions of users in popular instant messaging applications like WhatsApp, Facebook Messenger, and Google Allo. The protocol relies on an app-specific central server to distribute public keys and relay encrypted messages between the users. Signal prevents passive attacks. However, it is vulnerable to some active attacks due to its reliance on a trusted key server. A malicious key server can distribute fake keys to users to perform man-in-the-middle or impersonation attacks. Signal applications support an authentication ceremony to detect these active attacks. However, this places an undue burden on the users to manually verify each other’s public key. Recent studies reveal that the authentication ceremony is time-consuming and confusing, and almost nobody adopts it.

Our goal is to explore various approaches for automatically detecting or preventing fake key attacks. We modified a local copy of the Signal server to demonstrate that active attacks are feasible. We then designed three defenses that automatically detect or prevent the attacks. We completed a threat analysis of the defenses and implemented some proof-of-concept prototypes for two of them. We analyze their strengths and weaknesses and outline avenues for future work.

Keywords: Signal protocol, authentication, secure messaging
ACKNOWLEDGMENTS

I would like to acknowledge my parents for their support during graduate work and research. I would like to thank my advisor, Kent Seamons, for his constructive discussions and feedback throughout my research, which improved our designs and my writing style. I would like to thank Daniel Zappala for his valuable insights and questions which strengthened our designs. I would like to especially thank Austin Kolander and Aaron Chan for helping me with the prototype implementations.
# Table of Contents

List of Figures vi

List of Tables vii

1 Introduction 1

2 Background 4

2.1 Signal Protocol ........................................ 4
2.2 Authentication Ceremony .................................. 6
2.3 Prior work ............................................. 6

3 Threat model 9

4 Defenses 11

4.1 Trust Network (TN) ........................................ 11
4.2 Anonymous Key Retrieval (AKR) .......................... 15
4.3 Integrating CONIKS with Signal .......................... 18
4.3.1 Background ........................................... 18
4.3.2 Multi-provider scenario ................................ 19
4.3.3 Single provider scenario ............................... 20
4.4 Monitoring heuristics ..................................... 21
4.4.1 Mass key update monitoring ........................... 21
4.4.2 Isolation monitoring ................................... 22
4.4.3 Monitoring key-update history on every key change ... 24
5 Threat analysis

5.1 Trust Network ................................................................. 26
5.2 Anonymous Key Retrieval ............................................... 28
5.3 Key transparency ............................................................. 30

6 Evaluation

6.1 Implementation Details .................................................. 31
6.2 On the use of Trust Network ............................................ 32
   6.2.1 Memory requirements at Client Side ............................... 32
   6.2.2 Efficacy of Trust Network: Finding mutual contacts .......... 32
6.3 On use of Anonymous Key Retrieval .................................. 33
6.4 False positives ............................................................... 35

7 Discussion

7.1 Comparison of defenses .................................................. 37
7.2 Is automatic detection (or prevention) compulsory? ............... 37
7.3 Exhaustion of one-time pre-keys ...................................... 38
7.4 Pull vs. push-based approaches ........................................ 39
7.5 Attacks on multiple clients .............................................. 39

8 Conclusion

References

9 Appendix

9.1 Efficacy of Trust Network: Finding mutual contacts ............... 45
List of Figures

2.1 Key exchange process to setup a secure connection in Signal Protocol.............. 5
2.2 Authentication ceremony in Signal App.................................................. 7

4.1 Mutual Contact: (I) Alice’s contacts share the identity keys of their contacts to Alice. She maintains a database of her contacts’ contacts keys. (II) Alice queries the key server for Bob’s key. The key server responds with Bob’s key bundle. Alice compares the two set of keys (i.e., I and II). If a mismatch is found, an attack is detected......................................................... 12

4.2 Contact acts as a relay: Alice randomly selects one of her contacts and sends an encrypted key retrieval request for Bob’s key bundle. The contact relays the request to the key server and relays the response back to Alice. ................ 13

6.1 Time taken to a) retrieve keys through Tor circuit b) bring online the Tor service on a client ............................................................... 34

9.1 Histogram showing the distribution of the number of mutual friends among the users................................................................. 45

9.2 Histogram showing the distribution of the number of mutual friends among the users assuming random addition of friends...................... 46
List of Tables

5.1 Taxonomy of fake key attacks and an analysis of three defenses. Hollow circle represents that attack can be detected and filled circle represents that attack can be prevented. .................................................. 27
Chapter 1

Introduction

The Signal protocol is an end-to-end encryption protocol that provides forward secrecy and was designed specifically for messaging applications. It is used by billions of users in popular instant messenger applications like WhatsApp, Signal Private Messenger, Facebook Messenger’s “secret chat” feature, Skype, and Google Allo’s incognito mode.

In Signal protocol, clients are associated with unique “identity keys” i.e., public keys that Signal clients themselves generate and publish them on Signal’s key server. Currently, messaging applications that use Signal protocol, rely on a trusted key server for distributing identity keys among the communicating clients. When Alice wishes to communicate with Bob, she requests for Bob’s key from the key server (and vice-versa). This is vital step where a malicious key server (MKS) can launch a MitM attack against Alice and Bob, by providing them fake keys. Presently Alice and Bob have two options to authenticate each other’s key — either they trust the key server and believe that the keys are correct or they manually compare keys with their contact when they start communication. The same process can be repeated whenever the keys of the users are updated.

Recent user studies [11] suggest that mostly users are oblivious to authentication (i.e., identity key verification) and why it is needed. Thus, they do not perform the manual authentication ceremony with their contacts. Ideally, key verification must be performed on every key update. However, legitimate key updates occur only when contacts re-install the messenger app. Thus, convincing users to verify keys (on any key update notification)

\[1\text{https://signal.org/docs/specifications/x3dh/#keys}\]

\[2\text{This is possible as all the messages between Alice and Bob are relayed via the MKS}\]
would usually result in confirmation that the key updates were not due to some attack. Such repeated confirmations might cause user fatigue and a penchant to ignore key update warning messages. **Hence there is a pressing need to somehow distinguish between benign key updates due to app re-installation and fake key updates.**

The aim of our research is to automate the detection and prevention of fake key attacks in Signal. Based on our novel heuristics and approaches we aim to distinguish between actual key updates (due to re-installation of app) and fake key updates by MKS (with a malign intent to launch attacks). This would ease the burden from users to manually verify the keys they receive from the key server.

This paper presents the design and analysis of the following three defensive approaches that safeguard Signal users from fake key attacks by an MKS.

- **Trust Network:** Whenever Alice receives a key bundle (for Bob) from the key server, rather than trusting the keys directly, she checks if any of her contacts also have a copy of Bob’s keys. If the keys provided by the server do not match any copy of Bob’s keys from her contacts, Alice assumes she is under a fake key attack.

- **Anonymous Key Retrieval:** Currently, Signal clients authenticate to the key server before requesting other users’ keys. To prevent targeted fake key attacks, we propose the key server support anonymous key retrieval to make it harder for the key server to conduct targeted fake key attacks. Even though key retrieval is anonymous, all conversation messages between Alice and Bob still be authenticated as they flow through the key server.

- **Key Transparency:** CONIKS [6] is key transparency approach, where multiple identity providers (analogous to key server) facilitate the key verification process. A client can upload its key bundle to any of the provider, but can retrieve its (or some other client’s) key bundle any of the providers. If some malicious provider attempts to present fake keys, clients (or other providers) can cryptographically detect its malicious intents. However, at present, Signal based apps have only one provider. Thus, CONIKS
(a multi-provider system) cannot be directly used with Signal. In this research, we propose a modified CONIKS design that can be integrated with the Signal protocol.

We present a detailed design and analysis of these three approaches and describe how they prevent (or at least detect) fake key attacks by an MKS. These defenses can serve as a deterrent to the key server to ever willingly launch a fake key attack (e.g., respond to a government subpoena) and an incentive to guard against a hacker breaking into the server and launching a fake key attack that could damage the reputation of the key server.
Chapter 2

Background

This chapter describes the Signal protocol, our threat model for fake key attacks, and related work.

2.1 Signal Protocol

The Signal [3, 9] protocol (hereafter referred to as Signal) provides end-to-end encryption for several popular instance messaging applications, including WhatsApp, Signal Private Messenger, Skype, Facebook Messenger’s secret chat feature, and Google Allo’s incognito mode. Billions of users exchange instant messages using Signal. Each Signal app relies on a app-specific centralized server to store and distribute public keys and to relay messages between users.

When a client (Alice) installs a Signal app, she registers her identity with the key server and uploads some long-term, medium-term, and one-time use public keys. The three types of keys generated by Alice are as follows:

1. The identity key: a long-term public-private key pair associated with Alice that she uses to authenticate herself to other Signal clients.

2. The signed pre-key: a medium-term public-private key pair signed by the identity key and used to achieve forward secrecy.

3. One-time pre-keys: optional keys to achieve strong forward secrecy. Every pair of clients uses different one-time pre-keys to establish an end-to-end encrypted connection.
When Alice first communicates with Bob, she retrieves Bob’s public keys from the key server. Next, she computes a triple Diffie-Hellman key exchange using her public keys and Bob’s public keys to derive some initial symmetric encryption keys for establishing a session with Bob. She encrypts her first message to Bob with the session key and appends her key bundle (in plain text) to the encrypted message, and sends it to the Bob.

When Bob receives Alice’s message, he uses Alice’s key bundle and his public keys to compute the same session key that Alice used to encrypt the message. The process for creating the session is shown in Figure 2.1. The entire communication between Alice and Bob is relayed via the Signal server.

As Alice and Bob continue to exchange encrypted messages, they achieve forward secrecy through a double-ratchet algorithm that produces new symmetric keys as Alice and Bob send and receive messages. If a symmetric encryption key is ever compromised, only a single message is revealed. The attacker learns nothing about any prior encrypted message.

Figure 2.1: Key exchange process to setup a secure connection in Signal Protocol.
2.2 Authentication Ceremony

Since all the messages between any two users (including key exchange) are relayed via a central server, it can suppress the actual keys of the user(s) and present fake keys to them to launch impersonation and MitM attacks. To detect such fake key attacks, Alice and Bob can perform an authentication ceremony to verify that they both have the correct identity key. Most messaging apps provide an interface (see Figure 2.2) for Alice and Bob to scan a QR code from each other’s phone if they are co-located to verify they have the correct key, or they can read the safety number to each other over a voice call. The overall security of the system then depends on the users to verify their identity keys manually.

2.3 Prior work

Prior research shows that users do not understand the need for the authentication ceremony and find it difficult to perform. Schroder et al. [8] demonstrated that a majority of Signal users failed to correctly verify their conversation partner’s key due to usability issues and an incomplete mental model. Herzberg and Liebowitz [5] conducted a laboratory user study where they provided high-level information about the risks of secure communication. Only 13% of the users were able to complete the authentication ceremony successfully. Similarly, Vaziripour et al. [11] conducted a laboratory study where pairs of participants received high-level instructions to make sure they were communicating with the person they intended, and only 14% of the participants completed the ceremony.

More recently, several researchers have designed and evaluated improvements to the authentication ceremony interface [4][10]. Vaziripour et al. [12] modified the authentication ceremony by improving the UI of the Signal Application. They reported that 90% of the participants were able to complete the ceremony using the re-designed version of Signal. However, despite these improvements, there were a significant fraction of participants who
Figure 2.2: Authentication ceremony in Signal App
were still unsure (or confused) about the purpose of the authentication ceremony, and they spent an average of 11+ minutes finding and completing the ceremony.

Several research efforts have explored enlisting the help of third-party services during key verification to reduce the burden on users. Keybase is a key directory that links multiple social media accounts of a person to their encryption keys to increase the confidence in the authenticity of the keys \textit{i.e.,} keys belong to the person associated with social media accounts. Vaziripour et al. [13] attempted to semi-automate the authentication process in the Signal app using social media accounts, similar to Keybase. They found that automating the authentication ceremony and distributing trust with additional service providers is promising; however, the third-party actors need to be more trustworthy and accountable than social media companies.

There are several challenges with using social media accounts to help automate the key verification process. First, users are required to have additional accounts where their public keys can be published. Second, we assume the social media platforms do not collude with the messaging providers. For example, Facebook owns WhatsApp and is positioned to collude easily. Finally, the onus is on the user to verify that the social media account owner is the person they are trying to authenticate.

Key transparency is another approach to key verification. CONIKS [6] (and its successor [2]), is designed for each service provider to maintain an auditable directory of keys. Service providers monitor each other to detect equivocation, and users can monitor the correctness of their key in the directory. CONIKS was primarily designed to work with multiple service providers. However, in this research, we analyze how to adapt CONIKS with Signal (single provider) to detect fake key attacks.
Chapter 3

Threat model

Our threat model consists of a malicious key server (MKS) as an active attacker that delivers fake identity keys to clients. It must be noted that, first identity keys are exchanged, clients communicate with each other via a central key server, which only stores and relays information between parties, but does not perform any computation. There are two types of fake key attacks an MKS can launch:

**Impersonation Attack**  When Alice wishes to communicate with Bob, the MKS impersonates Bob by providing a fake key to Alice. The MKS communicates directly with Alice as if it were Bob. Bob does not participate in any of the communications between Alice and the MKS, and remains oblivious to the attack. This attack applies to both new and existing secure Signal connections. For new connections, when Alice retrieve’s Bob’s key bundle, the MKS simply provides fake keys for Bob. For an existing connection, the MKS must falsify a key update for Alice, indicating that Bob has changed keys, and providing the new key bundle. In the latter case, this will trigger a warning or notification for Alice, depending on the state of her conversation with Bob [14].

**Man In The Middle Attack (MitM)**  In this attack, the MKS impersonates Alice to Bob and vice versa. The attacker can read, modify, and inject messages into the conversation. The MKS is also positioned to impersonate Alice or Bob to each other and initiate a new conversation with just one of them.
To conduct this attack on a new connection, the MKS first generates two fake identity keys. When Alice attempts to create a secure Signal connection with Bob by retrieving Bob’s identity key, the MKS suppresses Bob’s key and presents one of the fake keys (as Bob’s key) to Alice. When Alice sends the first message to Bob containing her identity key, the server replaces Alice’s key with the other fake key. The MKS can also launch a MitM attack against clients with existing connections by sending fake key updates to both Alice and Bob.

Furthermore, both impersonation and MitM attacks can be launched in two different modes. In a *pair-targeted attack*, the MKS attacks a single pair of participants (Alice and Bob). In a *client-targeted attack*, the MKS launches the attack on Alice and all of her contacts.
In this chapter we first describe our three defensive strategies to safeguard against fake key attacks followed by some general monitoring heuristics that aid our proposed defense schemes.

4.1 Trust Network (TN)

With TN defense scheme, whenever Alice’s client receives Bob’s identity key from key server\(^1\) before trusting the keys directly, it verifies the authenticity of the keys from her “trusted” contacts. In TN we *assume* that Alice has few initial secure connection with some of her contacts after she joined the app’s network (we term such contacts as trusted contacts\(^2\)). Later, when she wishes to establish a secure connection with a new user (say Bob), she attempts to identify some trusted mutual contacts between her and Bob. If there exist some trusted mutual contacts between them, she directly confirms from them whether the received identity keys (of Bob) are correct. If there does not exist any mutual contact, she randomly selects any of her trusted contacts (*e.g.* Carol) to ask for Bob’s key on her behalf. If MKS was launching fake key attacks (*e.g.*, pair-targeted attacks) only for Alice and Bob, MKS would present correct keys of Bob to Carol’ client. Alice’s client on receiving the correct keys from Carol’ client, would detect that Alice is under attack.

We now describe our Trust Network defense strategy in detail.

---

\(^1\)The process is followed for first time key retrieval (*i.e.*, on new connection) as well as on every key update (*i.e.* for existing connections).

\(^2\)We suggest users resorting to manual key verification to have such secure E2EE connections.
1. *Populating the key dataset (contacts’ view):* Whenever Alice starts communicating with a new user, her client requests the new user’s client to send all of its contacts’ identity keys. For example, if Alice starts communicating with Bob, on receiving the request from Alice’s client, Bob’s client responds with the set of its contacts’ identity keys. This way Alice maintains the identity keys of all its contacts’ contacts.

Figure 4.1: Mutual Contact: (I) Alice’s contacts share the identity keys of their contacts to Alice. She maintains a database of her contacts’ contacts keys. (II) Alice queries the key server for Bob’s key. The key server responds with Bob’s key bundle. Alice compares the two set of keys (*i.e.*, I and II). If a mismatch is found, an attack is detected.

In our design, Alice’s friends return two entities corresponding to each of their contact — hash of the identity (*e.g.* phone number of the contact) and hash of identity key. The
use of hash function limits the privacy leak as Alice only learns about mutual friends instead of learning about all the contacts of her friends. She maintains a database that contain hash of the identity and identity keys of its contacts’ contacts. It must be noted that, whenever any Signal client would obtain a key update message (a push notification from key server) regarding a client C, it would send a database update message to all its contacts that it has new identity key of client C. Thus, Alice would always have the latest keys of its contacts’ contacts.

2. Alice before creating a new connection with Bob, or on obtaining a key update notification from MKS for Bob, tries to identify some mutual contact between the Bob and itself.

   (a) **Key from Mutual Contact:** If she finds some mutual contact (using the aforementioned database), she compares the key (from database) with the one provided by MKS in the previous step. On successful match, Alice is sure it’s a legit key update, else she is shown a warning that she is under attack.

   (b) **Contact as a relay:** If there does not exist any mutual contact, Alice randomly selects any one of her contacts (e.g. Carol) to act as a proxy for her. Alice asks
Carol to obtain Bob’s key bundle from the MKS (on her behalf). MKS which intends to launch attack against Alice, naively provides correct key of Bob to Carol. Carol in return relays Bob’s key to Alice. Alice compares this key with the one provided by MKS. On successful match assumes Alice assumes her to be safe, else she confirms that she is under attack. This process is shown in Fig 4.2

It can be argued that Carol can act maliciously. He can present Bob’s fake key to Alice to annoy her with false ”attack warning”. To thwart this situation, the key server generates a long-term asymmetric key pair and sends the public key to the clients at the time of registration. The key server signs the key retrieval response with the corresponding private key. Whenever a Alice retrieves Bob’s keys from Carol, she first verifies the key server signature with corresponding public key (of the key server) she owns. This prevents her malicious contacts from presenting a fake key. However, Carol can still identify that with whom Alice would communicate by reading the Alice’s request or Key server’s response. Henceforth, when Alice would relay her query via Carol, she would first generate a session key at her end only. Next, she would append the session key with the request (request——SessionKey) and eventually encrypting it with public key of the key server (\( Enc(request||SessionKey)_{KSPublicKey} \)). When Alice would relay this encrypted message via Carol to key server, she could not decrypt it. However, key server can successfully decrypts it. In return, key server would encrypt the response with the session key that was embedded in the request. This would prevent Carol from deciphering the response. In order to make relay request indistinguishable from normal request, all the Signal clients would include a self-generated session key in normal requests for key bundle as well.

A client’s key bundle from the key server has a timestamp in it. The *timestamp* is the time at which a client uploaded its key to the key server. Every key a client receives either while populating the key dataset from friends or through a relay comes with an associated
timestamp. To verify a key, they first compare the timestamps. If timestamp matches, then they match the keys for verification.

4.2 Anonymous Key Retrieval (AKR)

Currently, Signal clients authenticate to the key server before retrieving the keys of other clients. We propose that clients be able to retrieve keys from the key server anonymously. To do so, clients need not send their unique identifier at the time of key retrieval. Thus, if an MKS intends to launch a targeted attack on Alice, it could not — it is not possible to distinguish Alice from other clients. This would prevent MKS to present fake key bundle to Alice, preventing the attacks.

Even though Alice does not identity herself to the key server, the server may attempt to identify Alice with metadata information. For instance, Alice may regularly uses the same IP address when connecting to the key server. To address these concerns, our design assumes an anonymization service such as Tor [1]. It has a robust architecture to provide anonymity to its users. Clients always query the key server for a key bundle using Tor. This makes it much harder for the key server to know when Alice is requesting a key bundle. This makes it extremely difficult for an MKS to launch pair-targeted attacks against Alice and Bob, as the server does not know the identity of the key bundle requester.

If the key server wishes to execute a client-targeted MitM attack on Alice, it must present Alice’s fake keys to all her contacts (and vice-versa). Key server can easily present Alice’s fake keys to everyone. However, it is difficult for the key server to present a fake key of Alice’s contacts to her as Alice would anonymously query the MKS for key retrieval. This would prevent such an attack.

In Signal protocol when Alice starts communication with Bob, Alice retrieves Bob’s keys from the key server and sends her key along with the first message to Bob. (Bob need not retrieve keys from the key server). At this step, the key server can launch a client-targeted MitM even if Alice uses Tor to retrieve Bob’s keys. When Alice requests for Bob’s key (even
via Tor), key server can present Bob’s fake keys to her. Next, When Alice sends the first message to Bob, key server can replace Alice’s keys with a fake key and can successfully execute MitM attack. Thus, in defense we propose that, instead Alice sending her key in the first message (to Bob), Bob should also retrieve Alice’s key from the key server using Tor. This would render MitM attack by MKS ineffective, as MKS is unaware of both Alice’s and Bob’s identity.

Presently, all communication (key exchange and actual chat) between Alice and Bob using Signal involves the key server as a relay. Thus, the key server is well-positioned to launch fake key attacks to the communicating parties. If there is some third party trusted channel to verify keys, such attacks can be prevented (or at least detected). We now explain why a third party communication channel is needed.

**Requirement for a secure third party channel:** The MKS can execute an attack on the established E2EE connections. Assuming Alice and Bob are not communicating with each other (but have an established E2EE connection), MKS can notify one or both of the clients a fake key update (with new fake keys). Both the clients before trusting the new keys would query the MKS for each others keys anonymously (via Tor). MKS being oblivious to the process would present correct keys as a response to the anonymous requests. After obtaining the correct keys both the clients can match and verify the keys. After confirming that keys are correct they can resume their communication, thus preventing the attack. However, If Alice and Bob are communicating with each other, the aforementioned approach could be susceptible to timing attacks. A determined adversary may exacerbate the attempts made by clients to obtain correct keys. It might correlate — the moment it sent fake key update notifications to the client, within a very short duration it received anonymous requests for key retrieval of the same two clients from Tor exit nodes. Assuming that same clients (under consideration) are trying to access keys via Tor, it sends with fake keys as a response to the anonymous requests as well. In order to defend against such a timing attack, we propose
to use a third party service \textit{i.e.}, \textit{Tor onion services}. Tor onion (or hidden) services are TCP-based network services that can only be accessed via Tor network. Tor clients access onion services via domain names that can only be resolved and accessed within a Tor network \cite{1}.

In our design we propose that all Signal clients should have onion services corresponding to their Signal account. When two Signal clients establish E2EE connection with each other, they securely share their onion \textit{urls} with each other.

In a scenario when Signal clients with existing E2EE connections receive new set of keys as key update message (from the MKS), rather than directly trusting the keys, a soft warning message would appear on screens of the device — “Do not share any sensitive information for some time until the verification process gets completed in the background”. The key verification process is as follows:

- Both clients turn on Tor onion service for some time (few minutes).
- Both clients request each other’s hidden service for the updated keys.
- Each of them matches the two set of keys; one obtained from MKS (as key update) and second obtained from the hidden service (of the other client).
- If there is a mismatch the client is alerted for the potential attack, and the key verification fails.

In a scenario where a legitimate key update message is received at least one of the clients have reinstalled the application. The client(s) who reinstalled the application would not have their onion services running as the reinstalled application has no information about the associated onion service. In such a case when the other communicating client would request the onion service for the set of keys, she would not be able to access it. This would give her the indication that she received the correct keys from the key server.

Moreover, all Signal clients are required to perform \textit{periodic key confirmation} by running their Tor services multiple times a day, such that they can publish their latest
identity keys. For instance, Suppose Alice decides to run her service three times a day\(^3\). She would publish her latest identity key such that Bob can use her Tor service to confirm that he is using the correct key for Alice.

4.3 Integrating CONIKS with Signal

CONIKS is a generic key verification system that uses key transparency and auditing to ensure that identity providers cannot equivocate about the public keys they advertise on behalf of users. The idea of key transparency is to commit an identity provider to advertising a single, consistent set of public keys, thus enabling those using the identity provider to monitor its advertisements for equivocation. This acts as a deterrence for identity providers announcing one public key for Alice but then secretly using another in a MitM attack.

4.3.1 Background

CONIKS has four types of actors: (1) Identity Providers are key distribution servers that advertise public keys of their clients. (2) Clients run software that relies on public key cryptography, such as a secure email system, and regularly communicate with identity providers to retrieve or monitor keys. (3) Auditors verify that identity providers are not equivocating and communicate with each other to ensure global consistency. Clients serve as auditors for their own identity provider. Identity providers can act as auditors for each other. (4) Users: people who use CONIKS for key verification.

When a user installs the CONIKS client, it generates its own private-public key pair and uploads the public key to some identity provider. The provider generates a Merkle binary prefix tree of all registered keys of its clients. The leaf node of the tree are keys of clients, including metadata bound to the key. The interior nodes of the tree are the hash of the content of the child nodes. At regular intervals, called epochs, the identity provider signs the root of the Merkle tree, along with metadata that includes the current epoch and the

\(^3\)She would notify all her contacts about the up time of her Tor service first time when she would establish a E2EE connection with them.
hash of the previous epoch’s signed tree root. Notice that since the STR includes a hash of the previous epoch’s STR, the provider commits to advertising a consistent history of key bindings.

When a client (or some provider) requests the public key of a client from the identity provider, it responds with the current signed tree root (STR) and some necessary subset of the Merkle Tree, called the authentication path. The requester places its key as a node in the tree at the appropriate location in the authentication path, then generates the root of the tree and compares this root with the STR given by the provider. Thus, the STR with an authentication path act as a proof of inclusion of the client’s public key by the provider. If a client discovers misbehavior by a key server, in the form of inconsistent STRs for the same epoch, then they can alert auditors and other clients and include the proof of the inconsistent STRs.

4.3.2 Multi-provider scenario

In general, the CONIKS architecture assumes there are multiple non-colluding identity providers. However, in messaging apps like WhatsApp and Signal there is only one key server that can act as a provider, and there is no existing consortium that provides non-colluding distributed entities that could act as identity providers. To integrate CONIKS with Signal, we first describe a hypothetical scenario that assumes the existence of third-party non-colluding providers. We describe how automated key verification using CONIKS would work with Signal in this case and explain the challenges involved.

In a multi-provider scenario, Alice chooses an identity provider and her client uploads Alice’s key bundle to this provider. The provider updates its Merkle prefix tree to include this new key and generates a fresh STR. At the next epoch, it makes this STR available to other providers for auditing for non-equivocation. The recipient provider, acting as an auditor, first confirms that the received STR has been correctly signed by the Alice’s provider. Next, it performs history verification i.e. whether the present STR contains the hash of the
previous STR from Alice’s provider. If both conditions are met, the auditor accepts the STR of the Alice’s provider.

When Bob wishes to communicate with Alice, his client queries several random auditors for the latest STR it received from Alice’s providers. Later, Bob does a key lookup for Alice. It compares the just received STR with the STRs provided by auditors. If they match, Bob trusts Alice’s keys and commences encrypted communication with Alice. Additionally, at every epoch all CONIKS clients perform key monitoring by requesting their own keys from their own provider to verify that their keys are not changed without their consent.

This approach will be able to detect equivocation by an identity provider; however, it does have several limitations. First, if Alice receives a key update message from its provider for Bob (with which she has existing E2EE connection), she cannot verify the new keys until the next epoch. At the upcoming epoch, Alice can verify the new keys by querying other providers for Bob’s key. Thus, key verification using CONIKS would result in delayed detection. Second, CONIKS can not distinguish between a MKS that creates a new key for Bob (and does not equivocate), as opposed to Bob changing his key and then falsely accusing the identity provider of being malicious. There is no cryptographic proof that Bob can offer to prove that an identity provider was malicious, as opposed to Bob simply re-installing the application. One might think this could be solved if Bob was required to digitally sign new keys with the old key. However, a legitimate key update message is generated only when the app needs to be reinstalled, meaning the original key is lost.

4.3.3 Single provider scenario

It is more realistic that secure messaging applications will continue to operate with a single provider. In this case, we propose delegating clients as auditors. At every epoch, a client can check for equivocation by comparing the STR they obtain from the central key server with the STR provided by its contacts. Similarly, clients monitor their own keys for inclusion in the Merkle tree advertised by the key server.
Because there is a single provider, it is important that clients can exchange STRs at every epoch and that they are not blocked by the provider. For example, a provider could identify STR exchanges if they are exchanged at regular, predictable intervals. A client should consider the blocking of all auditing messages as malicious behavior because it is unlikely that all of its contacts are disconnected from the Internet at the same moment. To avoid extensive blocking we suggest that clients can exchange STRs use randomized delays each epoch or piggyback the STR with regular messages.

4.4 Monitoring heuristics

As we will see subsequently, the defenses we describe may not be able to detect or prevent all attacks. Thus, to augment these defense we now describe our three heuristics that aid our aforementioned defense schemes to defend against fake key attacks.

4.4.1 Mass key update monitoring

If an MKS wishes to launch a client-targeted MitM attack on Alice within a short duration, it needs to send fake key updates to Alice, resulting in large number of key updates within a very short duration of time. A legitimate key update only happens, when a user re-installs app. The attack is trivial to detect — it is highly unlikely that a high percentage of Alice’s contacts would simultaneously re-install the application resulting in a burst of key update messages. It will be more stealthy to spread this attack over a long period of time, during which Alice may ignore occasional notifications about key changes or assume this is normal.

Suppose an MKS launches client-targeted attacks (both MitM and impersonation) on existing connections. For the impersonation attack, assume the MKS intends to impersonate all of Alice’s contacts to Alice. To succeed, the MKS needs to send fake key updates to Alice for all her contacts. If a naive MKS sends all the fake key updates to Alice at once, Alice would trivially detect the attack because legitimate key updates occur only when a user re-installs the app. It is highly unlikely that all of Alice’s contacts would simultaneously
re-install the application and generate such a burst of key updates. If the MKS tries to be
more stealthy and spread out the key updates using a just-in-time approach, Alice could
eventually detect the attack is in progress by observing that each new conversation with an
existing contact coincides with a key update message. NOTE: I think the naive case is an
effective deterrent and an MKS will avoid it. The stealthy case is more realistic, but this
means Table 1 may need to be changed to detection for monitoring key updates.

4.4.2 Isolation monitoring

If an MKS wishes to launch a client-targeted impersonation attack on Alice, the MKS must
isolate Alice by disrupting all her outgoing messages and not forwarding incoming messages to
her. If Alice is unable to connect to all of her contacts, it’s possible there is a client-targeted
impersonation attack in progress. It is also possible that Alice herself is not connected to
the Internet. Thus, before concluding that Alice is under attack, the client first checks for
general Internet connectivity.

To detect this attack, Alice’s client regularly monitors for complete isolation by sending
a connection verification message to some of her contacts’ clients at regular intervals, called an
isolation monitoring interval. If none of her contacts’ client’s responds within a specified time
limit, Alice is alerted. It must be noted that Alice should only consider herself non-isolated
if she gets a response from a contact whose key has not been changed during the present
isolation monitoring interval. This precaution is necessary since an MKS can issue a fake key
update messages and then impersonate one of Alice’s contacts and issue a fake connection
verification message.

Determining the exact time period for isolation monitoring is not trivial. In a recent
study (involving 440 mobile devices) [], researchers reported that the probability that a
disconnected interval for a device lasts for less than one minute is 0.68, less than one hour
is 0.96, and less than one day is 0.998. It can be abbreviated as $DI_{time}$. For example,
$DI_{1hr} = 0.96$ If the time period for isolation monitoring is $t$, then the probability that the

single contact might not respond is $1 - DI_t$, and the probability that the ‘$n$’ contacts might not respond is $(1 - DI_t)^n$.

For example, assuming the time period for isolation monitoring is 1 hour and Alice sends isolation verification message to 3 of her friends (every interval) — the probability that none of the contacts would reply to Alice that interval is $(1 - 0.96)^3 \text{ i.e. } 0.00006$. If she has more contacts, than the probability would approach zero. If no response arrives at Alice from any of her contacts during a monitoring period, it is considered an attack. But, this might happen due to some network connectivity issues, and our approach would incorrectly classify this as attack. However, the probability of falsely determining isolation monitoring as an attack is negligible. Even with $n = 2$ (i.e. only two contacts are selected for verifying isolation), the attack can be detected in an hour with probability 0.9984.

However, to be absolutely sure that Alice has been completely isolated, Alice’s client must verify isolation with all of her contacts. Moreover, we must ensure that Alice’s client must not increase the load on the app’s network by sending verification messages to all of its contacts at once. Thus, to distribute the load, we further divide the isolation monitoring period into different intervals. In first interval, Alice’s client would send verification message to few of her contacts. If Alice’s client does not receive a response from any of her previously selected contacts within the first interval, it sends an isolation verification request to more random contacts in the next interval. Alice’s client keeps increasing the number of contacts exponentially (every interval) for sending verification message. The process is halted when either Alice’s client receives a response from any of the previously selected contacts or no more contacts are left for sending the verification message.

The number of intervals are decided by the following formula:

$$\sum_{i=1}^{I} K^i = |Total\ Contacts|$$

(4.1)
Here \( K \) is some constant, \( \text{Total Contacts} \) is the total number of Alice’s contacts with unchanged keys, and \( I \) is the total number of Intervals to be found. Moreover, \( K^i \) represents the number of contacts selected for sending verification message in the \( i^{th} \) interval.

For example, if we assume \( K \) to be 2 and Alice has 30 \( \text{Total Contacts} \), then \( I \) would be computed as 4. If isolation monitoring period is selected as one hour, then it would be divided into 4 intervals each of 15 minutes. In the first interval, a verification message would be sent to 2 random contacts; if no response is received in the next interval, verification messages are sent to 4 other random contacts and then to 8 and 16 random contacts in subsequent intervals.

If Alice’s client did not receive a response from any contact in all the intervals, it confirms that Alice is completely isolated — i.e., she could not communicate with any of her existing contacts in the isolation period.

4.4.3 Monitoring key-update history on every key change

In some of our proposed defensive approaches, key verification may take significant time. An MKS could launch an attack for a relatively short duration, and then perform a key update to restore the correct keys to the clients. By the time actual key verification process completes, the clients might obtain the correct keys. As a result, the clients would assume they are safe and the attack never happened. A vigilant MKS could always conduct short duration attacks that would not be easy to detect by our proposed approaches. As a fail-safe mechanism, we suggest that the Alice share Bob’s key-update history with Bob, after receiving the key-update message (new keys).

For example, assume Alice already has Bob’s identity key \( K1 \). The MKS sends a fake key update message (regarding Bob) to Alice. Now, Alice has fake key \( FK \) as the new identity key for Bob. Next, Alice sends a key-update history (that Alice owns) to Bob. This message contains the old key(s) of Bob i.e. \( K1 \). Since the MKS intercepts the message, it either simply drops the message or responds with a confirmation message that Alice has
correct keys. At some later time, when the MKS terminates the attack, it provides Bob’s correct keys to Alice. If Bob has re-installed the app by that time and generated a new key \( K^2 \), it sends \( K^2 \), else it provides the old key \( K^1 \). The next time Alice sends Bob’s key-update history to Bob (with the date and time at which Alice received the keys), she includes keys \( K^1, FK, K_1 t \) and \( K_1 t \). On reception, Bob compares the key history timestamps obtained from Alice with the timestamp when Bob installed the app (\( I_t \)). If there is a key update in key update history, after \( I_t \), it means there has been an attack. Bob will be alerted and Bob’s client also sends an alert message to Alice, confirming the attack.
Chapter 5

Threat analysis

This chapter contains an analysis of the three defenses against the various fake key attacks and whether the attacks are prevented or detected, as shown in Table 1. Prevention occurs when Alice is able to determine a priori that an MKS has provided her with a fake key for Bob and she refrains from communicating with him. Detection occurs when Alice is already communicating with Bob and later determines that she is under attack.

5.1 Trust Network

The trust network (TN) approach defends against both MitM and impersonation attacks. In most cases, it prevents the attacks, and in other cases it only detects them.

The TN prevents all pair-targeted attacks (Rows 1, 2, 5, 6) and client-targeted attacks for new connections (Rows 3 and 7) when Alice has a mutual contact with Bob that has previously sent her his correct keys.

If Alice does not share a mutual contact with Bob, she can request that one of her contacts (Connor) obtain Bob’s key from the MKS. If Connor queries the MKS for Bob’s key immediately after Alice asks him, the MKS might suspect that Connor is asking for Bob’s key on behalf of Alice by correlating Connor’s request with Alice’s message to Connor. To prevent the correlation, Connor can pause for a random period of time before querying the MKS for Bob’s key. Meanwhile, Alice has two options — (1) if she opts for prevention, she must wait for Connor to reply with the correct key, or (2) she can opt for detection and continue her communication with Bob using the key obtained from the MKS without waiting.
<table>
<thead>
<tr>
<th>ATTACKS</th>
<th>DEFENSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of attacks</td>
<td>Mode of attacks</td>
</tr>
<tr>
<td>MITM</td>
<td>Pair Targeted</td>
</tr>
<tr>
<td></td>
<td>Existing</td>
</tr>
<tr>
<td></td>
<td>New</td>
</tr>
<tr>
<td></td>
<td>Existing</td>
</tr>
<tr>
<td>Impersonation</td>
<td>Pair Targeted</td>
</tr>
<tr>
<td></td>
<td>Existing</td>
</tr>
<tr>
<td></td>
<td>New</td>
</tr>
<tr>
<td></td>
<td>Existing</td>
</tr>
</tbody>
</table>

Table 5.1: Taxonomy of fake key attacks and an analysis of three defenses. Hollow circle represents that attack can be detected and filled circle represents that attack can be prevented.
for Connor’s response. Once Connor responds, Alice can detect whether or not she was under attack.

Suppose an MKS launches a client-targeted MitM attack against all of Alice’s existing connections (Row 4). If Alice receives all the key updates for her contacts within a very short duration of time, she may have no contact remaining to act as a relay to query the key server. Mass key update monitoring can detects this attack (see 4.4). For client-targeted impersonation attacks against existing connections (Row 8), mass key update monitoring and isolation monitoring detects these attacks (see 4.4).

5.2 Anonymous Key Retrieval

AKR prevents a pair-targeted MitM attack for both new or existing E2EE connections. Using AKR, Alice and Bob can anonymously request each other’s key from the MKS. Anonymous key requests make it difficult for the MKS to reliably present fake keys to just Alice and Bob and no one else. Thus, a pair-targeted MitM attack for new connections is prevented (Row 1).

If an MKS launches this attack on an existing connection (Row 2), Alice and Bob will query each other’s Tor onion service after receiving the key update message (see 4.2). A mismatch confirms that they are under attack, and they could stop using the app to communicate to thwart the attack.

To launch an MitM attack, the MKS needs to present fake keys to both of the communicating parties. For a client-targeted attack against new connections, the MKS must present fake keys to Alice’s contacts and vice-versa. When Alice’s contacts anonymously request Alice’s keys from the MKS, it can easily present fake keys for Alice. However, when Alice anonymously requests keys for any of her contacts, the MKS has no guarantee a request is from Alice and not someone else desiring to communicate with her contacts.

This would render a client-targeted MitM for new connections ineffective (Row 3). However, a client-targeted MitM attack for existing connections can be prevented (Row 4) with
the use of Tor services (Row 2 explanation applies here also). If the MKS presents a fake key update for Alice to Bob, Bob can verify the new key bundle by accessing Alice’s Tor service.

AKR prevents pair-targeted impersonation attacks for new connections (Row 5) as well — MKS cannot distinguish a particular Signal client from all. However, a pair-targeted impersonation attack for existing connections can only be detected (Row 6). In this case, Alice and Bob already have an established connection. Assume the MKS attempts to impersonate Alice to Bob. It sends a fake key update message to Bob (but not to Alice). Before trusting the keys, Bob attempts to access Alice’s Tor service. However, since Alice has not received the key update message, she would not run her Tor service. The absence of a Tor service reply indicates to Bob that he received a legitimate key bundle (see 4.2). However, Alice and Bob are actually under attack. To detect such an attack, we propose a periodic key confirmation. Later, when Alice runs her Tor service and publishes her latest identity keys, Bob would retrieve her latest key from her Tor service. By comparing the identity keys that he initially received through the key update and to those he retrieved later through the Tor service, Bob detects that he is under attack.

However, a client-targeted impersonation attack on new connections can be detected (Row 7) through monitoring by having Alice anonymously request for her key bundle from the MKS (Anonymously monitoring for own keys). When the MKS receives her AKR request, it responds with the fake key bundle to Alice as well since the attack consists of the MLS impersonating Alice to all her contacts. However, the MKS cannot reliably distinguish between requests from Alice and her contacts. Thus, Alice confirms that she is under attack. Later, using off-line communication, she can notify all of her contacts that she is under attack.

For client-targeted impersonation attacks against existing connections (Row 8), mass key update monitoring and isolation monitoring detects these attacks (see 4.4). When the MKS impersonates Alice to all her contacts, Bob can confirm Alice’s key from her Tor service (at pre-defined times).
5.3 Key transparency

In general, for both MitM and impersonation attacks, key transparency schemes only detect the attacks. If an MKS presents fake keys to one or both clients, it is detected at the next epoch through auditing for non-equivocation or key monitoring. In a multi-provider scenario, the providers perform the auditing, whereas, in a single-provider scenario, the clients perform the auditing.

For a pair-targeted MitM attack for both new and existing connections (Rows 1 and 2), all clients except Alice and Bob obtain the same STR. Auditors would recognize that the MKS has provided Alice and Bob with an incorrect STR, and thus incorrect identity keys. Client-targeted MitM attacks on new connections (Row 3) can also be detected by auditing. If Alice wishes to establish a new connection with Bob, MKS would present fake keys and an incorrect STR to both of them. Both Alice and the new client can detect they have received the incorrect STR by asking the STR from the other clients.

A client-targeted MitM attack on existing connections (Row 4) can be detected using mass key update monitoring (see 4.4).

A pair-targeted impersonation attack for both new and existing connections (Rows 5 and 6) is also detected by auditing. If MKS impersonates as Alice to Bob, Bob would obtain fake keys with incorrect STR. As a result, auditors would easily identify the mismatch between the STR obtained from Bob and other clients. Client targeted impersonation attack on new connections can also be detected using auditing (Row 7).

For client-targeted impersonation attacks against existing connections (Row 8), mass key update monitoring and isolation monitoring detects these attacks (see 4.4).
Chapter 6

Evaluation

6.1 Implementation Details

We now describe the implementation details of each of various entities involved in our experiments. All the in-lab machines, had Intel(R) Core(TM)i7-7700K CPUs @ 4.20 GHz, provisioned with 16 GB RAM. For implementing the key server, we used open-source Signal server OWS 1.88 and for implementing clients we relied on Signal-Android 4.23.4. We modified the Signal key server to launch MitM (or impersonation) attacks. We added a client module to the key server such that it can perform the standard ratcheting encryption (like general Signal clients). We used Signal client CLI\(^1\) for the same. To successfully launch attacks, the messages of the victim(s) client are redirected to MKS’s client CLI, where it successfully performs encryption (or decryption) operation on the received messages.

For the Trust Network’s prototype, we mostly made changes on the android Signal client. All the verifying messages are sent exactly in the same way a standard communication message would be sent.

For the AKR prototype, we use the Tor Android Library\(^2\) at the Signal client. To anonymously retrieve keys, the client sets up a Tor circuit and retrieves keys using that circuit. We used TinyWebServer as a local server where identity keys of the users are available (on an android device) and then proxy server’s traffic via Tor SOCKS proxy.

---

\(^1\)https://github.com/AsamK/signal-cli
\(^2\)https://github.com/jehy/Tor-Onion-Proxy-Library
6.2 On the use of Trust Network

To test the feasibility of adopting the TN-based defense, we set up five lab machines as Signal clients (modified for Trust network) and one machine as Signal server (modified to launch fake key attacks). Signal clients communicated with each other via the Signal server. Our Signal server (using client CLI) launched both MitM and impersonation with all the variations shown in the Table 5.1. We were successful in automatically defending the clients against the fake key attacks.

6.2.1 Memory requirements at Client Side

In TN, every Signal user is supposed to maintain a database consisting of hashed identities (e.g. phone numbers) and identity keys of its contacts’ contacts (refer to chapter 4.1 for details). It can be questioned that what would be the memory overhead of storing such information in the mobile devices (considering their limited memory).

We assume a well established network; on an average each user has 200 contacts. To store the information for one contact in a database entry requires 72 bytes — 32 bytes (identity hash) + 32 bytes (identity key hash) + 8 bytes (timestamp). Bob’s client (which has an established E2EE connection with Alice’s client) would send the 200 database entries (corresponding to each of his contacts) to Alice. Similarly, Alice would receive database entries from all of her 200 contacts. Thus, Alice would need to maintain a database of only 2.88MB (200 * 200 * 72 bytes).

6.2.2 Efficacy of Trust Network: Finding mutual contacts

The trust network (TN) approach prevents an MKS from launching an attack on Alice and Bob only if they have some mutual friends. How likely is it that two friends on Facebook have some mutual contacts?

To answer this question, we analyzed a Facebook dataset[7] that consists of ≈ 0.6 million users. The dataset contains a graph G(V,E), where V is the vertices representing
users and E is the edges representing the friend relationship between two users. For our analysis, we selected users who had 10–500 friends. We refrained from selecting users with more than 500 friends since they could potentially be spammers.

From our analysis, we found that an average user has at least one mutual friend for 75 percent of their connections. The average number of mutual friends for these connections is 6.25.

Next, we considered a scenario where a new user (e.g. Alice) joins the aforementioned social network and starts building her own friend network. We analyzed that, what is the fraction of Alice’s friends, with whom she has at least one mutual friend.

For this, we isolate an existing user from the given dataset by removing all its edges from the given graph. For example, we removed Alice from the graph, and simulated a scenario, where she starts building her friend network from the beginning. In every iteration, we created an edge between Alice and any one of her contacts (Bob), that has not been visited yet. We calculated whether she has mutual contact(s) with Bob or not? If yes, then we counted the number of mutual contacts. In the end when Alice ends up having edges with all her contacts, we calculated the fraction of Alice’s friends with whom she has mutual contact(s) and also the the average number of mutual contacts she has with them.

We repeated the same process for all the selected users of the database and calculated the averages. We found that a user has at least one mutual friend in 64% of their connections while creating their friend network. The average number of mutual friends for these connections is 5.7.

6.3 On use of Anonymous Key Retrieval

Again, we used the five in-lab machines as Signal clients (modified for AKR) and one computer as a Signal server for launching fake key attacks. For AKR, our Signal clients anonymously queried for the identity keys of the other clients. Our requests did not include the sender’s
unique identity, and Signal clients sent such key retrieval requests to our Signal server via Tor. We were able to prevent/detect the attacks presented in Table 5.1 successfully.

Additionally, it can be argued that accessing keys via Tor (or Tor service) could be time-consuming. Thus, to measure the average time required to obtain a key bundle from the key server through Tor, we requested a key bundle through Tor 100 times per day for ten days using both WiFi and 4G networks. The time includes a) setting up Tor circuit and b) retrieve Keys from the key server using that circuit. Figure 6.1 shows the results of our experiment. Over WiFi, it took an average of 10.4 s to retrieve keys from the key server using Tor ($AKR_{WiFi}$); it took an average of 11.88 s over a 4G network ($AKR_{4G}$).

![Figure 6.1: Time taken to a) retrieve keys through Tor circuit b) bring online the Tor service on a client](image)

Moreover, to defend against attacks on existing connections, a client is required to host a Tor onion service on its mobile device to aid the key verification process. A client’s Tor
service is offline most of the time, and it is brought online during a particular time of the day.

To measure the average time required to bring a hidden service online on an Android device, we brought up a Tor service 100 times a day for ten days (both on 4G and WiFi). The time in bringing a Tor service online includes a) setting up a local server, b) setting up a Tor circuit, c) publish the Tor service’s identity in the Tor network, such that some Tor relay could act as an introduction point to the Tor service. Figure 6.1 shows the results. The average time taken to make Tor service accessible is 8.4 seconds on WiFi (TorServiceUP_WiFi) and 8.5 s on 4G (TorServiceUP_4G).

6.4 False positives

A false positive is a mis-classification of a legitimate key update as a fake key attack. None of the three proposed defense schemes are susceptible to a false positive.

In the Trust network, a Signal client verifies the key bundle provided by MKS to the one obtained from the Trust network — either by using (1) keys from mutual friends or (2) from contact as a relay. In both the cases, the keys a client retrieves have a “timestamp” associated with them. (The timestamp is the time at which a user has uploaded her key to the key server.) The two sets of key bundles — one from the MKS and other from the Trust network, are matched if they contain the same timestamp value. Thus, it prevents against raising the false alarm.

In AKR, we relied upon (1) anonymous key retrieval and 2) keys from Tor service; for key verification. In the first case, the keys are retrieved anonymously from the key server. The key server contains the latest copy of the legitimate keys, and thus would provide the correct key bundle to the anonymous requester. For the second case, key owners manage the tor services. Naturally, they would maintain their latest copy of the key bundle on the Tor service.

Additionally, the clients also opt for regular key monitoring i.e., they anonymously ask for their own keys from the key server, to verify that the key server hosts their correct
and legitimate key bundle. All these preventive measures ensure correctly identifying the legitimate key updates.

In CONIKS, when there will be a legitimate key update, and the non-malicious key server presents correct STR to every user, the STR of all the users would be the same, and the auditing process would not raise any alarm.

However, there can be some false positive in our Isolation monitoring heuristic. In case the messenger server is not responding for a prolonged period, the client would assume that they are under attack, as they are unable to communicate with all of her contact for a long period.
Chapter 7

Discussion

7.1 Comparison of defenses

The three defensive approaches have different strengths and weaknesses. Using the TN approach, if Alice has some mutual contacts with Bob, she can prevent a fake key attack without significant delay. She compares the key the MKS provides with the keys for Bob that she obtains from her mutual contacts. However, this approach introduces privacy concerns since Alice now knows all the mutual contacts she has with all her contacts. A privacy-conscious user might refrain from adopting TN as it would leak her complete contact network to all of her contacts.

On the other hand, AKR does not have these privacy concerns but does introduce a delay in the prevention (or detection) of the attacks. There are overheads like running a Tor hidden service at specified times every day and obtaining the keys of the users via Tor. These overheads might be unacceptable to delay-sensitive users. Key transparency approaches like CONIKS aim to detect the attacks. There is a deployment challenge due to no existing systems that rely on such approaches for key verification. However, AKR and TN are immediately deployable in instant messaging apps.

7.2 Is automatic detection (or prevention) compulsory?

Our proposed approaches attempt to defend against fake key attacks automatically. However, like other existing systems, they also entail some overheads. For example, in TN, if Eve has ‘n’ contacts, she is supposed to know the identity of all the contacts of these ‘n’ people. Thus,
if she somehow manages to build a large number of contacts, she could possibly infer a large portion of the app’s social network \textit{i.e.}, who is friends with whom. To safeguard against such kind of enumeration attack, all Signal clients would first hash the identity and corresponding identity key (of their contacts) before sending it to their contacts. This would limit the information leak about the social network (of app) to the Signal clients. (A Signal client can only infer about mutual contacts that it has with other contacts). Privacy-conscious people might not be willing to share their identity (\textit{e.g.}, phone numbers) with unknown people. On the other hand, in AKR, a Signal client is supposed to run a hidden Tor service. Some users may not be willing to use Tor (or it is blocked in their organization/country); such users can opt-out from using our proposed approaches (temporarily or permanently). But, they can still resort to a manual key verification process.

7.3 Exhaustion of one-time pre-keys

The addition of key monitoring to deter fake key attacks increases the number of key bundle requests, leading to an exhaustion of one-time pre-keys. There are two options to address this: (1) Alice must more frequently upload new one-time pre-keys to the key server, or (2) Alice could generate a session key with only the identity key and signed pre-key, without using one-time pre-keys. The first option requires Alice to push one-time pre-keys to the server more frequently. Option 2 reduces the security of the protocol.

One-time pre-keys are \textit{optional} in the Signal protocol. If the key server runs out of pre-keys for Bob, the server returns only Bob’s identity key and signed pre-key in the key bundle. When this occurs, the session keys are generated using only the two keys returned.

We propose the following approach when Alice sends a message to Bob: Initially, Alice generates a session key based on \textit{only} Bob’s identity key and signed pre-key; Alice encrypts the first message and sends it to Bob. In the same message, Alice piggybacks her one-time pre-keys to Bob. Bob also generates a session key (same as Alice) based only on an identity key and signed pre-key; Bob decrypts the first message and obtains Alice’s one-time pre-keys.
In the second message, Bob also piggybacks his one-time pre-keys keys with the message and encrypts it using the first session key. On reception, Alice decrypts the second message to obtain Bob’s one-time pre-keys. Now, both Alice and Bob have three types of keys from each other and can generate further session keys and encrypt their data according to the current protocol specification.

7.4 Pull vs. push-based approaches

WhatsApp, a popular messaging app with billions of users, supports a push-based model in its central server for distributing key updates. Whenever the key server receives a new key bundle for Alice, it pushes a key update message to Alice’s contacts.

Our analysis assumes a push-based model. The push model not only aids our detection strategies, but it is more efficient than a pull-based model — Bob’s contacts all receive Bob’s latest key at nearly the same time. Legitimate key updates do not occur very often; only when a client re-installs the app. Hence, push notifications require fewer resources at both the client and the server compared to a pull-based model.

The Signal Messenger app supports a pull-based model—clients pull key updates on-demand from the server. Whenever Alice opens a chat window to communicate with Bob, she issues a pull request to the server for Bob’s key bundle. Pull requests ensure that Alice obtains Bob’s current key before sending the message. The pull model means that some of Bob’s keys stored on other clients may be out-dated for anyone that hasn’t recently communicated with Bob. We may be able to tailor our defensive approaches for a pull-based model, and we intend to explore this in future work.

7.5 Attacks on multiple clients

In our threat model, we have considered broadly two types of scenarios — pair-targeted attacks (two specific Signal users Alice and Bob) and client-targeted attacks (Alice and all of her contacts). Both types of attacks and their mitigation strategies are analyzed in Chapter
5. The third type of attack is a partial client-targeted attack where the MKS attacks Alice and some of her contacts.

For a partial client-targeted attack, the MKS has two options: (1) provide Alice with fake key updates for a few of her contacts, or (2) send a fake key update of almost all of Alice’s contacts to Alice. With the first option (assuming TN defense strategy), Alice would have multiple contacts left whose keys have not changed. Let’s assume Alice has 10 contacts, and she obtains a key update message for 4 contacts. She could verify keys of these 4 from the remaining 6 contacts if some of the remaining 6 are mutual contacts with the Alice and the 4 clients. If not, she could use any of the 6 clients as a relay (ref. chapter 4.1). For the second option, let us assume Alice receives fake key updates for 8 contacts. If she receives key updates simultaneously, an attack can be easily detected using the burst of key updates heuristic. However, if the MKS wishes to present these fake keys to Alice gradually, the attack would be detected as a pair-targeted attack for Alice.
Chapter 8

Conclusion

We designed three automated key verification solutions to detect or prevent fake key attacks in Signal. We also implemented a fake key attack in our own Signal server to demonstrate that the attack is feasible. We also built proof-of-concept prototype of two of the defenses. Finally, we analyzed these approaches and reported the pros and cons of each. They can be deployed today with minimal effort.

The three defensive approaches have their strengths and weaknesses. The Trust network works using existing infrastructure but leads to privacy leaks among mutual friends. The AKR approach performs strongly in threat model analysis but requires third-party infrastructure. It also has some usability issues compared to the other defenses due to the delay in sending the first message to a new connection. CONIKS is a detection only approach and is privacy-preserving but requires multiple key servers to perform at its full potential. Currently, there is no feasible infrastructure available that provides multiple non-colluding providers to use in CONIKS for messaging systems.

Our proposed defenses serve as a deterrence against an MKS launching a fake key attack. They increase the security for most users without requiring any user interaction. Security-conscious users can still choose to perform manual key verification for critical connections to guarantee prevention against fake key attacks.

In the future, we would like to conduct a large scale simulation on one of the messaging system. We could use the simulations to do an in-depth analysis of the defensive mechanisms
in a real network. We would also tune the hyper-parameters that we used in our defensive approaches, like the regular monitoring time period, isolation duration, and delays.
References


Appendix

9.1 Efficacy of Trust Network: Finding mutual contacts

Below are the distribution of mutual friends for the user’s in the network that we tested in our analysis [7].

![Histogram showing the distribution of the number of mutual friends among the users.](image)

Figure 9.1: Histogram showing the distribution of the number of mutual friends among the users.

Figure 9.1 shows the distribution of the number of mutual friends among the users where there is a mutual friend hit. The y-axis is on the log scale. It was interesting to see that few users even had around 100 mutual friends on average.

Figure 9.2 shows the distribution of the number of mutual friends among the users where there is a mutual friend hit. Users with mutual friends 2, 3, 4 or 5 are more than
Figure 9.2: Histogram showing the distribution of the number of mutual friends among the users assuming random addition of friends

users with just 1 mutual friend. We think it is because there are higher chances of having a group of friends that are interconnected from school, society or work instead of just having 1 mutual friend. The other reason could also be the Facebook friends suggestion that depends a lot on mutual friends. This may lead users to add even those friends who were not too close but suggested by Facebook and known by the user.