A Security Analysis of the Signal Protocol's Group Messaging Capabilities in Comparison to Direct Messaging

by
Matthew Jansen

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented March 19, 2020
Commencement June 2020

AN ABSTRACT OF THE THESIS OF

Matthew Jansen for the degree of Honors Baccalaureate of Science in Computer Science presented on March 19, 2020. Title: A Security Analysis of the Signal Protocol's Group Messaging Capabilities in Comparison to Direct Messaging.

Abstract approved:_____

Michael Rosulek

Signal is a multimedia messaging application developed by OpenWhisper Systems in 2015 which allows its users to communicate securely between one another through the use of a complex encryption scheme. The set of algorithms used in combination to provide the services of the Signal application to their users is called the Signal Protocol. OpenWhisper Systems has documented the direct (or peer-to-peer)messaging capabilities of the Signal Protocol with great detail and plenty of work has been done in the security analysis of the protocol in a two-party context, as well as group contexts to some extent; however, there is still obscurity behind how the Signal Protocol implements group (or peer-to-group) messaging or and if any differences in Signal's group messaging protocol, compared to direct messaging, could result in the loss of security to any extent. In order to understand the protocol further, we developed a modified Signal application to discover how group messaging operations take place. Through this analysis we were able to discover that the Signal Protocol uses direct messaging to send group messages one at a time instead of sending one message to a server and fanning out messages to the recipients, or by using a customized protocol. Further, we claim that Signal's group messaging implementation has no negative impact on security properties provided to Signal's users, in comparison to direct messaging.

Key Words: Signal, encryption, cryptography, end-to-end, messaging

Corresponding e-mail address: jansemat@oregonstate.edu

A Security Analysis of the Signal Protocol's Group Messaging Capabilities in Comparison to Direct Messaging

by
Matthew Jansen

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented March 19, 2020
Commencement June 2020

Honors Baccalaureate of Science in Computer Science project of Matthew Jansen presented on March 19, 2020.

APPROVED:

_____

Michael Rosulek, Mentor, representing College of Electrical Engineering and Computer Science

_____

Yeongjin Jang, Committee Member, representing College of Electrical Engineering and Computer Science

_____

Dave Nevin, Committee Member, representing the Oregon Research & Teaching Security Operations Center (ORTSOC)

_____

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, Honors College. My signature below authorizes release of my project to any reader upon request.

_____

Matthew Jansen, Author

1 Introduction

The Signal Protocol, developed by Open Whisper Systems, is a cryptographic algorithm which allows its users to send and receive encrypted messages - as well as other media - from their devices that have the application installed. What makes the Signal Protocol unique among other instant messaging applications is its use of "end-to-end encryption." This method of encryption ensures that the data sent from one party to another is encrypted and decrypted at the intended sender and recipient(s), rather than an intermediary host. As it is stated on Signal's website, "[Open Whisper Systems] can't read your messages or see your calls, and no one else can either." The Signal Protocol accomplishes this goal by employing several prevalent cryptographic primitives and algorithms in order to achieve specific security properties. These properties can ensure the confidentiality, origin integrity, and message authenticity of texts and other media sent and received using the Signal Protocol, which Open Whisper Systems implements through the Signal application.

Signal is the successor of the TextSecure encrypted chat application. Four years after its initial release in May 2010, TextSecure V2 was released in February 2014 and included an additional feature: private group chats[1]. We will define "direct messaging" as communications between two parties, while "group messaging" entails communications between more than two parties. Having the ability to create, send and receive group-messages is very desirable, and is a feature seen among most instant messaging applications. However, unlike the security properties that are provided by the algorithms which support direct messaging in the Signal Protocol, there seems to be a lack of documentation regarding the security guarantees provided by Signal's implementation of group messaging. The purpose of this paper is to explore how the Signal Protocol is implemented to provide group messaging, and to observe what security properties are achieved when group messaging is executed. To do this, we will first model how we expect the Signal application to act based on the extensive documentation of the Signal Protocol's implementation of direct messaging, as well as model how a malicious actor might attempt to bypass these security guarantees. Next, we will discuss how the implementation of group messaging might deviate from direct messaging and what impact that may have on the security properties provided by the Signal Protocol. Finally, we will observe how Signal implements group messaging and investigate any differences by comparing and contrasting function calls and cryptographic key use found in the actual implementation of group and direct messaging within an application of the Signal Protocol.

2 Background

Signal provides end-to-end encrypted messaging for its users, but how did Open Whisper Systems design their protocol to accomplish this feat? In order to answer this question, this section will serve as a basis for learning the cryptographic primitives used in the Signal Protocol. A cryptographic primitive is a low-level algorithm, commonly used as a building block to construct more complex cryptographic protocols. After the purpose for these building blocks are understood, we can use these building blocks to construct the Signal Protocol one step at a time. The Signal Protocol even uses custom-made cryptographic algorithms to accomplish the security goals of the application, which will be covered in detail later in this section.

2.1 Security Properties

Before defining the cryptographic primitives used within the Signal Protocol, it is vital that an understanding is built of what these building blocks are used to accomplish. Three well known security goals of any secure messaging application include confidentiality, message authenticity and origin integrity.

- **Confidentiality** is the guarantee that messages sent between the two or more parties can only be viewed by authorized individuals, and that non-intended recipients cannot view the message — this accomplished by the use of encryption and decryption schemes.

[1] https://signal.org/blog/private-groups/

- **Origin Integrity** is a security goal which guarantees when a user Alice receives a message from another user Bob, Alice knows this message did in fact comes from Bob – this can be accomplished by the use of cryptographic signatures sent between two parties.
- **Message Authenticity** is the guarantee that a message between two parties has not been altered in any way by an unauthorized party – this can be accomplished by the use of message authentication codes (MACs).

These security properties are vital to any encrypted messaging application, without these adversaries may intercept and read messages intended for other users, impersonate another user, and/or edit the contents of messages intended for other users.

Based on previous work [6], which examines the security guarantees provided by various other encrypted messaging applications, we find that the Signal Protocol additionally provides forward secrecy, future secrecy, participant consistency, destination validation, causality preservation, message unlinkability, message repudiation and participant repudiation. We will base our definitions of each property by the descriptions provided by Unger et al. [6], as well as Schliep and Hopper [10] as follows:

- **Forward secrecy** is a feature of key agreement protocols which prevents an adversary who compromises the message keys of a target user from decrypting any past messages.
- **Future secrecy** is another feature of key agreement protocols which prevents an adversary who compromises the message keys of a target user from decrypting any future messages in the conversation to some extent.
- **Participant consistency** is a security trait derived from the encryption and authentication of the message transcripts from a given conversation. This ensures that each participant in a communication channel has the same list of members in the channel. In the context of a conversation, this ensures that the set of members in this channel is identical for each user.
- **Participant repudiation** (or Deniability) relies on the encryption and authentication of message transcripts, as well as the fact that Signal servers do not store conversation metadata. As a result, any participant is able to deny their participation in a conversation as long as private keys are not compromised.
- **Message unlinkability** is similar to the previously mentioned Participant repudiation / Deniability trait. To define this trait, let's assume that a judge has been convinced that a participant authored a message. Even with this assumption, having message unlinkability ensures that this fact does not provide evidence that they authored other messages within the conversation.
- **Message repudiation** is similar to Message unlinkability. For a given conversation and all its encryption/decryption keys, a user can deny that they authored a particular message.
- **Destination validation** is the assurance that a user can verify that they were in fact the intended recipient of a given message from another user.
- **Causality preservation** is a side effect to the implementation of the Double Ratchet algorithm within the Signal Protocol – if a message is delayed and another more-recent message is received before the original arrives to its destination, implementations can opt to wait for the first message to be received before decrypting the other.

The properties mentioned above describe the protections that are in place which safeguard Signal's users. To understand how these safeguards are implemented, we will take an in-depth look at the algorithms which support Signal's key distribution and message encryption functions.

2.2 Cryptographic Primitives

"Cryptographic primitives" are functions which act as building blocks to more advanced cryptographic algorithms. However, before we discuss these primitives, it is essential that we first cover some basic cryptographic notation that will be used throughout the rest of this paper:

- **Input** and **output** will relate to strings of characters which are either fed into or are the result of (respectively) a given function.
- A **function** can be thought of as a routine which performs a given task – a function may have 0 or more inputs, as well as 0 or more outputs. A function may be thought of as "deterministic" if it returns the same result when it's called with the same input values.
- A **key** can be considered either a piece of data necessary for an encryption/decryption routine, or as a second input to a specified function. This input is not available to an adversary.
- A **constant value** is a fixed number which is available to an adversary.

Using encryption/decryption schemes, cryptographic signatures and message authentication codes to accomplish confidentiality, message authentication and origin integrity is an example of using cryptographic primitives to accomplish specific security goals. Now, we will attempt to build the Signal Protocol by defining a few notable primitives and advancing them to produce the algorithms used in the protocol.

The first set of cryptographic primitives we will define includes a hash function and a pseudorandom permutation (otherwise known as block ciphers) – both of which take input and provides a deterministic output that appear to be random (in the case of block ciphers, the mappings from given inputs to their outputs appear to be random if the key input is unknown) [3]. A hash function is a primitive which takes one input of varying length and produces an output of fixed length, while a block cipher requires two inputs (an input and a key) and provides a single output where both inputs and the output will be of the same length.

For hash function implementations, it is simple to generate an output from a provided input, however if the output length is large enough, it should be considered infeasible to determine the input for a given output — this property is known as "Pre-image resistance." Similarly for block ciphers, it should be infeasible to determine the input to the function, given that they key is unknown; however, unlike hash functions, block ciphers are invertible if their key is known. Current day implementations of these primitives include the SHA-1 and SHA-2 family of hash functions, as well as AES (Advanced Encryption Standard) encryption/decryption implementations for block ciphers. For encryption implementations, note that the two inputs provided include a message block and a key, in order to produce a ciphertext block.

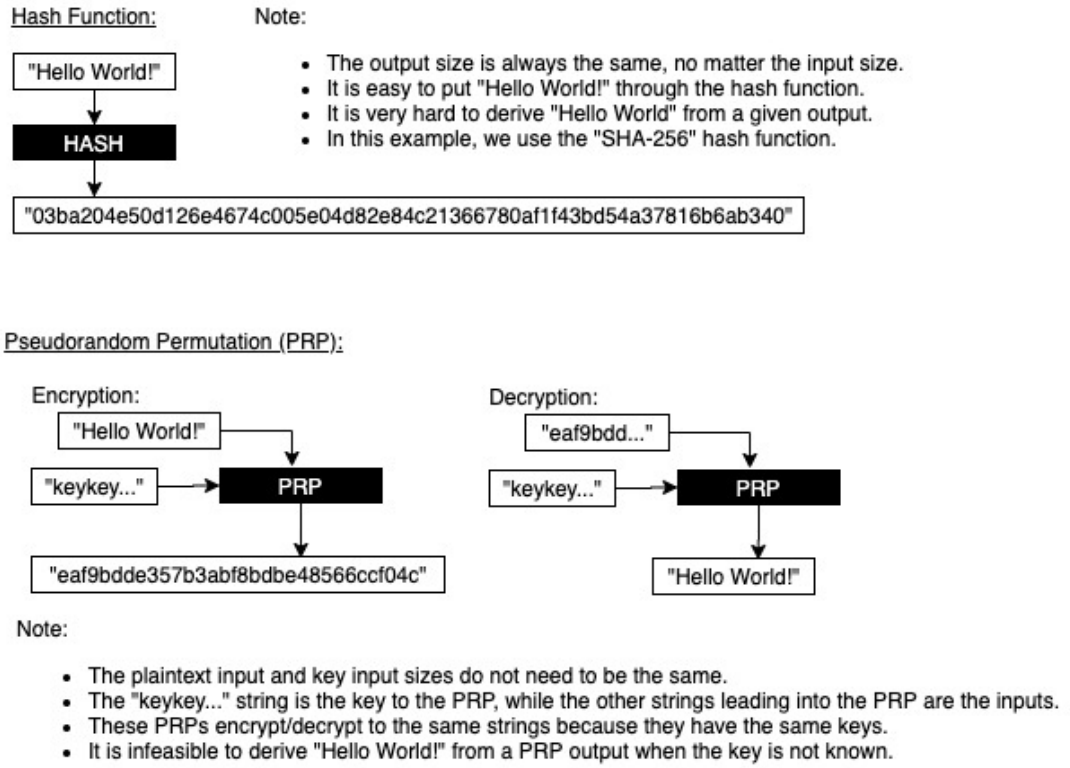

Figure 1: Hash Function and Pseudorandom Permutation Definitions

Moving forward, we will be simplifying our upcoming definitions for the sake of a more complete understanding over a larger audience. The second set of cryptographic primitives build off of previously defined constructions. The next primitive we will define is a Hashed Message Authentication Code, otherwise known as a HMAC. This construction is commonly used to verify the integrity of a message using some pre-determined constants, a message, and a hash function. The creation of a HMAC and an example of its use can be noted in Figure 2.

The next construction builds off of the HMAC and is called the HMAC-based Key Derivation Function, otherwise known as a HKDF. A key derivation function is similar to a block cipher, in that it takes several inputs as arguments to the function, however it differs from previously mentioned primitives as a HKDF provides multiple outputs, all of which are still indistinguishable from random. As seen in Figure 2, the construction of a HKDF depends on the available construction of a HMAC. The implementation of these constructions should be in accordance with their respective RFC [3, 4].

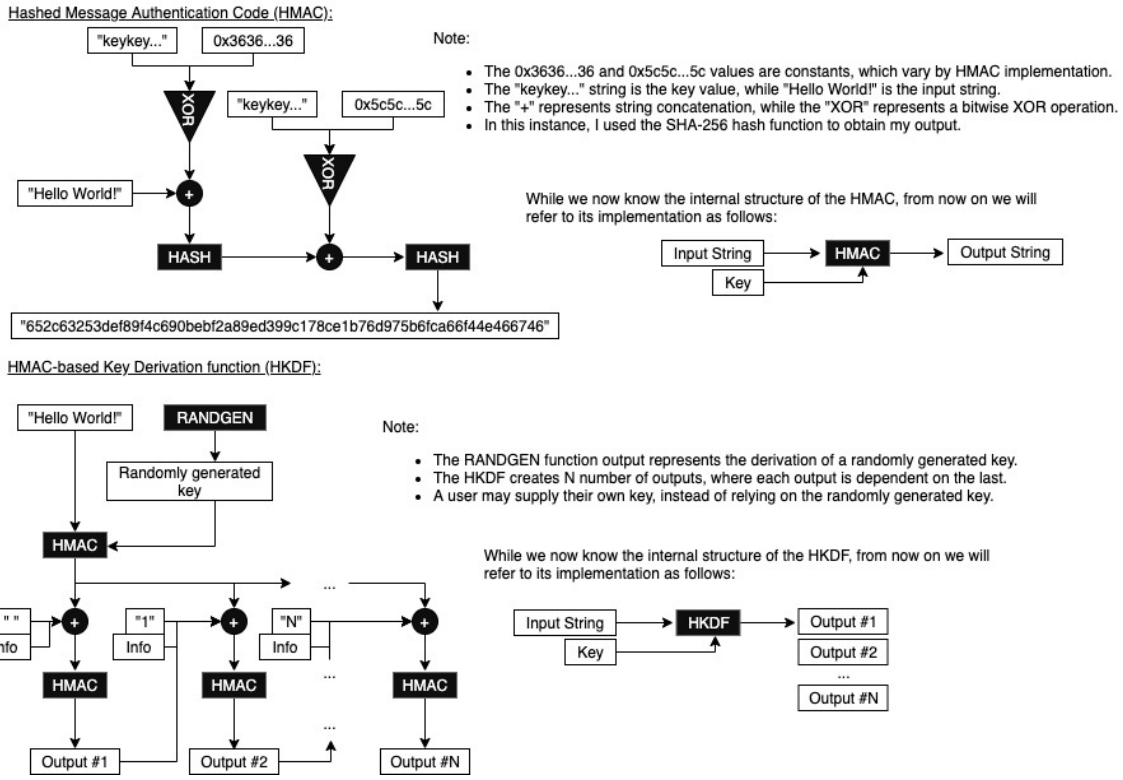

Figure 2: HMAC and HKDF Definitions

As seen previously, encryption/decryption schemes require the users involved to only own one key. This is called “symmetric cryptography,” as only one key is required. However, there also exists “asymmetric cryptography,” also known as “public key cryptography,” which involves a user having two keys – one public key that is known to all, and one private key that is known only to the individual user. Another cryptographic primitive that employs the use of asymmetric cryptography are digital signatures. Digital signatures involve signing a message you wish to send to another party using your private key. This way, when the other party receives your message, they can verify the integrity of the message you sent by comparing the signature of your message using your public key. If the computation needed to perform this signature verification matches up with your public key, then the recipient knows the message was indeed sent by you, since you are the only one who knows your private key. There are several implementations of digital signature algorithms, including RSA, DSA, and elliptic curve digital signatures.

Each of the earlier-mentioned primitives are important in successfully implementing the security properties that the Signal Protocol wishes to provide. But before we discuss how these building blocks are arranged in order to create the Signal Protocol, we must first cover how keys can be securely transferred between users. The concept of securely exchanging cryptographic keys over a public channel is a widely discussed topic that has many proposed solutions, and one of the more reputable algorithms for doing so is called the "Diffie Hellman Key Exchange Algorithm.”

In this algorithm, two parties wish to securely generate a symmetric key and do so using their own set of pre-generated asymmetric keys. The algorithm generally goes as follows:

1. The users, who we'll call Alice and Bob, first agree on a set of global constants that are publicly available. We will name the constant that is used and transferred between Alice and Bob g.
2. Alice and Bob each take a copy of one of the global constants and perform a mathematical operation on their copy using their private key, resulting in the output g_a for Alice, and g_b for Bob.
3. Alice and Bob exchange g_a and g_b. So now, Alice has g_b and Bob has g_a.
4. Alice and Bob perform the same original mathematical operation on the other party's copy, resulting in the output s_a and s_b for Alice and Bob (respectively).
5. Alice and Bob should now share the same shared secret g_{ab}, and in doing so, should have not revealed their private key to a public channel.

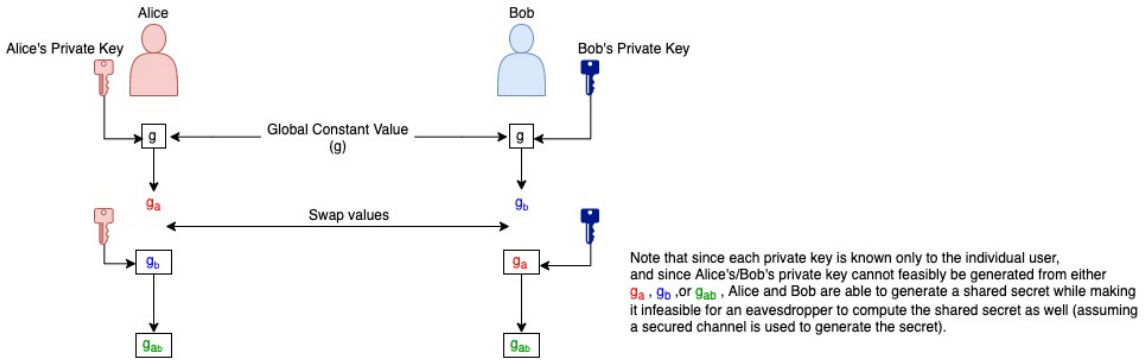

Figure 3: Diffie-Hellman Key Exchange Diagram

For these key exchanges, g_a and g_b are referred to as "public keys." More information regarding the mathematical operations performed in this algorithm can be found in the original work by Diffie and Hellman [2], as well as Rosulek [9]. In addition to Alice and Bob not having to share their asymmetric private, according to the discrete logarithm problem [7], attempting to find the private key used by each party using the information passed between each party during the key exchange is considered computationally infeasible if large constants and asymmetric keypairs are. Moving forward, when we refer to computing a Diffie-Hellman key exchange operation between two keys, such as:

$$DH(publicKey_{alice}, privateKey_{bob}),$$

We are referring to performing a mathematical operation on the two keys, such that the following equation holds:

$$DH(publicKey_{alice}, privateKey_{bob}) \equiv DH(privateKey_{alice}, publicKey_{bob})$$

2.3 Extended Triple Diffie-Hellman (X3DH) Protocol

The Signal Protocol takes the Diffie-Hellman Algorithm one step further by repeating the algorithm for different sets of keys which are stored for different amounts of time, they coined this recipe for secure key-exchange as the "Extended Triple Diffie-Hellman Algorithm," [12] or "X3DH" for short. In this algorithm, each user owns several different types of asymmetric key-pairs: identity keys are long term and are usually generated at install-time, signed ephemeral keys are replaced at a certain interval (for example, every month), and one-time pre-keys are used once for every iteration of the protocol. These keys, including a set of one-time pre-keys and a signature of the user's ephemeral key, are sent to and hosted at a server owned by Signal (this set of keys is referred to as a "pre-key bundle") which allows for the application to build shared keys between users even when one of those users might be offline. Below, Figure 4 shows the series of Diffie-Hellman key exchanges that are performed in order to generate a shared secret between two users.

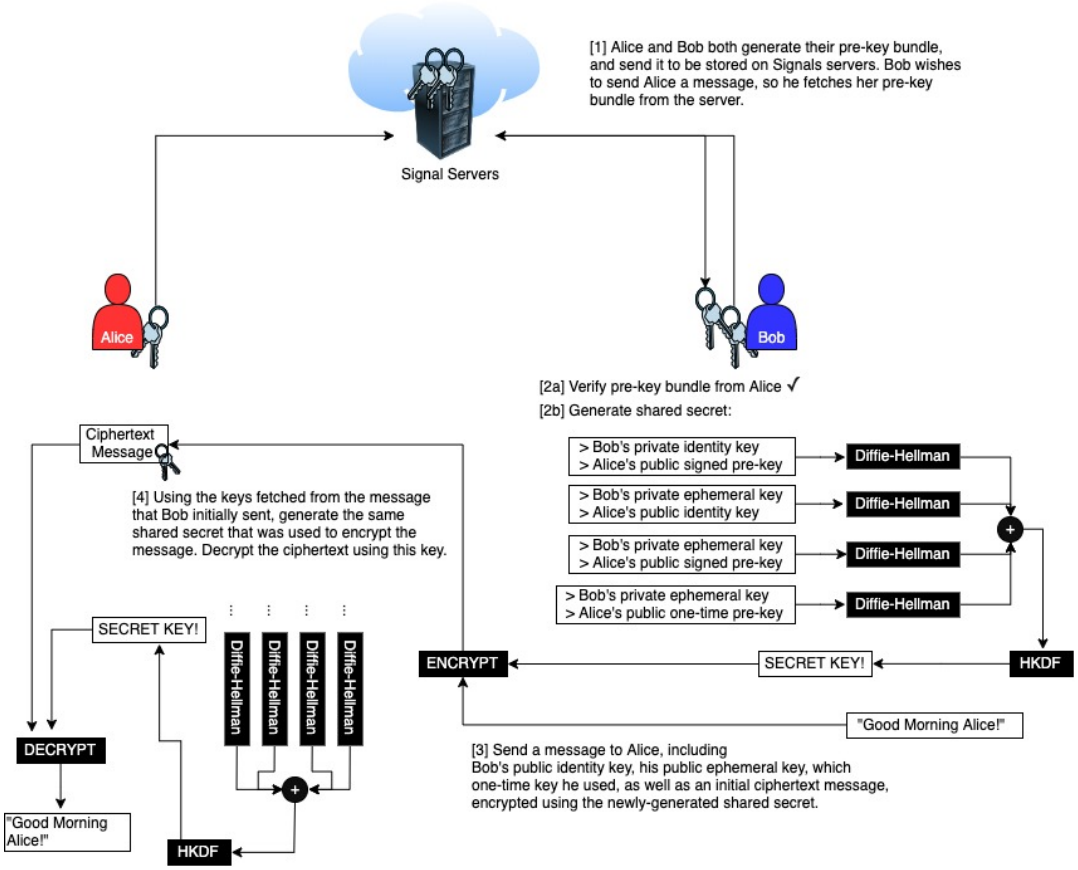

Figure 4: X3DH Key Exchange Example

Once a shared secret is generated between two users, they may use this shared secret as the initial cryptographic key for a symmetric encryption scheme in a post-X3DH algorithm, such as the Double Ratchet algorithm, which will be mentioned later. The implementation of the X3DH algorithm provides mutual authentication between users and forward secrecy over a public channel when generating the secret key. This prevents an adversary from impersonating a legitimate user, and if a user's keypair is compromised, prevents an adversary from decrypting past secret keys. In addition, both users do not need to be online when this key exchange is performed – because these keys are held on a server hosted by Signal, one party can generate a shared secret for a future conversation with another user.

2.4 Double Ratchet Algorithm

Once a shared secret key has been generated between two users, this key can be used to generate further message encryption keys or can be used itself as a message encryption key. In order to generate these keys such that all of the previously mentioned security properties can be achieved, the algorithm in which the key generation and message encryption takes place must be designed specifically to uphold these goals.

Let's say that we are creating our own hypothetical encryption scheme and take the following security feature into consideration: forward secrecy. In order for our message encryption algorithm to succeed in providing this feature, the algorithm must ensure that in the case a user's encryption key for a specific message is compromised by an adversary, it would still be considered infeasible for the same adversary to attempt to compromise any previous message encryption keys, even with the key they currently possess. In order for our encryption algorithm to accomplish this, each message encryption key must be passed through a sort of hash function (or something similar) in order to ensure that an adversary cannot feasibly guess previous keys (as we recall, coming up with the input of a hash function, given the output, is considered infeasible). Adding

this primitive into our message encryption scheme disallows adversaries from finding previous encryption keys (giving us forward secrecy,) but how can we prevent adversaries from calculating future encryption keys (in other words, having the encryption scheme provide future secrecy?) One solution entails incorporating a Diffie-Hellman key exchange once a certain amount of messages or time has passed, which results in the generation of a fresh key iteration. With this, old keys are discarded, a new key is created, and in the case that an adversary had compromised an encryption key, the creation of a new iteration of keys prevents an adversary from generating new keys using the old key they possess. Coming back to Signal, in order to exchange encrypted messages while meeting the previously mentioned security properties, Open Whisper Systems built an algorithm that does so effectively by ensuring fresh keys are used for every encrypted message, which they dubbed the "Double Ratchet Algorithm" [11].

In the implementation of the Double Ratchet Algorithm, keys are refreshed, or "ratcheted" in two different ways: using a symmetric ratchet, and an asymmetric ratchet. Performing these ratchets refreshes the data used to generate message encryption keys, as well as the actual message encryption keys. These ratchets are performed using chains of HKDFs, which entail having the input key of the current HKDF chain be the output material of the previous HKDF execution. There are two of these chains running concurrently which provide the symmetric keys used to encrypt and decrypt messages.

In the asymmetric ratcheting phase, the HKDF chain known as the "root chain" is constantly refreshing the HKDF chains which are used to generate the message encryption/decryption keys. The key value which is used in the initial root chain HKDF execution is the output of a Diffie-Hellman key exchange, using the keypairs from both users — given an authenticated channel — this output will only be known to the parties involved in the key exchange. For the initial input value of the root chain HKDF, a shared secret generated using the X3DH Algorithm is used; however, after the first asymmetric ratchet of the root chain, subsequent inputs to this HKDF chain will be the output of the previous ratchet. The second of the two HKDF outputs will be used as the starting input of two new chains, the sending and receiving chains, which will be used for the symmetric ratchet. When asymmetric ratchets take place, the current symmetric key chains are potentially discarded (if they aren't still needed for the decryption of in-transit messages) and a new sending/receiving chain pair are created, introducing freshness and allowing the Double Ratchet algorithm to provide future secrecy. Figure 5 shown below gives a graphical overview of the asymmetric ratcheting phase.

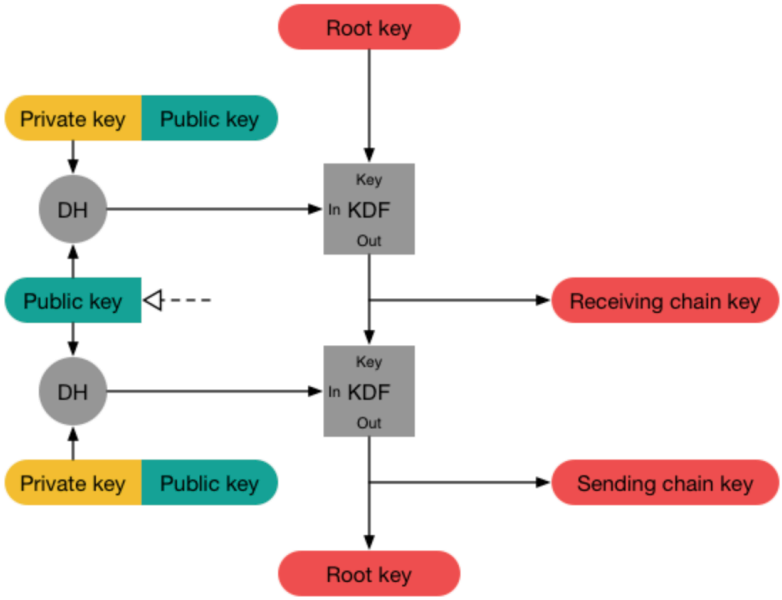

Figure 5: Asymmetric Ratcheting Phase of the Double Ratchet [11]

In the symmetric ratcheting phase, symmetric message encryption/decryption keys are refreshed within the sending and receiving chains. To ratchet the sending and receiving chains, the first of the two outputs from the previous HKDF execution for the chain is fed as the input to the a new HKDF execution, while the second of the two outputs will be used as a key for a new message's encryption/decryption operation. This use of HKDFs in this fashion allows the device to discard symmetric keys after they are used to encrypt or decrypt a message, and in doing so, prevents an adversary from decrypting past messages in the event of a symmetric key compromise – this is what provides forward secrecy to the Signal Protocol. Putting these two ratcheting mechanisms together, below in Figure 6 we can see how the root chain and sending/receiving chains may interact with each other in the event of a conversation between two parties. When a message is sent or received, a symmetric ratchet is performed to the corresponding chain in order to obtain a fresh key for message encryption/decryption. Furthermore, when a new Diffie-Hellman ratchet public key is received from a user, then an asymmetric ratchet is performed in order to replace both of the chain keys.

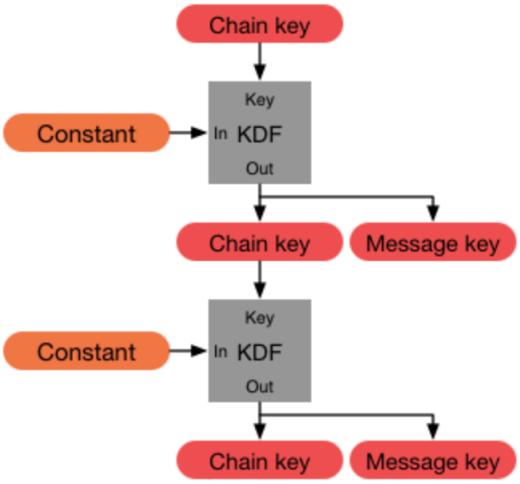

Figure 6: Symmetric Ratcheting Phase of the Double Ratchet [11]

2.5 Putting Everything Together

What connects the earlier discussed X3DH Algorithm and the Double Ratchet Algorithm is the generation of an initial shared secret key. This shared secret is the final output of the X3DH Algorithm and is coincidentally the starting point for the Double Ratchet Algorithm. When a session ends between two users after a varying amount of time has passed or messages have been sent, a new session may be created again using these two algorithms in succession.

Now that we have an understanding of how these algorithms work in combination to provide end-to-end encrypted messaging for Signal's users, we need to recognize and interpret any changes that might occur when we translate this protocol to a group messaging context. If you noticed, none of the previously mentioned algorithms discussed group contexts. In the next section we will discuss how we expect an application of the Signal Protocol to act and how threat actors might attempt to attack the protocol. Following this, we will deliberate the impact that group contexts have in all of the security properties and algorithms that we discussed in this section.

3 Security Model

A security model is exactly what it sounds like: it's a model that attempts to lay out how a system incorporates different security policies within its scope as a piece of software. In the context of the Signal Protocol, we identify the scope of the system in question as an application that implements the C/Java Signal library. The

Signal source code libraries can be found on GitHub[2] , and the official Signal Application, which is hosted by Open Whisper Systems and can be downloaded on most smartphones (iPhone/Android), as well as on Windows/Apple/Debian-based Unix operating systems[3] . In this chapter we will be modeling how the Signal Application is implemented by taking a thorough look at its source code, modeling how an adversary might attack the Signal application, and modeling how the Signal application mitigates the previously defined attacks.

3.1 System Model

A system model, used frequently in software engineering and architecture, can be utilized to describe how an application is implemented in pieces and how each of those pieces of the application can work together in order to successfully complete specific use-cases. These use-cases describe functionalities that the software needs to accomplish in order to meet the needs of its users. In this context, we can model how the Signal Application implements the Signal Protocol by identifying use-cases of the application and describing how the application uses user and system resources to accomplish these tasks. Our goal is not to do a source code review of the Signal Application - this has already been done and the results have shown that the source code is professionally written. In our system model, we will map the flow of data used within Signal, such that an understanding of the application may be developed.

The diagram below shows a "black box" system model of the Signal Application, which represents how the software works without analyzing the complex internal structure of the application. Another way of looking at this can be described as looking at the application from the user's point of view. From the network perspective, you'll notice several parties involved: Alice, Bob, and the Signal servers - if Alice wishes to send a message to Bob, the message is fed into the application by Alice, which is then encrypted and sent over the users network to the Signal servers. These servers hold the encrypted messages while Bob is offline. When Bob comes online, his device will query the Signal servers, which will then send him Alice's encrypted message. The application will then decrypt Alice's message, and output the unencrypted data to Bob - note that at any point during this communication, there is no instance where a party is sending unencrypted information over the wire. In other words, not even the Signal servers have access to the cryptographic keys which can decrypt Alice's and Bob's messages — only Alice and Bob own those keys, which can only be decrypted on their own device — this is how the Signal Application implements end-to-end encryption.

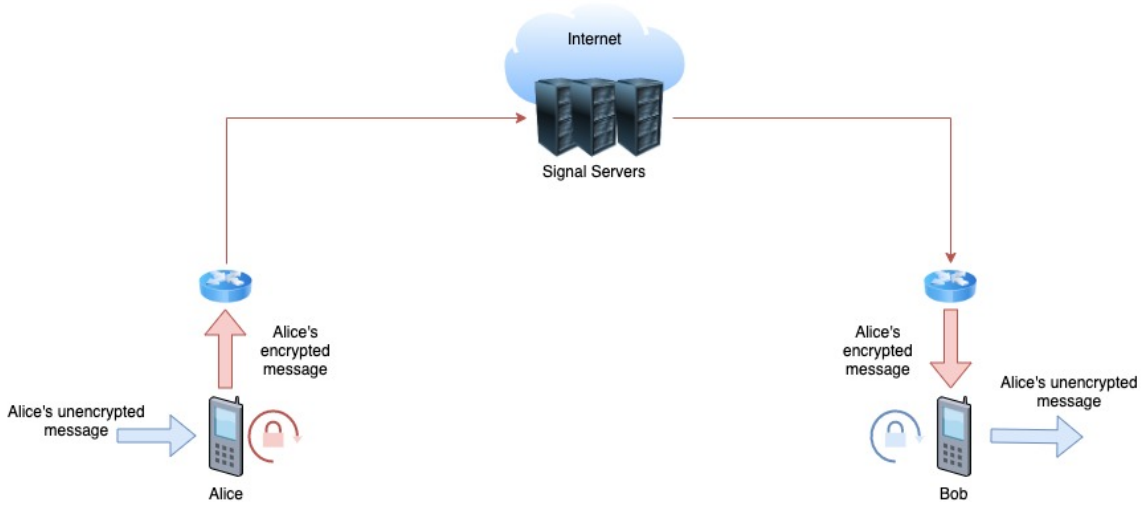

Figure 7: Signal Application Black Box System Model

[2] https://github.com/signalapp

[3] https://signal.org/download/

As we focus on what happens within each device, we begin developing a “white box” system model of the Signal Application. Simply put, instead of viewing an application as a box that accepts some input and gives some output, the internal structure and architecture of the application is recognized and understood. From this perspective, we see input coming from both Alice and Bob - where Alice’s input is her message to Bob, and Bob’s input is the end-to-end encrypted message that he sent to Alice. When Alice wishes to send a message, she records her data in a buffer within the application, and when the command is sent to the application to relay the data to Bob, the information in the buffer is sent to an encryption schema which encrypts the data. The encryption schema also takes a secret key as input which is generated simultaneously using the Double Ratchet Algorithm. When this encryption schema finishes its operation, the output will be in the form of encrypted data and will be sent out to be decrypted and viewed by Bob. Similarly, when encrypted data from Bob is sent to Alice's device, the data is sent to a decryption schema along with a decryption key, and the output is the original and unencrypted message for Alice to view and process.

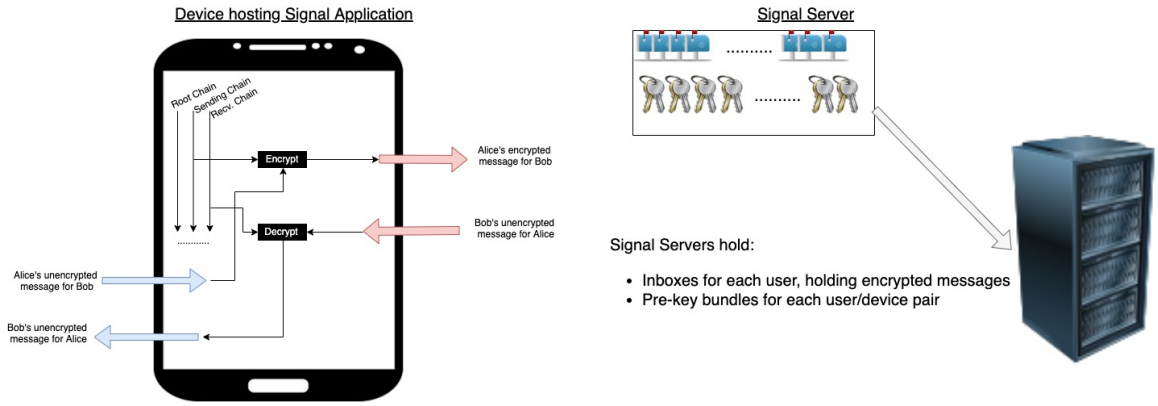

Figure 8: Signal Application White Box System Model

In addition to devices with the Signal Application installed, we can also clarify the use of the Signal Servers. These servers are used to store information for each user, as well as their devices. Specifically, each user’s inbox is stored on these servers, which hold the encrypted messages that are destined for each user. Once a user comes online, their device pings the server, and the server will return any new encrypted messages that are destined for them. In addition, the Signal Servers hold pre-key bundles which are used to initiate the handshake needed to generate a shared secret key between two users, using the X3DH Algorithm. Using this system model gives us the infrastructure needed to explain most use-cases for the Signal Application. With this understanding in place, we can observe how an attacker may try to exploit this architecture using an adversary model.

3.2 Adversary Model

In order to create our adversary model, we need to define how we expect an attacker to approach breaking any of the security properties that are provided by the Signal Protocol. Although we will be taking a look at some source code for Signal later, I would like to emphasize that we will not be attacking the code. Instead, now that we have defined how we expect Signal to work using our system model, we will go about defining the different roles that an attacker may engage in, and depending on their role, characterize what attacks they might be able to employ. We will be building our adversary model off of previously built models [8], such that we divide the roles our attacker may take into three positions. These positions describe the medium to which a malicious actor may try to break the Signal Protocol. These roles include a malicious device/user, the malicious network owner, or the malicious Signal server.

For the first role that was mentioned, the "malicious device/user," we are defining two different situations where attacks are based at the application level. The first situation entails a user who has malicious intent in their use of the Signal Application, and the second entails a user with no malicious intent in their use of Signal, however their device itself is compromised. In either of the two situations, the malicious actor is assumed to have full control over the device. While an attacker has taken this role, they may attempt to break the confidentiality of the messaging scheme by obtaining encryption/decryption keys from the device's memory, impersonate a user/device pair by stealing device identifiers and private keys, or a combination of the previous two by decrypting, altering, and re-encrypting messages in order to break the message authentication provided by Signal.

The next role that an attacker may take is the "malicious network user." Here, we are describing a malicious actor who has taken control of a piece of the computer network infrastructure which sits between the Signal Application user and the internet – more specifically, the Signal servers. While in this position, we are assuming that the attacker has full monitoring capabilities over the network, and the malicious actor may perform a network packet-capture on the network traffic originating from a target user. In addition to capturing network traffic, the malicious network user may attempt to perform network-based attacks on a target user, such as spoofing traffic from a legitimate peer or a Signal server or deploying a man-in-the-middle host to intercept or alter any incoming/outgoing network traffic.

The last role we are defining that an attacker may take is the "malicious Signal server." Here, we are describing a malicious actor that has compromised the servers that hold the inboxes for each user, and the pre-key bundles of each user/device pair. While in this role, we are assuming that the malicious actor has full control over these servers and have the potential to drop communications from a specific party and drain a user's one-time pre-keys from their pre-key bundle. The impact of dropping communications from a specific source includes potentially forcing the user to use another form of communication which may not be encrypted. Additionally, the impact of draining a user's one-time pre-keys from their pre-key bundle consists of potentially reducing the strength of future-generated shared secrets between two parties using the X3DH protocol. Instead of using four Diffie-Hellman operations to obtain a shared secret, only three are used before the outputs are passed through a HKDF, causing the forward secrecy of the shared secret key to depend solely on the lifetime of the signed pre-key [12].

Now that we have described the different roles a malicious actor may take to attack Signal, we need to define the scope of the assessment we will be performing. We will be documenting and analyzing how the Signal Application implements group messaging by evaluating the chain/key values used during message encryption and decryption. Since this cannot be done from the perspective of a "malicious network user" or a "malicious Signal server" attacker, we will reduce the scope of this assessment to that of a "malicious device/user" actor. In our assessment, we will assume that we may compromise a variety of cryptographic keys at any single point in time. In other words, we will choose and evaluate the severity of compromising a specific key; however, we will not be able to steal more than one key.

3.3 Mitigation Model

Now we will describe a mitigation model that is used by the Signal Application to prevent attacks defined as in-scope within our adversary model. In order to map out how Signal mitigates attacks performed by a malicious user or a malicious actor who has compromised a legitimate user's device, we will split mitigations into two categories: mitigations performed during the initial key exchange (or during an execution of the X3DH Algorithm), and mitigations performed when a valid session is already in place (during the Double Ratchet Algorithm execution). In both cases, we will give instances of attacks and mitigations in the case that a user's private identity key, signed pre-keys, or one-time pre-keys are compromised, and in the case of the Double Ratchet Algorithm, we will give attack/mitigation examples where the root, sending or receiving chain keys, or message keys are compromised.

During an execution of the X3DH algorithm, the keys that may be compromised are a user's identity keys, signed pre-keys, or their one-time pre-keys. The following security considerations are summarizations of Marlinspike et. Al [12].

- A user has no cryptographic guarantee that they are exchanging keys with their intended recipient unless both parties authenticate themselves over an authentic channel.
- If a party doesn't use one-time pre-keys in their X3DH protocol run, or if a malicious actor drains another users' one-time pre-keys, then an initial message that is sent may be replayed and accepted by the recipient. If this were to occur, this would allow users to reuse their initial keys. To mitigate this, both parties need to ensure that one-time pre-keys are used during their protocol runs and that their stock of one-time pre-keys hosted at the Signal server doesn't run out.
- If a third party has compromised Alice or Bob's private identity key, then given a transcript of the initial message that occurred between two parties, the third party could confirm that the communications occurred between Alice and Bob, breaking the deniability scheme within the X3DH Algorithm. Additionally, a compromised identity key can result in the complete impersonation of that party to others. To mitigate this, a user must replace their identity keys if they suspect them to be compromised.
- Compromised signed pre-keys may also have disastrous effects on the security of older or future values of the shared secret between two parties.
    - If one-time pre-keys are used during the protocol run, and a party's private identity key and signed pre-key are compromised, the older secret key value that was generated may not be compromised as long as the one-time pre-key that was used is deleted.
    - If one-time pre-keys are not used during the protocol run, and a party's private identity key and signed pre-key are compromised, then so is the older secret key value that was generated.
    - Compromised signed pre-keys may enable future attacks such as passive calculation of secret key values, and the impersonation of arbitrary other parties.

      In any case, to mitigate signed pre-key compromise, a user must replace any keys that they suspect to be compromised.

During a protocol run of the Double Ratchet Algorithm, in addition to private key compromise, other private resources available to the user may be stolen from an attacker. Here are some examples, et. al Marlinspike [11]:

- A malicious entity who has compromised a legitimate user device may be able to decrypt encrypted messages if old plaintexts or keys can be recovered. To mitigate this, it is recommended that old plaintexts and keys are securely deleted.
- A compromised user device may have their private keys stolen, which can lead to a number of disastrous situations:
    - An attacker could use the compromised private identity keys to make their own new sessions using new X3DH protocol runs.
    - An attacker could substitute their own ratchet keys via an active man-in-the-middle attack and impersonate the user who owns the compromised device.
    - An attacker could modify the random number generator within the device such that future ratchet keys are predictable.

  In any case, if a user suspects that their private keys have been stolen, they should replace them immediately.

Now that we have defined how we expect the Signal Application to function, how we suspect attackers might attack Signal in a given scope, and how those attacks might be mitigated, we can move forward in defining group contexts for the Signal Protocol. In the next section, we will be adding on to our previous definitions and security properties, discussing previous findings, and describing how we will assess group messaging within the Signal Application.

4 Group Messaging

Taking a step back and looking at unencrypted messaging applications in general, one key feature you might think of is the ability to send and receive messages in a group setting. Having this capability makes it much more convenient for more than two people to engage in conversation without miscommunications occurring. Possessing this function in an unencrypted messaging application can be as easy as repeatedly sending the same message to several recipients. However, does this seemingly easy implementation of group messaging have the same conversion to an encrypted messaging protocol? If not, what security properties in our encrypted messaging protocol are at risk of being violated in the context of group messaging? In this section, we will be answering these questions by understanding the differences in the security properties associated with direct and group messaging, summarizing previous findings and discussing their implications. We will end this section by discussing how we can customize an implementation of the Signal application in such a way that we may characterize the Signal protocol for group messaging.

4.1 The Implications of Group Messaging

As stated earlier, we know that Signal seeks to bring confidentiality, message authenticity, origin integrity, and much more to their messaging application. In order to understand how group messaging differs from direct messaging, we are first going to define additional security properties that are sought when describing an encrypted group messaging protocol. After this, we will describe some of the difficulties that are encountered while creating an encrypted group messaging protocol. We will be defining each new group messaging property by the descriptions provided by Unger et al. [6], as well as Schliep and Hopper [10].

- **Computational Equality** ensures that all chat participants share an equal computational load.
- **Trust Equality** guarantees that no participant is more trusted or takes on more responsibility than any other.
- **Subgroup Messaging** means that messages can be sent to a subset of participants without forming a new conversation.
- **Contractible Membership** ensures that after the conversation begins, participants can leave the group without having to start a new protocol run.
- **Expandable Membership**, similar to contractible membership, ensures that members can join the group without having to start a new protocol run.

The above-mentioned properties are important to encrypted group messaging protocols as they create a guideline for controls over group session management. These controls not only have to do with security, but with computational cost as well. For example, trust equality is a sought after so that there is no single target in which a compromise might have an avalanching affect to other users. Computational complexity comes into concern when groups grow in size, which is why many of the above properties of group messaging have to do with performing actions without having to start a new conversation, or perform a new protocol run.

Well if we wish to implement encrypted group messaging, why can't we just use a multi-party key distribution protocol, similar to X3DH, in order to generate an initial shared secret for everyone in the group? Keep in mind that each group member needs to contribute to the generation of a new group key. Thus the addition of more members into a group key distribution protocol consequentially increases the complexity of the computation needed to generate a key. In addition, we mention later that previous work [1] has shown that having a group key used within a Double Ratchet protocol run will deprive the session of its ability to provide future secrecy, as the chain keys are not asymmetrically ratcheted unless the group undergoes a major edit.

An older blog post made by Moxie Marlinspike after group messaging was released in 2014[4] describes how TextSecureV2 (the predecessor to Signal) battled group message security, however this blog post did not

[4] https://signal.org/blog/private-groups/

describe in depth how the application solved these problems. Since then, no documentation has been released from Signal regarding the implementation of the group protocol, and no documentation regarding the group protocol has been released by anyone with as much clarity as the other algorithms used in Signal such as X3DH [12] or the Double Ratchet Algorithm [11].

4.2 Previous Findings and their Implications

Previous work from [1] has stated that the utilization of the Double Ratchet algorithm within Signal's Java implementation is dubbed as the "Senders Keys" variant of the Signal Protocol, as opposed to the “Pairwise” form. The “Sender Keys” variant of the protocol is also known as "server-side fan out," and entails the sender of a message delivering the encrypted message to the Signal server, where it will copy the message and send it to each participant in the group message. Although this method reduces the complexity of the sending operation on the sending users' side, in the case of an adversary discovering a users' sending key, the adversary could potentially eavesdrop on all future messages and impersonate the target user. As stated in [1], this is due to the key chains only being rotated (via an asymmetric ratchet) when the group is edited in some way (a user leaves the group, the group’s name is changed, etc.). Hence, the "Senders Keys" variant of the Signal Protocol does not provide future secrecy. The “Senders Keys” and “Pairwise” variants of Signal’s Java implementation, otherwise known as the “server-side fan out” and “client-side fan out” respectively, is detailed below in figure 9.

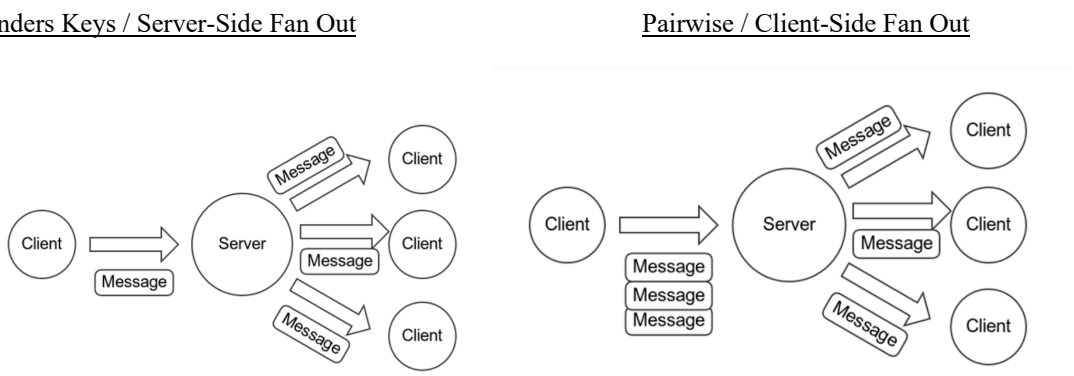

Figure 9: “Senders Keys” and “Pairwise” Variants of Group Messaging

However, in a previous analysis done on the Signal Protocol and its group messaging implementation [8], it was stated that the Signal Application applies "client-side fan out," which entails creating a pairwise conversation with each other user in a group message. This described method, although more complex than the "Senders Keys" variant and requiring more work done by each individual user, provides future secrecy in its implementation. The analysis performed by Rosler et al. also mentions using the Java implementation of the Signal Protocol, although, the authors also describe how they were able to break this implementation by describing two proof-of-concept attacks that may be performed: burgling into a group, and forging acknowledgements. In this case, the act of adding yourself back into a group, even after you have been kicked out, can compromise the security of all encrypted messages sent in the future, hence breaking future secrecy.

The two previously described works have brought forward the notion that the Signal Protocol cannot guarantee future secrecy; nevertheless the reasons behind each seem to contradict each other as one declares that the Java implementation of the Signal Protocol entails server-side fan out, while the other asserts that pairwise encrypted messages all originate from the sender. Conflicting viewpoints, in addition to the lack of documentation regarding the implementation of X3DH and the Double Ratchet algorithm in a group setting, brings up the following question: How is Signal's group messaging protocol actually implemented?

In order to find out exactly how Signal's group messaging protocol is implemented, we will be disassembling a command-line tool used to implement the Signal Protocol's Java library in such a way that users can send and receive messages from their command line prompt. After extracting the Java classes associated with this tool, the uncompiled code will be altered in order to track the path of execution during the messaging process. By doing this, we will be able to identify possible differences in the path of execution between direct and group messaging. In the next section, we will discuss the thoughts and design behind this customized tool and consider how our results can characterize Signal's group protocol.

5 Assessing the Group Messaging Protocol

In this section, we will discuss Signal's Java Implementation, and the different applications that use it. Next, we will describe in detail how it's possible to edit these applications in order to obtain verbose information regarding the implementation of direct and group messaging, so that we may map out the execution of the Double Ratchet Algorithm. After obtaining our results, we will use this information to summarize how group messaging works compared to direct messaging, and how that may affect the security properties that are sought after for encrypted messaging protocols.

Previous analyses of Signal's group messaging capabilities detail attacks against the protocol, comparing Signal to other messaging applications, and even analyzing the cryptographic algorithms used in the protocol [5]. However, our analysis will differ by examining the behavior of Signal under group messaging contexts. This analysis will not examine vulnerabilities in the group session management employed by Signal, nor will it focus on the cryptography of the underlying protocols. Instead we will focus on documenting the behavior of Signal under group messaging contexts by altering the code base of libsignal-protocol-java to log specific function calls. By doing this for direct and group messaging, we will have a first-hand glance at how the path of execution differs between these two modes of operation. The question we are asking is "How does Signal implement group messaging, and how does it differ from direct messaging?"

5.1 Senders Keys vs. Pairwise Keys Message Distribution

Before diving into Signal's code base, we will briefly discuss the differences in asymptotic complexity between the previously defined Sender Keys and Pairwise Keys variants of group messaging.

		number of exponentiations		number of symmetric encryptions		bandwidth		PCS
		sender	per other	sender	per other	sender	per other	
sender keys	setup	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	✗
	ongoing	0	0	1	1	$O(1)$	$O(1)$	
pairwise Signal	setup	$O(n)$	$O(n)$	0	0	$O(n)$	$O(n)$	✓
	ongoing	$O(n)$	$O(1)$	$n-1$	1	$O(n)$	$O(1)$	

Figure 10: Asymptotic differences between Sender Keys and Pairwise Keys variants [1]

As shown in the figure above, each group messaging variant has benefits over the other in some capacity. The Sender Keys variant provides increased efficiency in the post-setup phase with decreased number of exponentiations and symmetric encryptions, as well as less bandwidth usage. Although the Sender Keys variant seems to be more efficient, it does not provide Post-Compromise Secrecy (PCS), otherwise known as

future secrecy, which is provided in direct messaging, and in turn provided by the Pairwise Keys variant of Signal group messaging.

There appear to be pros and cons to both variants of group messaging, however with respect to the additional level of security provided by future secrecy, implementing the Pairwise Keys variant of Signal group messaging would ensure that future messages can't be decrypted (to some degree) after a key compromise.

5.2 The Java Signal Protocol Library (libsignal-protocol-java)

The Signal Protocol has been implemented in several languages, each of which are placed into libraries that can be found on the *signalapp* project page of GitHub[5]. Specifically, we will be looking at Signal's Java implementation, which we will refer to as *libsignal-protocol-java*[6]. This library is used to communicate with Signal's messaging service and is used in the implementation of Signal's official Android application. To use this library to communicate with the Signal servers, a client must generate keypairs and register themselves as a user of Signal before being able to send and receive both text and media messages

Within this library, there are several directories and standalone Java files that are used to implement the protocol. We will not be going through all of these as they are not all relevant to the topic at hand, and instead we will focus on the files which relate to direct/group message encryption and decryption:

- *Libsignal/groups/*
  - *Ratchet/SenderChainKey.java*
  - *Ratchet/SenderMessageKey.java*
  - *State/SenderKeyRecord.java*
  - *State/SenderKeyState.java*
  - *State/SenderKeyStore.java*
  - *GroupCipher.java*
  - *GroupSessionBuilder.java*
  - *SenderKeyName.java*
- *Libsignal/ratchet/*
  - *ChainKey.java*
  - *MessageKeys.java*
  - *RootKey.java*
- *Libsignal/state/*
  - *SessionRecord.java*
  - *SessionState.java*
  - *SessionStore.java*
- *Libsignal/SessionCipher.java*

At first glance, we notice that several files within the *group* directory are similar to those seen throughout the rest of the code base (*GroupCipher.java* is similar to *SessionCipher.java*, there are *ChainKey* and *MessageKey* files for both direct and group messaging, etc.). This appears to be re-implementations for specific functions such that they will work under a group context.

We have acknowledged the existence of a group messaging directory within the *libsignal-protocol-java* code base and we know there exists separate constructors and methods that specifically align with group messaging. From this information, we may form our hypothesis. If we send/receive group messages using our customized Signal application, then the application will execute function calls within the group messaging directory from within *libsignal-protocol-java* and will describe how group messaging is

[5] https://github.com/signalapp

[6] https://github.com/signalapp/libsignal-protocol-java

performed within Signal. We will use direct messaging as a baseline and observe what differences take place in order to send and receive group messages.

5.3 signal-cli

In addition to the official Signal Application, there are also programs which aren't official Signal applications but still employ the same Signal libraries and interact with the Signal servers such that all the same features of the official application are available on the unofficial one. An example of an unofficial Signal application includes the *signal-cli* tool, hosted by *AsamK* on Github[7]. We will be using this tool to assess the group messaging functionalities of Signal by altering the *libsignal-protocol-java* library hosted within the application. We weren't able to get any official version of the Signal Application under a debugger; this may be the subject of future work.

The *signal-cli* tool serves as a command-line interface for the *libsignal-protocol-java* library, and provides support for key generation, verification, as well as sending/receiving messages via the Signal servers. Focusing on the sending/receiving features of the application, *signal-cli* allows us to send and receive messages from a group, on the condition that the client provides a unique group identifier. Moving forward, we will discuss how we will alter the Signal library that is being used in order to get the information we need to further understand group messaging.

5.4 Modifying *signal-cli*

The following steps will describe how we will edit the *libsignal-protocol-java* library held within the *signal-cli* tool.

1. Clone the *libsignal-protocol-java* and *signal-cli* repositories from GitHub.
2. Decompress the *libsignal-protocol-java* library located within the *signal-cli* source code. In order to not confuse these two libraries, we will refer to the library located within the *signal-cli* repository as $libsignal\text{-}protocol\text{-}java_{CLI}$, and the other as $libsignal\text{-}protocol\text{-}java_{SRC}$. This decompression operation will result in the creation of a new directory containing subdirectories of *.class* files. These *.class* files are the compiled, but uncompressed *.java* source code files that make the Signal Protocol.
3. Locate and edit the file you wish to alter within the $libsignal\text{-}protocol\text{-}java_{SRC}$ library, and copy it into the top directory within the $libsignal\text{-}protocol\text{-}java_{CLI}$ archive. Compile the altered *.java* file, converting it into a *.class* file, and replace it with its respective *.class* file in the $libsignal\text{-}protocol\text{-}java_{CLI}$ archive directory.
4. Finally, recompile the $libsignal\text{-}protocol\text{-}java_{CLI}$ directory, converting it back into a Java archive (with a *.jar* file extention). Copy this Java archive to where it was originally located within the *signal-cli* repository.

After following these steps, running the *signal-cli* binary will in-turn run the Java source code that you edited when verifying yourself to the Signal server, as well as when sending and receiving both direct and group messages. While considering the best method of mapping out the protocol while including as much information as possible, we found that including all HKDF output information for each sent and received message would give us an appropriate mapping of the Double Ratchet execution. In order to accomplish this, we need to print the root key, sending/receiving chain key, message key, and padded plaintext message for each direct and group message that is sent and received. Within the Java source code of the *libsignal-protocol-java* library, there are the following functions that we will print information from in order to make this happen:

- *org/whispersystems/libsignal/SessionCipher.java*
    - *CiphertextMessage encrypt(byte[] paddedMessage)*
    - *byte[] decrypt(SignalMessage ciphertext, DecryptionCallback callback)*

[7] https://github.com/AsamK/signal-cli

- *org/whispersystems/libsignal/groups/GroupCipher.java*
    - *byte[] encrypt(byte[] paddedMessage)*
    - *byte[] decrypt(SignalMessage ciphertext, DecryptionCallback callback)*

Within the *groups* and *ratchet* sub-directories in the *libsignal-protocol-java* library, there are plenty more files and methods that provide us with valuable information, however we will be focusing on these two "*encrypt*" and *"decrypt"* methods in particular. Both of these encryption methods take a padded plaintext message as an input. The direct-message encryption method returns a CiphertextMessage-type object which contains Signal Protocol information and the actual serialized ciphertext, while the group-messaging encryption method only returns the ciphertext as a byte array. Similarly, with the decryption methods, both provide a byte array of ciphertext as an input and a plaintext byte string as an output. For each of these, we are able to add code which prints out a variety of information that is available within the method, which includes the following:

- Which method was called (direct/group encryption/decryption)
- Root chain key
- Sending/Receiving chain key
- Message key
- The encrypted/decrypted plaintext message

5.5 Running the Altered *signal-cli*

First, we want to see how direct messaging is mapped in our Double Ratchet execution. As a hypothesis, we expect the *signal-cli* application to hit the direct message encryption/decryption methods when sending and receiving messages. From the figure below, we can see two messages being sent from our command prompt to the same user. By doing this, we can see that the root key is the same, while all the other keys are different. This would make sense as the root key can stay the same while progressing the sending key chain in our ratchet.

```
19:25 mjansen@macbook-pro /Users/mjansen/Desktop/signal/signal-cli-0.6.5/bin
% ./signal-cli -u +13609791283 send -m "hi" +15038287794
*** Direct Message Encryption Triggered ***
*** File/Method :: SessionCipher.java / CiphertextMessage encrypt() ***
****** Root Key: 43F92C1466FE23D44C093447113369A92899A60B4BAFBE16C0E57E98A8D84548
****** Sender Chain Key: 8952CC9D1E65BE82B76118CBD0B84C5AC6FAC55D1045EA1C102F9D1BE7813E26
****** Message Key: 6C3D12DF1167742698EE44E7A964D62D9F11199D1AF2DEB12819ADC7E33448E7
****** Padded Plaintext: -hi2 m@@[@@@L@@,u@@^I2b@q@@@@w8@@@-@

19:28 mjansen@macbook-pro /Users/mjansen/Desktop/signal/signal-cli-0.6.5/bin
% ./signal-cli -u +13609791283 send -m "hi" +15038287794
*** Direct Message Encryption Triggered ***
*** File/Method :: SessionCipher.java / CiphertextMessage encrypt() ***
****** Root Key: 43F92C1466FE23D44C093447113369A92899A60B4BAFBE16C0E57E98A8D84548
****** Sender Chain Key: 8D370D832D0C210C08EE43A0364BAE79A33EF11BD4C853175A918F619E017F3A
****** Message Key: 65F91BF1F031BF3E6976FDA2462626831C85980FE6C0C47352F8CE140F6102D6
****** Padded Plaintext: -hi2 m@@[@@@L@@,u@@^I2b@q@@@@w8@@@-@

19:29 mjansen@macbook-pro /Users/mjansen/Desktop/signal/signal-cli-0.6.5/bin
%
```

Figure 11: Direct Messaging Using Altered signal-cli #1

After waiting a short amount of time (approximately 20 minutes), we repeated this operation, this time only sending one message. Note that each key, including the root chain key, the sending chain key, and the message key, are all different. This implies that between the two users, an asymmetric ratchet took place, replacing the root chain key and the other underlying chain keys as well.

```
19:44 mjansen@macbook-pro /Users/mjansen/Desktop/signal/signal-cli-0.6.5/bin
% ./signal-cli -u +13609791283 send -m "hi" +15038287794
*** Direct Message Encryption Triggered ***
*** File/Method :: SessionCipher.java / CiphertextMessage encrypt() ***
****** Root Key: 01A511D92C19D700A5040CE2A95BB167EB569F3EBE7A46D5B0E0711BA8FE15EE
****** Sender Chain Key: D3D88F6C0F47DFE50EE9B9DF3B0484A250F7E2591FE6905E7B7B19C55981E639
****** Message Key: B543102331C229A83F4E8E179DB0ED93CA18410FDE3A5440DE868CF29762A91F
****** Padded Plaintext: -hi2 m@@[@@@L@@,u@@^I2b@q@@@@w8@@@@-@

19:44 mjansen@macbook-pro /Users/mjansen/Desktop/signal/signal-cli-0.6.5/bin
%
```

Figure 12: Direct Messaging Using Altered signal-cli #2

Now that we've seen the behavior of direct messaging, we may begin observing how group messaging operates. We expect that the *signal-cli* application will use the *GroupCipher* methods, as well as others within the *groups* directory in *libsignal-protocol-java*. In order to begin group messaging within the *signal-cli* application, we will first create a group and send an initial group message using the mobile application (in this case, the iOS Signal Application was used). Then, we will obtain the group-ID by receiving a message within the *signal-cli* application and use that group-ID as an argument to send group messages. We show how this is done in the figures below.

```
*** Direct Message Decryption Triggered ***
*** File/Method :: SessionCipher.java / byte[] decrypt() ***
****** Root Key: C3258FAF902870CAEA0ED2A74E43C8C46B5F04DFFB9EF41E63F48B86C7DCDD45
****** Padded Plaintext: ^Test first group message!@@@@@Gf@f@q@T(2 @@@@K`9U@[c@@D@@@@}␣%@@@␣JW@8@琐@-`@

Envelope from:  (device: 0)
Timestamp: 1579239617513 (2020-01-17T05:40:17.513Z)
Sent by unidentified/sealed sender
Sender: +15038287794 (device: 1)
Message timestamp: 1579239617513 (2020-01-17T05:40:17.513Z)
Body: Test first group message!
Group info:
  Id: 4n8VovWnp0dm6WbkB3HOVA==
  Name: Test
  Type: DELIVER
Profile key update, key length:32
```

Figure 13: Receiving the initial group message, including the group-ID

```
22:17 mjansen@macbook-pro /Users/mjansen/Desktop/signal/signal-cli-0.6.5/bin
% ./signal-cli -u +13609791283 send -g 4n8VovWnp0dm6WbkB3HOVA== -m "hi"
*** Direct Message Encryption Triggered ***
*** File/Method :: SessionCipher.java / CiphertextMessage encrypt() ***
****** Root Key: BF594A8DF84F28ED077A160D52EA41484A7B6E2C775D41CC07C616EEC6FF1469
****** Sender Chain Key: C5BEE0B36A6047B6A08BE275661DEE22CD738B761175023C3727E4BA8E9E6D8F
****** Message Key: 31971DBEE29287408E1E7E39C37DA97F7E8B9F97B1ADB9FB37AE83CDC7FCE699
****** Padded Plaintext: !hi@@@@Gf@f@q@T8@@-@

*** Direct Message Encryption Triggered ***
*** File/Method :: SessionCipher.java / CiphertextMessage encrypt() ***
****** Root Key: C3258FAF902870CAEA0ED2A74E43C8C46B5F04DFFB9EF41E63F48B86C7DCDD45
****** Sender Chain Key: 158CBFACF3ED70CB8793341FB38957ED412CDB0141B8992D4C4DE9F06D63CCA0
****** Message Key: CD7A5BC8FFF55BA45FC45910CB454F0C6211C1840D92CFD414C2B5D7AC0F4C85
****** Padded Plaintext: !hi@@@@Gf@f@q@T8@@-@

22:17 mjansen@macbook-pro /Users/mjansen/Desktop/signal/signal-cli-0.6.5/bin
%
```

Figure 14: Sending a group message using the group-ID

As you see from the above figures, it appears that the group messages are being sent using the direct messaging methods. Although print statements were entered in every method of the *groups* directory, sending and receiving messages using the group identifiers always results in direct message encryption and decryption methods. Even after sending consecutive group messages with either a small or large amount of time between each message resulted in either none or multiple asymmetric ratchets, respectively, similar to direct messaging.

This concludes our experiment. Now, we will use the above-mentioned data that we have collected to characterize and comment on Signal's group messaging algorithm.

5.6 Examining Our Results

In summary, we can summarize our results by the following:

- The act of sending and receiving messages destined for a single host will execute the methods associated with the encryption and decryption of messages meant for one destination.
- Sending messages to the same destination host, one after the other in a quick fashion, resulted in two symmetric ratchets and no asymmetric ratchet. Sending two direct messages with several minutes in between each message resulted in two symmetric ratchets and an asymmetric ratchet. It is unknown if the number of messages or the time difference between the messages caused the asymmetric ratchet.
- The act of sending and receiving messages destined for multiple hosts will execute the methods associated with the encryption and decryption of messages meant for one destination. In all, the sending and receiving of messages tagged with a unique group-ID resulted in no methods from the *groups* directory within the *libsignal-protocol-java* library to be called.
- As with direct messaging, sending several group messages with little time between each message resulted in no asymmetric ratchet, while sending them over a large amount of time resulted in at least one asymmetric ratchet.

After reviewing these results, we arrived at the conclusion that each user in a group being tied to a unique group-ID, and aside from that, using the direct messaging protocol to communicate with each host. Having a group protocol that relies heavily on underlying direct message communication such as this meets all of the group messaging security properties that were discussed earlier.

The first property, *computational equality*, includes having each group participant share computational load. This is true as each member of the group decrypts the same message that is received by the group, no member needs to compute more than another to decrypt the same message. The next property, *trust equality*, ensures that no participant takes on more responsibility than another. This is also true, as each member assumes the role of an administrator of the group – there exists no group member who is not allowed to add another group member or edit the group description. *Subgroup messaging* allows users to send a message to a subset of participants without forming a new conversation, which is possible since Signal relies on direct messaging and would only constitute leaving a member out of the recipients list of the group. Lastly, we have *contractible and expandable membership*, allowing group members to leave or join (respectively) the group without having to start a new protocol run. Simply leaving or joining the group and sending an update query to the remaining members of the group after the operation is complete allows these properties to be met.

In addition to these group messaging properties being met, we may assume that the original direct messaging security properties are met as well, as the direct messaging protocol remains unchanged through the group messaging algorithm. In essence, our results show that the *signal-cli* implementation of the Signal Protocol, which uses the *libsignal-protocol-java* library from Open Whisper Systems, upholds both the direct and group security properties that are sought after by end-to-end encrypted messaging applications. If we recall, our hypothesis entailed exploring Signal's group messaging implementation by observing the flow of execution from within the files in the *group* subdirectory. Although our hypothesis was not initially met due to direct messaging functions being called instead of group messaging functions, we were still able to observe Signal under a group context and document the behavior of the application.

6 Conclusion

Through discussing the design and implementation of cryptographic primitives, using them to build the Signal Protocol, and modeling the environment to which applications that use the Signal Protocol operate, a firm understanding may be built regarding how the Signal Protocol operates in a direct-message setting. Initially, we argued that Signal's documentation around group messaging is lackluster, which is concerning as it is important to understand to ensure the security properties of the protocol are upheld. Through tests which included command-line interfaces to Signal's Java implementation of the protocol, we communicated in a direct and group setting to multiple devices and compared the keys which were used to move forward in the ratchet, as well as to encrypt and decrypt messages. By doing so, we found that the protocol for group messaging rests upon several executions of the direct messaging algorithm. Future work includes replicating this experiment over numerous implementations of the Signal Application, such as over official desktop and mobile applications.

REFERENCES

[1] Patricia S. Abril and Robert Plant. 2007. The patent holder's dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan. 2007), 36-44. DOI: https://doi.org/10.1145/1188913.1188915

[2] Sarah Cohen, Werner Nutt, and Yehoshua Sagic. 2007. Deciding equivalences among conjunctive aggregate queries. J. ACM 54, 2, Article 5 (April 2007), 50 pages. DOI: https://doi.org/10.1145/1219092.1219093.

[3] David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY.

[4] Sten Andler. 1979. Predicate path expressions. In *Proceedings of the 6th. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. ACM Press, New York, NY, 226-236. DOI: https://doi.org/10.1145/567752.567774.

[5] Patricia S. Abril and Robert Plant. 2007. The patent holder's dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan. 2007), 36-44. DOI: https://doi.org/10.1145/1188913.1188915

[6] Sarah Cohen, Werner Nutt, and Yehoshua Sagic. 2007. Deciding equivalences among conjunctive aggregate queries. J. ACM 54, 2, Article 5 (April 2007), 50 pages. DOI: https://doi.org/10.1145/1219092.1219093.

[7] David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY.

[8] Sten Andler. 1979. Predicate path expressions. In *Proceedings of the 6th. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. ACM Press, New York, NY, 226-236. DOI: https://doi.org/10.1145/567752.567774.

[9] Patricia S. Abril and Robert Plant. 2007. The patent holder's dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan. 2007), 36-44. DOI: https://doi.org/10.1145/1188913.1188915

[10] Sarah Cohen, Werner Nutt, and Yehoshua Sagic. 2007. Deciding equivalences among conjunctive aggregate queries. J. ACM 54, 2, Article 5 (April 2007), 50 pages. DOI: https://doi.org/10.1145/1219092.1219093.

[11] David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY.

[12] Sten Andler. 1979. Predicate path expressions. In *Proceedings of the 6th. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. ACM Press, New York, NY, 226-236. DOI: https://doi.org/10.1145/567752.567774.