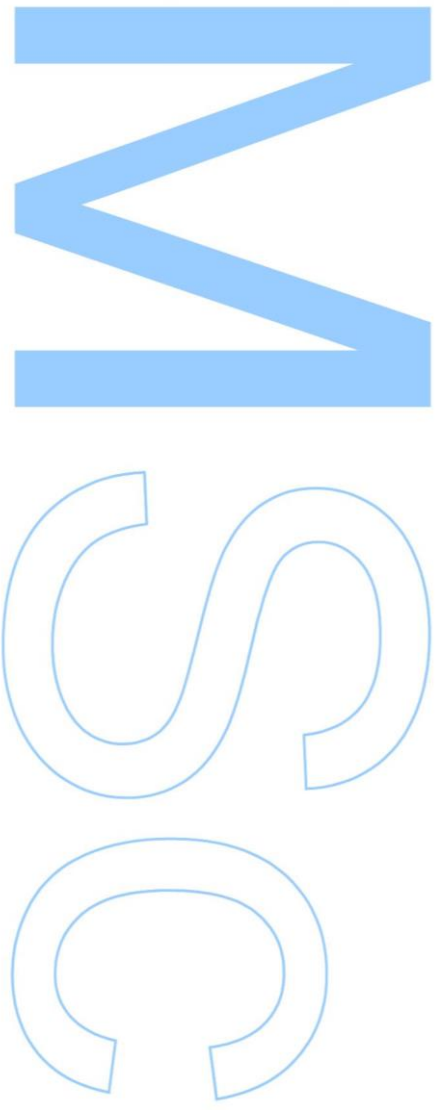
Security Analysis of a Closed-Source Signal Protocol Implementation

João Diogo Gaspar Alves
Mestrado em Segurança Informática
Departamento de Ciências de Computadores
2018

Orientador
Manuel Bernardo Martins Barbosa, Faculdade de Ciências da Universidade do Porto

Coorientador
Pedro Ricardo Morais Inácio, Faculdade de Engenharia da Universidade da Beira Interior



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Abstract

In recent years, with the reveal of mass surveillance programs targeting online communication came a growing interest for properties such as privacy and security of that very same communication. One way to ensure those properties is by using cryptographic protocols such as the Signal Protocol to secure the communication channel. However, this protocol is fairly recent therefore there is a limited amount of study regarding its security and its implementations' security. With this work, we seek to tackle that problem by providing some more knowledge regarding this specific topic.

After performing a relevant literature review in order to understand the state of the art and also to identify the underdeveloped areas that should be focused, we performed a security analysis of a popular Signal Protocol implementation. For that, we took a *hands-on* approach which involves reverse engineering an application using said implementation, analysing its interaction with the filesystem, dynamically tracing some parts of the protocol in execution and testing the protocol's behaviour in an unlikely yet possible device cloning scenario. A secondary objective of this work, resulting from the approach taken, is to incentivize the analysis and audit of closed-source applications and demystify the associated difficulty and complexity. In this work, we intentionally refrained from analysing network data and elliptic-curve and signature scheme implementations as it was considered to be out of scope.

Overall, our results matched the initial expectations set by the existing pertinent literature, with no major flaws being detected. However, our last test scenario did generate some erratic and elusive behaviour that could be considered a small security vulnerability and a hint of the existence of potential serious vulnerabilities, if some of the observed behaviour is controllable.

In the end, we contributed with yet another independent analysis using a different approach that found no major problems with the protocol and its implementations, besides our specific test scenario, in accordance with the existing literature. In the specific test scenario we found some interesting investigation clues to be pursued in future research.

Resumo

Nos últimos anos, a descoberta de programas de vigilância de larga escala direcionados para a comunicação online tem gerado um aumento do interesse acerca de propriedades como a privacidade e a segurança dessas mesmas comunicações. Uma forma de garantir essas propriedades é através da utilização de protocolos criptográficos como o Signal Protocol para protegerem o canal de comunicação. No entanto, este protocolo é relativamente recente logo a quantidade de estudos sobre a segurança do protocolo e das suas implementações é relativamente pequena. Com este trabalho, pretende-se contribuir para a redução desse problema acrescentando mais conhecimento acerca deste tópico especifico.

Depois de realizada uma revisão da literatura mais relevante de forma a identificar o estado da arte e as áreas de estudo subdesenvolvidas que deveriamos focar neste trabalho, foi realizada uma análise de segurança a uma implementação popular do Signal Protocol. Para isso foi utilizada uma abordagem prática que envolve a utilização de técnicas de engenharia reversa sobre uma aplicação específica, a análise da interação desta aplicação com o sistema de ficheiros subjacente, a deteção dinamica de chamadas de funções por parte da aplicação de forma a mapear fragmentos do protocolo e finalmente testar o comportamento da aplicação e do protocolo num cenário de teste improvável mas possível em que um dispositivo que utiliza a aplicação é clonado. Um objetivo secundário deste trabalho, fruto da abordagem escolhida, passa por incentivar a análise e auditoria de aplicações *closed-source* e desmistificar as dificuldades associadas. Neste trabalho evitamos intencionalmente analisar informação associada a tráfego de rede e implementações de esquemas de assinatura ou curvas elípticas por considerarmos que ultrapassavam o âmbito do mesmo.

Em geral, os resultados obtidos correspondem ás expectativas iniciais definidas pela revisão de literatura inicial, não tendo sido encontradas falhas significativas. No entanto, o último cenário de teste gerou algum comportamento errático e obscuro que pode ser considerado uma vulnerabilidade pequena e que pode ainda indiciar a existência de possíveis vulnerabilidades mais sérias caso seja

possível controlar este comportamento.

Por fim, contribuímos para o nosso problema central através da apresentação de mais uma análise independente que utiliza uma abordagem diferente e que não encontrou falhas ou problemas significativos no protocolo e respectivas implementações para além do nosso caso de teste específico, de acordo com a literatura existente. No caso de testes analisado na fase final do trabalho foram encontrados indicios significativos que justificam investigações futuras.

Acknowledgments

Most of all, I would like to thank my parents and my brother who have certainly been the biggest supporters not only of my education but also my personal development. Secondly, I would like to thank my friends, especially André Baptista and Miguel Cardoso, for the occasional help but mostly the opportunities to unwind and relax. Lastly, I would like to thank Ole Ravnås for taking the time to answer my questions regarding the Frida toolkit.

In addition, a very special thanks goes out to my advisors, professors Manuel Barbosa and Pedro Inácio, for the guidance, availability and help provided throughout this chapter of my life.

Contents

List of Tables

List of Figures

Listings

Acronyms

AD Associated Data

ADB Android Debug Bridge

AEAD Authenticated Encryption with Associated Data

API Application Programming Interface

APK Android PacKage

ASCII American Standard Code for Information Interchange

AVD Android Virtual Device

DEX Dalvik Executable

DH Diffie-Hellman

DHR Diffie-Hellman Ratchet

DoS Denial-of-Service

DRA Double Ratchet Algorithm

E2EE End-to-End Encryption

ECC Elliptic-Curve Cryptography

EK Ephemeral Key

GUI Graphical User Interface

HKDF HMAC-based Key Derivation Function

HMAC Hash-based Message Authentication Code

IDE Integrated Development Environment

IK Identity Key

IMEI International Mobile Equipment Identity

JAR Java ARchive

KDF Key Derivation Function

MAC Message Authentication Code

OPK One-time Pre-Key

OWS Open Whisper Systems

PRF Pseudo-Random Function

RTP Real-time Transport Protocol

SIM Subscriber Identity Module

SK Secret Key

SMS Short Message Service

SPK Signed Pre-Key

X3DH Extended Triple Diffie-Hellman

XML Extensible Markup Language

Chapter 1

Introduction

The Signal Protocol is a cryptographic protocol developed in 2013 by Open Whisper Systems (OWS), that provides End-to-End Encryption (E2EE) between communicating parties. Existing implementations allow for voice calls, video calls, text messaging and multimedia messaging. The protocol is notorious for its usage in popular mobile messaging applications such as WhatsApp [27], Facebook Messenger [25], Google Allo [26] as well as its homonym and original application Signal [42], and also for some other high profile adaptations coming in the near future such as Skype [22].

1.1 Motivation and Scope

With the growing concerns around privacy and the exposed large scale mass surveillance programs affecting various sorts of online communication [16], the technologies that offered some surveillance protection started to receive additional attention. Another factor that also increased the popularity of such technologies was the creation of products that hide the burden and complexity associated with this protection from the user, providing intuitive, user-friendly, functional alternatives with security as an added bonus against mass surveillance. One of such technologies is the Signal Protocol briefly mentioned earlier.

Due to the widespread popularity of the applications mentioned earlier for using the Signal Protocol, a lot of warranted security concerns have been raised. However, due to the protocol and its various application implementations' recency and fast evolution, the existing work in this security field is still very limited, especially when compared to the wide reach and impact of this technology. In the identification of this knowledge gap lies the main motivation behind this work.

In this work, we will attempt to take a more *hands-on* approach by using reverse engineering and dynamic instrumentation techniques in our analysis, as a way to bridge the gap between security analysis of open-source and closed-source software. Initially, we are going to review the Signal Protocol itself, and later, we will analyse one of its most relevant implementations by using the previously mentioned reverse engineering and dynamic instrumentation techniques but also by artificially manipulating the environment in order to recreate the conditions necessary for our experiments.

The scope of this analysis will mainly focus on the cryptographic and session mechanisms and their interaction with the application's host environment. It should also be noted that we will intentionally refrain from analysing any network data, the XEdDSA and VXEdDSA signature schemes [35] and the Curve25519 elliptic curve implementation [5], as we consider this to be outside of the scope of this work.

1.2 Problem Statement and Objectives

One big gap we found in the available and reviewed literature was the lack of application auditing or implementation inspection that involved extracting and analysing real world closed-source implementations and not just analysing the existing open-source implementations. Therefore, with this analysis we will attempt to help tackle our main problem, which is the study and analysis deficit regarding the Signal Protocol, particularly centred around existing closed-source applications. The contributions we seek to provide will be explained in greater detail in section 1.4.

Another smaller issue that this work looks to address is the intimidating and deterring view that surrounds closed-source software auditing and analysing. We will attempt to use a couple different techniques such as reverse engineering and dynamic instrumentation to show that with the existing analysis tools and techniques for software analysis, the workload and difficulty gap between open-source and closed-source analysis is much smaller than generally perceived.

In order to answer these problems, we defined a set of objectives against which we will later compare what was accomplished. These objectives are:

- to provide yet another explanation of the Signal Protocol, to help demystify it and its apparent complexity, and also as a critical component deserving of an introduction due to its relevance to the rest of our analysis;
- to extract a protocol implementation from the most impactful application in order to validate

the existence of the protocol's major components and its functioning;

- to identify how the application interacts with its operating system's filesystem, more specifically how it handles the long term data storage, and what risks this might represent, which is an underexamined area in the recent relevant literature;
- to use dynamic instrumentation to allow function tracing to analyse the protocol's execution on the application, validating the knowledge acquired in previous steps by recording them during runtime;
- to assess the application's behaviour in a specifically devised scenario, which is another understudied topic which some of the existing literature identified as notable for future works [2].

1.3 Approach

After defining the problems we attempt to tackle with this work, as well as the objectives that we will look to accomplish in order to tackle those problems, we can now outline the approach that will be taken.

We will begin by selecting the most relevant application using the Signal Protocol, and attempt to reverse engineer it, in order to access its protocol implementation. We will use the results of this reverse engineering process to attempt to identify the protocol and its different components, or at least, identify the implementation used by matching it to any of the existing open-source Signal Protocol implementations.

Then, we analyse the filesystem to try and identify how the application interacts and stores its information on the system, which includes sensitive information such as communication data and also cryptographic material.

After this, we attempt to use a fairly novel technique which consists of using dynamic instrumentation to trace and possibly manipulate function calls during the application's runtime, and if possible, trace the protocol execution or parts of it. Another goal is to also verify that the implementation previously analysed is truly being executed during runtime.

Finally, we intend to design, recreate and analyse the application's behaviour in an unlikely yet possible scenario consisting of cloning the application data to another device and studying the application and protocol's behaviour.

1.4 Main Contributions

The main contributions we assume this work will have can be enumerated as follows, with the order representing its expected impact:

1. The contribution to the specific field, pertaining to the Signal Protocol's security by taking a different and more empirical approach where we do not blindly trust what we are told about the implementation. Instead we extract the implementation from the application itself by using reverse engineering techniques, analyse the extracted implementation and how it interacts with the filesystem, namely for storage purposes, then validate our findings using dynamic instrumentation to trace the implementation in action, and conclude by simulating an specific scenario.

2. To attract attention to the fairly recent although still unstable but very powerful analysis technique that is dynamic instrumentation using the Frida toolkit [36], particularly in mobile environments, which hopefully, given its open-source nature, may inclusively speed up the project's development. We also contribute to the Frida toolkit by releasing our adapted scripts for Java method and class tracing.

3. Clarify and expose the Signal Protocol so that its security and strengths can be better understood and apparent even to the more sceptical.

1.5 Document Overview

This document can be divided in two major parts. First we have a theoretical part, which starts with a brief introduction to the work developed in this project. This is followed by a state of the art chapter dedicated to introducing some of the basic concepts and ideas used later in the document, as well as a review of the most relevant specialized literature and the most important works. After this, we have a chapter dedicated to presenting the Signal Protocol and other related mechanisms, concluding the theoretical part of this document. Then, the empirical part begins with a chapter describing the selected approach, where we choose a target application, describe our testing environment, define some brief testing guidelines and introduce some of the tests we will implement. This is followed by a chapter describing precisely the implementation of the previously mentioned tests and yet another chapter where we present and discuss the main results and make some improvement suggestion.

Finally, there is one last chapter where we draw the main conclusions regarding this work, and explore its limitations as potential clues for future work.

In appendix we have some of the scripts, code excerpts and logs that were considered too large to be part of the main body of the document, but are also integral to this work.

Chapter 2

State of the Art

2.1 Introduction

In this chapter we begin by briefly introducing, in section 2.2, some of the simpler notions and concepts that form the foundation or allow for easier understanding of the ideas and technologies used in this work. Then, in sections 2.3 and 2.4, we will also cover the relevant specialized literature work that contributes to the specific branch of knowledge that this work also focuses. We start by presenting the more relevant scientific works (in section 2.3), and finish with a few specific attacks and analyses presented more informally that have targeted some of the Signal Protocol's implementations (in section 2.4). It should be noted that some of the work mentioned in this chapter was only published recently, during this security analysis.

2.2 Background

This section contains a brief explanation of some of the key concepts and ideas that will greatly aid the full comprehension of this particular work and also the relevant reviewed literature we will cover in sections 2.3 and 2.4.

2.2.1 Basic Security Principles

We will begin by defining the basic information security principles or properties used to characterize and even evaluate the level of security provided by a specific technology.

- Confidentiality - The principle of confidentiality concerns the protection of private information by restricting the access to said information to the intended parties only.
- Integrity - The principle of data integrity concerns the assurance of consistency, accuracy and reliability of data, as a critical component of a system which uses and/or stores data. It is often important to validate this principle throughout the entire life span of the data, or at least from its entry point to its exit point in a specific system.
- Availability - The principle of availability concerns the uptime of the system, so that its functionalities and information is available on demand. As an example, Denial-of-Service (DoS) attacks specifically target this principle.
- Authentication - The principle of authentication concerns the process of proving an identity. This is a critical step in order to achieve confidentiality.
- Non-repudiation - The principle of non-repudiation or deniability concerns the ability of a user denying his past actions. This principle is critical for assigning responsibility as consequence of actions taken by any agent in a system.

2.2.2 Basic Cryptographic Concepts

We will now define a few cryptographic concepts and ideas that will help fully understand the Signal Protocol presented later, as well as its core mechanisms.

- End-to-End Encryption (E2EE) - E2EE is a property of communication systems in which, the communications are encrypted on the sender's device and are only decrypted on the intended recipient's device. This means that any passive network eavesdroppers will be unable to gain information regarding the contents of the previously mentioned communications. This kind of systems requires all parties to have a pre-shared secret or to negotiate a new shared secret on the spot using key agreement protocols such as the Diffie-Hellman (DH) key exchange.
- Forward Secrecy - The cryptographic principle of forward secrecy or perfect forward secrecy, specifically concerns key agreement protocols and is a property that assures that the compromise of the long-term secrets cannot compromise older, established sessions [11]. Regarding the Signal Protocol we will analyse later, this is done by mostly using the long-term secret for authentication during the key agreement negotiation and by using additional ephemeral keys in this process, that are deleted after usage.

- Future Secrecy - The not very common cryptographic principle of future secrecy was defined by Moxie Marlinspike from OWS as the *self-healing* capability of the mechanism now called Diffie-Hellman Ratchet (DHR), which is able to continue to generate secure unpredictable keys even after the compromise of its preceding cryptographic material [7, 24]. The DHR will be explained in detail in section 3.3.3.

- Pseudo-Random Function (PRF) - A PRF is an efficient, deterministic function which takes, at least, a hidden or explicit seed and a piece of data as inputs, and whose output cannot be realistically distinguished from a source of random data. These functions are then used as sources of pseudo-random data themselves.

- Message Authentication Code (MAC) - A MAC is a small piece of information that is attached to a message in order to validate and protect its authenticity and integrity. A possible MAC construction would consist of a PRF with the message as input, and also an additional input key used for authentication, and the output being the attached piece of information that could be used to verify integrity and authenticity. A specific subset of MAC are Hash-based Message Authentication Code (HMAC) which use a cryptographic hash function.

- Cryptographic Ratcheting - A cryptographic ratchet, similarly to a mechanical ratchet wheel, is a device that advances only in one direction, which is realistically impossible to revert. In the specific case of the cryptographic ratchet, a robust hashing function or PRF is used to advance the ratchet, using its own state as an argument, making it extremely hard to reverse, but trivial to advance [7]. This serves similar purposes as that of a PRF, in the sense that is provides a very hard to reverse source of new pseudo-random data.

- Key Derivation Function (KDF) - A KDF is a function which takes a secret and allows the derivation of one or more keys by using a PRF. In the cases where this PRF is a HMAC hash function, this mechanism is named HMAC-based Key Derivation Function (HKDF).

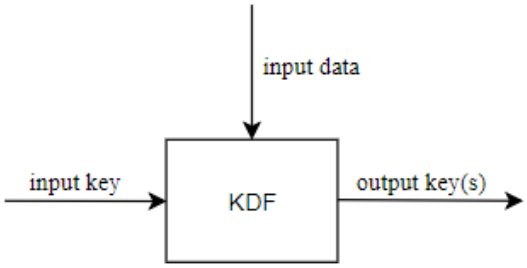


Figure 2.1: KDF example.

- Elliptic-Curve Cryptography (ECC) - ECC is a public-key cryptography variant based on the algebraic structure of elliptic curve over finite fields. It is used as an alternative to the RSA cryptosystem and the main advantage it provides is smaller key sizes.

- Diffie-Hellman (DH) Key Exchange - The DH key exchange is a secure method for cryptographic key exchange between two parties, over public channels, that is very popularly used to establish a secret session key for other cryptographic protocols. It takes advantage of the discrete logarithm problem, or similar problems, to efficiently and safely share a secret between two parties, even if the exchange is eavesdropped [10]. In Figure 2.2 there is a simple, unsafe, example that illustrates the key exchange in action. To note that it is only unsafe due to the simplicity of the numbers used in the example for demonstration purposes.

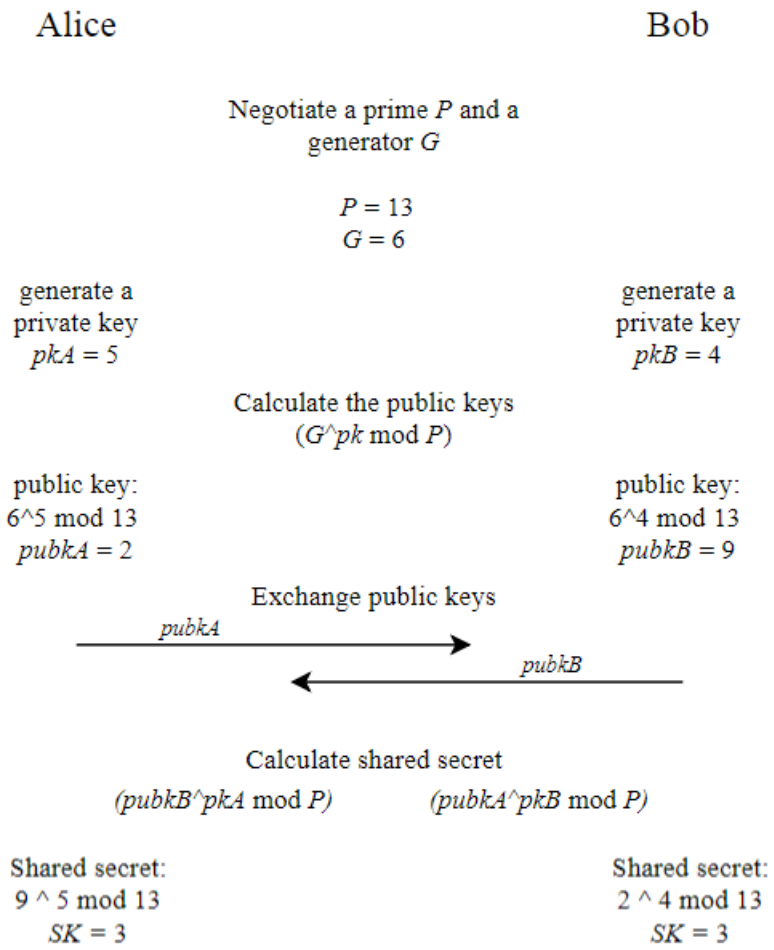


Figure 2.2: DH key exchange example.

2.2.3 Program Analysis Concepts

Here we will define some specific program analysis concepts that will be used later in this work. We begin by defining reverse engineering and some other basic concepts, and later we define some more specific and advanced ideas such as dynamic instrumentation.

- Static Analysis - Static analysis is a type of computer software analysis that does not involve executing the tested software in any way, and instead usually focuses on static components such as the software's source code. Source code reviewing is an example of static analysis.
- Dynamic Analysis - Dynamic analysis is a type of computer software analysis that focuses on monitoring the tested software during its runtime. This type of analysis is generally more complex than static analysis, but allows the detection of problems that are much harder or even impossible to find using static analysis. Traditional software testing generally always includes some type of dynamic analysis. Interactive debugging is an example of dynamic analysis.
- Reverse Engineering - Reverse engineering is the process by which an object is deconstructed back to its various constituents in order to learn more about this object. This practice is used in software engineering and one of its possible usages consists of deconstructing an application by reversing the compilation process in order to retrieve the original program constituents, such as the source code.
- Decompilation - The decompilation process takes an executable file and attempts to reverse the compilation process, ideally outputting a source file which could be compiled again to generate the originally inputed executable. As much of the high level information such variable and function names is usually lost in the compilation process, the output of the decompilation process can be obfuscated or generic pseudo-code.
- Dynamic Instrumentation - Dynamic instrumentation or dynamic binary instrumentation is a technique that allows the analysis of a binary application during runtime by injecting various small pieces of instrumentation code that is executed as part of the regular instruction stream. These small pieces of code can be used to inject debugging code in the target application. This technique is very powerful as often it is impossible to detect certain software bugs and flaws simply by examining the source code.
- Tracing - Tracing, in the context of this analysis, is the usage of dynamic instrumentation to follow a function call during runtime time, offering insight on the application logic by allowing

an easy association between application functionality and executed source code. An example of a tracing script output can be found in Listing C.1.

2.3 Literature Review

One of the more complete and relevant works regarding the Signal Protocol's security belongs to Cohn-Gordon et al. [6], and consists of the first formal security analysis of the protocol. This work focuses exclusively on the Signal Protocol and it attempts to provide a formal description and proof for the security of Signal's cryptographic core. In order to achieve that, this analysis focuses on the key-agreement stage of the protocol and defines a specific threat model that encompasses the following protocol properties:

- Correctness - a completed exchange should generate the same keys for all parties involved, and different exchange should generate different keys;
- Secrecy - a completed exchange should not leak any information to parties not participating in the exchange;
- Authentication - if one party shares a key with another party, then only these two parties should know anything about the shared key;
- Forward secrecy - an attacker that compromises a party's long term secret should not be able to learn anything about keys derived for previous completed sessions.
- Identity hiding - a passive attacker should not be able to identify communicating parties.

Then, a security model is presented and applied to the Signal Protocol, followed by a section that aims to prove the security of the Signal Protocol, as defined in the previous sections.

This formal security analysis concludes with *"(...), our analysis proves that several standard security properties are satisfied by the protocol, and we have found no major flaws in its design, which is very encouraging." (pp. 20).*

Another analysis that focuses on the Signal Protocol and its open-source implementation code belongs to Rubín [39], which explicitly excludes client applications using the protocol from the analysis scope. This is a simpler work that does a brief contextualization of the Signal Protocol and other instant messaging protocols, followed by a protocol analysis that included the XEdDSA and VXEdDSA

signature schemes, the X3DH key agreement protocol and the Double Ratchet Algorithm (DRA). Then, an implementation analysis is performed on the available open-source implementation by analysing the available code. The results obtained by this analysis indicate that no major flaws were found, presenting only some observations such as the lack of implementation of the more secure header encryption variant of the protocol.

Another broader security analysis was conducted by Mujaj [32], and targeted six different mobile applications (Signal, WhatsApp, Wire, Viber, Riot and Telegram) and their respective messaging protocols. These protocols were studied according to their own official documentation, and the six applications using them were tested in a set of common scenarios such as initial contact and partner authentication. These tests were conducted and analysed exclusively by observing the application's behaviour. The observed results for both the Signal and the WhatsApp applications (considered more relevant in the scope of our very own security analysis) indicated no major flaws once again, although some improvement suggestions were made such as allowing the blocking of outbound messages in case a contact changed long term keys and an out-of-band identity authentication had not been made.

Other authors, such as Rösler et al. [38], focused on analysing the security of the Signal Protocol specifically regarding group messaging which, at the time of their work, still represented a big gap in the existing relevant literature. They targeted three specific applications, Signal, WhatsApp and Threema. The approach they present is heavily inspired on the existing literature, by defining a model containing the pertinent security and reliability goals from other works that targeted one-to-one communication to the group communication setting. The security goals for dynamic group conversation can be split into three major categories: content confidentiality, integrity and group management confidentiality.

In their testing methodology they used multiple Android devices to test the different protocols using the respective official applications. However, for the protocol descriptions they used Signal's open-source implementation for the Signal application, but they only used network traffic analysis and unofficial protocol implementations for the remaining two applications. It should be noted that although WhatsApp uses the Signal Protocol, its group message implementation may partially diverge from Signal's as it is not specifically defined by the Signal Protocol.

Amongst the more relevant findings, it was discovered that in the case of WhatsApp, group management messages and reception acknowledgements are very vulnerable to malicious server attackers. We also learn that group messages do not make use of the complete DRA, skipping the DHR stage and thus, sacrificing the Future Secrecy property.

Two other forensic analyses by Thakur [43] and by Anglano [1] focused on the WhatsApp application. However, these analyses date back to 2013 and 2014, which predates the implementation of the Signal Protocol on the WhatsApp application, so they are not very relevant to our work. Still, some of the findings might be relevant for our analysis, namely the way the application interacts with the filesystem which is not specified by the Signal Protocol.

Finally, there is a book chapter by Dai et al. [9] which follows a very similar approach to the one in our work, taking advantage of both static and dynamic analysis techniques such as decompilation and dynamic instrumentation of the target applications. However, the goals of this analysis were to understand the application's chat log database backup encryption mechanism, and the origin of the key used in that encryption, in order to be able to decrypt maliciously stolen backups. In the end, the database encryption and key retrieval mechanisms on new devices were explained, with the encryption key being located on the application directory after retrieval and its retrieval mechanism involving sending the WhatsApp server the compromised device's google account name. They also expanded on the impacts of this vulnerability, and a few recommendation were made.

2.4 Other Sources

In the security field, another popular source of knowledge are technical blogs or conference presentations. This is a common way for busy researchers to spread and publish very specific empirical knowledge without taking the time to write a proper scientific document. We will also take into consideration a few relevant works that fit into this category.

One conference presentation that stands out belongs to Aumasson and Vervier [2], and involves a security analysis on the Signal mobile application and underlying Signal Protocol. The authors define a loose methodology that is used to uncover a few vulnerabilities such as MAC bypass, lack of public key validation allowing the use of invalid keys, type confusion and integer overflow vulnerabilities in C lib callbacks, Real-time Transport Protocol (RTP) packet underflow and message replay attacks. The authors then concluded that their analysis only covered part of the application scope, and more logic bugs or protocol edge case vulnerabilities should be expected.

A recent and interesting blog post, authored by Barda et al. [3], also disclosed a set of vulnerabilities targeting WhatsApp individual and group conversations. These vulnerabilities were the ability to spoof the identity of the sender of a message in a group chat, the ability to locally spoof conversations by forging replies from the other party, and the ability to send private group messages that are only

visible to preselected individuals, but whose replies are visible to everyone.

2.5 Summary

In this chapter we covered some basic ideas and concepts that lay the groundwork for the rest of our work. We also covered the most relevant works from various authors that targeted the Signal Protocol and the WhatsApp application with their different analyses.

We now transition to another theoretical chapter which focuses on presenting an overview of the Signal Protocol, its main components and some additional features.

Chapter 3

Signal Protocol Overview

3.1 Introduction

In this chapter we will briefly dive into the Signal Protocol and its major components, with the goal of providing an overview of the protocol that is the focus of this security analysis. In section 3.2 we will cover the X3DH key agreement protocol which is the first major component, in section 3.3 we will cover the DRA which is the second major component and in section 3.4 we will cover the particular case of group communication.

The Signal Protocol is a cryptographic protocol used for asynchronous communication that provides E2EE as one of its key features. The protocol itself is composed of a collection of smaller components such as the DRA, the elliptic curve implementation Curve25519 and the X3DH.

3.2 Extended Triple Diffie-Hellman

The X3DH key agreement protocol is used to negotiate a shared secret between two communicating parties. The protocol is designed for asynchronous communication where one party has posted information to a server that can be picked up at any point by the other party in order to send back encrypted data or establish the shared secret. In addition to that, the protocol also provides mutual authentication, forward secrecy and cryptographic deniability [29].

3.2.1 Preliminary stage

Before initiating a protocol execution, the X3DH protocol takes three parameters: an elliptic curve implementation, a hashing algorithm and an identifying American Standard Code for Information Interchange (ASCII) string. It is also designed to involve three specific parties. Those parties are two active actors, which we will name Alice and Bob, and one passive actor, the server, which might be comprised of a single or multiple entities.

For the purpose of this explanation, Alice takes the initiative and wants to send some encrypted data in order to establish a secure communication channel with Bob by negotiating a shared secret. On the other hand, Bob wants to be available to receive encrypted data that allows him to establish secure communication channels with any other parties attempting to communicate with him. However, Bob is not always online and Alice doesn't know how to reach Bob, therefore they both need to previously establish a relationship with the server. The server can store messages going either way until the receiving party retrieves them. The server also stores the contact data that Alice and Bob post in order to be reachable by other parties.

The X3DH protocol makes use of the following four distinct types of key pairs:

- Identity Key (IK) which are used as long-term secrets and authentication;
- Ephemeral Key (EK) which are generated on every protocol run;
- Signed Pre-Key (SPK) which are used for a predetermined amount of time;
- One-time Pre-Key (OPK) which are usable only once before being erased.

The outcome of the X3DH protocol should be a 32 byte shared Secret Key (SK).

3.2.2 The X3DH protocol in action

The X3DH protocol can be split into three distinct stages:

- the posting of public identity keys and pre-keys to the server (the contact data mentioned earlier);
- the retrieval of that contact data from the server by the party that will take initiative in establishing communication. This party then sends out the initial message;

- the reception and processing of the initial message.

Each party needs to post a set of its public keys to server. This set is composed by the IK which should only be posted once (although it is possible for it to be altered and require updating), one SPK and respective signature using IK which should be updated periodically, and finally a subset of multiple OPKs which should be replenished once the server notifies that its supply is running low.

As for the private keys, aside from the IK, they should be deleted shortly after their previously posted public counterparts are replaced or consumed. A small delay between consumption or replacement and deletion is advised to account for delayed messages using old pre-key pairs. This deletion step concerns one of the protocol's key features, forward secrecy.

Once the key pairs have been generated and the contact information (public keys) have been posted to the server, it is time to establish communication.

The initiating party (Alice) fetches a contact bundle from the server containing the other party's (Bob) public contact information (IK, SPK, pre-key signature and one OPK provided by the server). The server should then delete this specific OPK used in the bundle so that each key is used only once. Next, Alice validates the signature and respective pre-key with the public IK, and generates a new EK pair. This EK will have a mitigating effect in cases of IK compromise, and also grants cryptographic deniability to the process. The private EK should be deleted as soon as SK is calculated. The SK calculation is:

$$\begin{aligned}
&\text{DH1} = \text{DH}(IK_A, SPK_B);\\
&\text{DH2} = \text{DH}(EK_A, IK_B);\\
&\text{DH3} = \text{DH}(EK_A, SPK_B);\\
&\text{DH4} = \text{DH}(EK_A, OPK_B);\\
&\text{SK} = \text{KDF}(\text{DH1} \,||\, \text{DH2} \,||\, \text{DH3} \,||\, \text{DH4}).
\end{aligned}$$

It should be noted that it is possible for the protocol to function properly without OPKs (in case they are depleted or just not being used). In such cases we ignore the DH4 step that uses the OPK and the final step of the SK calculation becomes:

$$\text{SK} = \text{KDF}(\text{DH1} \,||\, \text{DH2} \,||\, \text{DH3}).$$

To go into a bit more detail regarding the key pairing for the previous DH steps, while the usage of both parties' IKs provides mutual authentication, the lack of direct interaction between IKs also

provides a certain degree of repudiation or cryptographic deniability as the keys that interact with the IKs are bound to be replaced or erased over time.

Alice then sends Bob a message containing her public IK, public EK, an id of Bob's OPK if used, and a message containing, at least, both parties' identity information which is encrypted with an Authenticated Encryption with Associated Data (AEAD) scheme [37] using either SK or another key derived from it and an Associated Data (AD) sequence based on both parties' IKs.

From Bob's point of view he fetches Alice's message from the server and uses the keys provided by Alice alongside his own private keys to calculate the various DH steps and reach the same SK. Then he constructs the standard AD sequence defined for the protocol with each parties' IKs and attempts to decipher the encrypted message. If anything goes wrong, the protocol aborts. Otherwise, the protocol is complete, the OPKs used are deleted for the sake of future secrecy and SK or a key derived from it are used in any subsequent communication protocol between the same parties.

With the shared SK established, we move forward to the next major component of the Signal protocol, the DRA.

3.3 Double Ratchet Algorithm

The DRA can be used to allow two parties who share a secret key to communicate using securely encrypted messages. In the context of the Signal protocol, this shared secret key will be the outcome of the X3DH protocol described previously [28].

For every new message, a new key is derived from a previously used key in a realistically irreversible way so that earlier keys cannot be recalculated. Each party also sends public DH attached to each message and the DH calculations are intertwined with the previously derived keys to prevent the calculation of future keys. These two properties prevent attackers from being able to compromise previous and future sessions, assuming the compromise of private cryptographic material is contained.

The DRA is composed of three major mechanisms we will explain individually.

3.3.1 Key Derivation Function Chains

The first mechanism is the KDF chains, which can be seen as a set of sequentially connected nodes. These nodes represent a predefined KDF which takes some input, and produces some cryptographically

secure output, provided the KDF satisfies the requirements of a cryptographic PRF. The input is usually composed by a key and possibly some more input data. The output should be one or more keys. At least one of the keys in the output is used as input for the next node in the chain. The KDF that is most commonly used alongside the Signal Protocol is the HMAC-SHA256.

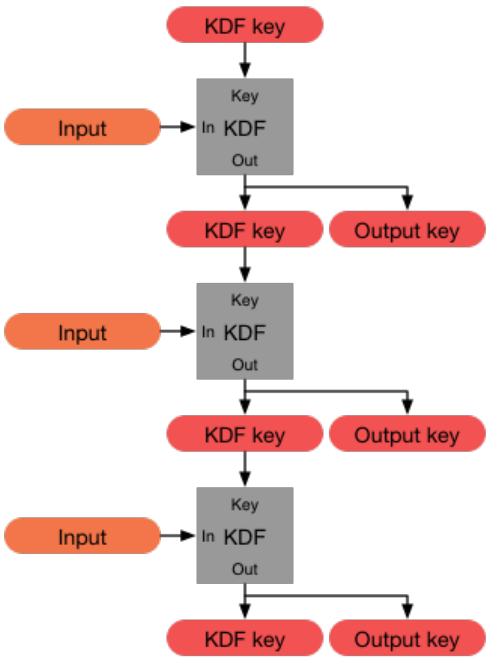


Figure 3.1: KDF chain example [28].

In Figure 3.1 a KDF chain is depicted. The grey KDF boxes represent the chain nodes, the red output keys represent the keys derived with the KDF keys representing the keys used to progress the chain. The extra input data in orange will be used at a later stage.

In the context of the Signal Protocol, every KDF chain should possess the following properties:

- resilience - the keys should appear random to an attacker, even if he has control over the additional KDF inputs (not the key), which can be accomplished with the choice of a secure KDF;
- forward security - previous keys should appear random to an attacker that compromises the KDF chain at some point, compromising current KDF keys, which can also be accomplished with the choice of secure KDF;

- break-in recovery - future keys should appear random and be safe to use even if the KDF chain was compromised at a previous point, which can be accomplished with the usage of a cryptographically secure variable extra data input for the KDF.

3.3.2 Symmetric-key Ratchet

The second mechanism is the Symmetric-key Ratchet, which consists of a specific KDF chain implementation. In this Symmetric-key Ratchet KDF chain, the KDF receives a key and a constant data value as inputs, and outputs a key to be fed to the next iteration of the chain, as well as a second key to be used as a message key. The constant nature of the extra data input means the symmetric-key ratchet does not provide break-in recovery.

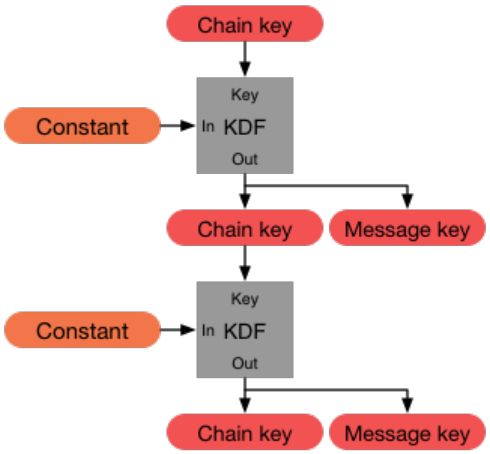


Figure 3.2: KDF chain implementing a symetric-key ratchet example [28].

In Figure 3.2 a KDF chain implementing two steps of a symmetric-key ratchet is depicted. The grey KDF boxes represent the chain nodes, the red output keys represent the keys derived, with the Chain keys representing the keys used to progress the chain and the message keys representing the keys used to encrypt/decrypt the communication messages. The constant extra input data is in orange.

3.3.3 Diffie-Hellman Ratchet

The third mechanism is the DHR. Previously in this document we have briefly described the DH protocol (section 2.2.2). In this section we will be analysing the DH protocol in a ratcheting context, and later we will analyse its usage when intertwined with the Symmetric-key Ratchet to form the DRA.

As soon as the shared secret is established, both parties in a two party communication generate a new DH key pair destined to be used for the DRA. These key pairs will be called ratchet key pairs. Whichever party sends a message should include in its header the public key of its ratchet key pair. When a ratchet public key is received from the other party, a new ratchet key pair is generated and a DHR step takes place. This sequence of events should happen intermittently, with each party generating and sending a public key one after the other.

In Figure 3.3, we can see the initialization process. First, Bob generates his DH key pair and sends Alice his public DH key. Then, Alice generates her own DH key pair and uses it to calculate the initial DH output. This concludes the initialization process for Alice, but not for Bob.

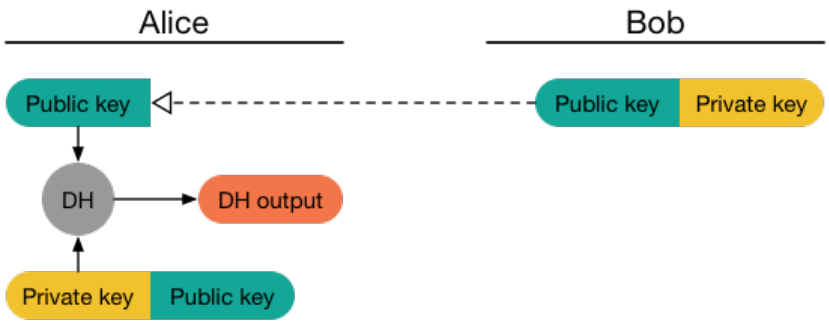


Figure 3.3: DHR example [28].

After this, in Figure 3.4, it is Alice's turn to share her own public DH key in order to allow Bob to calculate the same DH output as Alice's. This concludes the initialization step for Bob. Bob can now generate a new DH key pair (because his last one was already used) and calculate a second DH output using Alice's first public DH key. Then it would be Alice's turn to do this and so on.

Now it is time to integrate the DHR with the Symmetric-key Ratchet described previously. The DH outputs described previously will be used as the extra data input of the KDF described in the Symmetric-key Ratchet. But before that we also need to explain what types of KDF chains make up

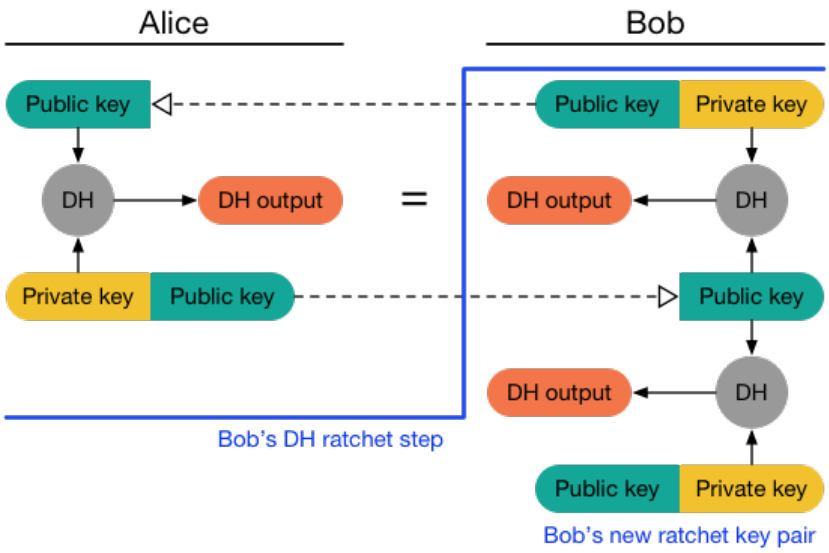


Figure 3.4: DHR example [28]. (cont.)

the Signal Protocol.

For a two party communication, each party needs to store three different types of chains: a single Root chain, any amount of Sending chains and any amount of Receiving chains. Both parties' Root chains should match each other, and each party's Sending chain should have a correspondent opposing party's Receiving chain. The Root chain's initial key should be derived from a shared secret. The extra keys derived from each node in the Root chain should be used as initial keys for Sending and Receiving chains. This symmetry alongside the initial shared secret allows each party to derive the necessary message keys to encrypt and decrypt all communication.

In other words, there is one major chain (Root), whose nodes derive two keys, one to progress the KDF, another to serve as the starting key of a new chain. These new chains will be used to Receive (Decrypt) or Send (Encrypt) the message data. The goal is for the Root chains in communicating parties to mirror each other, while the secondary KDF chain they generate should complement each other (e.g. Bob's Sending chain should correspond to Alice's Receiving chain).

The role of the DHR in this system will be to provide a variable cryptographically secure extra data input to the Root chain's KDF nodes, in order to protect the algorithm against contained cryptographic material compromises. The deletion of older chain information, alongside the unpredictability of the DHR greatly restrain the impact the previously mentioned compromises, thus providing break-in

recovery (mentioned earlier).

In Figure 3.5 we have an example of a Root chain making use of the KDF chain mechanism, the Symmetric-key Ratchet mechanism and the DHR mechanism.

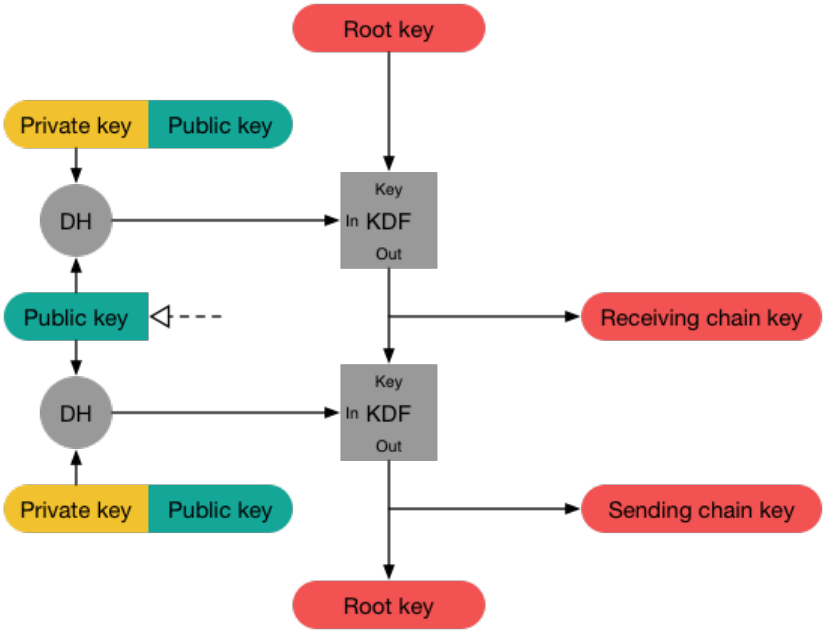


Figure 3.5: KDF chain using a symmetric-key ratchet and a DHR example [28].

3.3.4 Double Ratchet Algorithm in action

As the name indicates, the DRA consists of two intertwined ratcheting mechanisms. On one hand we have the symmetric key ratchet which produces the message keys mentioned previously to encrypt/decrypt the messages themselves. These keys can be deleted once they have been used but can also be stored for longer periods of time without compromising future or past secrecy, as the message keys themselves are not used for advancing any of the KDF chains. On the other hand we have the DHR. This mechanism is used in advancing the Root chain by using the result of a DH calculation between two DH key pairs (one generated by each party) as a data input to the KDF (in addition to the previous node's key). The second key produced by the KDF used on the root chain generates a new Sending/Receiving chain. By default the DHR takes place every communication round-trip, meaning new chains are not generated for consecutive messages, but rather the existing chain is extended until the other party responds.

In Figure 3.6 we have a depiction of the entire DRA for one of the parties (let's assume Alice's). The scheme is divided between the DHR mechanism and the three distinct types of KDF chains. Initially, Alice receives Bob's public DH key and together with her own private DH key, she progresses the Root chain in order to derive the initial key for a Sending chain. Alice then advances this Sending chain in order to derive a message key to encrypt a single outgoing message. Alice should also send her current DH public key so Bob can derive his own Receiving chain. Then, Bob replies to Alice with two messages. This means Alice receives another of Bob's public DH keys and calculates another DH that is used to progress the Root chain once more. This, in turn, will create a new chain, which will be a Receiving chain. Bob sent two messages, therefore this Symmetric-key Ratchet chain will need to progress twice, in order to generate two message keys to decrypt Bob's two messages. And this cycle repeats itself, with Alice generating a new DH key pair that she uses to progress the Root chain, which creates a new Sending chain that she progresses three times in order to send three messages.

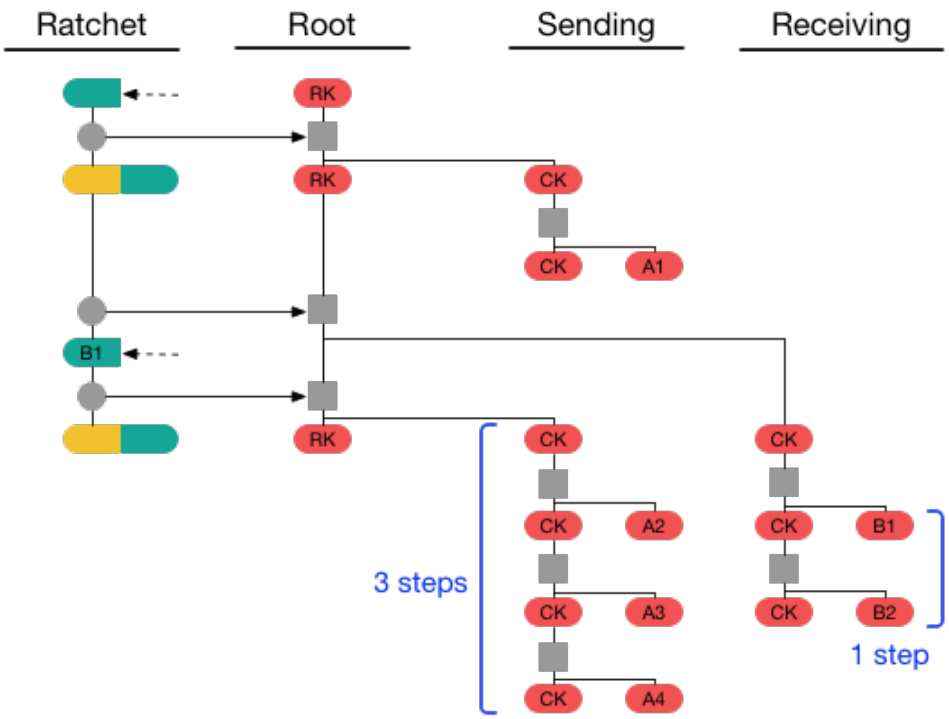


Figure 3.6: DRA in action for one of the parties [28].

As a side note, DRA handles out-of-order messages by adding the message number within the chain to the header, alongside the previous chain's length. This means that a user knows exactly how many messages he is missing and when he can safely delete an older chain if he has already

successfully received all its messages. The protocol is not specific regarding the choice of behaviour to ensure the missing messages are received or to accept their loss.

3.3.5 Double Ratchet Algorithm with Header Encryption

Another slightly more complex implementation of the DRA includes a mechanism called Header Encryption. We will discuss this mechanism briefly although it should be noted there was no evidence of its implementation during our analysis, and the reviewed literature came to similar conclusions, that this mechanism is most likely not implemented in most applications as it is not part of the open-source implementation [39].

The particularity of this implementation of the DRA is that every Root chain KDF node outputs an extra key. This extra key will be used to encrypt or decrypt (depending on the generated chain by that Root chain KDF node being a Sending or Receiving chain) the header of the messages generated by the next chain. The initial header encryption or decryption key for the messages in the first instance of each chain are part of the initially shared secrets, as we can see in Figure 3.7.

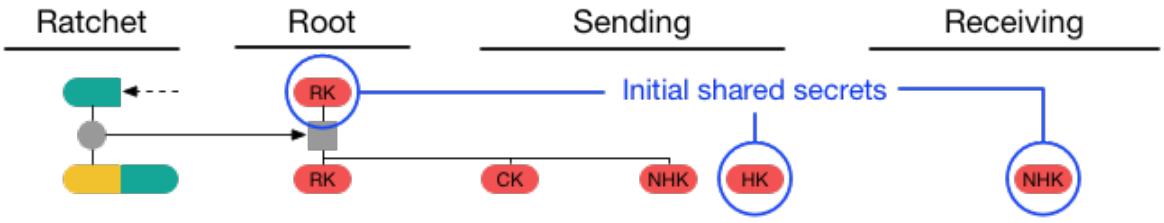


Figure 3.7: DRA with header encryption in action [28].

3.4 Signal Protocol applied to Group Communication

There are a few potential ways to apply the one-on-one Signal Protocol functioning described previously to a group setting, and the protocol documentation does not specify this. The easiest and most intuitive one would be to decompose group communication to a series of one-on-one interactions. However, following this approach there would be a very large number of KDF chains and messages being sent and it would greatly impact performance in large group scenarios.

Another more efficient approach is used by application like WhatsApp and it functions a little

differently. A user generates a new key pair and a new symmetric key. This key pair will be called Signature key pair and the symmetric key will be used to start a new KDF chain. The user combines his public Signature key and initial KDF chain key into a new key called Sender Key, which he then sends to all the group participants individually using the original one-on-one protocol explained in earlier sections [44].

After sharing this initial secret, every message sent by the user will consist of advancing the KDF and generating a message key, similarly to what was used in earlier sections. This message key will be used to encrypt the message, and the private Signature key will be used to sign it. The user then sends this message to the server who takes care of the distribution, minimizing the number of messages sent (only one message sent to the server instead of N, where N is the number of group participants). The user only needs to store one KDF chain per group participant, instead of the traditional Root, Sending and Receiving chains. This implementation does however sacrifice the Future Secrecy property and is explored further by Rösler et al. [38].

3.5 Summary

In this chapter we covered the Signal Protocol by explaining in detail its main mechanisms: the X3DH and the DRA. We also covered the slightly more complex variant of the DRA which includes header encryption, and the possible group communication adaptations for the Signal Protocol.

This marks the end of the theoretical part of this work. From this point onward we will focus the empirical component of our analysis.

Chapter 4

Technical Approach

4.1 Introduction

In this chapter we will outline the steps taken in order to perform our security analysis of the Signal Protocol. We begin by defining and justifying our target in section 4.2. Then, in section 4.3 we explain the testing environment that was used. In section 4.4 we define some brief guidelines for the future testing process. We use section 4.5 to introduce the Frida toolkit and the setup we will use for our tests. Finally, in section 4.6 we describe the specific test scenario we devised, justify its origin and briefly discuss its expected behaviour.

4.2 Target Choice

The first step of our analysis was the selection of a specific protocol implementation. Out of all the various implementations of the Signal Protocol available, the WhatsApp mobile application stands out thanks to its impressive user base [8], as well as the fact it is one of the few applications that enforces the usage of the Signal Protocol. Based on this, we can assume that WhatsApp is the most popular implementation of the Signal protocol. And within the WhatsApp universe, the most popular application is the one for Google's Android operating system [40]. On top of this, we can also add the existence of good logistical support for the operating system (in terms of emulation software, documentation) and the availability of various reverse engineering resources for Android applications that we will expand upon later. Based on this information, we opted to focus on the Android WhatsApp mobile application, and its respective Signal Protocol implementation.

As a brief introduction, WhatsApp is a communication application implemented in Java programming language for the Android Operating system, that employs E2EE by using the Signal Protocol to protect its user communication data from network eavesdropping attacks. The application allows users to send text messages, perform voice calls and share various types of media with each other and with groups. There is also a significant number of extensions that allow the augmentation of the application's functionalities and services [21].

The WhatsApp application and the Signal Protocol it uses are built for asynchronous communication, so it should be safe to assume it uses some sort of mailbox system from which authenticated users retrieve their respective encrypted messages.

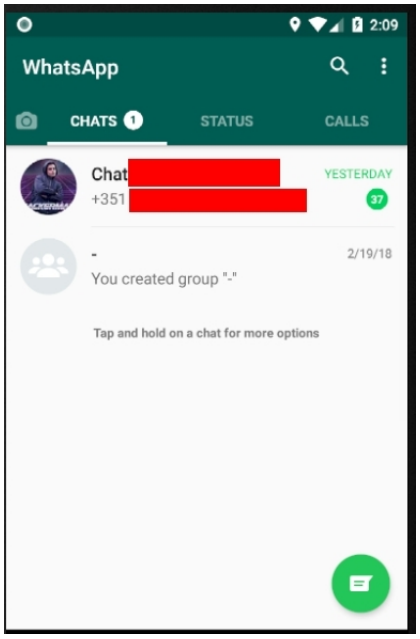


Figure 4.1: The WhatsApp application home screen.

For access to its services, the application requires the validation of a registered phone number that it will use as a username both to connect its users via the phone's contact list, but also to authenticate with the service provided. However, the cryptographic and security properties of the Subscriber Identity Module (SIM) card are not used in any way beyond the initial Short Message Service (SMS) phone number validation. A proof of that is the ability to use the application's services on emulated devices by just using a completely different phone to authenticate a phone number. This raises questions and concerns regarding how the application ensures and validates the user's identity binding and if it is possible to spoof that authentication process, which is explored later in a test scenario.

4.3 Testing Environment

For the testing and analysis environment, we decided to use Google's official Android emulator [20], that is traditionally packaged with the Android Studio Integrated Development Environment (IDE). We briefly tested other solutions such as Genymotion [17], but the official Android emulator ended up causing the least compatibility issues, while at the same time providing the most control over the testing environment. The emulated Android Virtual Device (AVD) instances used were running x86 Android 8.0 Oreo.

The connection between the host computer and the emulated devices was established partially by using the Graphical User Interface (GUI) provided by the Android emulator and also by using the Android Debug Bridge (ADB) which opens a remote terminal to the emulated device [18].

The Android Logcat command-line tool was also used to log unexpected behaviour for further analysis [19].

4.4 Testing guidelines

After the selection of the implementation and testing environment, we began to reverse engineer the application in order to identify and analyse its protocol implementation. The goal is to produce identifiable pseudo-code that can be analysed. This process is still fairly experimental so there was some initial uncertainty regarding the quality of the obtained results.

The next step will be analysing the protocol's presence in the filesystem. The sensitive data, from cryptographic material to contact information and message logs needs to be stored in the filesystem. Part of this analysis will be looking at how that storage is accomplished, and if pertinent, suggesting potential improvements.

Another step will be attempting to dynamically trace protocol functions during runtime using the Frida framework for dynamic instrumentation of the mobile application. This technique will be explained in the next section.

Finally, we will attempt to analyse the application's behaviour in a novel test scenario where multiple identical accounts try to authenticate themselves simultaneously. The approach to this scenario will be expanded upon in section 4.6.

In retrospect, the general approach used for this work ended up being very similar to the one presented by Dai et al. [9], but at the time of its inception it intuitively seemed like the most logical path towards closed-source application analysis. We consider that this proximity between approaches only strengthens our choices.

4.5 Frida

Frida is a dynamic instrumentation toolkit that allows the injection of Google's V8 Javascript engine into targeted processes spawned by native applications on operating systems like Windows, Linux, iOS, Android and more. The injected engine has full access to the process's memory and allows function hooking and much more. The toolkit also allows for other operation modes but we will only focus on its injection mode [36].

We can summarize Frida's architecture down to two main components. On one hand we have *frida-core* which can be interfaced with by using various Frida language bindings available such as *frida-node*, the Node.js binding. On the other hand, we have the *frida-agent* which is the injected component of Frida that communicates with the *frida-core*. The injected Javascript engines are part of *frida-agent*, as well as a dynamic instrumentation library.

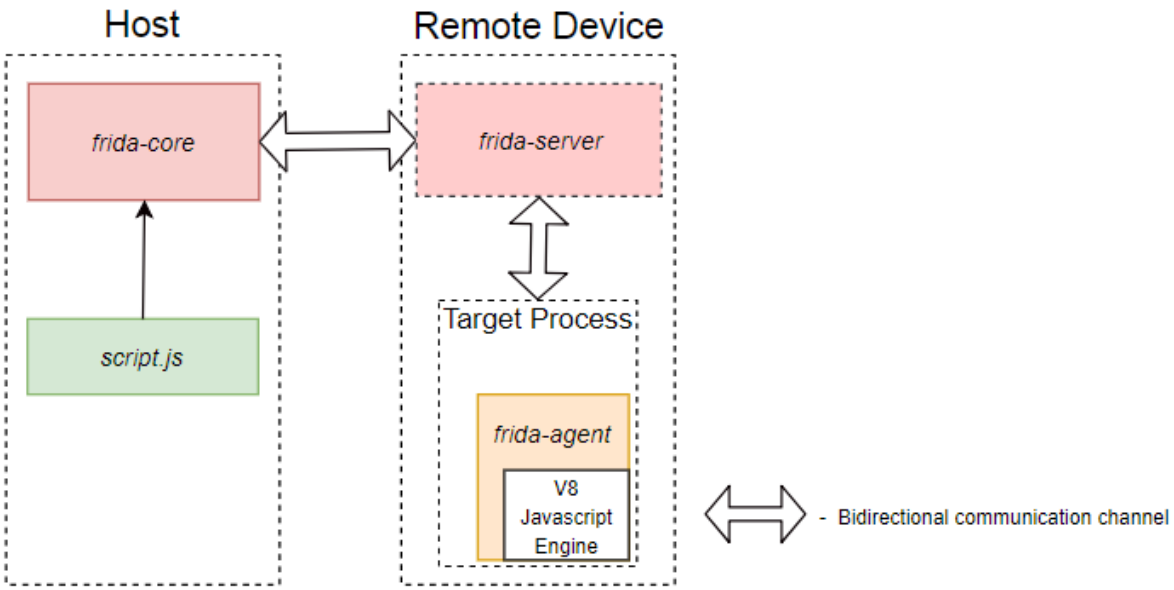


Figure 4.2: Frida componenet interaction in a remote target host scenario.

In addition to this, if the process being injected and *frida-core* are not on the same host, it is

possible to use another component called *frida-server* to handle the communication between the previously mentioned *frida-core* and *frida-agent*. In Figure 4.2 we can see a simplified diagram with the interaction between Frida components in a remote target host scenario.

Some simple tools are also provided alongside Frida, that allow for basic Application Programming Interface (API) tracing, process listing, hookable function listing and more.

4.6 Test Scenario

The scenario devised for this analysis consists of allowing the attacker to access the victim's device and copy some of WhatApp's mobile application data, including the various databases and configuration files.

In theory, we can speculate this attack could allow multiple devices to be connected and using a single account without issues if the messages are not consumed from the WhatsApp server mailbox, or it could also allow multiple devices to authenticate as the same user and use the same user account but only as a common starting point and have unpredictable behaviour afterwards, as different sessions consume messages arbitrarily from the mailbox, corrupting both conversations. The attack could also be detected by WhatsApp that would terminate active sessions using multiple devices for example, by analysing hardware or software serial numbers. It is also possible that the actual scenario behaves differently from any of the previous predictions.

The purpose of this test scenario is to study the protocol implementation's behaviour and assess if the mobile application offers any kind of protection or detection against such attack. Although copying privileged files from a mobile device may seem a bit far-fetched, similar attacks could potentially be achieved by other means such as hijacking a filesystem backup software account.

Initially the environment used for this scenario consisted of two distinct emulated virtual devices, running on the same host and networking hardware, with identical International Mobile Equipment Identity (IMEI) serial numbers. However, it was later replicated using an emulated device and a legitimate mobile phone which meant using distinct networking hardware and different IMEI serial numbers.

With this scenario we hope to obtain some information regarding the WhatsApp session management algorithm and its interaction with the Signal Protocol.

4.7 Summary

Over the chapter we briefly introduced and justified the choice of our testing target which will be the WhatsApp application for Android. We also explained our testing environment which will be comprised of two AVDs although a real device was also used for our specific scenario. Then, the guidelines for our testing were briefly outlined, followed by an introduction to the Frida toolkit and our specific test scenario.

Chapter 5

Implementation and Experiments

5.1 Introduction

After previously highlighting the steps needed to conduct our security analysis, we now describe their implementation and associated experiments.

This chapter is divided in section 5.2 where we describe the implementation of our static analysis techniques (reverse engineering and filesystem analysis), section 5.3 where we describe the implementation of our dynamic analysis using the Frida toolkit, and finally section 5.4 where we describe the tests done in our test scenario.

5.2 Static Analysis

In this section we will explore and describe the static analysis component of our work.

5.2.1 Reverse Engineer

The first step of the static analysis component will be to reverse engineer the application. For that, the WhatsApp mobile application Android PacKage (APK) was downloaded to host machine, before being subsequently transferred and installed on the emulated Android devices. It is important to note that this step was repeated a few times as new versions of the application came out and compatibility with the previous versions stopped being supported. This analysis will be using various versions of the WhatsApp application although no major changes that affected the Signal Protocol were detected.

In order to begin the reverse engineering process, we start by converting the Dalvik Executable (DEX) files contained in the APK to a Java ARchive (JAR) file (using tools such as dex2jar [34]). These JAR files can then be decompiled using any of the existing Java decompilers. For this work, different free decompilers were tested, including JDCore [12], Jadx [41], CFR [4] and Fernflower [30]. In Table 5.1 we have a comparison of decompiled pseudo-code volume for each of the mentioned decompilers. In this table we can clearly see that CFR and Jadx provide the best results in terms of pseudo-code volume. However, CFR is very marginally superior in terms of volume and supports advanced Java features such as Java 8 lambdas. Therefore, we opted to use the CFR decompiler. As such, from this point onwards, any mention of decompilation will indicate the usage of CFR. It should be noted that this should not be considered an extensive and in-depth decompiler comparison. Such analysis would be beyond the scope of this document.

Table 5.1: Decompiler comparison.

Java Decompilers:	CFR	Fernflower	JDCore	Jadx
Output size:	35,9 MB	16,1 MB	27,4 MB	35,9 MB

A substantial amount of time was dedicated to trying to understand and map the generated pseudo-code. The goals were to identify the cryptographically relevant parts of the code containing the key pair generation and storage, the Signal Protocol (X3DH and DRA) implementations and their interaction with the underlying filesystem, namely the KDF chain and associated keys storage. But after making some progress and trying to follow the heavily obfuscated pseudo-code to find the details of the implementation, we came across a series of decompilation errors, caused by the immature and experimental nature of the decompilation process. However, the recovered pseudo-code was enough to allow us to notice the similarities between the open-source Signal Protocol Java implementation provided by OWS, and the implementation present on the actual application. It also revealed the usage of Google Protocol Buffers alongside the previously mentioned implementation [15].

One case where such implementation similarities are apparent despite the obfuscation can be seen in Listings 5.1 and 5.2, which showcase a KDF associated function called *expand*. The first example comes from the reverse engineered application pseudo-code and the second example belongs to OWS's open-source Signal Protocol Java implementation [33]. Many more similar examples are available, such as Figure 5.1, where we can see similarly named files whose names appear to have escaped obfuscation. It should also be noted that part of the folder structure was also recovered and matches the open-source implementation aswell. This indicates that, although not perfectly identical, there is a lot of similarity between the open-source implementation and WhatsApp's implementation.

This confirms the claims made in WhatsApp's whitepaper conclusion [44].

```
private byte[] b(byte[] arrby, byte[] arrby2, int n2) {
        double d2 = (double)n2 / 32.0;
        int n3 = (int)Math.ceil(d2);
        byte[] arrby3 = new byte[]{};
        ByteArrayOutputStream byteArrayOutputStream = new
            ByteArrayOutputStream();
        int n4 = this.a();
        int n5 = n2;
        for (n2 = n4; n2 < this.a() + n3; n5 -= n4, ++n2) {
            Mac mac = Mac.getInstance("HmacSHA256");
            mac.init(new SecretKeySpec(arrby, "HmacSHA256"));
            mac.update(arrby3);
            if (arrby2 != null) {
                mac.update(arrby2);
            }
            mac.update((byte)n2);
            arrby3 = mac.doFinal();
            n4 = Math.min(n5, arrby3.length);
            byteArrayOutputStream.write(arrby3, 0, n4);
        }
...
    }
```

Listing 5.1: Code excerpt from the recovered pseudo-code.

Figure 5.1: Recovered pseudo-code filename correspondance with the open-source Java implementation filenames.

```
private byte[] expand(byte[] prk, byte[] info, int outputSize) {
    try {
      int                   iterations     = (int) Math.ceil((double)
          outputSize / (double) HASH_OUTPUT_SIZE);
      byte[]                mixin          = new byte[0];
      ByteArrayOutputStream results        = new ByteArrayOutputStream();
      int                   remainingBytes = outputSize;
      for (int i= getIterationStartOffset();i<iterations +
          getIterationStartOffset();i++) {
        Mac mac = Mac.getInstance("HmacSHA256");
        mac.init(new SecretKeySpec(prk, "HmacSHA256"));
        mac.update(mixin);
        if (info != null) {
          mac.update(info);
        }
        mac.update((byte)i);
        byte[] stepResult = mac.doFinal();
        int    stepSize   = Math.min(remainingBytes, stepResult.length);
        results.write(stepResult, 0, stepSize);

...
    }
```

Listing 5.2: Code excerpt from the official Signal Protocol Java implementation.

The open-source implementation is also incomplete, not containing the storage mechanisms, which explains the existence of significant differences. Still, the proximity between implementations comes as no surprise, given WhatsApp partnered with OWS for the implementation of the Signal Protocol on its communication and messaging application. This is also the likely scenario regarding the protocol implementations of other popular applications that partnered with OWS to implement the Signal Protocol.

Knowing this, we opt to assume the correct implementation of all the protocol mechanisms rather than committing to the arduous and often impossible task of validating them in the obfuscated, incomplete pseudo-code we extracted from the application. This decision is also influenced by the amount of existing work already covering this implementation [38, 39]. Instead, we move on to the next step where we analyse the filesystem presence of the application as a vector to reveal its storage

mechanisms which are not part of any implementation.

5.2.2 Filesystem Analysis

Because the application needs to store the cryptographic material in a more permanent way, we can safely assume the application must use some sort of long-term secure storage technique in the Android device's filesystem. Thus, we proceed and perform an analysis of the filesystem presence of the WhatsApp application.

The Android operating system has a directory destined for application data under the path *../data/data/*. It is here that we find the majority of the files associated with the WhatsApp application in a folder named *com.whatsapp*. We will now highlight the interesting finds in this directory.

Our first find was a couple Extensible Markup Language (XML) files containing all sorts of information. From statistical information to the registered phone number. There is also a *keystore.xml* which contains a server public key and a client key pair but it didn't appear to be relevant to the Signal Protocol.

A more interesting find was a plain SQLite database file named *axolotl.db* (a reference to Axolotl Ratchet, an older name for the DRA). As the name suggested, this database contains the cryptographic material associated with the DRA. The tables worth highlighting and respective presumed functionalities after analysing are as follows:

- *identities* - contains the known identities (contacts in the application context) and respective public keys. The first entry in this table is the user's contact and his public and private keys.
- *prekey_uploads* - contains a register of pre-key upload timestamps, presumably in order to track their expiration;
- *prekeys* - contains pre-key cryptographic material;
- *sender_keys* - contains cryptographic material associated with group communication;
- *sessions* - contains the last received message and respective timestamp from each contact messages including group settings;
- *signed_prekeys* - contains signed pre-key cryptographic material.

We also found another interesting SQLite database named *msgstore.db*, which contained all sorts of information associated with the content of the communications using the WhatsApp application. This information ranges from plain text messages to group participant history, voice call log, media references and payment transaction. Additionally, upon further analysis we also learned that this database is also encrypted and used as the application's content backup. This was verified by decrypting the crypt12 file format backup (a specific file format belonging to WhatsApp) using the knowledge first introduced by Dai et al. [9] and implemented by Ibrahim [31]. As the application doesn't allow for an easy transfer of cryptographic material between devices, the alternative is to transfer or backup the message content itself. By default, the local backups are encrypted with a key in another file in the application directory called *key*. Dai et al. [9] explored this mechanism in greater detail, and inclusively formulated an attack vector around it. An interesting observation is that despite the protocol changes, this database alongside a few other less interesting findings were already available in the forensic analyses by Thakur and Anglano [1, 43].

5.3 Dynamic Analysis with Frida

In this security analysis, we decided to employ a novel technique for dynamic analysis in various environments including mobile (Android or iOS). That technique consists of using the Frida toolkit to dynamically instrument a specific process (in our case, WhatsApp's process) and use this to trace function calls in real time. This technique also allows some more interesting functionalities such as argument and return value manipulation, as well as backtrace access. It is important to note the instability associated, once again, with the experimental and immature nature of this technique and the Frida toolkit itself.

The first step is to transfer Frida's Android x86 server to the device used in testing. This server communicates with the *frida-core* running on the host, and is responsible for the injection of the Javascript engine and the script's execution. It is necessary to grant this server executable file execution permissions.

The second step was to somehow enumerate the hookable functions in the targeted process. Frida provides only simple tool called Frida-discover which is meant to do exactly this. However, due to some incompatibility we ended up not determining, this tool crashed the target process. Therefore, we decided to write our own tiny enumeration script, as can be seen in Listing 5.3.

```
Java.perform(function() {
   Java.enumerateLoadedClasses({
      "onMatch": function(className){
         console.log(className)
      } });
},0);
```

Listing 5.3: Frida hookable class enumeration fuction.

The final step involved hooking any interesting functions from the list of functions enumerated previously, in order to dynamically trace the application's functioning. For the example in Listing C.1, we decided to hook the function *org.whispersystems.libsignal.c.c.a* which was the obfuscated KDF *expand* function from Listing 5.1. The trace log shows the sequential timeline and backtrace obtained for sending one single text message to a group conversation.

The script used is available in Appendix A and allows for method or class tracing. It should be noted that hooking too many functions will crash the process, and the argument printing functionality needs to be manually tailored to specific argument data types.

5.4 Test Scenario

The final step of the selected approach involved artificially manipulating the testing environment to try and analyse the behaviour of the application and the protocol in a specific scenario.

Following the attack outlined in section 4.6, we copied the relevant application files from legitimate Device A onto malicious Device B. It is also necessary to modify the user ID and group ID of the files, as the Android operating system attributes a unique set of IDs to each application as a sandboxing mechanism [14]. Having done this, Device B is able to successfully authenticate itself as the original user from Device A, and it exposes all the content in the copied files (which was already exposed by exploring the copied files themselves), as visible in Figure 5.2.

After successful authentication, we attempted to message an existing group conversation with both authenticated devices. What happened was that both devices could communicate with the group, yet they did not receive each other's messages. However, the remaining behaviour was not constant. Some inbound messages were received on both devices, some only on one device, some outbound messages were not received by any of the other group participants yet others were received

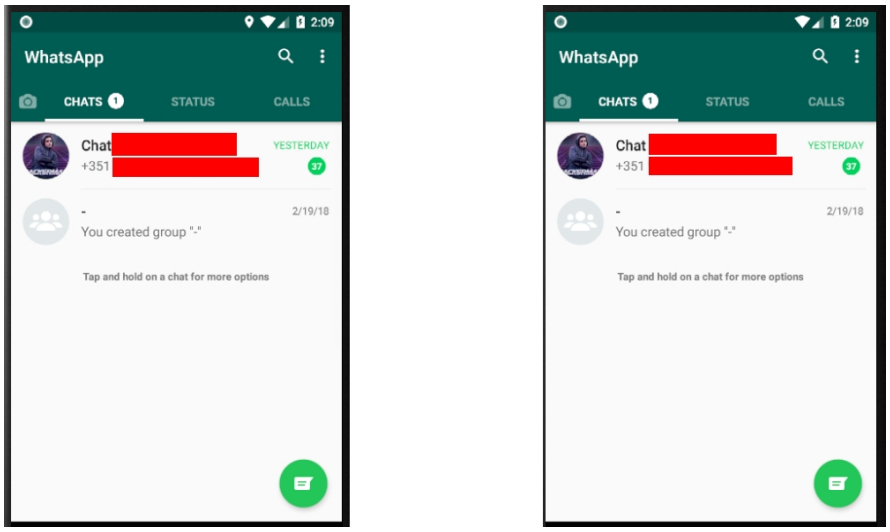


Figure 5.2: WhatsApp home screen on device A on the left, device B on the right

by everyone. This was reproduced in our virtual testing environment, but also using a real device.

We decided to do a second test, where we, once again, copied the relevant files from Device A to Device B. This lead us back to the situation depicted in Figure 5.2. The difference in this test was that at this stage, we decided to add a single new contact to each of the devices. We then initiated a conversation with this new contact. The idea behind this test was to fully trigger the entire Signal Protocol, as the previous test was using existing conversations therefore only using the DRA but not the X3DH. Each device sent a message containing a word followed by a number. The number is used to determine which device sent the message first chronologically. The responses are also numbered following the same logic.

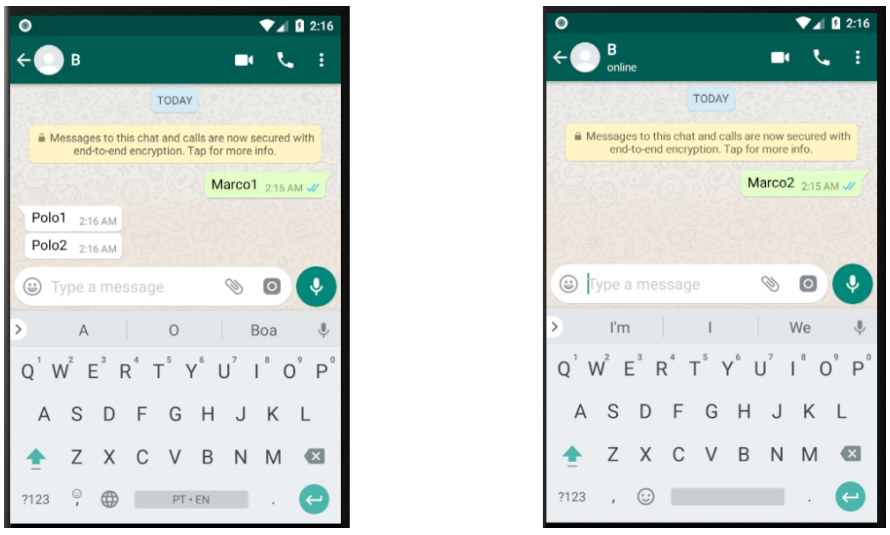


Figure 5.3: Scenario testing. Device A on the left, Device B on the right.

In Figure 5.3 we initiated two distinct conversations with our common contact using two distinct devices using the same session and user authentication. Interestingly, Device B's message ("Marco2") arrived first and Device A's message ("Marco1") arrived second despite that not being the chronological order they were sent in. After that, the new contact responded with two messages of his own, which arrived only on Device A. Interestingly, Device A was also the last one to setup a conversation if we base ourselves on the order the initial messages arrived for our contact.

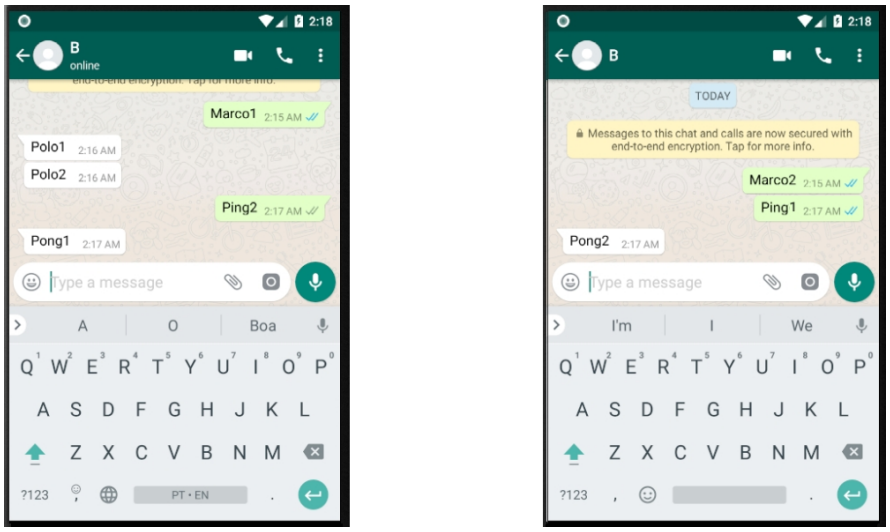


Figure 5.4: Scenario testing. Device A on the left, Device B on the right. (cont.)

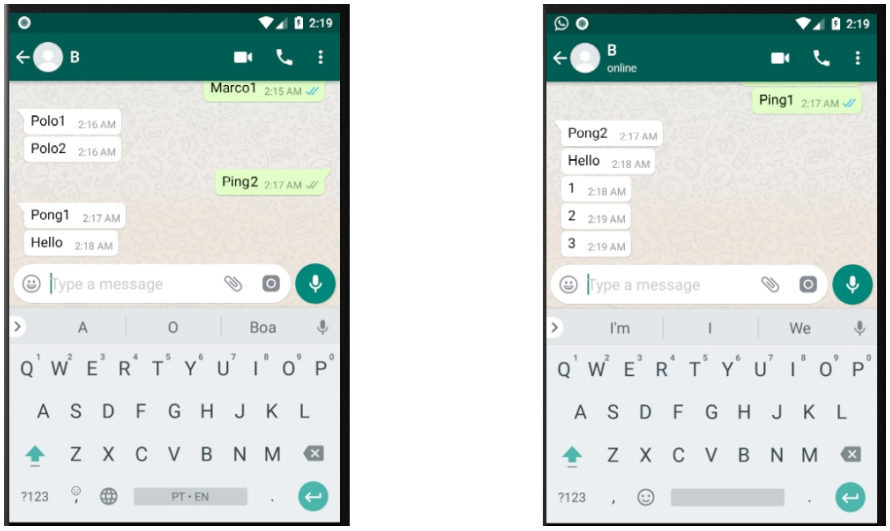


Figure 5.5: Scenario testing. Device A on the left, Device B on the right. (cont..)

In Figure 5.4 we send two more messages from our devices. This time, Device B takes the initiative, followed by Device A. Our contact received these messages chronologically and responded with two more messages. This time, the first message was received on Device A and the second

message was received on Device B.

In Figure 5.5 our contact took the initiative and sent four messages. The first message containing ("Hello") was received on both Device A and Device B. The following three messages arrived only on Device B.

In Figure 5.6 we can see the entire conversation from our common contact's perceptive.

Figure 5.6: Scenario testing. Commom contact message log prespective.

As we can see from the previous test, the observed behaviour of the application and the Signal Protocol in our test scenario seems very elusive. However, scenarios like the one in Figure 5.5 where the message was delivered to both devices while they are unaware of each other's existence might represent serious security vulnerabilities.

5.5 Summary

In this chapter we covered the steps necessary to implement our various tests and techniques, from the initial reverse engineering to the final test scenario.

In the next chapter we compare the obtained results from this chapter against the expectations and objectives set earlier in this document.

Chapter 6

Main Results

6.1 Introduction

In this chapter we will look at the results obtained and briefly discuss them and their potential impacts. In section 6.2 we briefly review the experiments and tests performed previously and the results obtained, followed by a comparison of said results against our initial expectations and objectives. Then, we use section 6.3 to make some security considerations regarding the contents of the previous section. We conclude the chapter's body with section 6.4 where we suggest some potential improvements to upgrade our analysis target's overall security.

6.2 Results

After the previous experimentation and testing phase, it is time to review the obtained results against the original expectations set by the reviewed literature.

6.2.1 Application Reverse Engineering

After selecting the most relevant application using the Signal Protocol, we attempted to reverse engineer it in order to obtain as close to an an approximation as possible to it's original source code. In this process we used the best performing Java decompiler, CFR, and managed to obtain a very good, although not perfect, code base. However, this code base was heavily obfuscated and contained several decompilation errors which hid unknown amounts of code. Still, the available pseudo-code

allowed the detection of many of the smaller Signal Protocol constituents, especially by comparison with the open-source Signal Protocol implementation provided by OWS.

We consider it safe to assume that, even if the implementations differ a little, WhatsApp's implementation is heavily based on that open-source implementation. As evidences of this similarity we have already provided the Listing 5.1 and Listing 5.2 examples, but more examples are featured in Appendix B.

This step of the analysis had as objectives to identify the protocol's constituents or the protocol's implementation. The assumptions and evidences gathered indicate we accomplished this goal, documenting a connection between the open-source implementation and the application's implementation which should be useful for further research and analysis.

6.2.2 Filesystem Analysis

The second stage of our work consisted of analysing the filesystem *footprint* of the application, mostly by attempting to identify its storage mechanisms and locate the stored information. For this we manually explored the application data directory on the Android operating system and analysed the various WhatsApp files. In this directory we found a database containing all the cryptographic material, a database containing the communication content, a few interesting configuration files and finally we found and inspected the application's local backup system by testing some of the findings in Dai et al. [9].

The objectives of this step were precisely to understand how the application handled the long term data storage on the filesystem, which is what was accomplished. The most relevant stored data was found in regular, unencrypted, SQLite databases. The backup mechanism, which saves only the database containing the communication content, does encrypt its database file using a key found in another file in the application directory, presumably because the backups can be stored is non-privileged filesystem directories.

6.2.3 Dynamic Instrumentation with Frida

The dynamic instrumentation step of this security analysis involved using the Frida toolkit to dynamically inspect and trace function calls related to the Signal Protocol in the WhatsApp aplication. For this, we wrote and adapted custom Frida scripts that allowed us to list all hookable functions and

to hook and trace actual functions, with the ability to manipulate their arguments, return values and even access their backtrace, which is extremely helpful for understanding the application logic.

The goals for this step were to use the function tracing ability to attempt to trace entire procedures of the Signal Protocol. Unfortunately, due to the immature nature of the toolkit there were some issues, namely process crashes, while trying to trace large protocol segments. However, it is possible to decompose the protocol down to smaller mechanism and trace their execution, as seen in Appendix C. The potential of this technique is huge, and in this analysis we only scraped the surface, and one of the goals was to expose and use these less common analysis techniques and tools to incentivize the audit and analysis of closed-source software.

6.2.4 Local Data Cloning Test Scenario

For our scenario testing we replicated a scenario where an attacker got access to the victim's application files, either by compromising the real device, a device filesystem backup or any other potential attack vector. We replicated this scenario by copying the application's local files from one device to another. This was tested between AVDs but also between an AVD and a real device.

For this stage the objectives we defined were to study the application's behaviour under extraordinary circumstances that could still be found in a real scenario. Given that extraordinary nature of the circumstances and the lack of knowledge regarding the functioning of certain application components, such as its session management, we can only speculate regarding the possible implication of the observed erratic and elusive behaviour.

6.3 Security Considerations

Overall, the general idea that there were no major flaws in the Signal Protocol implementation that we got from the reviewed literature was correct. The implementation analysed in this work showed many similarities with the much more accessible and already tested open-source implementation. This corroborates what is stated on the WhatsApp whitepaper, where they state they are using the open-source implementation in their application [44]. However, that implementation is not complete, missing, for example, the storage mechanisms and the group communication implementation.

Its precisely the long term data storage mechanisms that we consider one of the aspects where the WhatsApp application could do a better job. In this particular topic, most if not all of the security is

entrusted to the operating system, with the only possible exception being the backup mechanism and its encryption, which then again uses a key that is also generally protected by the operating system alone and can easily be compromised, as seen in Dai et al. [9]. There are a lot of potential upgrades for this aspect that will be suggested in the next section.

Another important aspect to consider is the elusive observed behaviour that, while limiting the scope of the potential speculation about the functioning of the application, do not provide us with concrete answers. However, the observed behaviour shows some promise regarding the existence of security flaws and exploitable vulnerabilities in such a scenario. One of the most notable of these flaws would be enabling the malicious cloned device to passively eavesdrop on the communication by receiving a copy of the incoming messages without being detected. This behaviour was already observed in the tested scenario amongst other erratic behaviour but we found no way to control it and make it consistent.

It's also important to add that behaviour elusiveness aside, the observed behaviour itself already constitutes a small security vulnerability at it allows for partial communication eavesdropping and corruption.

6.4 Improvement Suggestions

One of the most notable aspects that could easily be improved based on the findings of this work is the WhatsApp application's long term storage solution. An idea that could improve this aspect would be the usage of the Android Keystore System to safely store the application's cryptographic material at all times [13].

Another simple mechanism that could protect the WhatsApp local files would be the usage of a password or pincode from which we could derive an encryption key used to encrypt the those files. This mitigation would also hinder the attack carried out in our test scenario.

Another interesting concept that could improve the Signal Protocol's security would be the removal of the trusted server entity, which still possesses a lot of information regarding its user's actions such as communication metadata (who messages who and when). However, this would necessarily cause many changes to the protocol itself, such as the possible adoption of a peer-to-peer architecture and making it much harder to preserve the asynchronous aspect of the Signal Protocol.

6.5 Summary

After covering the results obtained in this work, briefly discussing them, sharing some security considerations, and also suggesting some potential security improvements for the analysis targets, we now proceed to the final chapter where we will extract the main conclusions of this work, and highlight potential clues for further research raised by this work.

Chapter 7

Conclusions

7.1 Introduction

In this final chapter we begin with section 7.2 where we draw the main conclusions from the work done previously. To finalize the main body of this document, we will use section 7.3 to explore the limitations of our work as clues that indicate possible future work.

7.2 Main Conclusions

At the beginning of this work, we set out to perform a security analysis on the fairly recent Signal Protocol and one of its implementations, because we identified the small amount of work in this field when compared to its impact to be a problem. In order to contribute to the solution to that problem, we defined an approach with a strong empirical focus which diverged a bit from the existing literature, followed it during implementation and discussed the achieved results.

We were able to accomplish, to varying degrees, the initially defined objectives. Hopefully, the maturing of the processes and techniques used in this analysis will continue and strengthen them even further, so that the tools to effectively audit and analyse any applications that deal with sensitive data reach wider audiences and the volume and quality of the work in this field continues to grow.

The literature review performed for this work was constant, as the novelty factor and rising popularity of our analysis targets would certainly attract the attention of other authors that would, in turn, publish their own work and findings. This means that the literature review and the experimental

stage of this work took place simultaneously.

For the majority of our analysis, the general results ended up matching the expectations set by the reviewed literature and pointed to no major flaws, even though there was room for improvement in some aspects. The situation changed a bit when we tested our custom scenario. While the observed results are still inconclusive in regards to the existence of critical security vulnerabilities, we consider the observed behaviour to already constitute a lower impact security vulnerability with many unexplored ramifications and implications.

7.3 Future Work

Throughout this work, there were many instances where we could have opted to do a deeper analysis. This, however, would have required us to recover the invested time somewhere else, most likely sacrificing part of the width of our defined approach. We intentionally opted not to do this as we felt a significant part of the value of this work belonged to its approach. We leave it for the future to perform deeper analyses to the already briefly covered areas.

Another area of this work which should be analysed and studied in the future is the defined test scenario. The elusive and erratic behaviour we observed during our tests provides a strong clue that points towards potential serious security vulnerabilities. It pertains to both the session algorithm, which we know very little about, and the Signal Protocol.

References

[1] Cosimo Anglano. Forensic Analysis of WhatsApp Messenger on Android Smartphones. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 11:201–213, Sep, 2014.

[2] Jean-Philippe Aumasson and Markus Vervier. Hunting For Vulnerabilities in Signal. Website: https://conference.hitb.org/hitbsecconf2017ams/sessions/hunting-for-vulnerabilities-in-signal/, Apr, 2017.

[3] Dikla Barda, Roman Zaikin, and Oded Vanunu. FakesApp: A Vulnerability in WhatsApp. Website: https://research.checkpoint.com/fakesapp-a-vulnerability-in-whatsapp/, 2018.

[4] Lee Benfield. CFR - another java decompiler. Website: http://www.benf.org/other/cfr/, 2018.

[5] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228. 2006.

[6] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Nov, 2017.

[7] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On Post-compromise Security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, 2016.

[8] Josh Constine. WhatsApp hits 1.5 billion monthly users. $19B? Not so bad. Website: https://techcrunch.com/2018/01/31/whatsapp-hits-1-5-billion-monthly-users-19b-not-so-bad/, Jan, 2018.

[9] Zhongmin Dai, Sufatrio, Tong-Wei Chua, Dinesh Kumar Balakrishnan, and Vrizlynn L. L. Thing. Chat-App Decryption Key Extraction Through Information Flow Analysis . In Abhik

Roychoudhury and Yang Liu, editors, *A Systems Approach to Cyber Security*, volume 15, page 3–18, Singapore, 2017. IOS Press.

[10] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, Nov, 1976.

[11] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2:107–125, Jun, 1992.

[12] Emmanuel Dupuy. Java Decompiler. Website: http://jd.benow.ca/, 2018.

[13] Google Inc. Android keystore system. Website: https://developer.android.com/training/articles/keystore, 2018.

[14] Google Inc. Application Sandbox. Website: https://source.android.com/security/app-sandbox, 2018.

[15] Google Inc. Protocol Buffers. Website: https://developers.google.com/protocol-buffers/, 2018.

[16] Glenn Greenwald and Ewen MacAskill. NSA Prism program taps in to user data of Apple, Google and others. Website: https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data, Jun, 2013.

[17] Genymobile Inc. Genymotion Android Emulator. Website: https://www.genymotion.com/, 2018.

[18] Google Inc. Android Debug Bridge (adb). Website: https://developer.android.com/studio/command-line/adb, 2018.

[19] Google Inc. Logcat command-line tool. Website: https://developer.android.com/studio/command-line/logcat, 2018.

[20] Google Inc. Run apps on the Android Emulator. Website: https://developer.android.com/studio/run/emulator, 2018.

[21] WhatsApp Inc. WhatsApp. Website: https://www.whatsapp.com/, 2018.

[22] Joshua Lund. Signal partners with Microsoft to bring end-to-end encryption to Skype. Website: https://signal.org/blog/skype-partnership/, Jan, 2018.

[23] Marco Ivaldi. raptor_frida_android_trace.js. Website: https://github.com/0xdea/frida-scripts/blob/master/raptor_frida_android_trace.js, 2018.

[24] Moxie Marlinspike. Advanced cryptographic ratcheting. Website: https://signal.org/blog/advanced-ratcheting/, Nov, 2013.

[25] Moxie Marlinspike. Facebook Messenger deploys Signal Protocol for end-to-end encryption. Website: https://signal.org/blog/facebook-messenger/, Jul, 2016.

[26] Moxie Marlinspike. Open Whisper Systems partners with Google on end-to-end encryption for Allo. Website: https://signal.org/blog/allo/, May, 2016.

[27] Moxie Marlinspike. WhatsApp's Signal Protocol integration is now complete. Website: https://signal.org/blog/whatsapp-complete/, Apr, 2016.

[28] Moxie Marlinspike and Trevor Perrin. The Double Ratchet Algorithm. Website: https://signal.org/docs/specifications/doubleratchet/, Nov, 2016.

[29] Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol. Website: https://signal.org/docs/specifications/x3dh/, Nov, 2016.

[30] Andrew McRae. FernFlower Java decompiler. Website: https://github.com/fesh0r/fernflower, 2018.

[31] Mohamed Ibrahim. How to Decrypt WhatsApp crypt12 Database Messages. Website: https://stackpointer.io/security/decrypt-whatsapp-crypt12-database-messages/559/, Oct, 2016.

[32] Aulon Mujaj. A Comparison of Secure Messaging Protocols and Implementations. Master's thesis, University of Oslo, 2017.

[33] Open Whisper Systems. Signal Protocol library for Java/Android. Website: https://github.com/signalapp/libsignal-protocol-java, 2018.

[34] Bob Pan. Tools to work with android .dex and java .class files. Website: https://github.com/pxb1988/dex2jar, 2018.

[35] Trevor Perrin. The XEdDSA and VXEdDSA Signature Schemes . Website: https://signal.org/docs/specifications/xeddsa/, Oct, 2016.

[36] Ole Ravnås. Frida - A world-class dynamic instrumentation framework. Website: https://www.frida.re/, 2018.

[37] Phillip Rogaway. Authenticated-encryption with associated-data. In *CCS '02 Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107. ACM, Sep, 2002.

[38] Paul Rösler and Jörg Schwenk Christian Mainka. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Jan, 2018.

[39] Jan Rubín. Security Analysis of the Signal Protocol. Master's thesis, Czech Technical University in Prague, May, 2018.

[40] Paul Sawers. Three-quarters of WhatsApp users are on Android, 22% on iOS (study). Website: https://venturebeat.com/2015/08/27/three-quarters-of-whatsapp-users-are-on-android-study-finds/, Aug, 2015.

[41] Skylot. Dex to Java decompiler. Website: https://github.com/skylot/jadx, 2018.

[42] Open Whisper Systems. Signal. Website: https://signal.org/, 2018.

[43] Neha S. Thakur. Forensic Analysis of WhatsApp on Android Smartphones . Master's thesis, University of New Orleans, 2013.

[44] WhatsApp Inc. WhatsApp Encryption Overview. Website: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, 2017.

Appendix A

Frida Tracing Script

Listing A.1 contains the Javascript script used to dynamically trace method calls and classes using the Frida toolkit. This script contains three mains functions which are:

- *trace* - this function receives a pattern as argument and attempts to match it to one of the existing classes or, if unsuccessful, passing that argument as a parameter to *traceMethod*.
- *traceClass* - if a class matching the initially inputed pattern was found, this function will iterate through all the class methods and pass them as argument to individual *traceMethod* calls.
- *traceMethod* - this function will attempt to hook a method it receives as argument. This involves hooking all existing method overloads as they all share the same name and class which is our specified pattern. It would be possible to hook specific overloads by slightly modifying the script. Once a method call has been hooked, it's possible to manipulate its arguments and return value, but also to print its backtrace which is massively helpful for function tracing, as can be seen on the logs in Appendix C.

```
function trace(pattern){
   var found = false;
   Java.enumerateLoadedClasses({
      onMatch: function(aClass) {
         if (aClass.match(pattern)) {
            found = true;
            var className = aClass.replace(/\//g, ".");
```

```
            traceClass(className);
        }
    },
        onComplete: function() {}
    });

    if (!found) {
        console.log("Class not found, tracing method")
        try {
            traceMethod(pattern);
        }
        catch(err) {
            console.error(err);
        }
    }
}

function traceClass(targetClass){
    var hook = Java.use(targetClass);
    var methods = hook.class.getDeclaredMethods();
    hook.$dispose;
    var parsedMethods = [];
    methods.forEach(function(method) {
        parsedMethods.push(method.toString().replace(targetClass + ".",
            "TOKEN").match(/\sTOKEN(.*)\(/)[1]);
    });
    var targets = uniqBy(parsedMethods, JSON.stringify);
    targets.forEach(function(targetMethod) {
        traceMethod(targetClass + "." + targetMethod);
    });
}

function traceMethod(targetClassMethod){
    var delim = targetClassMethod.lastIndexOf(".");
    if (delim === -1) return;
    var targetClass = targetClassMethod.slice(0, delim)
    var targetMethod = targetClassMethod.slice(delim + 1,
        targetClassMethod.length)
    var hook = Java.use(targetClass);
    var overloadCount = hook[targetMethod].overloads.length;
    console.log("Tracing " + targetClassMethod + " [" + overloadCount + "
```

```
      overload(s)]");
   hook[targetMethod].overloads.forEach(function (method) {
      method.implementation = function () {
         console.warn("\n*** entered " + targetClassMethod);
         // print backtrace
          Java.perform(function() {
            var bt = Java.use("android.util.Log").getStackTraceString(Java.use(
"java.lang.Exception").$new());
            console.log("\nBacktrace:\n" + bt);
          });

         // print args
         if (arguments.length) console.log();
         for (var j = 0; j < arguments.length; j++) {
            console.log("arg[" + j + "]: " + arguments[j]);
         }

         // print retval
         retval=method.apply(this, arguments);
         //console.log("\nretval: " + retval);
         console.warn("\n "+Date.now()+" *** exiting " + targetClassMethod);
         return retval;
      }
   });
}

function uniqBy(array, key){
        var seen = {};
        return array.filter(function(item) {
                var k = key(item);
                return seen.hasOwnProperty(k) ? false : (seen[k] = true);
        });
}

setTimeout(function() { // avoid java.lang.ClassNotFoundException
   Java.perform(function() {
      trace("org.whispersystems.libsignal.c.c.a"); //MSGs de grupo
   });
}, 0);
```

Listing A.1: Script used with Frida to dynamically trace method calls. Adapted from [23].

Appendix B

Implementation Evidence

In this appendix we list more examples that evidence the correlation between the reverse engineered pseudo-code and the open-source implementation, despite the obfuscation. These obfuscated pseudo-code methods can then be traced in order to identify protocol components in action.

```
public final d b() {
    Object object = this.a(c);
    object = new a(this.e.a((byte[])object, "WhisperMessageKeys".getBytes(),
        80));

    return new d(object.a, object.b, object.c, this.b);
}
```

Listing B.1: Code excerpt from the recovered pseudo-code.

```
public MessageKeys getMessageKeys() {
    byte[] inputKeyMaterial = getBaseMaterial(MESSAGE_KEY_SEED);
    byte[] keyMaterialBytes = kdf.deriveSecrets(inputKeyMaterial,
        "WhisperMessageKeys".getBytes(), DerivedMessageSecrets.SIZE);
    DerivedMessageSecrets keyMaterial = new
        DerivedMessageSecrets(keyMaterialBytes);

    return new MessageKeys(keyMaterial.getCipherKey(), keyMaterial.getMacKey(),
        keyMaterial.getIv(), index);
```

```
}
```

Listing B.2: Code excerpt from the Signal Protocol Java implementation. [33] Matches Listing B.1.

```
public a(byte[] arrby) {
  try {
    arrby = d.a(arrby, 32, 32, 16);
    this.a = new SecretKeySpec((byte[])arrby[0], "AES");
    this.b = new SecretKeySpec((byte[])arrby[1], "HmacSHA256");
    this.c = new IvParameterSpec((byte[])arrby[2]);
    return;
  }
  catch (ParseException parseException) {
    throw new AssertionError(parseException);
  }
}
```

Listing B.3: Code excerpt from the recovered pseudo-code.

```
public DerivedMessageSecrets(byte[] okm) {
  try {
    byte[][] keys = ByteUtil.split(okm, CIPHER_KEY_LENGTH, MAC_KEY_LENGTH,
        IV_LENGTH);
    this.cipherKey = new SecretKeySpec(keys[0], "AES");
    this.macKey    = new SecretKeySpec(keys[1], "HmacSHA256");
    this.iv        = new IvParameterSpec(keys[2]);
  } catch (ParseException e) {
    throw new AssertionError(e);
  }
}
```

Listing B.4: Code excerpt from the Signal Protocol Java implementation. [33] Matches Listing B.3.

```
public final org.whispersystems.libsignal.f.c<f, c> a(e object,
    org.whispersystems.libsignal.a.c c2) {
```

```
  object = d.a((e)object, c2.b);
  object = new b(this.b.a((byte[])object, this.a, "WhisperRatchet".getBytes(),
      64));
  return new org.whispersystems.libsignal.f.c<f, c>(new f(this.b, object.a),
      new c(this.b, object.b, 0));
}
```

Listing B.5: Code excerpt from the recovered pseudo-code.

```
public Pair<RootKey, ChainKey> createChain(ECPublicKey theirRatchetKey,
    ECKeyPair ourRatchetKey)
  throws InvalidKeyException
{
  byte[] sharedSecret = Curve.calculateAgreement(theirRatchetKey,
      ourRatchetKey.getPrivateKey());
  byte[] derivedSecretBytes = kdf.deriveSecrets(sharedSecret, key,
      "WhisperRatchet".getBytes(), DerivedRootSecrets.SIZE);
  DerivedRootSecrets derivedSecrets = new
      DerivedRootSecrets(derivedSecretBytes);
  RootKey  newRootKey  = new RootKey(kdf, derivedSecrets.getRootKey());
  ChainKey newChainKey = new ChainKey(kdf, derivedSecrets.getChainKey(), 0);
  return new Pair<>(newRootKey, newChainKey);
}
```

Listing B.6: Code excerpt from the Signal Protocol Java implementation. [33] Matches Listing B.5.

Appendix C

Tracing Logs

In this appendix we list some function tracing logs containing method calls, method call backtraces and method exits. These logs were used to validate and trace the Signal Protocol in action.

Listing C.1 was recorded while sending one outbound message via WhatsApp.

```
Tracing org.whispersystems.libsignal.c.c.a [5 overload(s)]

*** entered org.whispersystems.libsignal.c.c.a

Backtrace:
java.lang.Exception
   at org.whispersystems.libsignal.c.c.a(Native Method)
   at org.whispersystems.libsignal.b.a.c.<init>(:270880)
   at org.whispersystems.libsignal.b.a.b.a(:270873)
   at org.whispersystems.libsignal.b.c.a(:271172)
   at com.whatsapp.jobqueue.job.SendE2EMessageJob.a(:163359)
   at com.whatsapp.jobqueue.job.d.call(Unknown Source:8)
   at java.util.concurrent.FutureTask.run(FutureTask.java:266)
   at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor...
   .java:1162)
   at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor...
   .java:636)
   at com.whatsapp.c.k.run(Unknown Source:7)
   at java.lang.Thread.run(Thread.java:764)



*** entered org.whispersystems.libsignal.c.c.a
```

```
Backtrace:
java.lang.Exception
   at org.whispersystems.libsignal.c.c.a(Native Method)
   at org.whispersystems.libsignal.c.c.a(:119558)
   at org.whispersystems.libsignal.c.c.a(Native Method)
   at org.whispersystems.libsignal.b.a.c.<init>(:270880)
   at org.whispersystems.libsignal.b.a.b.a(:270873)
   at org.whispersystems.libsignal.b.c.a(:271172)
   at com.whatsapp.jobqueue.job.SendE2EMessageJob.a(:163359)
   at com.whatsapp.jobqueue.job.d.call(Unknown Source:8)
   at java.util.concurrent.FutureTask.run(FutureTask.java:266)
   at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor...
   .java:1162)
   at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor...
   .java:636)
   at com.whatsapp.c.k.run(Unknown Source:7)
   at java.lang.Thread.run(Thread.java:764)

*** entered org.whispersystems.libsignal.c.c.a

Backtrace:
java.lang.Exception
   at org.whispersystems.libsignal.c.c.a(Native Method)
   at org.whispersystems.libsignal.c.c.a(:119559)
   at org.whispersystems.libsignal.c.c.a(Native Method)
   at org.whispersystems.libsignal.c.c.a(:119558)
   at org.whispersystems.libsignal.c.c.a(Native Method)
   at org.whispersystems.libsignal.b.a.c.<init>(:270880)
   at org.whispersystems.libsignal.b.a.b.a(:270873)
   at org.whispersystems.libsignal.b.c.a(:271172)
   at com.whatsapp.jobqueue.job.SendE2EMessageJob.a(:163359)
   at com.whatsapp.jobqueue.job.d.call(Unknown Source:8)
   at java.util.concurrent.FutureTask.run(FutureTask.java:266)
   at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor...
   .java:1162)
   at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor...
   .java:636)
   at com.whatsapp.c.k.run(Unknown Source:7)
   at java.lang.Thread.run(Thread.java:764)
```

```
1537425377073 *** exiting org.whispersystems.libsignal.c.c.a

1537425377075 *** exiting org.whispersystems.libsignal.c.c.a

1537425377077 *** exiting org.whispersystems.libsignal.c.c.a
```

Listing C.1: Log containing *expand* method trace.

Listing C.2 contains the tracing log of the function in Listing B.1, and was recorded for sending one outbound message to a group conversation.

```
Tracing org.whispersystems.libsignal.e.c.b [1 overload(s)]

*** entered org.whispersystems.libsignal.e.c.b

Backtrace:
java.lang.Exception
    at org.whispersystems.libsignal.e.c.b(Native Method)
    at org.whispersystems.libsignal.l.a(:271449)
    at com.whatsapp.jobqueue.job.g.call(Unknown Source:25)
   at java.util.concurrent.FutureTask.run(FutureTask.java:266)
   at java.util.concurrent.ThreadPoolExecutor.runWorker ...
   (ThreadPoolExecutor.java:1162)
   at java.util.concurrent.ThreadPoolExecutor$Worker.run ...
   (ThreadPoolExecutor.java:636)
   at com.whatsapp.c.k.run(Unknown Source:7)
   at java.lang.Thread.run(Thread.java:764)

1538250426016 *** exiting org.whispersystems.libsignal.e.c.b
```

Listing C.2: Log containing *getMessageKeys* method trace.

Listing C.3 contains the tracing log of the function in Listing B.1, and was recorded for receiving one inbound message from a group conversation.

```
*** entered org.whispersystems.libsignal.e.c.b

Backtrace:
java.lang.Exception
   at org.whispersystems.libsignal.e.c.b(Native Method)
   at org.whispersystems.libsignal.l.a(:271769)
```

```
  at org.whispersystems.libsignal.l.a(:271654)
  at org.whispersystems.libsignal.l.a(:271642)
  at com.whatsapp.mx.a(:97990)
  at com.whatsapp.mx.a(:98062)
  at com.whatsapp.mx.run(:98148)
  at java.util.concurrent.ThreadPoolExecutor.runWorker ...
  (ThreadPoolExecutor.java:1162)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run ...
  (ThreadPoolExecutor.java:636)
  at com.whatsapp.c.k.run(Unknown Source:7)
  at java.lang.Thread.run(Thread.java:764)

1538251356673 *** exiting org.whispersystems.libsignal.e.c.b
```

Listing C.3: Log containing *getMessageKeys* method trace.

Comparing Listings C.2 and C.3 we can clearly notice the different path taken by the program logic in the two different scenarios, thanks to the backtrace information.