New Machine Identity for Compromised Credentials in Machine-to-Machine (M2M) Communication

Wil Liam Teng

Reuben College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2025

Acknowledgements

First and foremost, I would like to express my gratitude towards my DPhil supervisor, Professor Kasper Rasmussen, for his continuous guidance. I would also like to give many thanks to Professor Bart Preneel, Professor Andrew Martin, Professor Ivan Martinovic, and Dr. Christopher Hargreaves for their feedback on the work of this thesis. I would further dedicate this thesis to my beloved parents and family whose emotional and moral support are the biggest reasons that I am able to embark on this journey and also complete it to the very end. I will not be forgetting to show my past and present selves some appreciation to be able to persevere through this long and lonely journey, even when things do not look promising. I am taking this opportunity to additionally extend my appreciation to all my friends that I have made throughout the course of my DPhil whose faith in me is bigger than my own. Finally, to anyone that I have fatefully met since my first day in Oxford: thank you.

Abstract

Machine-to-machine communication (M2M) refers to the communication between machines without the active intervention of human users. As the credentials of a machine that form the machine's identity are made up of secret information, this brings detrimental effects to the security of M2M communication if the secrecy of the machine credentials is compromised by an adversary. This thesis strengthens the security of M2M communication protocols to address the problems of credential compromise in three different applications of M2M communication: remote access applications using the Secure Shell (SSH) protocol, automated development workflows within internal organisational networks, and end-to-end encrypted instant messaging applications using the Signal protocol.

We improve the SSH protocol to provide a novel detection mechanism against an adversary that can simultaneously compromise the long-term identity keys of both the SSH client and server, offering the detection of the adversary even when the adversary can access and compromise the SSH server as root. We additionally propose a framework as a solution to counter the credential leakage problem in automated development workflows within internal organisational networks where credentials are often hard-coded or embedded into the application source code or automation scripts, mitigating the implications of a credential leakage to an adversary that compromises the credentials and uses them to access application services in the system. We further propose improvements to the Signal protocol to enhance the key authentication of the protocol with the purpose of providing the detection of an active Man-in-the-Middle adversary that compromises all secrets of a communicating client, with our solution built directly into the Signal protocol without requiring any out-of-band channel or user interaction and without introducing additional parties into the communication system.

We provide computationally secure solutions in each of the M2M application domains to facilitiate real-world deployments without requiring specialised machines. We further perform the necessary security analysis to prove our novel security guarantees while also preserving the existing guarantees of the solutions that our improvements build on. Finally, we implement proof-of-concept software for each of our solutions to demonstrate its practicality.

Contents

List of Figures

List of Abbreviations

AE . . . . . Authenticated Encryption

AEAD . . . . . Authenticated Encryption with Associated Data

AES . . . . . Advanced Encryption Standard

AKE . . . . . Authenticated Key Exchange

API . . . . . Application Programming Interface

CA . . . . . Certificate Authority

CBC . . . . . Cipher Block Chaining

CDH . . . . . Computation Diffie-Hellman

CRL . . . . . Certificate Revocation List

DDH . . . . . Decisional Diffie-Hellman

DSS . . . . . Digital Signature Standard

DevOps . . . . Development Operations

DL . . . . . Discrete Logarithm

DoS . . . . . Denial-of-Service

E&M . . . . . Encrypt-and-MAC

ECC . . . . . Elliptic Curve Cryptography

ECDH . . . . . Elliptic Curve Diffie-Hellman

ECDSA . . . . Elliptic Curve Digital Signature Algorithm

ECQV . . . . . Elliptic Curve Qu-Vanstone

EtM . . . . . Encrypt-then-MAC

EUF-CMA . . Existential Unforgeability under Chosen Message Attacks

HMAC . . . . Hash-based Message Authentication Code

HKDF . . . . . HMAC-based Key Derivation Function

IIoT . . . . . Industrial Internet-of-Things

IoT . . . . . Internet-of-Things

IKE . . . . . Internet Key Exchange

IPsec . . . . . Internet Protocol Security

IND-CCA . . Indistinguishability under Chosen Ciphertext Attacks

INT-CTXT . . Ciphertext Integrity

ISP . . . . . Internet Service Provider

IV . . . . . Initialisation Vector

KDF . . . . . Key Derivation Function

M2M . . . . . Machine-to-Machine

MAC . . . . . Message Authentication Code

MtE . . . . . MAC-then-Encrypt

MFA . . . . . Multi-Factor Authentication

MitM . . . . . Man-in-the-Middle

OCSP . . . . . Online Certificate Status Protocol

OIDC . . . . . Open ID Connect

PCS . . . . . Post-Compromise Security

PKI . . . . . Public Key Infrastructure

PUF . . . . . Physical Unclonable Function

RSA . . . . . Rivest-Shamir-Adleman

SAML . . . . . Security Assertion Markup Language

SHA . . . . . Secure Hash Algorithm

SSH . . . . . Secure Shell

SSO . . . . . Single Sign-On

TOFU . . . . . Trust on First Use

TPM . . . . . Trusted Platform Module

TLS . . . . . Transport Layer Security

VPN . . . . . Virtual Private Network

X3DH . . . . . Triple Diffie-Hellman

XOR . . . . . Exclusive-OR

ZKP . . . . . Zero Knowledge Proof

1 Introduction

Contents

Machine-to-machine (M2M) communication refers to the communication that takes place between machines without the interaction of a human user. M2M communication is used every day by millions of devices for automated and scheduled tasks that require routine communication with other devices. This includes processes such as automatic backup, automatic software updates, and many, many other things which would otherwise need manual effort. In addition M2M communication is the foundation for the so-called Development Operations (DevOps) and Internet-of-Things (IoT) as well as most industrial processes, e.g., manufacturing, healthcare, and power generation. Without realising, M2M communication has manifested itself in many processes of modern human lives, such as instant messaging, remote access, home automation etc. By 2029, the worldwide market volume for IoT alone is projected to reach 1.56 trillion US dollars (Statista, 2024).

An essential security component in M2M communication is the establishment of

a secure communication channel between machines, and this often translates into the authentication of machine identities to ensure that the machines involved in the communication are indeed who they claim to be. This is because without robust authentication schemes, a secure connection could inadvertently be established with a machine controlled by an adversary, potentially leading to unintentional exposure or modification of transmitted information, e.g., modified execution commands. Thus, machine identity authentication and secure channel establishment are intrinsically interconnected and often go hand-in-hand, as authentication serves as a prerequisite for secure communication.

In practice, machine identity authentication requires a machine to demonstrate the possession of its credentials, i.e., secret information known only to that machine. With connections being made without human interaction in M2M communication, the secrets for a machine required to authenticate a machine's identity and to bootstrap a secure channel should be available to the machine every time the connection is made. Unlike secrets retained in human memory, this often means that the secrets are stored on the machine locally in the clear, or perhaps protected by another key which is itself stored in the clear, which does not provide much more security from a determined adversary. A real-world example to illustrate this can be seen in the popular end-to-end encrypted instant messaging application, Signal, which stores the decryption key for the encrypted local database of its Windows client in plaintext within a text file (L. Abrams, 2018). Similarly, the private host keys for the OpenSSH (OpenSSH, n.d.) server used for server authentication are also stored unencrypted.

The unique challenge that is inherent to M2M communication lies in the fact that anyone with access to a machine allows them to extract the cryptographic secrets from the machine, which then grant access for the adversary to trivially authenticate themselves as the compromised machine's identity, and to subsequently access the service independently from the compromised machine from which the secrets were extracted. These threat actors can range from an insider such as a company's IT staff member to an external adversary that observes or controls the

connection, or worse, compromises the machine itself. This is not just a speculation of an adversary that exists merely in theory since an adversary such as this has, in fact, resulted in breaches at large companies (Kovacs, 2022; McDaniel, 2022; Engelberg, 2021; Kost, 2024). This unique challenge of M2M communication raises our first research question:

RQ1 What security guarantees can be provided for a compromised machine even after an adversary has compromised its secrets?

The credential leakage problem in M2M communication is further exacerbated by the traditional reliance of the security of a cryptographic system on on the secrecy of a machine's stored secrets, in accordance to Kerckhoffs' Principle (Kerckhoffs, 1883). From a computationally secure perspective, this means that an adversary with access to the machine's secrets becomes indistinguishable from the legitimate machine, which enables a full impersonation by the adversary. While certain secrets may be harder to extract than others, e.g., cryptographic keys residing in the Trusted Platform Module (TPM) of a machine, the reason this problem persists ultimately stems from the fact that the authentication in M2M communication rests solely on the secrecy of such secrets. While it is also true that this has been the conventional assumption for a cryptographic system, the security of a M2M communication system could be significantly strengthened by having security guarantees in place even in the event of a secret compromise. This brings us to our second question, in relation to the first one:

RQ2 What constitutes a new machine identity that facilitates in providing security guarantees for the compromised machine in the event of a secret compromise?

Motivated by real-world scenarios where secrets are frequently exposed, this thesis aims to investigate on how we can improve the authentication in M2M communication that account for situations involving compromised secrets. Rather than focusing on the security methods to make it hard for a secret compromise to occur in the first place, we look at a different but closely related scope of the problem

by instead considering security after the unfortunate event where these secrets are compromised. Given that the use of secrets remains the predominant method for authentication in practical applications, abandoning this approach entirely is not a viable solution. As a starting point we look at other machine authentication methods that complement the usage of a secret for a more robust machine identity and one that is resilient to a secret compromise in various M2M applications. Our third and final research question, that builds upon the previous two, is then:

RQ3 How should an authentication protocol be designed to leverage the new machine identity and provide the post-compromise security guarantees?

1.1 Research Scope

M2M communication spans a wide range of applications. In this thesis, we narrow down our research scope into three key M2M application domains, each explored in a dedicated chapter from Chapter 5 to Chapter 7. These are, namely, remote access applications using the SSH protocol (Chapter 5), automated development workflows within internal organisational networks (Chapter 6), and end-to-end encrypted messaging applications using the Signal protocol (Chapter 7).

While SSH is traditionally thought as a human-centric remote access and command execution application, more often than not SSH is also used for automated tasks where no human interaction is involved. Examples of these include executions initiated through a shell script and a job scheduler such as `cron`. For these automated SSH tasks, password-based and key-based authentication remain prevalent despite SSH supporting other authentication methods. The credentials in such cases, i.e., the passwords or the private SSH keys, are often stored in the clear on the client machine, typically in a local text file. This use case of SSH highlights that it falls within the context of M2M application domains.

In end-to-end encrypted messaging applications, we look into the Signal protocol as our research scope, which is the de facto standard for secure communication in this domain. Although such applications were designed with human users in

mind and assume the presence of human users to manually type the messages to be sent, the focus of this thesis is the protocol itself, independent of its user-facing implementations or applications. The Signal protocol allows the users to receive or send messages without the users manually authenticating themselves to the server for each message. In one instance, the keys generated by the protocol is stored encrypted on a device with a decryption key that is stored in plaintext (L. Abrams, 2018). As such, in spite of its user-centric origins, we consider the protocol to be within the broader scope of M2M application domains.

1.2 Thesis Contributions

Our foremost contribution in this thesis is the introduction of a new and stronger adversary model in the three M2M applications as previously listed in Section 1.1. We describe the adversary model more broadly in Chapter 4 along with the rationale and we motivate its significance. The adversary model captures the real-world adversarial ability to compromise *all* respective secrets of a machine in each M2M application, while also maintaining the adversarial power of controlling the communication channels. Despite the stronger adversary model, we are able to prove security guarantees for the compromised machine in each of the M2M applications, pushing the boundaries of M2M authentication protocols and challenging the previously established notion that providing security guarantees for a compromised machine is trivial given that the adversary has complete knowledge of all the machine's secrets and can fully impersonate it.

Remote access applications using the SSH protocol. Chapter 5 solves the problem of long-term identity key compromise for the SSH protocol, which currently still remains an open problem. We introduce a defense mechanism to protect against the compromise of a SSH client's secrets by the adversary which allows the security of the communication channel between the SSH client and the SSH server to heal itself. Our recovery mechanism allows the detection of the adversary if the adversary uses the compromised secrets of the SSH client to login and impersonate the client, and that even after the adversary compromises the SSH

server's private key during the adversary's session. We provide an improved design for the SSH Transport Layer Protocol and the SSH Authentication Protocol with the security analysis of the protocols to prove that our new security properties hold with respect to the stronger adversary, while also preserving the existing guarantees that the original SSH protocol has. We integrate our changes to the protocol into the OpenSSH implementation, and we illustrate the practicality and feasibility of our solution evaluated from the experimental results on the performance overhead of our modified protocol against the original SSH protocol. The evaluation shows that our modified SSH protocol only has a negligible performance overhead compared to the original SSH protocol which will be unnoticable in real-world applications.

Automated development workflows within internal organisational networks. In Chapter 6, we mitigate the problem of key (mis-)management of embedding or hardcoding a secret into application source code or execution scripts in the clear for automation purposes, as is a common albeit bad practice for development workflows in an internal organisational network with a centralised trusted server. As this opens up opportunities for an adversary with access to the machine or the source code and execution scripts themselves to compromise the secret and impersonate the machine, the compromise often goes unnoticed by the central trusted server. We present our solution that makes it possible for the server to have a mechanism in place for detecting the compromise by using a novel machine identity based on the action sequence (or 'actions') of a machine. We provide a proof-of-concept of our solution, which we compare its performance with the SSH protocol. Our experimental evaluation shows that the performance is within a feasible range with the performance of the SSH protocol introduced in Section 2.5.

End-to-end encrypted instant messaging applications using the Signal protocol. We solve the issue of key authentication in the Signal protocol that currently requires a pair of communicating users to authenticate their key fingerprints in an out-of-band fashion to detect a Clone-in-the-Middle's presence, i.e., an adversary that clones a machine's secrets and then inserts itself in the middle of the communication. Our solution alleviates the requirements of user intervention

as well as an out-of-band channel. Chapter 7 provides our solution that builds directly on top of the Signal protocol with respect to an adversary that clones *all* secrets of a client device while also not requiring additional parties to Signal's communication architecture. We achieve this by associating the identity of each client with a recursively updated key fingerprint of the client device as its identity and by taking advantage of Signal's relay server, the powerful adversary can be detected. We describe our concrete changes to the Signal protocol and we analyse our new security guarantees with respect to the stronger adversary model while also preserving the existing guarantees that the original Signal protocol provides. We then demonstrate the practicality of our modified protocol by implementing them into the open-source Signal protocol library which we use to compare and analyse the experimental performance of our modified protocol with the original Signal protocol. Our results show that our changes do not incur a significant performance overhead for retaining the current user experience of the original Signal protocol.

1.3 Chapters Overview

We start off with Chapter 2 with the background concepts necessary to understand the rest of the thesis, including a high-level introduction to the primitives used as building blocks for the design of the cryptographic protocols as well as the descriptions to the real-world security protocols that the work of this thesis is based on: the Secure Shell (SSH) protocol and the Signal protocol.

In Chapter 3, we go into providing a comprehensive literature review of the landscape of the large body of existing research work relevant to the contribution of this thesis, covering the design of M2M authentication methods and protocols as well as the existing research on Single Sign-On (SSO) systems, SSH protocol, and the Signal protocol.

We then proceed to describe in a wider context in Chapter 4 the stronger adversary model that we propose along with the motivation for doing so as well as the design goals of our solutions to defend against the adversary, both of which shape the methodologies and the cryptographic techniques that we employ in our solutions.

Chapter 5 to Chapter 7 outline the details the main work of this thesis briefly introduced in Section 1.2 in the three M2M application domains as listed in Section 1.1, with each chapter containing the specific system and adversary model for the respective M2M application domain, the concrete specification of our proposed solution, the security analysis of the proposed solution, and the implementation of the solution.

Chapter 8 picks up on where the previous chapters left off and continue the discussion on the topics that are necessary to prepare our solutions presented in this thesis for real-world deployments, such as the topics of persistent adversarial access, the trust in and the availability of the third party, and credential management.

Last but not least, we summarise and conclude the thesis in Chapter 9, and we provide the future work that can be undertaken as an extension of the work in this thesis.

1.4 Publications

Part of the work in this thesis is published in conference proceedings. In the following, we list the publications that form the basis of the chapters to show that the research work presented in this thesis has undergone rigorous academic reviews. The full details of the publications can be found from the cited entry for the publication in the bibliography.

Chapter 5 *Wil Liam Teng and David Soler Garcia and Kasper Rasmussen*
How to Heal a Cracked Shell: Post-Compromise Security for Long Term Identity Keys in SSH.
(Currently under submission: Teng, Soler Garcia, et al., n.d.)

Chapter 6 *Wil Liam Teng and Kasper Rasmussen*
Actions Speak Louder Than Passwords: Dynamic Identity for Machine-to-Machine Communication.
ARES '23: Proceedings of the 18th International Conference on Availability, Reliability and Security (Teng and K. Rasmussen, 2023)

Chapter 7 *Wil Liam Teng and Kasper Rasmussen*
Attack of the Cloned: In-Band Active Man-in-the-Middle Detection for the Signal Protocol.
(Currently under submission: Teng and K. Rasmussen, n.d.)

2
Background

Contents

We open this chapter with a brief overview of the cryptographic primitives and

their associated security guarantees. This chapter then introduces the high-level concept of a cryptographic protocol with the cryptographic primitives as its building blocks, followed by an introduction to the concept of an adversary model and the more advanced and high-level security guarantees from a cryptographic protocol. This chapter further provides the background for two real-world cryptographic protocols: the Secure Shell (SSH) protocol and the Signal protocol to serve as the foundation for the later chapters of the thesis. We give an informal introduction to these cryptographic concepts as background just so the readers can grasp their basic ideas and be familiar enough with them to understand their usage in the later chapters of the thesis. For a more detailed comprehension, we invite interested readers to examine the formal descriptions of these cryptographic concepts.

2.1 A Brief Primer on Cryptographic Primitives

Cryptography refers to the study of techniques and algorithms to secure the data exchanged between two parties in the presence of an adversary. Cryptography is classified into two categories: symmetric key cryptography and asymmetric key cryptography. Symmetric key cryptography uses the same secret key for both parties whereas asymmetric key cryptography uses a public-private keypair where only the private key is kept secret. Cryptographic techniques and algorithms provide important security guarantees that form the basic building blocks of cryptographic protocols. In this section, we cover the more basic security properties such as confidentiality, authentication, and non-repudiation. We then look into the concept of a cryptographic protocol and the Diffie-Hellman key exchange protocol used in many practical protocols.

2.1.1 Message Confidentiality Schemes

Confidentiality is the security guarantee that the message (or data) is only accessible to authorised parties. This security guarantee is achieved through encryption schemes, also known as ciphers. An encryption scheme consists of three algorithms:

- A key generation algorithm, $\texttt{KeyGen}(\lambda) \rightarrow (K_E, K_D)$, that takes in an input λ representing a security parameter, e.g., the key size, and outputs a key pair (K_E, K_D).

- An encryption algorithm, $\texttt{Enc}_{K_E}(M) \rightarrow C$, is a transformation process that takes the generated encryption key K_E and a message M as inputs, and outputs ciphertext C, which is a scrambled representation of the original message M.

- A decryption algorithm, $\texttt{Dec}_{K_D}(C) \rightarrow M$, is a process to recover the original message M from the ciphertext C. The algorithm takes in the generated decryption key K_D and a ciphertext C as inputs, and outputs the original message M.

For an encryption scheme to be considered secure, it must satisfy the notion of indistinguishability under (adaptive) chosen ciphertext attacks (IND-CCA), which represents the strongest security for an encryption scheme. Informally, IND-CCA security ensures that even if an adversary can obtain ciphertexts for messages of their choice and decrypt arbitrary ciphertexts, the adversary cannot distinguish between a challenge ciphertext generated from the encryption algorithm and a random message with a probability better than guessing (i.e., at most $\frac{1}{2}$) (Katz and Lindell, 2007).

In symmetric key cryptography, the encryption key and the decryption key are identical, i.e., $K_E = K_D$, but in asymmetric key cryptography, the encryption key is the public key and the decryption key is the private key. An example of a symmetric key encryption scheme is the Advanced Encryption Standard (AES) (Rijmen and Daemen, 2001) based on the Rjindael cipher (Daemen and Rijmen, 2002) and an example of asymmetric key encryption scheme is the RSA encryption scheme (Moriarty et al., 2016). In the later parts of this thesis, we treat the key generation and decryption implicitly. We also denote the output of a symmetric key encryption of a message m under a key k as $\{m\}_k$ and the output of an asymmetric key encryption of a message m under a public key pk as $E_{pk}(m)$.

2.1.2 Message Authentication Schemes

Message integrity is the security guarantee that a message (or data) is not modified during transmission whereas message origin authentication is the security guarantee that verifies the authenticity of a sender of a message. Together, they form the security guarantee of message authentication which ensures that the messages originate from a verified source and that they are not modified during transmission (ISO, 1989). Message authentication is provided by a message authentication scheme which consists of three algorithms:

- A key generation algorithm, $\texttt{KeyGen}(\lambda) \rightarrow (K_S, K_V)$, that takes in a security λ parameter such as the key size as an input and outputs a key pair (K_S, K_V).
- A signing algorithm, $\texttt{Sign}_{K_S}(M) \rightarrow \sigma$, that produces a signature σ over a message M using the signing key K_S.
- A verification algorithm, $\texttt{Verify}_{K_V}(M, \sigma) \rightarrow$ True or $\perp$, that outputs a True value to verify if the signature σ is valid using the verification key K_V. A signature is valid if the signature is generated from the signing algorithm under the corresponding signing key K_S, i.e., if $\texttt{KeyGen}(\lambda) \rightarrow (K_S, K_V)$ and $\texttt{Sign}_{K_S}(M) \rightarrow \sigma$. The algorithm outputs a failure symbol $\perp$ otherwise. The inputs here are the message M to be verified and the signature σ.

The security of a message authentication scheme relies on the concept of existential unforegeability under (adaptive) chosen message attacks (EUF-CMA), which is the strongest available security for such schemes. The informal definition of EUF-CMA states that even after observing valid message-signature pairs of an adversary's choice, the adversary only has a negligible probability of generating (or 'forging') a message-signature pair where the message has not previously been signed (Katz and Lindell, 2007).

A symmetric key message authentication scheme is known as a Message Authentication Code (MAC) where the sigining key and the verification key are equal, that is, $K_S = K_V$, and that the signature σ is also known as a MAC or a tag. For

an asymmetric key message authentication scheme, this is achieved with a digital signature where K_S and K_V are distinct. An example of a MAC algorithm is a Hash-based MAC or a HMAC (NIST, 2008) based on a hash function (covered later in Section 2.1.5) and an example of a digital signature is the Digital Signature Standard (DSS) (NIST, 2023) specifying the RSA algorithm (Moriarty et al., 2016). In the subsequent parts of the thesis, the key generation algorithm is treated implicitly and we use the term 'Verify' to denote the execution of the verification algorithm. We also adopt the notation $\texttt{MAC}_K(M)$ to represent the MAC of a message M under the symmetric key K and $\texttt{Sign}_{sk}(M)$ to represent the digital signature of a message m under the private key sk.

2.1.3 Authenticated Encryption Schemes and Authenticated Encryption with Associated Data Schemes

An Authenticated Encryption (AE) scheme is a symmetric-key cryptographic primitive that provides both security guarantees of message confidentiality and message authentication. This is achieved through the combination of an encryption scheme and a MAC scheme. The three modes of composition for the two cryptographic primitives are: Encrypt-and-MAC (E&M), MAC-then-Encrypt (MtE), and Encrypt-then-MAC (EtM). E&M encrypts the message and computes the authentication tag over the plaintext message independently with two separate processes. However, this is vulnerable to CCAs as an adversary can modify the ciphertext and then observe the results of the authentication tag verification for cryptanalysis. MtE first generates the authentication tag over the plaintext message and subsequently encrypts the message to produce the ciphertext. This mode was used in older versions of TLS (pre-1.2) but this mode presents practical issues as the ciphertext has to be decrypted before the authentication tag can be verified. EtM encrypts the message and subsequently generates the authentication tag over the ciphertext. EtM is the widely accepted approach for composition and it is deployed in real-world protocols such as IPsec and TLS 1.2.

Where an AE scheme can be realised by combining separate encryption and MAC schemes possibly with distinct keys, an Authenticated Encryption with Associated Data (AEAD) scheme is a symmetric key cryptographic primitive providing both security guarantees of confidentiality and authentication under a single key. An AEAD scheme additionally supports the authentication of optional unencrypted associated data such as an IP address. An example of a EtM AEAD scheme is ChaCha20-Poly1305 (Nir and Langley, 2015). Similar to an encryption scheme (in Section 2.1.1), an AEAD scheme consists of three algorithms:

- A key generation algorithm, $\texttt{KeyGen}(\lambda) \rightarrow K$, that takes as input a security parameter λ (e.g., key length) and outputs a symmetric key K.

- An encryption algorithm, $\texttt{Enc}_K(N, M, A) \rightarrow (C, T)$, that takes as inputs the following: the generated key K, a nonce or initialisation vector N that is unique for each encryption under the same key, a message M, the optional associated data A. The algorithm outputs a ciphertext C of the message M and an authentication tag T.

- A decryption algorithm, $\texttt{Dec}_K(N, C, A, T) \rightarrow M$ or $\perp$, that takes as inputs the generated key K, the nonce or the initialisation vector N, the ciphertext C, the associated data A, and the authentication tag T. The algorithm decrypts the ciphertext and also verifies the authentication tag. It returns the original message M if the authentication tag is valid, i.e., if the ciphertext and authentication tag are produced from the encryption algorithm with the same key used in the decryption algorithm ($\texttt{Enc}_K(N, M, A) \rightarrow (C, T)$ and $\texttt{KeyGen}(\lambda) \rightarrow K$). Otherwise, it outputs a failure symbol $\perp$.

The security of an AEAD scheme combines both notions of IND-CCA security as described in Section 2.1.1 for the ciphertext and EUF-CMA security Section 2.1.2 for the authentication tag (although the AEAD scheme equivalent for EUF-CMA security is called ciphertext integrity or INT-CTXT security (Bellare and Namprempre, 2000)). In the chapters that follow, we treat the key generation algorithm,

the nonce or the initialisation vector, and the decryption of the ciphertext as implicit. We denote an AEAD operation on a message M under the key K as $\{M\}_K, \texttt{MAC}_K(M)$ where $\{M\}_K$ represents the ciphertext and $\texttt{MAC}_K(M)$ represents the authentication tag (or MAC).

2.1.4 Message Origin Non-Repudiation

Message origin non-repudiation refers to the security guarantee that the signer of a message cannot deny the act of signing the message. As only the owner of a private key possseses the private key and thus only the owner of the private key can be the signer of a message, this means that a digital signature on a message can only originate from the signer. Thus, a digital signature additionally provides the security guarantee of non-repudiation, in addition to message authentication. While a MAC algorithm similarly provides message authentication, it does not provide non-repudiation. This is because either party that owns the symmetric key can generate a MAC on the message.

2.1.5 Hash Functions

Hash functions are functions that take in an input of an arbitrary length and outputs a unique fixed-length value as the representation or the fingerprint of the input, also called as a hash. Hash functions do not require a key as an input and thus do not provide the authentication guarantee themselves. However, hash functions are often used by message authentication algorithms to compute the hash of a message and operate on the hash for a more efficient operation, instead of operating on the original message directly. This is due to the unique properties of the hash functions (Menezes et al., 2018): (1) any change in the input will produce an entirely different hash, (2) one-way, which means that the hash cannot be used to recover the original input, (3) second pre-image resistant, which means it is hard to find a second input that produces the same hash, and (4) collision-resistant, which means that it is infeasible to find a hash collision, i.e., two inputs that produce the same hash. An example of a hash function is the variants of Secure Hash

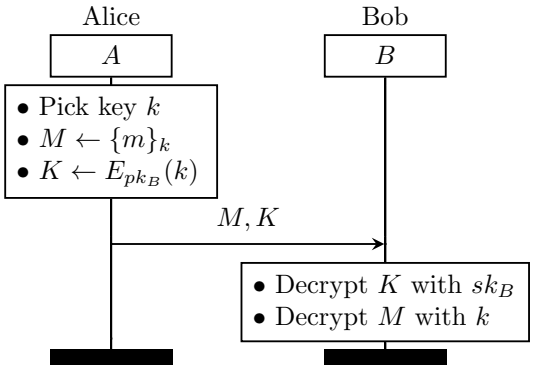


Figure 2.1: An example of a cryptographic protocol. This protocol implements a hybrid key technique to send a large message m efficiently. Alice is assumed to know that the public key pk_B belongs to Bob.

Algorithm 2 (SHA-2) (NIST and Dang, 2012) such as SHA-256 and SHA-512. In this thesis hash functions will be denoted $\mathtt{H}(\cdot)$.

2.1.6 Key Derivation Functions

Key derivation functions (KDFs) are cryptographic functions used to derive one or more secret keys as outputs from one or more input values, depending on the construction of the KDFs. The inputs might be cryptographically weak, meaning not uniformly ditributed, such as a password, and might also be partially known to the attacker, making them unsuitable for direct use as cryptographic keys (Krawczyk, 2010). The purpose of a KDF then is to convert such inputs into keys that are suitable for cryptographic operations. For the derived keys to be cryptographically strong, a KDF must produce outputs that satisfy the security requirements of: (1) high entropy, that is, they should retain enough entropy so that they resist brute-force attacks even when parts of the inputs and outputs are known to the adversary, and (2) pseudorandomness, that is, being indistinguisable from random. An example of a KDF is a HMAC-based Extract-and-Expand Key Derivation Function (HKDF) (Krawczyk and Eronen, 2010). We use the notation $\mathtt{KDF}(\cdot)$ to denote key derivation functions.

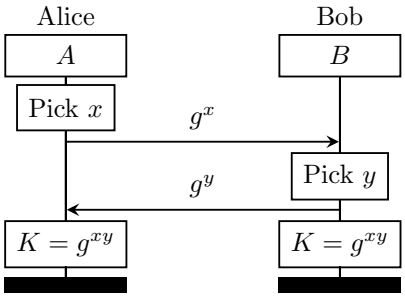


Figure 2.2: The Diffie-Hellman key exchange protocol.

2.2 Cryptographic Protocols

A cryptographic protocol is a series of steps that achieves higher-level and sometimes application-specific security guarantees using cryptographic primitives as building blocks. The cryptographic primitives are usually treated as black-boxes in a protocol and the steps in a protocol also include sending or receiving specific messages to other communication parties. For a better visualisation, Figure 2.1 shows an example of a cryptographic protocol. The protocol showcases a hybrid key encryption technique for Alice to send a large message m to Bob, since encrypting a large message directly using an asymmetric key encryption function is considered inefficient. In this example, we assume that Alice already knows the public key of Bob. Alice starts by picking a random symmetric key k and encrypting the large message m with the fresh key k. Alice then encrypts the fresh key k with Bob's public key and sends both the encrypted message and the encrypted key to Bob. On Bob's end, Bob first uses his private key sk_B to decrypt K and retrieve the symmetric key k. Bob then uses the key k to decrypt the ciphertext to obtain the message m.

2.2.1 Diffie-Hellman Key Exchange Protocol

A more specific type of a cryptographic protocol is a key exchange protocol that allows two parties to contribute their cryptographic materials called a keyshare to generate a shared symmetric key known only to the two parties. Here we provide an overview of the Diffie-Hellman key exchange protocol (Diffie and Hellman, 1976) that is widely adopted in practical applications as a building block, which we will see in the two real-world protocols later in this chapter.

As depicted in Figure 2.2, Alice and Bob respectively picks a random exponent as their respective (ephemeral) private key. Alice and Bob then calculate their respective public key by raising g to the random exponent. Mathematically, g is the generator of a cyclic multiplicative group of a prime order in which the discrete logarithm is hard. Alice and Bob then exchange the public keys with each other and generate the symmetric key K by raising their partner's public key with their own private key. The protocol also assumes that the ephemeral private keys are securely erased and never reused so that they cannot be recovered.

Against a passive adversary, that is, an adversary that only observes the messages in the communication channel (as opposed to an active adversary introduced in Section 2.3), the security of Diffie-Hellman key exchange protocol depends on the hardness of three mathematical problems, in decreasing order of the hardness: the Discrete Logarithm (DL) problem, the Computational Diffie-Hellman (CDH) problem, and the Decisional Diffie-Hellman (DDH) problem. The DL problem states that given g^x, it is hard to compute x. The CDH problem builds on the DL problem and states that it is hard to compute g^{xy} given g^x and g^y. Similarly, the DDH problem is a simpler problem than CDH that states given g^x, g^y, and a random number r, determine if $g^{xy} = r$. Throughout this thesis we assume that all three problems are hard, and in the proofs in the security analyses of the later chapters, we will mention only the DL and the CDH problems.

2.3 Adversary Models

Before we go into the background on real-world protocols, we have to first define the power of an adversary. An adversary in the context of cryptographic protocols refers to a party that does not follow and deviate from the steps of a protocol with the purpose of breaking an underlying security guarantee of the protocol. As an example, an adversary attempts to break an encryption to reveal the content of the message by trying all possible combinations of a symmetric key (brute-force attacks). An adversary model is the assumptions made on an adversary's abilities and goals in a cryptographic system or simply, a cryptosystem. Each aforementioned

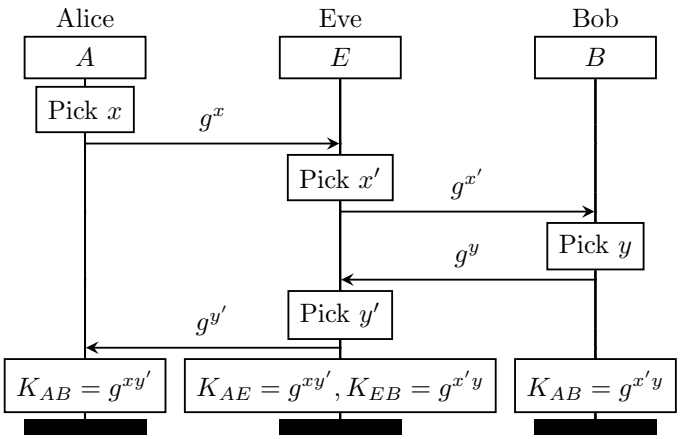


Figure 2.3: A Man-in-the-Middle attack on the Diffie-Hellman key exchange protocol.

cryptographic primitive has its respective adversary model but in this thesis we focus on the adversary models relevant at the level of cryptographic protocols.

Real-world cryptographic protocols for the transmission of messages are often designed with the classical adversary model of Dolev-Yao (Dolev and Yao, 1983) in mind. This adversary model captures the ability of an adversary to control the communication channels and manipulate the messages sent through the channels. Specifically, the model assumes that the adversary is an active adversary where the adversary is able to eavesdrop, intercept, delay, modify, replace, replay, and even drop a message. This is also sometimes described as *the adversary carries the message* or that *the adversary acts as a Machine-in-the-Middle* (MitM). The adversary in this model is additionally assumed to not be able to break the underlying security guarantees of the cryptographic primitives in the protocol, including not having access to the cryptographic keys, and that the adversary does not have physical access to the device. The Dolev-Yao adversary model has been the de facto standard for designing cryptographic protocols because it reflects real-world communication where the messages exchanged between two machines over the Internet pass through multiple machines with different owners, e.g., different access points and different Internet Service Providers (ISPs). The SSH protocol (Ylonen, 2006a) and the Transport Layer Security (TLS) protocol (Dierks and Rescorla, 2008), for example, are designed with this adversary model in mind.

For a concrete example, consider the MitM attack on the Diffie-Hellman protocol as depicted in Figure 2.3. If we assume Eve as a Dolev-Yao adversary, then Eve is able to impersonate Alice to Bob and Bob to Alice. From Alice's and Bob's point of view, it seems as though the shared secret key K_{AB} is only known between the two of them. In reality, Eve has the same keys as Alice and Bob without either of them knowing. This allows Eve to break the secrecy of the generated session key K_{AB} which should be known only to Alice and Bob. Subsequently, this allows Eve to decrypt the messages exchanged between Alice and Bob, and to modify or inject messages to either Alice or Bob on the channels. The reason for this attack is mainly because the origins of the exchanged messages are not authenticated, which reinforces the idea that the establishment of a secure channel and authentication go hand-in-hand as mentioned in Chapter 1. This example again exemplifies that an adversary model is a necessary component for the design of a cryptographic protocol.

2.4 Security Guarantees from a Cryptographic Protocol

We discuss the more advanced and high-level topics from a cryptographic protocol in this subsection. These topics will be extensively applied throughout the remainder of the thesis.

2.4.1 A Secure Channel

Now that we have covered the Dolev-Yao adversary model, a secure channel ensures message confidentiality, message authentication, and replay protection in the communication between parties against such an adversary. Message confidentiality and message authentication have been previously explained in Section 2.1.1 and Section 2.1.2 respectively. Replay protection refers to the prevention of the adversary from reusing messages captured from a previous run of a protocol in a future session.

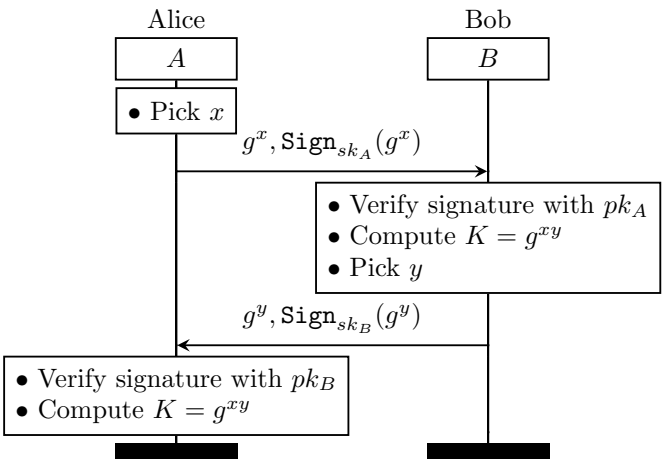


Figure 2.4: An authenticated Diffie-Hellman key exchange protocol. The protocol assumes that Alice and Bob each possesses their long-term private key to generate the signatures, i.e., sk_A for Alice and sk_B for Bob. The protocol also assumes that Alice and Bob each has the long-term public key from their respective partner for signature verification, i.e., pk_A for Alice's public key and pk_B for Bob's public key.

2.4.2 Authenticated Diffie-Hellman Key Exchange

Figure 2.4 shows an authenticated Diffie-Hellman key exchange protocol to counter the MitM attack from Figure 2.3. With the added digital signatures over the ephemeral public keys g^x and g^y, the protocol authenticates the messages to the respective parties and also ensure the integrity of the messages. This prevents the adversary from impersonating the party on either end of the communication because a successful impersonation requires the adversary to forge the digital signatures. This is computationally infeasible by the EUF-CMA security definition and the assumption of the adversary not capable of breaking the underlying cryptographic primitives in the Dolev-Yao model. Interestingly, the protocol in Figure 2.4 does not fulfill the definition of a secure channel as specified in Section 2.4.1 since the protocol does not prevent a Dolev-Yao adversary from replaying the messages to each party in a future run of the protocol.

2.4.3 Perfect Forward Secrecy

The authenticated Diffie-Hellman key exchange protocol in Figure 2.4 also brings forth an important security guarantee: perfect forward secrecy (PFS). PFS is the security guarantee that the compromise of a party's long-term private key does

not compromise the secrecy of previously established session keys. Indeed, from Figure 2.3, the compromise of long-term private keys sk_A and sk_B for the signature generation does not provide the adversary the capability of deriving the session K, due to the CDH assumption and the secure removal and unreusable assumption of the ephemeral private keys, which are the random exponents x and y. Another example of a protocol providing the PFS guarantee is the SSH protocol, introduced in Section 2.5 and Figure 5.3, and the Signal protocol, introduced in Section 2.6.

2.4.4 Post-Compromise Security

Post-compromise security (PCS) refers to the security guarantee that enables the restoration of a secure channel at a future time after an adversary has compromised the secret keys temporarily. This stands in contrast to the previous PFS guarantee, as PFS provides the secrecy of past session keys. PCS, on the other hand, provides the secrecy of future keys and thus it is also known as the 'self-healing' guarantee of a protocol. PCS is an inherent and prominent security guarantee in the Signal protocol, introduced in Section 2.6 as the foundation for one of the main contributions of this thesis in Chapter 7. First coined and formally defined by Cohn-Gordon, Cremers, Dowling, et al., 2020, PCS is achieved in the Signal protocol through the Double Ratchet algorithm, which updates the derived session keys at discrete time intervals known as epochs. If an adversary compromises the session keys on epoch t, then the Double Ratchet algorithm requires at most two epochs $t + 1$ and $t + 2$ for the adversary to be unable to derive the updated session keys, and thus locks the adversary out of future communication. This assumes that the adversary is passive in these two epochs and that the two epochs represent the 'healing speed' of the protocol (Blazy, Boureanu, et al., 2023). We refer to Section 2.6 for a more detailed description of the Double Ratchet algorithm and the definition of epochs.

2.4.5 Key Authentication

The security guarantees that we have discussed thus far assume that the communicating parties know that the corresponding key, particularly a public key,

truly belongs to a party. This security guarantee is known as key authentication. Key authentication is essential as the public keys themselves are transmitted over an insecure communication channel such as the Internet. Without proper authentication, this means that the public keys themselves are vulnerable to an adversary intercepting and modifying the transmitted keys to impersonate a certain identity.

The devastating effect of not authenticating a public key can be seen through the MitM attack on the Diffie-Hellman key exchange protocol in Figure 2.3. Even the authenticated variant of the protocol to prevent this attack illustrated in Figure 2.4 depends on the assumption that the public keys of the parties involved are correctly authenticated. This again demonstrates that security guarantees are not preserved unless the public keys themselves are verified to belong to the intended parties.

In practice, key authentication is achieved through various methods. The most common is through the use of a digital certificate, which is a digital document that binds the details of an identity, such as a domain name or an IP address, to a public key through a digital signature. The certificate is digitally signed by a trusted third-party called the Certificate Authority (CA) who vouches for the legitimacy of the public key. We denote the certificate of a party A as Cert_A. The infrastructure for the management, issuance, and distribution of certificates is known as a Public Key Infrastructure (PKI). Alternatively, key authentication can be realised by other approaches such as the Trust-on-First-Use (TOFU) model (introduced in Section 2.5.1), where a key is accepted as authentic the first time it is received and the key is stored for future verification; and user-assisted out-of-band key authentication (introduced in Chapter 7) where users manually verify key fingerprints via a separate communication channel.

2.5 The Secure Shell Protocol

As the work in Chapter 5 is based on the Secure Shell Protocol, we provide a brief introduction to the protocol in this section as background. The SSH Protocol is a protocol for establishing a secure channel for network services over an insecure

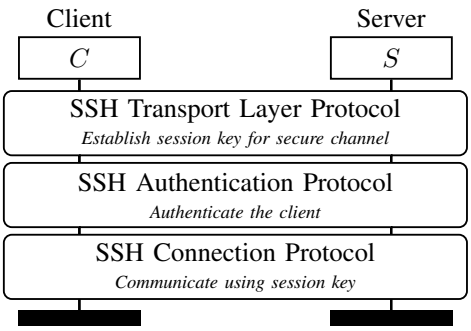


Figure 2.5: The SSH Protocol architecture as described by the RFC (Ylonen, 2006a). The protocol is run by a client and a server. The SSH Protocol has three major components: the SSH Transport Layer Protocol, the SSH Authentication Protocol, and the SSH Connection Protocol.

network. As shown in Figure 2.5, the SSH Protocol composes of three subprotocols: the SSH Transport Protocol (Ylonen, 2006c), the SSH Authentication Protocol (Ylonen, 2006b), and the SSH Connection Protocol (Ylonen, 2006d). The SSH Transport Protocol is first executed to authenticate the server and to establish a secure connection between the client and the server. It is normally run over a TCP/IP connection or any other reliable data stream. The SSH Authentication Protocol authenticates the client-side user to the server on top of the secure channel established by the SSH Transport Layer Protocol. The SSH Connection Protocol runs after the SSH Authentication Protocol where it multiplexes the encrypted tunnel into logical channels for different applications such as setting up secure interactive shell sessions, and forwarding arbitrary TCP/IP ports and X11 connections. In this thesis we will only be focusing on the first two protocols: the SSH Transport Layer Protocol and the SSH Authentication Protocol as these are where the mutual authentication of the server and the client occur. We describe the specifications of the SSH Transport Layer Protocol and the SSH Authentication Protocol together with our proposed changes in both protocols as our solution to the credential leakage problem of the SSH protocol in more details in Chapter 5.

2.5.1 Trust Models

According to the RFC (Ylonen, 2006a), an SSH server should have a host key as its identity keypair. During the key exchange in the SSH Transport Layer Protocol,

the client authenticates the server by verifying the server's signature as the server's proof of identity using this host key to have the assurance that the client is indeed contacting the right server. However the client needs to know beforehand that the (public) host key actually belongs to the server. The RFC (Ylonen, 2006a) suggests a strategy for the client to check the server's name-key association: the Trust-on-First-Use (TOFU) model. When a client connects to a server for the first time, the client accepts the host key without any verification. The client then saves the host key in its local database, and the host key of future connections to that server is compared against the stored key. Note that the RFC (Ylonen, 2006a) also permits implementations to provide additional verifications for checking the correctness of a public host key. One such example is to generate the hash of the public host key and compare the key fingerprint via an out-of-band communication channel such as through telephone or emails. In Chapter 5, we follow this trust model for the server's host key as stated in the system model of Section 5.1.1. Note that our solution in Chapter 5 to the protocol's credential leakage problem is similarly applicable with the certificate-based method for the server's name-key association that is also suggested by the RFC (Ylonen, 2006a).

2.6 The Signal Protocol

The Signal protocol, or simply Signal, is a protocol used for establishing and updating session keys between two clients, a sender and a recipient, for encrypting messages in an asynchronous instant messaging environment. Based on the Triple Diffie-Hellman (X3DH) handshake and the Double Ratchet Algorithm, Signal provides important security guarantees, namely, end-to-end message content confidentiality and authentication, perfect forward secrecy, and post-compromise security. Our solution in Chapter 7 is built on top of the Signal protocol, so we provide an overview of Signal as background in this section.

2.6.1 Phases of the Signal Protocol

Signal is further divided into four subprotocols, one for each phase of communication, namely, the registration phase, the session establishment phase, the asymmetric ratcheting phase, and the symmetric ratcheting phase. The registration phase is for a client to generate and commit its *prekey bundle* (explained later in this section) to the server. In the session establishment phase, a sender establishes a communication session with a recipient by first retrieving the prekey bundle of the recipient from the server and then computing the session keys to generate the first *envelope* (details in Section 7.2.3) containing the encrypted content. After the sender sends the first envelope to the recipient and the recipient receives the envelope and decrypts the content, the sender and the recipient are said to have entered the first time epoch of the communication session. The communication session enters a new time epoch if the roles of the sender and the recipient are swapped, and this occurs in the asymmetric ratcheting phase. In this phase, the sender generates new keying material to update (or *ratchet*) the session keys and sends the keying material along with the first envelope of the new time epoch to the recipient. In the same time epoch, i.e., when the roles of the sender and the recipient do not change, if the sender sends additional consecutive envelopes after the first envelope of that time epoch, the symmetric ratcheting phase takes place. Session keys are "rolled over" in this phase without introducing new keying material for each consecutive envelope, and the chain of envelopes is known as the message chain.

2.6.2 Algorithms in the Signal Protocol

We encapsulate the detailed steps of the four phases into algorithms which we use later in Section 7.2 to describe our improved Signal protocol, and we list the algorithms here. We explain the details of the algorithms that generate or update the session keys in each of the four phases of the Signal protocol, i.e., the registration phase, the session establishment phase, the asymmetric ratcheting phase, and the symmetric ratcheting phase.

The generation and the update of the session keys require three cryptographic primitives: a Diffie-Hellman key generation function and three key derivation functions (KDFs). The Diffie-Hellman key generation function is denoted $(g^x, x) \leftarrow \texttt{DHGen}(\lambda)$ which takes in a security parameter λ and outputs a Diffie-Hellman keypair. The first KDF is denoted $k_1 \leftarrow \texttt{KDF}_1(x_1, x_2, x_3, x_4)$ that takes in four key material inputs x_1, x_2, x_3, x_4, and outputs one symmetric key k_1. The second KDF is denoted $(k_1, k_2) \leftarrow \texttt{KDF}_2(x_1, x_2)$ that takes in two key material inputs x_1, x_2 and outputs two symmetric keys k_1, k_2. Lastly, the third KDF is denoted $(k_1, k_2) \leftarrow \texttt{KDF}_3(x_1)$ that takes in only one key material input x_1 but outputs two symmetric keys (k_1, k_2).

Note that all public-private key pairs generated in the algorithms are in the form of Diffie-Hellman key pairs, and the steps in the algorithms are analogously applied if Elliptic Curve Diffie-Hellman (ECDH) is used. We denote a time epoch as t and the number of envelopes sent or received by a client as n. We collectively refer to the secrets that a client stores as state, denoted $s^{t,n}$ for the state at a time epoch t after the processing of envelope n. Also note that the state of the client is updated accordingly after the execution of the algorithms. The algorithms are described below:

Prekey Bundle Generation

Algorithm 1: Generation of prekey bundle for the registration phase of a client.

Function `PkbGen(`λ`)`

1. $(g^{x_i}, x_i) \leftarrow \texttt{DHGen}(\lambda)$
2. $(g^{x_s}, x_s) \leftarrow \texttt{DHGen}(\lambda)$
3. $(g^{x_o}, x_o) \leftarrow \texttt{DHGen}(\lambda)$
4. $\sigma \leftarrow \texttt{Sign}_{x_i}(g^{x_s})$
5. $pkb \leftarrow (g^{x_i}, g^{x_s}, g^{x_o}, \sigma)$
6. $s^{0,0} \leftarrow (x_i, x_s, x_o)$
7. **return** $(pkb, s^{0,0})$

The Signal protocol starts with the registration phase where a client generates and registers its prekey bundle to the server. The generation of a client's prekey bundle follows Algorithm 1. The algorithm first generates three Diffie-Hellman

keypairs, known respectively as the identity keypair (g^{x_i}, x_i), the signed prekey keypair (g^{x_s}, x_s), and the one-time prekey keypair (g^{x_o}, x_o). The algorithm then generates a digital signature on the public signed prekey using the private identity key. An example of a digital signature that is based on the DL problem (introduced in Section 2.2.1) is the Elliptic Curve Digital Signature Algorithm (ECDSA), which relies on the elliptic curve variant of the DL problem. The Signal protocol instead uses a Schnorr-like digital signature scheme based on Edward curves. The prekey bundle pkb consists of the three public keys and the signature. The initial state $s^{0,0}$ of the client consists of the three corresponding private keys. Both the prekey bundle and the initial state are the outputs returned by the algorithm.

Ephemeral Keys Generation

Algorithm 2: Ephemeral Diffie-Hellman keypairs generation for a sender in session establishment or asymmetric ratcheting as required by the Triple Diffie-Hellman key agreement algorithm or the Double Ratchet algorithm respectively.

Function EphKeyGen(λ, $s^{t,n}$, phase)
  if phase = sessEst **then**
1 $(g^{x_b}, x_b) \leftarrow \texttt{DHGen}(\lambda)$
2 $(g^{x^1}, x^1) \leftarrow \texttt{DHGen}(\lambda)$
3 $epk^{t+1} \leftarrow (g^{x_b}, g^{x^1})$
4 $esk^{t+1} \leftarrow (x_b, x^1)$
  else if phase = asymRatch **then**
5 $(g^{x^{t+1}}, x^{t+1}) \leftarrow \texttt{DHGen}(\lambda)$
6 $epk^{t+1} \leftarrow (g^{x^{t+1}})$
7 $esk^{t+1} \leftarrow (x^{t+1})$
8 $s^{t+1,0} \leftarrow s_A^{t,n} \cup esk^{t+1}$
9 **return** $(epk^{t+1}, s^{t+1,0})$

Before a sender establishes or updates session keys in the phase of session establishment or in asymmetric ratcheting respectively, the sender first generates new Diffie-Hellman keypairs as described in Algorithm 2. For session establishment (sessEst), the algorithm generates two keypairs: a base keypair (g^{x_b}, x_b), and the first ratchet keypair (g^{x^1}, x^1); in asymmetric ratcheting (asymRatch), the algorithm

generates a single ratchet keypair $(g^{x^{t+1}}, x^{t+1})$ for the new time epoch $t+1$. The algorithm then collects the freshly generated public keys into a tuple epk^{t+1} and private keys into a tuple esk^{t+1}. The current state $s^{t,n}$ of the sender is then updated to $s^{t+1,0}$ that includes the freshly generated private keys. The algorithm then returns the tuple epk^{t+1} and the updated state of the sender as outputs. The generation of new keypairs is not required for a recipient of an envelope in session establishment or asymmetric ratcheting.

Message Key Generation

The Diffie-Hellman key pairs generated previously in Algorithm 1 or Algorithm 2 are then used to establish or update the session keys in Algorithm 3 in session establishment or asymmetric ratcheting respectively for the subsequent generation of an envelope. As the secrets stored in the state of a sender is different from a recipient, some steps in the algorithm are executed differently depending on whether the client running the algorithm is a sender or a recipient, but the output of the symmetric session keys are identical for both clients. For these steps we label the operations that are taken respectively by a sender or a recipient in Algorithm 3. If a step is not labelled whether it is taken by a sender or a recipient, both the sender and the recipient execute the same operation. The algorithm is divided into three parts depending on the phase of the communication: session establishment (sessEst), asymmetric ratcheting (asymRatch), and symmetric ratcheting (symRatch).

For session establishment, a sender executing this algorithm first retrieves its private identity key x_i, private base key x_b, and the first private ratchet key x^1 from its state. The sender also retrieves its communicating partner's, i.e., the recipient B's public identity key $g^{x_{i_B}}$, public signed prekey $g^{x_{s_B}}$, and public one-time prekey $g^{x_{o_B}}$ from the recipient's public keys. Conversely, a recipient B executing this algorithm first extracts its private identity key x_{i_B}, private signed prekey x_{s_B}, and the private one-time prekey x_{o_B} from its state. The recipient also extracts its communicating partner's, i.e., the sender's public identity key g^{x_i}, public signed prekey g^{x_b}, and the first public ratchet key g^{x^1} from the sender's public keys. Both the sender and the

Algorithm 3: Message key generation for generating or updating session keys based on the the Triple Diffie-Hellman key agreement algorithm for the session establishment phase, or the Double Ratchet algorithm for the asymmetric ratcheting phase as well as the symmetric ratcheting phase, for a client and its communicating partner B depending on the client's role.

Function `MessKeyGen`($s^{t,n}$, phase, pk_B)

if phase = sessEst **then**

1 Sender: $(x_i, x_b, x^1) \leftarrow s^{t,n}$

Recipient: $(x_{i_B}, x_{s_B}, x_{o_B}) \leftarrow s^{t,n}$

2 Sender: $(g^{x_{i_B}}, g^{x_{s_B}}, g^{x_{o_B}}) \leftarrow pk_B$

Recipient: $(g^{x_i}, g^{x_b}, g^{x^1}) \leftarrow pk_B$

3 $ms \leftarrow \mathtt{KDF}_1(g^{x_{s_B} x_i}, g^{x_{i_B} x_b}, g^{x_{s_B} x_b}, g^{x_{o_B} x_b})$

4 $(rk^1, ck_0^1) \leftarrow \mathtt{KDF}_2(ms, g^{x_{s_B} x^1})$

5 $(ck_1^1, mk_1^1) \leftarrow \mathtt{KDF}_3(ck_0^1)$

6 $(rk, ck, mk) \leftarrow (rk^1, ck_1^1, mk_1^1)$

7 Sender: $s \leftarrow s^{t,n} \setminus (x_o, x_b, x^1)$

Recipient: $s \leftarrow s^{t,n} \setminus (x_{o_B})$

else if phase = asymRatch **then**

8 Sender: $(x^{t+1}, rk^t) \leftarrow s^{t,n}$

Recipient: $(x^t, rk^t) \leftarrow s^{t,n}$

9 Sender: $(g^{x^t}) \leftarrow pk_B$

Recipient: $(g^{x^{t+1}}) \leftarrow pk_B$

10 $(rk^{t+1}, ck_0^{t+1}) \leftarrow \mathtt{KDF}_2(rk^t, g^{x^t x^{t+1}})$

11 $(ck_1^{t+1}, mk_1^{t+1}) \leftarrow \mathtt{KDF}_3(ck_0^{t+1})$

12 $(rk, ck, mk) \leftarrow (rk^{t+1}, ck_1^{t+1}, mk_1^{t+1})$

13 $s \leftarrow s^{t,n} \setminus (rk^t, ck_n^t, mk_n^t)$

else if phase = symRatch **then**

14 $(rk^t, ck_n^t, mk_n^t) \leftarrow s^{t,n}$

15 $(ck_{n+1}^t, mk_{n+1}^t) \leftarrow \mathtt{KDF}_3(ck_n^t)$

16 $(rk, ck, mk) \leftarrow (rk^t, ck_{n+1}^t, mk_{n+1}^t)$

17 $s \leftarrow s^{t,n} \setminus (rk^t, ck_n^t, mk_n^t)$

18 $s^{t,n+1} \leftarrow s \cup (rk, ck, mk)$

19 **return** $s^{t,n+1}$

recipient then generate the initial session keys of the communication session based on the Triple Diffie-Hellman (X3DH). The three session keys are the root key rk, the chain key ck, and the message key mk. The chain key and the message key are derived from the root key, and the message key is used to encrypt the content of an envelope. After the session keys are generated, the sender then removes its private one-time prekey x_o, private base key x_b, and the first private ratchet key x^1 from its

state. The recipient similarly removes its private one-time prekey x_{o_B} from its state.

In the asymmetric ratcheting phase, for a sender executing this algorithm, the sender first retrieves the private ratchet key x^{t+1} for the new epoch $t+1$ as well as the root key rk^t for the current time epoch t from its state. The sender additionally retrieves its communicating partner's, i.e., the recipient B's public ratchet key g^{x^t} for the current time epoch from the recipient's public keys. In contrast, a recipient executing this algorithm first extracts its private ratchet key x^t and the root key rk^t for the current time epoch from its state. This is followed by the recipient extracting its communicating partner's, i.e., sender A's, public ratchet key $g^{x^{t+1}}$ for the new epoch. Both the sender and the recipient then generate the session keys for the new epoch $t+1$ based on the Double Ratchet algorithm, after which they subsequently remove the three session keys of the current time epoch t from their respective state.

For symmetric ratcheting, no new key material is introduced. A sender and a recipient both derive the new session keys the same way in this phase according to the Double Ratchet algorithm. The sender and the recipient derive the next chain key and message key for the new envelope $n+1$ by passing the chain key for the current envelope n into the KDF. The respective state of the sender and the recipient is then updated by removing the chain key and the message key for the current envelope n.

As a last step, the newly established or updated session keys are added into the state of both the sender and the recipient. The algorithm returns the updated state containing the new session keys as output.

2.7 Chapter Summary

In this chapter, we establish the cryptographic basics necessary to comprehend the concept of cryptographic protocols with the cryptographic primitives as its building blocks for the foundation of the work in this thesis. We further introduce the Diffie-Hellman key exchange protocol and the Dolev-Yao adversary model due to its prevalence in the design of cryptographic protocols, as well as advanced topics pertaining to cryptographic protocol design. We conclude this chapter with the

background of two real-world protocols: the SSH protocol and the Signal protocol that form the basis for the work of two chapters in the thesis.

3
Related Work

Contents

We first start off this chapter with a general review of the current machine authentication methods and the state-of-the-art machine-to-machine (M2M) communication schemes used in various applications. We realise that improvements can still be made to the current solutions to address the issue of credential leakage using computationally secure methods. We then explore more closely the related work in each of the specific M2M application domains that we have chosen as our research scope (listed in Section 1.1), namely, Single Sign-On (SSO) systems, the Secure Shell (SSH) protocol, and the Signal protocol.

3.1 M2M Authentication

In this section, we discuss the related research areas on the current machine authentication methods as well as the state-of-the-art M2M authentication protocols for various applications.

3.1.1 Machine Authentication Methods

While the authentication methods for a human have been well-established (what you know, what you have, what you are), this is not the case for the authentication methods for a machine. The authentication methods for a machine can be broadly categorised into three main families: secret-based, context-aware, and hardware-based. Traditionally the most common method to authenticate a machine is through its possession of a secret that comes in different forms. These include passwords, certificates (and the corresponding private keys) (Cooper et al., 2008; Campagna, 2013), Application Programming Interface (API) keys (Microsoft, 2022; Google Cloud, 2022; IBM, 2022; AWS, 2022; GitHub Docs, n.d.), and SSO access tokens (Steiner et al., 1988; Hardt, 2012; Sakimura et al., 2014; Lockhart and Campbell, 2008). Hardware-based authentication methods authenticate a machine by its responses to the challenges issued by a verifier based on a secret derived from the underlying hardware of the machine or an externally generated secret loaded into the secure hardware, e.g., through Trusted Module Platforms (TPMs) (Zhou and Zhenfeng Zhang, 2010; Khalil et al., 2015) or Physical Unclonable Functions (PUFs) (Braeken, 2018; Chatterjee et al., 2018; Rührmair, 2022). Although hardware-based can also be considered as secret-based, since they typically involve secrets stored within the secure hardware, the key distinction here is that the hardware-based family requires a dedicated hardware component on a machine, as opposed to a machine solely depending on the possession or knowledge of secret information without a secure hardware. Context-aware authentication methods (Loske et al., 2019) identify a machine based on the measurements of physical features of a machine's surroundings, such as geolocation (Denning and MacDoran, 1996; F. Zhang et al., 2012) and proximity (Brands and D. Chaum, 1994; K. B. Rasmussen and Capkun,

2010), which require making assumptions about the physical environment of a machine. Without making any assumption on a machine's physical environment and its underlying hardware, in this thesis we propose novel computationally secure solutions that strengthen a machine authentication method, enabling it to withstand credential compromises by an adversary.

3.1.2 M2M Communication Schemes

We then examine the widely deployed practical M2M authentication schemes designed for various application purposes to establish a foundational understanding for the work of the thesis. The industry standard for a client machine to authenticate a server over the web is the Transport Layer Security (TLS) protocol (Dierks and Rescorla, 2008). TLS ensures a client that it is connecting to a server which is the legitimate owner of its public key, before establishing a secure channel between the client and the server. For network administrators, the Secure Shell (SSH) protocol (Ylonen, 2006a) is the most common protocol to secure the messages exchanged during a remote login from a SSH client to a SSH server. Internet Key Exchange (IKE) protocol (Harkins and Carrel, 1998) is used in Internet Protocol Security (IPsec) for creating secure Virtual Private Networks (VPNs) between two machines. In instant messaging applications, the Signal protocol (Signal, 2023b) is the de facto standard to provide end-to-end encryption for user messages. Recently, M2M authentication protocols have been proposed (Esfahani et al., 2017; Thammarat and Techapanupreeda, 2021; Gao et al., 2020; Renuka et al., 2019; Lara et al., 2020; Hussen et al., 2013) specifically for resource-constrained devices in Internet-of-Things (IoT) networks that put heavy emphasis on the performance of the protocols. Although these M2M authentication schemes are designed for their specific use cases, the most prevalent authentication method used in these M2M communication schemes are secret-based, as discussed in the previous section. Thus, the security of these protocols rests on the secrecy of the cryptographic keys that the machines store. While this has been the convention for many protocols, in practice this

assumption might not always be fulfilled, especially if we consider a powerful real-world adversary that has access to the machines, as we explain later in Chapter 4. To also consider security definitions that account for this stronger adversary, the work in this thesis provides new countermeasures in the M2M application domains as set in our research scope in Section 1.1.

3.2 The Secure Shell (SSH) Protocol

We look more closely into the existing work on the SSH protocol in this section as the SSH protocol serves the basis of our work in Chapter 5. Existing research focuses on the second and reworked version of the Secure Shell Protocol (SSH-2). It proposes improvements to the protocol to mitigate brute-force attacks (Cao et al., 2019), chosen-plaintext attacks (Bellare, Kohno, et al., 2002; Bellare, Kohno, et al., 2004), attacks on the Trust-on-First-Use model (explained in Section 2.5.1) of server authentication (Wendlandt and Perrig, 2008; Ali and Smith, 2004), and also fixes the privacy issues of the protocol (Roy et al., 2022). Other work provides security analysis on the protocol (Williams, 2011; Albrecht, Degabriele, et al., 2016). There are also papers that discover attacks against the protocol (Song et al., 2001; Albrecht, Paterson, et al., 2009; Bhargavan and Leurent, 2016; Bäumer et al., 2024). A 2023 attack on SSH (Ryan et al., 2023) demonstrates that it is possible for a passive adversary that only monitors the protocol's messages to probabilistically derive the server's private key used to generate the server's signature sent in the clear during server authentication from seven years' worth of SSH traffic. The main reason for this appears to be stemming from the different classes of hardware failures in the vulnerable implementations, and the solution offered by the study is for the server to simply re-verify the signature after generation. As this is also caused by the fact that the signature key is static across all sessions, our solution in Chapter 5 solves this problem by having additional entropy added to the server signature, in addition to solving the SSH client credential leakage problem.

Surprisingly, the problem of SSH client credential leakage has received little attention in the extensive literature on SSH, even though this problem poses a serious

threat to most of the security guarantees of the SSH protocol. Harchol et al. (2018) and Plaga et al. (2018) propose solutions to prevent the leakage of a SSH client's keys. Harchol et al. (2018) provides a distributed key manager in enterprise-level SSH to mitigate this problem. Their methodology uses threshold cryptography that requires the introduction of multiple servers to the SSH architecture for the storage and management of the cryptographic materials required for client authentication. Plaga et al. (2018) introduces Hardware Security Modules (HSMs) for the storage of SSH client private keys to prevent an adversary's access to the client's keys.

In contrast, we focus on the security post-compromise of a SSH client's key in Chapter 5. We build our solution for this problem directly into the SSH protocol without requiring additional servers or parties to the SSH architecture and without specialised hardware such as a HSM. Our solution in Chapter 5 is thus suitable for a broader scope of applications, and not limited to just enterprise-level SSH systems.

3.3 Single Sign-On (SSO) Systems

Although not strictly a M2M authentication scheme, we explore Single Sign-On (SSO) systems here because the goals and architecture of SSO systems bear some resemblances to our work in Chapter 6. SSO systems go hand-in-hand with other user authentication mechanisms, and the attractiveness of SSO system stems from the fact that they reduce a user's reliance on having to manage multiple secrets for multiple application services. Specifically, SSO systems allow a user to access multiple servers using only a single password, and simultaneously without having to reveal a user's password to the servers. This is done through a trusted third party called the identity provider, whom a user has registered the password with. Kerberos (Steiner et al., 1988) is arguably the first deployed SSO system to secure network services in MIT. Next, a family of SSO systems, OAuth 2.0 Authorisation Framework (OAuth) (Hardt, 2012), OpenID Connect (OIDC) (Sakimura et al., 2014), and Security Assertion Markup Language 2 (SAML2) (Lockhart and Campbell, 2008) are deployed for cross-organisational user authentication and authorisation through web browsers. Various research proposals have since been made to enhance the

security and privacy of SSO systems. One interesting line of research (Agrawal et al., 2018; Baum et al., 2020; Rawat and Jhanwar, 2020; Y. Zhang et al., 2020) focuses on distributing the risk of a single identity provider compromise into a number of identity providers to prevent the identity provider from becoming a single point of failure. Another research area focuses on solving the privacy issues of user activity tracking by a curious identity provider or collusive servers. The proposed SSO schemes (Fett et al., 2015; Zhiyi Zhang et al., 2021; Guo et al., 2021; Han, Chen, Schneider, Treharne, and Wesemeyer, 2018; Han, Chen, Schneider, Treharne, Wesemeyer, and Wilson, 2019; Zhenfeng Zhang et al., 2020) provide user untraceability and unlinkability from the identity provider and servers. This shows that SSO systems have garnered much research attention and are still an active research area. However, where SSO systems excel in user authentication, our focus is on M2M communication in which there is no human user involved. We compare our proposed solution with existing SSO systems in more detail in Section 6.1.3.

3.4 The Signal Protocol

First proposed by Signal (2023b), the Signal protocol (or Signal) is based on the Extended Triple Diffie-Hellman (X3DH) key agreement protocol (Marlinspike and Perrin, 2016b) and the Double Ratchet Algorithm (Marlinspike and Perrin, 2016a) where it has since garnered a substantial interest from researchers. The analysis of Signal has first been formalised in Cohn-Gordon, Cremers, Dowling, et al. (2020), and Cohn-Gordon, Cremers, and Garratt (2016) coined the term *Post-Compromise Security* (PCS), which refers to the self-healing property of Signal. Due to its widespread practical uses, Signal has since attracted other research work (Canetti et al., 2022; Alwen et al., 2019; Bienstock et al., 2022; Blazy, Bossuat, et al., 2019; Blazy, Boureanu, et al., 2023). Our work in Chapter 7 is largely inspired by research focusing on key authentication and cloning attacks in Signal.

3.4.1 Signal Key Authentication

The security guarantees of Signal only hold if a pair of communicating users authenticate their key in a secure channel (Marlinspike and Perrin, 2017), which helps in detecting Man-in-the-Middle (MitM) attacks. The authentication method provided in Signal depends on the pair of users comparing the *Safety Numbers* or fingerprints of the identities of their devices out-of-band (Marlinspike, 2016), which are derived by hashing the identifiers and the long-term public keys of the devices, and this process of comparing the fingerprints out-of-band is known as an authentication ceremony (Ellison, 2007).

As the fingerprints are calculated on purely static information, Dowling and Hale (2021) considers an adversary that can manipulate the encoded fingerprints to perform an active MitM attack. Their proposed detection mechanism improves on Signal's derivation of the fingerprints from a hash function to a Message Authentication Code (MAC), and provides the additional guarantee that there is no MitM attack from session establishment up until the time epoch where the fingerprints are compared. These existing solutions build on the idea that key authentication requires some degree of active user intervention. While most messaging applications adopt this authentication ceremony, user studies (NEAL, 2017; Vaziripour, Wu, O'Neill, Whitehead, et al., 2017; Vaziripour, Wu, O'Neill, Metro, et al., 2018; Schröder et al., 2016) have shown that this presents usability and practicality problems as not more than 25% of participants are able to successfully complete the authentication ceremony.

WhatsApp (2023), which also uses the Signal protocol as the underlying messaging protocol, has recently published a new security feature called *Key Transparency* (Lawlor and Lewi, 2023) to tackle these usability issues by performing key authentication without user intervention; their solution uses an additional auditable key directory maintained by the server and a third-party audit record, as well as a separate verification process. WhatsApp's method is similar to DECIM (Yu et al., 2017), which proposes a solution to detect endpoint compromise in messaging involving an append-only log and a log maintainer that can then be queried by users

to detect key misuse. Malvai et al. (2023) improves on the scalability issues of Chase et al. (2019). Malvai et al. (2023) addresses the issue of ever-growing append-only logs by introducing the notion of 'compaction', and the issue of server equivocation, i.e., malicious servers not showing legitimate logs, by proposing a lightweight broadcast protocol to achieve consistency of the logs from the clients' view.

3.4.2 Cloning Attacks in Signal

Cloning attacks in the context of Signal have been studied in Cremers, Fairoze, et al. (2020); Cremers, Jacomme, et al. (2023); Dowling, Günther, et al. (2022); Barooti et al. (2023). The solution proposed by Cremers, Fairoze, et al. (2020) embeds an increasing counter into the protocol as part of the associated data for each envelope sent, whereas Cremers, Jacomme, et al. (2023) improves Signal's session management scheme to detect a cloned device by checking if envelopes are encrypted with a key from a session not recognised by an honest device. Both solutions assume a clone operating only through the interface of the messaging applications and not having direct access to the keys.

To reduce the reliance on out-of-band authentication, Dowling, Günther, et al. (2022) proposes a clone detection mechanism that reintroduces the long-term identity keys of clients to generate signatures on past envelopes. The solution assumes an active MitM adversary that has access to the state of the clients, including the long-term keys, but still requires an out-of-band channel for users to detect the compromise of the long-term keys. As the solution proposed by Dowling, Günther, et al. (2022) requires three rounds of in-band communication before an out-of-band detection can take place, Barooti et al. (2023) generalises Signal into a ratcheted communication protocol and associates each ciphertext sent in the protocol with an ordered indexing called an ordinal. Their solution detects a message forgery by an adversary that can control the communication channels and expose the state of a device (similar to our adversary model). By attaching the history of previously sent and received ciphertexts' ordinals (and hashes) to a subsequent ciphertext, the

receiver of a ciphertext can check if it has received an envelope forgery by comparing its view of the ordinals history with the received set of ordinals.

While Barooti et al. (2023); Cremers, Fairoze, et al. (2020); Cremers, Jacomme, et al. (2023) require a client to keep track of a different state for each of its communication peers, our proposed solution in Chapter 7 only needs a client to keep track of a single hash regardless of its number of communication peers. Our solution also detects a much more powerful MitM adversary than Cremers, Fairoze, et al. (2020); Cremers, Jacomme, et al. (2023); Dowling, Günther, et al. (2022) that not only has full control of the communication channels but also a one-time direct access to all of a client's secret keys, including its long-term identity keys.

3.5 Chapter Summary

In this chapter, we presented the existing work related to the systems that will be covered later in the thesis. We start with the general landscape of the research related to machine authentication methods and M2M communication schemes used in various applications. We also reviewed the extensive literature on the Secure Shell (SSH) protocol since its inception, as related to our work in Chapter 5. We then proceed to the existing body of research on Single Sign-On (SSO) systems that inspires the design for our solution in Chapter 6. Lastly, we present the research work on the Signal protocol, with a focus on the areas on Signal's key authentication and cloning attacks on Signal, to position our work in Chapter 7.

4

Adversary Model, Design Goals, and Methodology

Contents

In this chapter we describe the overarching theme of the research work presented across the subsequent chapters of this thesis. We begin with the description of our stronger adversary model used consistently (but with application-specific differences) across the subsequent chapters. This is followed by the design goals which our solutions aim to accomplish with respect to the given stronger adversary model. Finally, we provide an overview of the techniques used in our research methodology to achieve the aforementioned design goals with respect to our adversary model.

4.1 A Stronger Adversary Model

As highlighted earlier in the thesis, the Dolev-Yao model (Dolev and Yao, 1983) has been the standard adversary model considered for the transmission of messages.

However to capture the additional ability of the adversary compromising all secrets stored on a device, the Dolev-Yao model is not sufficient and thus in this thesis we introduce a novel adversary model. In the later chapters we apply this adversary model extensively, save for the application-specific differences of each chapter.

We note that an adversary model involving the compromise of a cryptographic key itself has already been considered by previous works. As an example, the security guarantee of perfect forward secrecy (PFS) considers an adversary that compromises long-term keys but not ephemeral keys (Blake-Wilson et al., 1997). Other works (LaMacchia et al., 2007; Krawczyk, 2005; Cremers and Feltz, 2012) improve on the notion of PFS to allow the additional compromise of ephemeral keys of a party but not both simultaneously in the same session. Meanwhile, key-compromise impersonation attacks (Blake-Wilson et al., 1997) are concerned with an adversary that impersonates other parties to the compromised party. Additionally, Post-compromise security (PCS), as we have discussed in Section 2.4.4, allows the healing of the communication channel after a compromise but requires the adversary to be passive temporarily.

We differentiate the work in this thesis from existing adversary models by taking into consideration an adversary that not only compromises all secrets of a party at a time (as opposed to only partial keys at a specific time in PFS), but also impersonates the compromised party to other participants in the system (as opposed to impersonating other honest participants in the system to the compromised party in key-compromise impersonation attacks). We further improve on the adversary model of the SSH protocol so that it considers a more powerful adversary that compromises all keys on a client, and at the same time we address this issue by providing the novel PCS guarantee for the SSH protocol (in Chapter 5). We also enhance the PCS guarantee in the Signal protocol (in Chapter 7) so that it addresses an active adversary and not just a passive adversary as previously required for PCS. This is in addition to the adversary's ability to control the communication channels from the Dolev-Yao model. At first glance this may seem trivial, since from a computationally secure perspective, an adversary that has exactly the same secrets

and ability (e.g., generating, sending, and receiving messages on the communication channel) as Alice is essentially indistinguishable from Alice. However, we show that new security guarantees can be achieved due to our design goal of not preventing the adversary from impersonating the compromised party, but instead detecting the presence of such adversary (more on this in Section 4.2).

4.1.1 Motivation for a Stronger Adversary Model

We have previously provided examples in Chapter 1 where secrets have been stored locally in the clear in machine-to-machine (M2M) communication. We have mentioned this presents a security risk against a determined adversary. In this section, we elaborate on what it means to be 'a determined adversary'. We look ahead on the abilities that we have given to the adversary in subsequent chapters and we give real-world examples to motivate why our adversary model does not rely on unrealistically strong assumptions, but rather on pragmatic considerations of adversaries for practical M2M applications. These real-world examples serve as the inspiration for our work in hardening the security of practical M2M protcols against these stronger adversaries.

Supply Chain Attacks

By 'a determined adversary', we refer to an adversary that can access the aforementioned secrets that are stored locally in the clear, which is a common theme across the adversary models in the next three chapters of this thesis. This does not necessarily refer to the adversary having physical access to the machine, since there are instances in the wild where an adversary is able to access the device without being physically in proximity of the machine. One such instance is through supply chain attacks. A recent supply chain attack on Microsoft (Poireault, 2023) had allowed the adversary to get hold of a data dump of Microsoft's signing key, which subsequently gave the adversary access to the Exchange Online and Azure Active Directory accounts of 25 organisations. The stolen signing keys of the company D-link has also allowed the adversary to sign and pass off the Plead

malware consisting of a backdoor and a password-stealer to its customers, which further allows the adversary to compromise user credentials (Goodin, 2018). The infamous Stuxnet worm that sabotaged Iran's nuclear programme involves the stolen signing keys corresponding to the certificates from the companies Realtek Semiconductor and JMicron technology, which allowed the malware to appear legitimate and bypass detection (R. Abrams, 2010).

Human Negligence

An adversary does not necessarily have to resort to supply chain attacks in order to gain access to credentials, as we have seen time and time again that the secrets can be unintentionally leaked through human errors due to the negligence of the users or developers in key mismanagement in the development environments themselves. A perfect example here is the bad practice of developers in hardcoding credentials into the source code of an application, and even renowned companies such as Samsung (Kovacs, 2022) and Toyota (McDaniel, 2022) are not immunune to this form of development bad practice. This security risk due to the mishandling of credentials also extends to secrets that are locally stored in environment variables of the operating system, which has already exploited by a real-world adversary (Engelberg, 2021). The data breach at Uber (Kost, 2024) has additionally demonstrated that not only a threat actor can easily purchase targeted credentials for employees of a certain organisation, but also when the credential compromise is coupled with Multi-Factor Authentication (MFA) fatigue attacks, the threat actor can easily obtain legitimate access by compromising the weakest link in the security chain - a human.

Physical Attacks

The examples that we have discussed thus far do not consider the adversary having physical access to the machine to compromise the machine's secrets. More often than not, physical attacks are notably more common than expected. Take, for example, government authorities, particularly border check officers, that have the legal power to confiscate electronic devices temporarily. This statutory mandate for search and seizure of electronic devices is not unique to a single jurisdiction, since it has been

exercised by many countries such as the US (Fisher, 2020), the UK (Toor, 2013), and Australia (Thomas, 2018) where in some cases the local data are authorised to be downloaded by the authorities (Toor, 2013). Not only does this expose users' private data, but it also provides a means for the secrecy of the cryptographic keys stored on the device to be broken, especially if the keys are stored locally in the clear on the device as we have previously established in Chapter 1. Even if the authorities themselves do not have direct access to the devices, there have been reports where tourists and residents have been coerced to install government spyware on their devices (Cox, 2019; Cox, 2018). On a smaller scale, physical attacks are also not entirely impossible if we consider lunchtime attacks by insiders.

Other Assumptions on the Adversary

Besides compromising all secrets of a machine, we look ahead on some of the other assumptions that we have made regarding the adversary in the subsequent chapters. Note that the adversary models differ across the later chapters and they are tailored to the specific M2M application domain in each chapter. We examine the commonalities among them here.

We additionally assume that during the process of compromising the cryptographic keys by the adversary, or also referred to the process of *cloning* in Chapter 7, the adversary does not modify the existing secrets or data stored on the machine, and that the adversary's impersonation after key compromise is operated on a physically separate machine from the victim machine. That is to say the adversary has a read-only access to the secrets of the machine. This is because we assume that the adversary demonstrates a level of intelligence and prudence to avoid actions that could arouse a user's suspicion and reveal the presence of an adversary. For example, changing the keys or any data on a user's device after an adversary has access to the device would raise the suspicion of a user on the presence of the adversary if the user checks that the modified secret is different from a backup or a logged copy of the secret.

Some security guarantees in the solutions presented in Chapter 5 and Chapter 7 operate under the assumptions that the adversary is passive, i.e., does not tamper with the messages of the protocol for some period of time. This assumption is also at the heart of the PCS guarantee that allows the broken security of the communication channel to heal (Cohn-Gordon, Cremers, and Garratt, 2016). In practice, this assumption reflects realistic conditions of real-world communication channels and is very likely to hold true in practical applications. As an illustration, data packets can be re-routed to a different path depending on a network's condition and an adversary listening on a specific path might miss the rerouted packets. Even if we consider an adversary that is on the same wireless local area network with a targeted device, the race condition that happens between the adversary and a legitimate access point makes it possible that some packets reaches the access point first before reaching the adversary, thus resulting in the adversary failing to capture a packet.

Lastly, in Chapter 5 and Chapter 7 we state that a machine's secrets can only be compromised once by the adversary. From what we have discussed so far in this section on the various methods that an adversary can gain access to the secrets of a machine in real-world scenarios, it seems as if the adversary has ample opportunities to compromise the secrets. This assumption is made to simplify the security analysis in the subsequent chapters, and we discuss in Section 8.1 that with some caveats, the security guarantees of our solutions still hold in spite of multiple compromises of the secrets of the same machine.

4.2 Design Goals

Once the specifics of our adversary model are clearly defined, we proceed to outline the design objectives that our solutions must achieve to effectively mitigate the identified threats posed by the adversary. This section details the primary design objectives that the solutions presented in the subsequent chapters aim to fulfill. We emphasise that like the adversary model, the design goals in each of the later chapters are distinct to the respective M2M application domain of the chapter. We provide a holistic examination of the shared characteristics of the design goals in this section.

We mentioned in Chapter 1 that our research objective is to achieve security guarantees for a compromised machine even when the adversary has both the control of the machine's secrets and the communication channels. We have also briefly touched on in Section 4.1 on how preventing the adversary from impersonating the compromised machine is trivially impossible from a computationally secure perspective when the adversary has access to the same secrets and the same communication channel as the compromised machine. In order to achieve our research objective meaningfully, we define our design goals as not on the prevention of the adversary impersonating the compromised machine, but instead we shift the focus of our goals to the detection of the adversary's presence. This does not imply that the detection mechanism must necessarily differentiate between the authentication of the compromised machine and that of the adversary. Rather, our design goals are oriented towards, at a minimum, to be capable of identifying the presence of an impersonation activity perpetrated by the adversary.

Recall from Chapter 3 that the security of M2M secure communication protocols used in practice rely on the secrecy of the secret keys. While this has been the prevailing assumption and one that is widely regarded as conventional, we stress that this is not compatible with our stronger adversary model. Also recall from Chapter 3 that the proposed alternative authentication methods to reduce this reliance require assumptions on the underlying physical environment and hardware of a machine. Consequently, our design goals are to aim for identifying solutions that are exclusively computationally secure, and those that are agnostic to the constraints in the underlying physical environment or hardware assumptions of a machine. The reason behind this is to allow for a broader applicability across various systems as these solutions can be implemented purely through software, such as via an update patch, ensuring the ease of deployment and flexibility for real-world applications.

Last but not least, we strive to have the design of our solutions to exclude user interaction. Not only do we not wish to reintroduce the involvement of the users that would render M2M communication unusable in practice, we have also indicated earlier in this chapter in Section 4.1.1 that human negligence is one of the primary

factors contributing to real-world key compromise. This also includes eliminating the need for an out-of-band communication channel, as this usually means that some degree of verification is required from the users on the out-of-band channel.

4.3 Methodology and Techniques

Now that we are familiar with the adversary model and the design goals, in this section we present a comprehensive overview of the intuitive techniques that we employ in our solutions to effectively counter the adversaries, while also ensuring the realisation of our design goals.

As our new adversary model now assumes that the adversary possesses the same secret state as a compromised machine and its corresponding abilities, we know that we cannot depend solely on the secret state to be able to detect the impersonation by the adversary. Naturally this necessitates us to look for information that can complement the existing status quo of the secret keys of a machine, ensuring that the machine's identity is not exclusively dependent on the keys. It is important to highlight that the adversary will still be able to access this new form of a machine identity. However, if we make the characteristics of the new identity to be dynamic and if we subject the identity to constant updates, with each previous iteration being overwritten on every access, then neither the adversary nor the machine would retain access to the prior identity before the adversary has the initial access to the machine. This approach offers a potential solution for identifying a 'difference in knowledge', which could be used to detect instances of impersonation. While this is an existing cryptographic technique, we employ this technique to improve the existing real-world protocols in Chapter 5 and Chapter 7 to gain novel security guarantees, thereby mitigating the credential compromise problem.

We additionally exploit some of the weaknesses of the adversary as laid out in Section 4.1.1. Remember from Section 4.1.1 that, after gaining initial access to the machine, the adversary utilises the acquired secrets on a different machine to prevent raising suspicions from the user or owner of the compromised machine. Thus, any new information that the machine generates on each future service access

is also inaccessible to the adversary, as the adversary does not have the ability to predict or retrieve this information without direct access to the machine. Looking forward to the solutions presented in the subsequent chapters (Chapter 5, Chapter 6, Chapter 7), we show how this property of the adversary can be leveraged to give rise to novel and stronger security guarantees. As another example, we can take advantage of the fact that the adversary might be passive for some period of time as mentioned in Section 4.1.1. This allows the preservation of the PCS guarantee (Cohn-Gordon, Cremers, Dowling, et al., 2020) that results in the self-healing of the communication channel after the compromise of the machine secrets, as presented in the solutions in Chapter 5 and Chapter 7.

4.4 Chapter Summary

This chapter offers a more comprehensive description of the fundamental aspects that underpin our solutions in the upcoming chapters. We have looked at a stronger adversary model that involves the adversarial access to a machine and subsequently the compromise of the machine's secrets, as well as the real-world examples and scenarios to motivate the stronger adversary model. This chapter also provides the design goals that are intended for our solutions presented in this thesis, including the focus on impersonation detection rather than prevention, with respect to our stronger adversary model. We then discussed an overview on the intuition behind the methodologies that comprises the techniques that our solutions deploy, such as using dynamic information to complement a secret key and using the weaknesses of the adversary, in order to attain the aforementioned design goals.

5

Post-Compromise Security for the Secure Shell (SSH) Protocol

Contents

The Secure Shell (SSH) protocol was first proposed in 1995 as a secure alternative to remote login tools like telnet. SSH was redesigned in 2006 as SSH2 which to this day remains one of the most important system administration tools for servers all over the world. In this chapter we will use SSH to refer to the latest

version as described by RFC4251 to RFC4254 (Ylonen, 2006a; Ylonen, 2006b; Ylonen, 2006c; Ylonen, 2006d).

SSH consists of three protocols, one to establish a secure session based on a Diffie-Hellman exchange and authenticate the server; one to authenticate the client; and one to handle the subsequent secure communication using the session key from the first protocol. The server uses a signature to authenticate itself, but the client has multiple options for authentication, including the use of passwords or asymmetric cryptography. In either case, the risk of credential leakage is a serious threat. An adversary that obtains the authentication credentials of a client, can impersonate that client indefinitely as the authentication keys (or passwords) never need to be changed.

The impact of such a compromise is even greater than in many other communication protocols, as the credentials will give the attacker (root) access to the server. With such access the attacker can read and modify any data on the server, on the hard-drives or in memory. This includes stealing the server's long-term key material and deleting any evidence that the attack happened. Such an attack is simultaneously impactful and undetectable, and there are countless examples of this happening to both companies and private individuals over the last 20+ years.

In this work we reexamine the SSH protocol and modify it to be secure in a much stronger adversary model and we apply the notion of post-compromise security that has been successfully used in protocols such as Signal (Cohn-Gordon, Cremers, Dowling, et al., 2020). Post-compromise security allows us to recover from credential compromise if the credentials have not been used by the adversary, and to detect the compromise otherwise. We cannot however rely entirely on the existing definition of post-compromise security, since in this case the adversary has the ability to modify the server's state after compromising the client.

We present a series of modifications to the SSH protocol such that new sessions are able to "heal" the communication channel after a compromise, assuming the adversary remains passive. This is a huge improvement in the practical security of SSH since leaked credentials would lose their value after the normal client has

started a new session. If an adversary does manage to use the leaked credentials before the legitimate user, our new protocol ensures that the user discovers this next time he/she tries to log in. Achieving this detection is not trivial in our threat model, since the adversary can arbitrarily modify the server after logging in with the stolen credentials. However by updating a ratchet key stored on the server as part of the login process we can ensure that the attacker cannot obtain the information he needs to hide the login from the client, or impersonate the server.

We prove the novel security properties for our construction, such as Post-Impersonation Detection and Post-Compromise Security. Simultaneously, we prove that the existing guarantees of the SSH protocol, i.e., Mutual Entity Authentication, Perfect Foward Secrecy, and Session Data Confidentiality and Authentication, are preserved. Following a recent cryptanalysis attack where SSH signatures collected over a period of time are used to derive the private key (Ryan et al., 2023), we additionally limit the exposure of the signatures used during the key-establishment phase of our protocol. This results in an additional guarantee we call Reduced Signature Exposure.

Finally we present an implementation (in C) of our protocol, built on top of OpenSSH. Despite a significantly improved level of security our implementation has a marginal overhead in the order of milliseconds of less than 5 ms (less than 55% increase), compared to the normal SSH protocol.

In summary, we present the following contributions:

- A new strong threat model that better reflects the wide range of attacks that the SSH protocol is subjected to in practice.
- A new version of the SSH Transport and Authentication protocols that incorporate post-compromise security notions into SSH.
- Security proofs of our three new guarantees that in addition to confidentiality and authentication provided by the original SSH protocol, also provides post-impersonation detection.

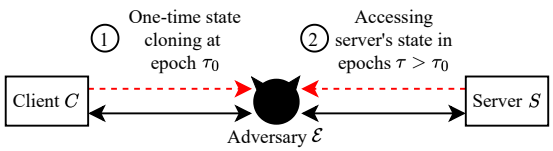


Figure 5.1: Our system and adversary model. The system is made up of a client C and a server S. An adversary $\mathcal{E}$ has access to the communication channel and acts as a man-in-the-middle. We allow the adversary to have the additional ability (as indicated by the red dotted arrow) to clone the state of the client once and gets access to the secrets of the client in epoch τ_0. With the client's secrets, the adversary can access the server's state in epochs τ_i in a SSH session.

- An implementation of our protocol (in C) based on OpenSSH that allows us to characterise the performance overhead of our protocol in a real network environment, and allow others to use, and build upon our proposal.

5.1 System and Adversary Model

In this section we detail the assumptions on the participants in the system and the adversary as depicted in Figure 5.1. In the system model, we describe the details of the participants and in the adversary model, we describe the adversary's capabilities as well as its goals. Parts of the work in this section are a joint work with David Soler Garcia, a collaborator from University of A Coruña.

5.1.1 System Model

Our underlying system model is identical to the SSH Protocol architecture. As shown in Figure 5.1, the system model consists of a client and a server that communicates via the SSH protocol. We assume that the server is always online, and the client is only online when it is communicating with the server.

Each completion of a protocol execution between a client and a server will be referred to as an *epoch*. We assume that the protocol executions for SSH Transport Layer Protocol (Figure 5.3) and SSH Authentication Protocol (Figure 5.4) are sequential, meaning that a new epoch cannot start until the previous epoch has finished. However, SSH sessions can still run concurrently after the keys for each independent session have been established, i.e., after the completions of protocol

executions from different epochs. Per-user (or in our case, per-client) sequential execution of key exchange can be desirable for situations such as brute-force and Denial-of-Service (DoS) attacks mitigation, audit and session management, as well as prevention of race conditions in authentication involving the same username. This also means that the epochs are represented as non-negative integers incremented in a strictly increasing way starting with the first epoch 0.

We also assume that the client keeps a local mapping of the server-host key association following the TOFU model and that the server also maintains the associations of the clients' public keys in the system as configured by a SSH administration prior to a client's initial login to the server. We further assume that the deletion of keys can be done securely by the client and the server as required by our modified SSH protocol and also the original protocol.

5.1.2 Adversary Model

We assume an adversary $\mathcal{E}$ has the ability to control the communication channel between the client and the server, and act as a Dolev-Yao adversary on the channel. This means that the adversary has the ability to modify, drop, and replay the protocol messages on the channel. The underlying cryptographic primitives are assumed to be secure. We assume that honest parties faithfully execute the operations of the protocol, while the adversary can arbitrarily deviate from the protocol.

We additionally give the adversary the ability of cloning the state of the client, which we will call a *corruption*, and gaining access to all secrets that the client has stored in its state. While the nature of the long-term secret depends on the authentication method (e.g., password, public key), in this work we assume that the adversary has access to all the private information stored on a client as required by the SSH protocol and this includes cryptographic keys and potentially passwords. We refer to Section 5.3 for further explanation on the secrets revealed by the corruption. The epoch where the adversary cloned the client's secrets is denoted τ_0. The adversary can use the cloned secrets to log into the server impersonating the client in a future epoch for an arbitrary number of times starting from the

next immediate epoch τ_1 after τ_0, i.e., $\tau_1 = \tau_0 + 1$. We denote τ_N as the epoch where the adversary last logs into the server before the client first logs into the server at $\tau_N + 1$ since τ_0, i.e., $\tau_N + 1 > \tau_N > \tau_0$.

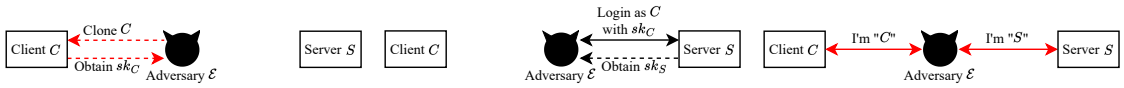
The adversary has two *Goals*:

- **Eavesdrop**: Being able to decrypt the encrypted data exchanged during a SSH session outside of the initial epoch of compromise τ_0 and outside the epochs following τ_0 where the adversary logs into the server impersonating the client. More precisely, this goal is applicable only to epochs t where $t < \tau_0$ and $t > \tau_N$.

- **Stealthy Impersonation**: Successfully logging into the server using the client's identity without being detected in a future epoch, or impersonating the server to the client without being detected.

The cloning ability of the adversary can result in trivial attacks that must be ruled out in order to have a meaningful model. For example, if the adversary clones a client it will then gain access to the information exchanged in the current epoch (epoch τ_0), but this should not be considered as accomplishing the adversary's *Goal* of **Eavesdrop**. Thus, we consider that the adversary can only accomplish their *Goals* in epochs that have not been compromised, while allowing corruptions to be executed to previous and future epochs. This corresponds to the Post-Compromise Security (PCS) and Perfect Forward Secrecy guarantees.

As mentioned, the adversary can also attempt to inject messages into the communication. This can lead to an impersonation, in which the adversary acts as the client and successfully initiates a SSH session with the server. The implications of an impersonation in a SSH scenario are more significant than in other Authenticated Key Exchange (AKE) protocols, as this would grant a shell to the adversary that allows them to access the state of the server. In order to provide the stronger security guarantees possible, we assume the adversary has root permission and thus can arbitrarily read or modify the state of the server. We note that this represents a strengthening over other PCS models, as it assumes a stronger type of adversary.



(a) The adversary $\mathcal{E}$ compromises a client and steals all the client's secrets, including the client's private key sk_C used for authentication.

(b) The adversary employs the stolen key from the client to create a legitimate SSH session with the server. Then, the adversary steals the server's private key. Notice that this is done using normal SSH operations after the adversary logs in as the client.

(c) With both private keys, the adversary can launch a Man-in-the-Middle attack without getting detected.

Figure 5.2: Stages of a stealthy MitM attack in the original SSH protocol by an adversary $\mathcal{E}$ as described by our adversary model. Note that a client compromise is sufficient to deploy the attack.

The additional capabilities allow the adversary to impersonate both the client and the server by requiring only one compromise to the client. Furthermore, they allow the adversary to prevent the attack from being detected. We provide an example by showing how the original SSH protocol is insecure under this new adversary model, as an undetectable Man-in-the-Middle (MitM) attack can occur. In Figure 5.2a the adversary compromises the client, allowing it to perform an impersonation in Figure 5.2b to create a SSH session with the server. This also allows the adversary to retrieve the server's private signature key. With this information, the adversary can perform a MitM attack as in Figure 5.2c. The adversary can further evade detection by modifying the server's internal state, for example, altering the persisted logs such as the client's last login time on the server, such that the following session between the client and the server will be accepted, allowing the attack to remain undetected.

We remark that the SSH protocols with their existing security guarantees are only concerned with an adversary that can only manipulate messages on the communication channels. Observe that this is equivalent to our adversary if our adversary does not use its cloning ability.

5.2 Protocol Description

We describe the full details of the SSH protocols with our changes to the protocols as our solution in this section. We propose our solution in the first two subprotocols

of the SSH protocol architecture, i.e., the SSH Transport Layer Protocol and the SSH Authentication Protocol. For each protocol we provide a message sequence chart, where we highlight our changes to the protocols in blue. We describe the steps that are unchanged in the protocols based on the respective RFC documents of the protocol, namely, RFC 4253 (Ylonen, 2006c) for the SSH Transport Layer Protocol, and RFC 4252 (Ylonen, 2006b) for the SSH User Authentication Protocol. Parts of the work in this section are a joint work with David Soler Garcia, a collaborator from University of A Coruña.

5.2.1 SSH Transport Layer Protocol

Recall from Section 2.5 that the SSH Transport Layer Protocol is the first protocol run by the client and the server to establish a secure channel during a login. This is achieved by establishing a fresh Diffie-Hellman shared key that is in turn needed to derive the six application keys used later in subsequent protocols. In order to achieve our novel security properties, we change the way that this shared key is derived by introducing the notion of a ratchet key and the concept of an epoch. On a high-level, a ratchet key is a symmetric key that is unique to a pair of client and server in a single epoch. We say that a new epoch is entered every time the same pair of the client and the server has successfully completed one instance of the SSH protocol (both the SSH Transport Layer Protocol and the subsequent SSH Authentication Protocol) until the ratchet key is updated in the persistent storage of the client and the server. Our changes begin in the key exchange phase of the protocol after the algorithm negotiation.

Version Identification

As depicted by Figure 5.3, the SSH Transport Layer Protocol begins with version identification for the client and the server. The version identification is for the client and the server to identify and agree with the protocol version and the SSH implementation version. This is done by exchanging the identification strings I_C and

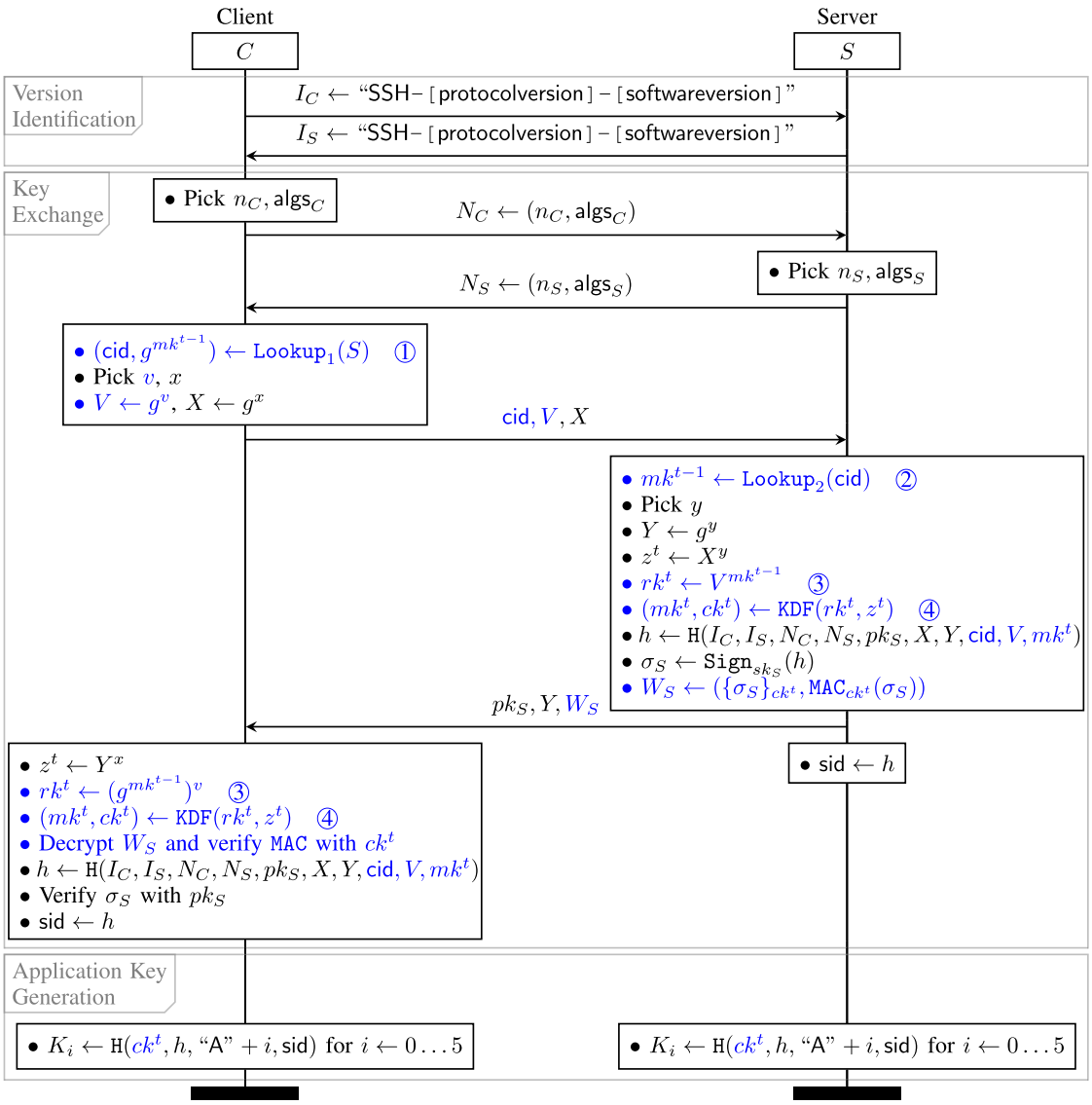


Figure 5.3: The SSH Transport Layer Protocol incorporated with our changes highlighted in blue and ',' is used for concatenation. `Lookup` is an algorithm that returns existing information from the persistent storage based on the input. In the first epoch, i.e., $t = 0$, ①: The lookup returns a null string for both cid and $g^{mk^{t-1}}$ to indicate the client has no previous records of S. The client picks a random identifier cid; ②: the lookup returns a null string to indicate there is no existing mk^{t-1} in the server's record; ③: this step is skipped; ④: rk^t is replaced with a constant.

I_S, which are literal strings that contain the protocol version (`protocolversion`) and also the software version (`softwareversion`) for the SSH implementation.

Key Exchange: Algorithm Negotiation

The key exchange of the protocol starts immediately after version identification with the client and the server sending the initial messages of the key exchange, N_C and N_S, respectively. The initial messages contain a random nonce (known as a cookie in the RFC (Ylonen, 2006c)) and the namelists of categories of the preferred and supported algorithms of each side. These categories are the key exchange algorithms, the server host key algorithms, the encryption ciphers, message authentication code (MAC) algorithms, and compression algorithms. The encryption ciphers, MAC algorithms, and compression algorithms are further divided into two categories for messages sent in each direction: from the client to the server, and the server to the client. Note that in the RFC, the key exchange initial messages are labelled as `SSH_MSG_KEXINIT`. We refer to the RFC of this protocol (Ylonen, 2006c) for further information on the algorithm negotiation of the key exchange.

Modified Key Exchange: Key Derivation and Server Authentication

The key exchange of the protocol proceeds with the client and the server running the negotiated key exchange algorithm and this is where our changes are applied. Although there exist RFCs for other key exchange algorithms for the SSH Transport Layer Protocol such as the RSA algorithm (Harris, 2006) and the elliptic curve algorithm (Stebila and Green, 2009), in this chapter we focus only on Diffie-Hellman and its variations. Notice that our changes to the protocol described in the following can still be applied analogously to other Diffie-Hellman-like key exchange algorithms. We put less emphasis on the RSA key exchange algorithms as they have been deprecated in 2022 by Baushke (2022).

Our changes are applied at the beginning of the negotiated key exchange algorithm. Here, the steps in the key exchange depend on the epoch when the protocol is executed by the client and the server, denoted by t in Figure 5.3. Recall that an epoch refers to the number of times that the SSH protocols (Transport Layer Protocol and Authentication Protocol) are completed by the same pair of the client and the server. If the protocol execution is not in the first epoch (i.e.,

$t > 0$), this means that the client has executed the protocol before with the same server. In this case the client retrieves the masked master key $g^{mk^{t-1}}$ (details in Section 5.2.2) for the server and the identifier cid for the masked master key that the client had stored in the previous execution of the protocol for the same server at epoch $t-1$. The client then proceeds to generate the two Diffie-Hellman keypairs, (x, X) and (v, V). Otherwise if this is the first epoch ($t = 0$) for the client and the server, the client picks a random identifier cid and proceeds to the Diffie-Hellman key pair generation. The client then sends the message that contains the two fresh public Diffie-Hellman keyshares with the identifier cid.

After the server receives the client's message, the server retrieves the master key mk^{t-1} from the previous epoch $t-1$ based on the received identifier cid. The current epoch of the protocol is $t = 0$ if the server does not have a record of cid. The server then proceeds to generate its own fresh Diffie-Hellman keypair (y, Y) and the corresponding shared secret z^t using the client's public key share X. If the current epoch is not the first epoch, i.e., $t > 0$, the server derives another Diffie-Hellman shared secret that we call the ratchet key rk^t for the current new epoch t by using the client's public key share V and the retrieved master key mk^{t-1} as the exponent. Two symmetric keys: the new master key mk^t and the cipher key ck^t for the new epoch t are then derived from the shared secret z^t and the ratchet key. If instead the epoch is $t = 0$, the ratchet key is simply replaced with a constant. The role of the master key is to detect if there has been an impersonation attempt by the adversary whereas the cipher key is to use to derive the actual keys used in the ciphers of the protocol. The server subsequently generates the exchange hash h by hashing together the public information that has been exchanged so far with the client along with the master key. We also include the random identifier cid and the additional client's public Diffie-Hellman keyshare V into the exchange hash as part of our solution. A signature σ_S is generated by the server on the exchange hash using the server's private host key sk_S. A MAC is then generated on the signature and the signature is encrypted. The encryption and MAC operations can be done either by deriving two independent keys, one for each operation, from the

cipher key, or using an AEAD encryption scheme that directly employs the cipher key as outlined in Section 2.1.3. As the last message of the protocol, the server sends its public host key pk_S corresponding to the private key used to generate the signature, the server's fresh public Diffie-Hellman keyshare Y, and the encrypted signature with its MAC. While the operation of wrapping the signature of the server might seem unorthodox, the motivation for this operation is to introduce additional entropy to the signature generated by the server's long-term host key, and to reduce the exposure of the signature every time it is sent in the clear when the protocol is executed during SSH logins. This in turn enables us to reduce the materials for advanced cryptanalysis even by an adversary that only monitors the traffic, as shown by a recent attack (Ryan et al., 2023).

Once received, the client similarly generates the fresh Diffie-Hellman shared secret z^t, and derives the master key and the cipher key accordingly based on the epoch of the protocol. With the cipher key, the client can now decrypt the server's wrapped signature and verify the corresponding MAC. If the verification is successful, the client has the assurance that the server has the master key and cipher key synchronised with the client, providing the guarantee that the server is the same server from past epochs. The client then proceeds to similarly generate the exchange hash and verify the server's signature using the server's public host key. If the signature verification succeeds, the server is authenticated by the client. The session identifier sid is then assigned the value of the exchange hash.

The rekeying process in the original SSH protocol is where the client and the server renegotiates the application keys triggered after a configurable threshold of time or data with the default being 1 hour or 1 GB respectively (Ylonen, 2006c). This is to maintain forward secrecy and mitigate cryptanalysis during prolonged sessions. During rekeying, the key exchange part of the original SSH Transport Layer Protocol in Figure 5.3 is repeated. The only exception is that the session identifier stays constant even when the value of the exchange hash changes. The same applies to our modified SSH Transport Layer Protocol during rekeying.

Modified Application Key Generation

The last step of the SSH Transport Layer Protocol for both the client and the server is the generation of the six application keys as represented by the K_i's for $i = 0 \ldots 5$ used for encrypting and producing the authentication tag of the subsequent messages in the protocol. These application keys are derived by hashing the new cipher key ck^t of the current new epoch, the exchange hash, and the session identifier. Specifically each K_i application key is derived using the formula $\mathtt{H}(ck^t, h, \text{“}\mathtt{A}\text{”} + i, \mathsf{sid})$, where $\mathtt{H}$ refers to a hash function and “$\mathtt{A}$” $+ i$ corresponds to the ASCII sequence from $\mathtt{A}$ to $\mathtt{G}$. The first two application keys, K_0 and K_1 are the initial initialisation vectors for the subsequent messages in the two directions: from the client to the server and from the server to the client respectively. The next two application keys, K_2 and K_3, are respectively used as the encryption keys for messages from the client to the server and from the server to the client. The last two application keys K_4 and K_5 are then the respective message authentication keys for messages from the client to the server and also the server to the client. In the original protocol the fresh Diffie-Hellman shared secret z^t is used in place of the cipher key.

5.2.2 SSH Authentication Protocol

Once the secure channel has been established from the SSH Transport Layer Protocol, the server authenticates the client through the SSH Authentication Protocol. The RFC for the protocol (Ylonen, 2006b) documents three authentication methods: public key, password, and host-based. The RFC (Ylonen, 2006b) states that all implementations of the protocol must support the public key authentication method and so we focus only on the public key authentication method in this section based on the details from the RFC. We further discuss the other two authentication methods in Section 5.5.

The public key authentication protocol starts with the negotiation of the public key algorithm and also the client's public key. The client sends the server the request message R_C containing: the username, the name of the public key algorithm, and the public key of the user. The server must reject the authentication request if

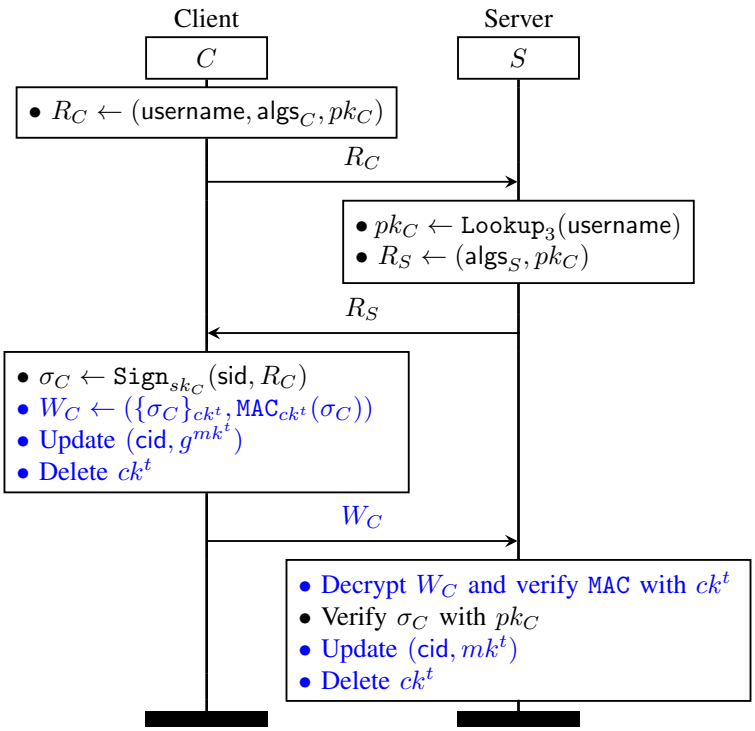


Figure 5.4: The SSH Authentication Protocol using the public key authentication method. `Lookup` is an algorithm that returns existing information from the persistent storage based on the input and ',' is used for concatenation. Our changes are highlighted in blue. After a successful run of this protocol, the server authenticates the client to possess the same updated cipher key and also the private key to the requested public key.

the server does not support the public key algorithm. The server also verifies that it recognises the client's public key and check whether this key is acceptable for authentication. If the server rejects the authentication, the server sends a failure message back to the client. Otherwise the server sends back the message R_S with the public key algorithm name and the public key of the client from the client's previous request. In the RFC, R_C and R_S are called `SSH_MSG_USERAUTH_REQUEST` and `SSH_MSG_PK_OK` respectively.

The actual authentication is performed when the client generates a signature σ_C on the session identifier from the SSH Transport Layer Protocol together with the request message R_C using the private key corresponding to the requested public key pk_C. In the original SSH Authentication Protocol, this signature is then sent to the server. For our modifications to the protocol, this signature is similarly wrapped by encrypting the signature and generating a MAC on the signature as

in the SSH Transport Layer Protocol, with both operations utilising the cipher key ck^t for the current new epoch t before the wrapped signature is sent to the server. The client then generates the corresponding Diffie-Hellman public key with the master key being the private key. We call this operation as 'masking' the master key, and we call the master key as being 'masked', or simply, the 'masked' master key. The client then stores or replaces the masked master key with the identifier cid from the SSH Transport Layer Protocol into its persistent storage for future sessions. The purpose for masking the master key is to prevent the adversary from accessing the master key directly when cloning the client, which helps to prevent further attacks by the adversary such as server impersonation to the client. We refer to Figure 5.2 for more details. The client then securely deletes the cipher key to prevent the leakage of the key.

The server then accordingly decrypts the wrapped signature and verifies the MAC with the same cipher key ck^t generated in this time epoch t. If the MAC verification is successful, this means that the server has the assurance that the client synchronises the cipher key correctly with the server. The server then proceeds to verify the signature σ_C of the client using the client's requested public key. A successful verification means that the client possesses the corresponding private key and therefore this implies a successful authentication. The new master key and the client's identifier cid are then updated by the server into its records. The server stores the master key without masking the key for the following reason: If the adversary attempts to log into the server while impersonating the client, the server will still be able to replace the master key before the adversary gains access to the server and this prevents the adversary from obtaining the master key before it accesses the server's persistent storage. Also since the adversary only gets access to the masked master key and not the actual master key itself from the client, the adversary has no choice but to brute force the master key. Consequently, the adversary is prevented from rolling back the master key to avoid detection. We call this security guarantee as Post-Impersonation Detection which we prove in Guarantee 5.1. Notice that the master key is updated into persistent storage on

both the client and the server only after authentication so that a Denial-of-Service (DoS) attack where an adversary without the cloning ability can desynchronise the master key between the client and the server, e.g., by simply dropping the protocol messages, can be prevented. In the last step of the protocol, the server then similarly deletes the cipher key in a secure way to prevent the leakage of the key.

This then concludes the public key authentication. Note that the server sends a final message to indicate to the client if the authentication is successful or if further authentication is required. We redirect interested readers to the RFC of this protocol (Ylonen, 2006b) for more details. In the rekeying process, the SSH Authentication protocol is not required to be executed by the client and the server again. Since the client and the server store the updated keys in the persistent storage during the SSH Authentication protocol, we require that the protocol is executed in the rekeying process as well. However, this will not impose a serious computational performance overhead as the completion time for the execution of this protocol is in the range of milliseconds, as shown by our experiments in Section 5.4.3. We note that this experimental result do not account for network latency and we discuss more on this in Section 5.4.3.

5.3 Security Analysis

In this section we present our security analysis for our improved SSH protocol. We prove that our solution provides new security guarantee as well as maintains and enhances the existing guarantees of the original SSH protocols. In each of our proofs, we enumerate the different ways the adversary can break the guarantee, and we prove by contradiction that each of them breaks our underlying assumptions. Each security guarantee corresponds to one of the two goals of the adversary: **Eavesdrop** and **Stealthy Impersonation** as listed in Section 5.1.2. We assume that the underlying cryptographic primitives are secure as per the definitions in Section 2.1.

5.3.1 New Guarantees

We prove that our solution provides three new guarantees. If the adversary clones the client and subsequently logs into the server impersonating the client, Guarantee 5.1 and Guarantee 5.2 provide the SSH Transport Protocol with the capability of detecting the presence of the adversary. These two guarantees correspond to the **Stealthy Impersonation** goal of the adversary. With respect to the goal of **Eavesdrop**, Guarantee 5.3 minimises the exposure of the server's signatures generated from the server's long-term signing key sent in the SSH Transport Protocol.

Guarantee 5.1 (SSH Transport Layer Protocol: Post-Impersonation Detection)**.** *If an adversary $\mathcal{E}$ successfully impersonates the client C to the server S at epoch $\tau_1 = \tau_0 + 1$ and the subsequent epochs $\tau_i > \tau_1$, then $\mathcal{E}$ will be detected by C when C tries to log into S at epoch $\tau_N + 1$.*

Proof. At epoch τ_N, as the client has not participated in the SSH Transport Layer Protocol since epoch τ_0, this means that the masked master key $g^{mk^{\tau_0}}$ of the client has not changed since epoch τ_0. On the other end, the master key of the server S and the adversary $\mathcal{E}$ has been updated to mk^{τ_N} because the adversary runs the protocol with the server impersonating the client at epoch τ_N. Hence, when the client runs the protocol with the server at $\tau_N + 1$, the Message Authentication Code (MAC) in message W_S of the SSH Transport Protocol does not match which causes the adversary to be detected.

Note that the the adversary possesses the masked master key $g^{mk^{\tau_0}}$ from cloning the client at epoch τ_0. Also observe that when the adversary logs into the server impersonating the client at epoch $\tau_1 = \tau_0 + 1$, the adversary obtains the server's private signing key sk_S and the master key mk^{τ_1} but not mk^{τ_0}. This is due to the server updating mk^{τ_0} with mk^{τ_1} during the protocol run with the adversary. Similarly, when the adversary logs into the server at subsequent epochs $\tau_i > \tau_1$, the adversary has the master key mk^{τ_i} in addition to sk_S.

For an adversary to break this guarantee, the adversary has to remain undetected by both the client and the server when the client initiates the protocol at epoch

τ_N+1. There are two ways for the adversary to achieve this. (1) While the adversary can now generate the server's signature σ_S with sk_S, the adversary is still required to compute ck^{τ_N+1} to avoid detection. (2) Alternatively, the adversary can roll back the master key stored at the server to mk^{τ_0}, i.e., the master key corresponding to the masked master key that the client stores when the adversary cloned the client at epoch τ_0, which prevents the client and the server from generating different cipher keys and allows the adversary to evade detection.

The first case (1) requires the adversary to also derive the fresh Diffie-Hellman shared secret z^{τ_N+1} and the ratchet key rk^{τ_N+1} as the cipher key ck^{τ_N+1} is derived from both z^{τ_N+1} and rk^{τ_N+1}. Observe that for z^{τ_N+1}, the adversary can generate and inject its own fresh Diffie-Hellman keypairs $(g^{x'}, x')$ and $(g^{y'}, y')$ into the protocol to derive the values $(z_C^{\tau_N+1}, z_S^{\tau_N+1}) = (g^{xy'}, g^{x'y})$ in a classic decrypt-and-encrypt Man-in-the-Middle (MitM) attack. For the same attack with rk^{τ_N+1}, the adversary can similarly generate and inject its own Diffie-Hellman keypair $(g^{v'}, v')$ to generate the same root key $rk_S^{\tau_N+1} = g^{mk^{\tau_N+1}v'}$ as the server. However, to generate the same root key as the client, the adversary must be able to generate $rk_C^{\tau_N+1} = g^{mk^{\tau_0}v}$. This, in turn, leaves the adversary with two choices: either derive or guess $rk_C^{\tau_N+1}$. While the adversary has both $g^{mk^{\tau_0}}$ and g^v from the client, if the adversary is successful in deriving $rk_C^{\tau_N+1}$, this breaks the Computational Diffie-Hellman (CDH) assumption and thus our assumption of secure cryptographic primitives. On the other hand, with such an assumption the probability of the adversary guessing the value of $rk_C^{\tau_N+1}$ is negligible.

In the second case (2), recall from the second paragraph of this proof that the adversary does not possess mk^{τ_0} as the server replaces mk^{τ_0} with mk^{τ_1} when the adversary logs into the server impersonating the client at each epoch τ_1. For the adversary to roll back the master key stored at the server to mk^{τ_0} and evade detection, the adversary must first obtain the value of mk^{τ_0}. With the masked master key $g^{mk^{\tau_0}}$ that the adversary has cloned from the client at epoch τ_0, the adversary has two ways of achieving this: either derive mk^{τ_0} from $g^{mk^{\tau_0}}$, or guess mk^{τ_0}. In succeeding to extract mk^{τ_0} from the masked master key $g^{mk^{\tau_0}}$, the adversary

breaks the Discrete Logarithm (DL) assumption, and thus our assumption of secure cryptographic primitives. However, with the assumption of secure cryptographic primitives, guessing mk^{τ_0} can only occur with a negligible probability. □

Guarantee 5.2 (SSH Transport Layer Protocol: Post-Compromise Security)**.** *If the client C logs into the server S at epoch $\tau_1 = \tau_0 + 1$, any future impersonation attempt by the adversary $\mathcal{E}$ in epochs $t > \tau_1$ is detected by C.*

Proof. At epoch $\tau_1 = \tau_0 + 1$, i.e., the next immediate epoch after the cloning epoch τ_0, if the client logs into the server before the adversary, this means that the adversary has neither logged into the server nor impersonated the client. Also recall from Section 5.1.2 that the adversary does not have access to the server's long-term signature key sk_S as this requires a login to the server. As a result, at epoch τ_1, the information that the adversary possesses from cloning the client in epoch τ_0 consists of only the client's masked master key $g^{mk^{\tau_0}}$ and the client's long-term keypair (pk_C, sk_C).

For the adversary to break this guarantee, the adversary has to be able to impersonate the server to the client without being detected at epoch τ_1. This means that the adversary has to fulfill two conditions: (1) derive the cipher key ck^{τ_1}, and (2) generate a valid server's signature σ_S.

The first case (1) proceeds similarly with Guarantee 5.1's first case. As the cipher key ck^{τ_1} is derived from both the fresh Diffie-Hellman shared secret z^{τ_1} and the ratchet key rk^{τ_1}, this requires the adversary to derive z^{τ_1} and rk^{τ_1}. For z^{τ_1}, the adversary can generate and inject its own fresh Diffie-Hellman keypairs $(g^{x'}, x')$ and $(g^{y'}, y')$ into the protocol to derive the values $(z_C^{\tau_1}, z_S^{\tau_1}) = (g^{xy'}, g^{x'y})$ in a classic decrypt-and-encrypt Man-in-the-Middle (MitM) attack. With the same attack on rk^{τ_1}, the adversary can similarly generate and inject its own Diffie-Hellman keypair $(g^{v'}, v')$ to generate the same root key $rk_S^{\tau_1} = g^{mk^{\tau_0}v'}$ with the server. On the client's end, the adversary must be able to generate the same $rk_C^{\tau_1} = g^{mk^{\tau_0}v}$ with the client. This again leaves the adversary with two choices: either derive or guess $rk_C^{\tau_1}$. Although the adversary has both $g^{mk^{\tau_0}}$ and g^v, if the adversary succeeds

in deriving $rk_C^{\tau_1}$, this breaks the Computational Diffie-Hellman (CDH) assumption and subsequently our assumption of secure cryptographic primitives. On the other hand, with such an assumption the probability of the adversary guessing the value of $rk_C^{\tau_1}$ is negligible.

The second case (2) requires the adversary to forge the server's digital signature. However, with the assumption of secure cryptograhic primitives, specifically with a EUF-CMA secure digital signature scheme (see Section 2.1.2), this means that a signature forgery can only occur with a negligible probability.

□

Guarantee 5.3 (SSH Transport Layer Protocol: Reduced Exposure of Long-term Signatures)**.** *An adversary $\mathcal{E}$ can access the long-term signatures σ_S of the server S only if the adversary clones the client.*

Proof. In order for the adversary to break this guarantee, the adversary has to obtain the server's signatures without cloning the client. This means that the adversary only has control of the communication channels and not the secrets of the client. Subsequently this also means that the adversary does not have access to the server's private key sk_S. Thus, to obtain signatures of the server, the adversary must be able to decrypt the wrapped signature encrypted with the cipher key ck^t from the message of W_S at any given epoch t. To achieve this the adversary has to derive the cipher key ck^t and this requires the adversary to either: (1) inject the adversary's own Diffie-Hellman keyshares into the protocol messages, or (2) deriving the cipher key ck^t itself passively without tampering with the protocol messages.

For the first case (1), the adversary can inject its own values of Diffie-Hellman keyshares, i.e., $(g^{x'}, x')$ and $(g^{y'}, y')$, into the protocol messages to derive the cipher key ck^t. The adversary, in turn, has to also generate the signature σ_S of the server to convince the client and satisfy the protocol. This in turn requires the adversary to forge the server's signature but with a EUF-CMA secure digital signature scheme under our assumption of secure cryptographic primitives, this can only occur with a negligible probability.

For the second case (2), deriving the cipher key ck^t passively requires the adversary to obtain not only the fresh Diffie-Hellman shared secret z^t of epoch t exchanged by the client and the server, but also the ratchet key rk^t derived from the master key mk^{t-1} of the previous epoch $t-1$. To fulfill this, the adversary has to compute the Diffie-Hellman private keys, i.e., the values of v, x, and y generated by the client and the server in the protocol in both epochs t and $t-1$, or the adversary can simply guess the values of the secret keys v, x, y, z^t, z^{t-1}, mk^{t-1}, rk^t, and ck^t. This similarly leaves the adversary with the choices of deriving or guessing the Diffie-Hellman private keys of both epochs t and $t-1$. As we have seen in Guarantee 5.1, successfully deriving the Diffie-Hellman private keys contradicts our assumption of secure cryptographic primitives as it contradicts the DL assumption and thus also the CDH assumption; and with the assumption of secure cryptographic primitives, guessing the secret keys v, x, y, z^t, z^{t-1}, mk^{t-1}, rk^t, and ck^t correctly can only occur with a negligible probability. □

5.3.2 Existing Guarantees

In addition to new guarantees, we also prove that our solution maintains the existing guarantees of the SSH protocols. Recall from Section 5.1.2 that existing SSH protocols are only limited to an adversary that has access to only the communication channels (except perfect forward secrecy). Also notice that this is similar to the adversary in our adversary model without the adversay's cloning ability. For the **Stealthy Impersonation** goal of an adversary with this restricted capability, Guarantee 5.4 preserves the mutual entity authentication between the client and the server for the SSH Transport Protocol and the SSH User Authentication Protocol. Similarly for the adversary's goal of **Eavesdrop**, Guarantee 5.6 keeps the confidentiality and authentication of the session data exchanged between the client and the server in the SSH Transport Protocol whereas Guarantee 5.5 retains the perfect forward secrecy of the previously established session keys.

Guarantee 5.4 (All Protocols: Mutual Entity Authentication)**.** *The client C authenticates the server S through the SSH Transport Layer Protocol, and S*

authenticates C through the SSH Authentication Public Key Protocol. An adversary $\mathcal{E}$ can impersonate C at epochs $\tau_i > \tau_0$ only if $\mathcal{E}$ clones C at epoch τ_0.

Proof. This guarantee can only be broken if the adversary can successfully impersonate (1) the server S in the SSH Transport Layer Protocol or (2) the client in the SSH Authentication Public Key Protocol respectively without cloning the client.

For the first case (1), in order for the adversary to impersonate the server, the adversary has to produce the signature σ_S and the MAC in message W_S of the SSH Transport Protocol. By our assumption of secure cryptographic primitives, specifically the EUF-CMA security for the signature scheme and the MAC scheme as defined in Section 2.1.2, the probability of the adversary forging the signature and the MAC is negligible.

For the adversary to impersonate the client without cloning the client in the second case (2), the adversary has to similarly produce the signature σ_C and the MAC in message W_C of the SSH Authentication Protocol. This again invokes our assumption of secure cryptographic primitives with a focus on the EUF-CMA security for the signature scheme and the MAC scheme where the probability of the adversary succeeds in forging the signature and the MAC is negligible.

□

Guarantee 5.5 (SSH Transport Layer Protocol: Perfect Forward Secrecy)**.** *An adversary $\mathcal{E}$ will not be able to derive the master key, the cipher key, and the application keys from the previous session epochs $t < \tau_0$.*

Proof. As the application keys are derived from the cipher keys ck^t, this requires the adversary to first derive the previous cipher keys for epoch $t < \tau_0$. Note that in epochs $t < \tau_0$, i.e., the epochs before the adversary clones the client, the adversary only has access to the communication channels and thus only the protocol messages. In the epoch τ_0, the adversary has the additional access to the client's masked master key $g^{mk^{\tau_0}}$. To obtain the past cipher keys for epochs $t < \tau_0$ and break this guarantee, the adversary has only three choices: (1) deriving past master and cipher keys using the protocol messages observed in previous epochs, (2) deriving past

master and cipher keys using the masked master key $g^{mk^{\tau_0}}$, and (3) simply guessing the previous master key, cipher key, or application keys.

In the first case (1), we emphasise that the adversary only has access to the messages of the protocol in the previous epochs $t < \tau_0$ before the adversary clones the client. This is identical to the adversary in the second case of the proof in Guarantee 5.3 where the adversary also has to derive the master and cipher keys passively. As we have previously established in the proof, this results in a contradiction by our assumption of secure cryptographic primitives.

In the second case (2), the adversary must be able to derive the past cipher keys ck^t for the previous epochs $t < \tau_0$ using the additional masked master key $g^{mk^{\tau_0}}$ that the adversary has cloned from the client. As the cipher keys are further derived from the ratchet key and the fresh Diffie-Hellman secret, we have shown before in the first case of the proof for Guarantee 5.1 that deriving the ratchet key using the masked master key also leads to a contradiction of our assumption in secure cryptographic primitives, specifically the DL and the CDH assumptions.

In the third case (3), the adversary guessing the correct previous master keys, cipher keys, or application keys can only occur with a negligible probability under the assumption of secure cryptographic primitives. More precisely, with secure key derivation functions and hash functions as defined in Section 2.1.6 and Section 2.1.5 respectively.

□

Guarantee 5.6 (SSH Transport Layer Protocol: Session Data Confidentiality and Authentication)**.** *Session data exchanged between the client C and the server S are known only to C and S, and the authenticity of the data is preserved. An adversary $\mathcal{E}$ can expose session data exchanged at epoch τ_0 only if $\mathcal{E}$ clones C.*

Proof. Breaking this guarantee means that the adversary is able to reveal or modify the session data exchanged between C and S in an epoch t without cloning the client. To do this, the adversary has to generate the application keys for the current epoch t. As the application keys themselves are derived from the cipher key ck^t,

which itself is derived from both the fresh Diffie-Hellman shared secret z^t and the ratchet key rk^t of the current epoch t, the adversary has to first compute these two values of z^t and rk^t. Recall that the current ratchet key rk^t is derived from the previous master key mk^{t-1} in epoch $t-1$ which is in turn derived from the fresh Diffie-Hellman keyshares in that epoch. In order for the adversary to succeed, the adversary has three choices: either (1) generate and inject the adversary's own fresh Diffie-Hellman keys, i.e., (X, x) and (Y, y), in both epochs t and $t-1$, (2) derive the freshly generated Diffie-Hellman private keys, i.e., the values v, x and y, by the client and the server in the SSH Transport Protocol in both epochs t and $t-1$, or (3) guessing the values of v, x, y, z^t, rk^t, mk^{t-1}, ck^t, and the application keys. These three cases of the adversary reflect the identical situation as described in both cases of the proof in Guarantee 5.3 where all cases contradict on the assumption of secure cryptographic primitives.

□

5.3.3 Summary

In summary, we preserve the existing security guarantees for the SSH protocol against a traditional adversary originally considered by the protocol, i.e., an adversary that only has access to the communication channels but not the secrets of a client. Specifically, Guarantee 5.4 prevents such an adversary from breaking the mutual entity authentication between the client and the server whereas Guarantee 5.6 prevents such an adversary from exposing the session data exchanged between the client and the server.

Our modifications to the two SSH protocols provide three novel security guarantees in the presence of a more powerful adversary that clones the client's keys and subsequently impersonates the client to the server. Guarantee 5.1 assures that the presence of this stronger adversary will be detected by the client if the adversary logs into the server impersonating the client before the client's own login. The detection from Guarantee 5.1 is still in place even if the adversary later compromises the server's private key and rolls back the master key stored at the server during

the adversary's session. In contrast, if the client logs into the server before the adversary does, then Guarantee 5.2 prevents future impersonation attempts by the adversary. As demonstrated in Guarantee 5.3, our protocol provides the added security against advanced cryptanalytic attacks that exploit the static signatures of the server, such as those discovered by Ryan et al. (2023), by reducing their exposure during protocol execution.

We additionally make sure that Guarantee 5.6 limits the consequences of the adversary's impersonation such that only the session data in the epoch where the cloning occurs is exposed in our stronger adversary model. Guarantee 5.5 preserves the existing forward secrecy guarantee of the SSH protocol to ensure that the adversary is not able to derive the master keys, cipher keys, and application keys from previous epochs.

5.4 Implementation and Evaluation

To demonstrate the practicality of our changes to the protocol, we implement a proof-of-concept for our solution. Based on our implementation, we evaluate the performance overhead of our solution against the original SSH solution experimentally. This section presents our findings on the implementation and the experiments.

5.4.1 Integration with OpenSSH

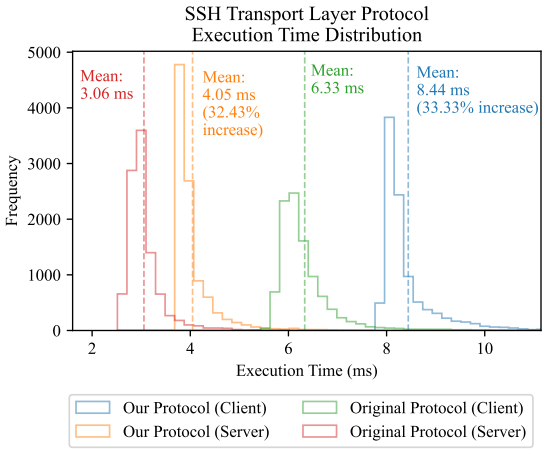
We build our changes for the SSH Transport Layer Protocol and the SSH Authentication Protocol on top of the existing implementation of OpenSSH, specifically the `openssh-portable` codebase with version `OpenSSH_9.8p1`. We make the source code for our implementation available on our self-hosted git server repository[1] where we provide the full details of our changes, and we only describe the relevant details in this chapter. We show that our changes can be implemented without changing the existing privilege separation mechanism (daztucker, 2019) for the server-side code in OpenSSH, and that our changes can be implemented using only the cryptographic primitives already available in OpenSSH. This means that our changes do not

[1] `https://malaria.cs.ox.ac.uk/git/wil/pcsssh-experiments`

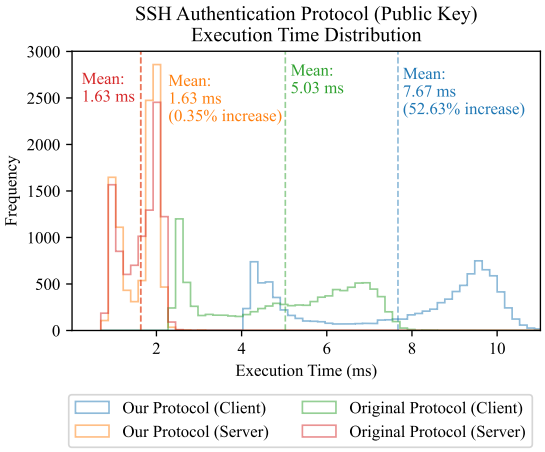
require the introduction of any new cryptographic primitives, and thus do not require any new dependencies. Additionally the changes that we introduce in our implementation can be compiled, built, and installed in the same way as the original OpenSSH components, e.g., the server in our implementation can be executed as a daemon and set up as a `systemd` service managed by the `systemctl` shell command.

5.4.2 Experiment Setup

We then use the implementation as the basis for our experiments to evaluate the performance overhead of our solution against the original SSH protocol. We installed both versions of implementation: our implementation and the original unchanged codebase of OpenSSH with the same version to run the experiments. The server for both versions of the experiments runs as a daemon, and the client connects to the server via the `ssh` command, after which the client immediately disconnects by executing the `exit` shell command. To evaluate the performance overhead, we measure the execution time of the SSH Transport Layer Protocol and the SSH Authentication Protocol on both the client's and the server's end when the client logs into the server. The measurements are repeated for n connections to obtain the distribution as well as the mean of the execution time for each protocol in each of the two versions of the experiments. For the reproducibility of the experimental results, we provide the details of relevant parameters for the experiment setup here. In both versions of the experiments, the key exchange algorithm is set to `curve25519-sha256`, and we only test the public key authentication method for the SSH Authentication Protocol. In order to gauge a good estimate on the computational overhead and make network latency negligible, both the client and server are run on a single Windows 11 machine equipped with Intel® Core™ i5-1145G7. All experiments are performed in Windows Subsystem for Linux (WSL) running Ubuntu 22.04.



(a) Distribution of execution time for SSH Transport Layer Protocol on the client's and the server's end in both versions of the experiments.



(b) Distribution of execution time for SSH Authentication Protocol with only the public key authentication method on the client's and the server's end in both versions of the experiments.

Figure 5.5: Distributions of execution time for the SSH Transport Layer Protocol and the SSH Authentication Protocol from $n = 10000$ SSH logins as the results of the experiments. The two versions of the experiments are based on the modified OpenSSH that contains our modified protocols and the original OpenSSH implementation.

5.4.3 Results Analysis

In our experiments, we collected the measurements for $n = 10000$ connections. Figure 5.5a and Figure 5.5b show the distribution of the execution time in milliseconds (ms) for the client and the server to complete the SSH Transport Layer Protocol and the SSH Authentication Protocol (public key authentication) respectively.

From Figure 5.5a, the mean completion time to execute the SSH Transport Layer Protocol where the key exchange takes place has only a small difference in both versions of the experiments on both the client's end and the server's end. On the client's end, the difference in mean protocol completion time is 2.11 ms (an increase of 33.33%) whereas on the server's end the difference in mean protocol completion time is 1.01 ms (an increase of 32.43%). Both differences in mean completion time are well under 5 ms (less than 35%). This shows that our changes to the key exchange of SSH are very practical and hardly introduce any significant performance overhead over the original key exchange on either side of the connection, which demonstrates the practicality of our solution.

From Figure 5.5b, the mean completion time to execute the public key authentication method of the SSH Authentication Protocol has only a similar slight difference in both versions of the experiments for the client and the server. The figure shows that the difference in mean protocol completion time on the client's end is 2.64 ms (an increase of 52.63%) and there is no observable difference in mean protocol completion time on the server's end. Similarly, the differences in the mean completion time are well under 5 ms (less than 55% increase). This again shows that our changes do not incur substantial performance overhead when the server authenticates the client using the public key method, thus solidifying our solution's practicality.

The experimental results presented in this section show that no impractical performance degradation can be observed on either the client or the server when our changes to the SSH protocol are implemented into the OpenSSH implementation, which shows that our solution does not add any deterioration to the user experience of OpenSSH. With our modified SSH protocols, not only are we able to achieve new security guarantees as previously shown in the chapter, but the experimental results obtained here conclude that our solution is also deployable for real-world usage.

While the experiments only cover computational overhead, we briefly discuss about the potential impact of our modifications to the SSH protocols on storage and communication overhead. We anticipate that the storage overhead on the server will scale linearly, since the server stores an additional master key for each client in the system. As for communication overhead, our changes introduce only fixed-length fields to the protocol messages, namely, an identifier, a Diffie-Hellman public key, the ciphertext of a fixed-length signature, and a MAC. These additions contribute a constant overhead to existing message sizes as determined by the specific parameters of the implementation of the cryptographic primitives, all the while without requiring any additional round trips (except rekeying) in the protocol. As such, we do not expect a significant increase in communication overhead to current SSH implementations. These overhead considerations are indeed important aspects for practical deployment, and so we plan to explore them further in our future work, as outlined in Section 9.2.2.

5.5 Further Considerations

In this chapter we present how our solution is integrated into the SSH Authentication Protocol using the public key authentication method (Figure 5.4) for the server to authenticate the user. We mainly focus on the public key authentication method as it is the authentication method that is required by the RFC (Ylonen, 2006b). However, we note that the core idea for our solution is still applicable to other SSH authentication methods. As examples, for password-based authentication, the cipher key can be used to encrypt the password on the client-side before the encrypted password is sent to the SSH server for verification; for host-based authentication, the steps of the protocol can proceed similarly with the public-key authentication using a host-tied public key. Since the goal of our solution in SSH Authentication Protocol is for the server to verify that the user (or client) has the same synchronised cipher key as the server, our only requirement for the authentication is for the client to be able to encrypt and produce a Message Authentication Code (MAC) on information related to the chosen authentication method with the new cipher key generated on the client's end, which shows the extensibility of our solution to various authentication methods.

5.6 Chapter Summary

This chapter addresses the problem of SSH client credential leakage problem, where if an adversary manages to access the secrets of the SSH client, the adversary can impersonate the client indefinitely when logging into the server without getting detected. Existing solutions proposes prevention methods to such credential leakage, and they either introduce additional parties into the SSH architecture (Harchol et al., 2018), or requires a specialised hardware (Plaga et al., 2018).

Instead of focusing on preventing the leakage of the SSH client's credentials, our solution focuses on the detection of the adversary post-compromise. We offer a solution built directly into the SSH protocol, specifically the SSH Transport Layer Protocol and the SSH Authentication Protocol, without changing the SSH

architecture or requiring any hardware on the client or the server. Our solution can thus be applied into a wider area of SSH systems. By associating a symmetric master key for a pair of client and server that gets updated on every login or rekeying, we can detect an adversary whenever the adversary impersonates the client or the server, which causes the key to be desynchronised between the client and the server. Our detection mechanism works against a powerful adversary that not only controls the communication channels and the SSH client's secrets, but also even when the adversary gets hold of the server's secrets after logging into the server impersonating the client using the client's secrets with root access.

We give an informal security analysis to prove that our solution provides the new detection guarantees and also to prove that our solution preserves the existing security guarantees of the SSH protocol. To demonstrate the practicality of our solution, we integrate our changes to the protocol into the codebase of OpenSSH as a proof-of-concept. We further show experimentally that our solution only incurs a tiny performance overhead that would be unnoticeable in real-world deployments, thus preserving the current user experience of OpenSSH.

6

Authenticating with Actions

Contents

In machine-to-machine (M2M) communication, embedding or hardcoding credentials into application source code or automated scripts as the identity of the

machine is a common (bad) practice to facilitate routine tasks in the development process. This approach enables the machine to authenticate its identity in a seamless execution of automated workflows, reducing manual human intervention and increasing efficiency. However, while it simplifies authentication, it also introduces significant security vulnerabilities. The hardcoded credentials that form the machine's identity can be exposed to unauthorised parties with access to the source code and scripts directly, or to the machines running them, increasing the risk of compromise.

Real-world security breaches in technology companies (Kovacs, 2022; McDaniel, 2022) highlight the dangers of inadequate secure coding practices. Although storing credentials locally, such as through environment variables, is often viewed as a safer approach, incidents like (Engelberg, 2021) showcase that these methods are still not safe from a determined attacker. When coupled with human-related security vulnerabilities, such as Multi-Factor Authentication (MFA) fatigue attacks as shown in recent events (Kost, 2024), poor identity or credential management could lead to further security risks. These practical scenarios underscore the need for a more robust and secure approach that balances usability with effective risk mitigation. With the number of machine identities rapidly outgrowing the number of human identities (VENAFI®, 2021), it is of paramount importance that a machine identity does not only help in securing the machine communication that takes place, but also simplify the management of such machine identities that provides scalability for the entity or organisation that owns them.

In this chapter we propose *ActionID* which is a novel solution to this problem. It would of course be desirable to entirely prevent credential theft in the first place, but since it is nearly impossible to fully prevent theft from, say, the administrator of the machine of a company, we take a different approach. Rather than trying to prevent credential theft altogether, we make sure that the stolen credentials can only be used for a very specific well-defined purpose, and only for a short amount of time, after which they will have to be renewed. We achieve this by associating the identity of a machine with not only its credentials, but also to every action

sequence it wants to execute. Not only does this limit what an adversary with a compromised credential can do, this association further allows a central entity to monitor the specific usage of each identity, and build up a picture of who is doing what in a company's network. If an adversary is forced to continuously renew stolen credentials, the chances of detecting a credential theft is also increased since the credentials will be renewed more often than before.

ActionID also provides a convenient way to set up and manage user accounts and access control policies for an entire organisation in one place. This is desirable because if access control decisions are made by individual services, then any changes to the access control policies, including adding new users or removing old ones, have to be applied locally to each individual service. *ActionID* enables a unified access control policy to be put in place in a central location, and gracefully handles both identity management and revocation of access. We present the protocols that make all this work efficiently along with a thorough security analysis that prove the necessary security guarantees hold. We also implement *ActionID* as a library to show the performance and scalability of the idea. From that we conclude that even our implementation in Python has a performance on par with other tools that can establish a secure connection, e.g., SSH.

We summarise our main contributions as follows:

- We propose a M2M scheme that utilises a novel machine identity: the assocation of identity with specific actions requested for a specific server. This forms a dynamic machine identity that reduces the implications of a machine compromise and credential theft.
- We design two protocols that make use of this dynamic identity to give strong security guarantees to the principals, even in the presence of a powerful adversary that has the capability to compromise machines.
- We prove that our protocols are secure with respect to the relevant security properties.

- We provide an implementation of *ActionID* as a Python library which makes it easy to deploy on a large class of devices and our implementation is available on GitHub. We use it to document the performance and scalability of the scheme.

This represents a brand new way to think about machine identity, not just as the owner of a private or symmetric key, but as a collection of actions backed up by a traditional identity. If any of the actions changes, that means the identity changes, so any access tokens issued are no longer valid. This effectively limits an adversary that did manage to steal credentials, to do only the same action as the owner of the credential, and only for a brief time, within the limits set out by the access control policies.

6.1 Motivation and Design

We first look into the specific problems arising from a typical M2M communication scenario in an organisation's network as our motivation. We then show how existing systems are not designed specifically to handle these problems. To address these problems, our solution is to associate a machine's identity not only to its credentials, but also to its actions and we further list the design goals that we aim to achieve with this idea.

6.1.1 Motivation

In Figure 6.1, we illustrate a general abstracted scenario of M2M communication in an organisational network between machines owned by the organisation. Without a human in the loop, a requesting machine, or simply a *requestor*, accesses a *server*'s services by running a *script*. A script typically contains the task to be completed by a server, which we term as *actions*, and the *credential* a requestor uses to authenticate itself to the intended server. Examples of actions include database scripts, shell commands, or even custom predefined strings readable and executable by a server; examples of credentials are mentioned in Section 3.1.1. We refer to

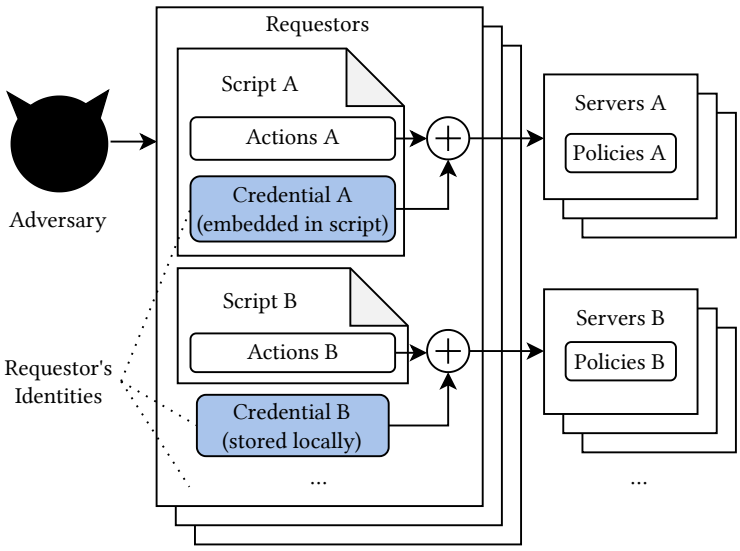


Figure 6.1: A general abstracted scenario of a M2M communication in an internal organisational network. A requestor runs scripts specifying the actions to be executed and the credential to authenticate to a server. Without a human supplying the credentials, the requestor's credentials are typically stored locally in the clear, e.g., hard-coded in the requestor's script, allowing an adversary that has access to the requestors to extract the credentials. A server enforces access control policies that manage the execution of the requestor's actions. Multiple servers may share the same policies.

Figure 6.8 for a concrete example of a requestor's script. Each server imposes and enforces *access control policies* of who can access what and checks if the execution of a requestor's actions is allowed, e.g., if a requestor's access to a file is authorised.

Without a human to actively input credentials when accessing a server, a requestor's credentials must be available locally to the requestor in the clear, e.g., a hard-coded password embedded in the script of the requestor, or a private key stored locally on the disk storage of the requestor. The credentials might in turn be encrypted by other secrets that are stored on a requestor unencrypted. Nonetheless, an adversary having physical access to a requestor can obtain the unencrypted credentials or extract the secrets to decrypt the wrapped credentials. As the credentials represent a requestor's only identity to a server, an adversary with such credentials can impersonate the victim requestor from another machine.

From a management's point of view, this scenario of a M2M communication has several disadvantages. Since the access control policies are managed by individual servers, changes to the access control policies shared by several servers require

local changes to the policies at each of the servers. Even with the access control policies in place, mismanagement of a server's access control policies, e.g., resulting from human errors (Verizon, 2020), allows for an adversary with compromised credentials to successfully request for the execution of arbitrary actions on behalf of the compromised requestor. As for the credential management of a requestor, each requestor has to manage a unique credential for each service offered by the servers. These management issues can quickly become unscalable for an organisation with a large amount of servers.

On one hand, the problem with relying solely on a secret information to identify machines is that the secret information can be stolen. On the other hand, it is also impossible to establish secure connections and differentiate machines without using any secret information. Ideally it would be perfect if there is a foolproof way of preventing credential theft, but since this is impossible to achieve, we take a different approach: instead of worrying about a piece of secret information getting stolen, we limit the consequences of a credential theft by restricting what the adversary with the stolen credential can achieve. Building on this approach, we propose a novel machine identity for use in a M2M communication where a machine's identity is not only made up of its credentials but also the actions it wants to execute. With our proposed system model, we simplify the management of such machine identities.

6.1.2 Design Goals

Our solution to address the problems presented in Section 6.1.1 is to associate a requestor's identity to not only its credentials, but also its actions. Here we list the five design goals based on this novel idea. The first four design goals focus on the security aspect of our system and the last design goal focuses on the credential and policy management of the system. We later discuss how our system achieves the last design goal in Section 6.2.1, and prove the four security-related design goals of the system in Section 6.4.

Mitigate Implications of Credential Leakage

Unlike existing systems where machine credentials are only based on secrets, an adversary with unrestricted access to credentials can impersonate the victim from a different machine. A server is then unable to differentiate between an adversary with a compromised credential from the legitimate machine based solely on a machine's possession of some secret. Our goal here is then for a requestor's actions to become an additional authentication factor for a certain requestor. While no protocol can provide security guarantees against an adversary with unrestricted access to a secret, we can mitigate the implications of such credential leakage. Specifically, we require such an adversary should not execute arbitrary actions but only the same set of actions allowed for the victim requestor. Even if the adversary has both the "valid" actions and the credentials, we further require the actions must first be declared before execution and that the declared actions can only be executed for a short time, after which the declaration is again needed. Not only do we restrict what an adversary with compromised credentials can achieve, we provide a way for detection of credential compromise by observing if a requestor's actions deviate from its "normal" actions while also increasing the chances for such detection.

Bind Identity to Actions

Based on our requirement of requestors declaring actions ahead of time, the declared actions form an essential part of the requestor's identity for its session with the server. An adversary with compromised credentials might attempt to circumvent the requirement by not declaring its actions at all, or by first declaring "valid" actions, and then sending undeclared actions for execution. To avoid the adversary from arbitrarily changing its identity using undeclared actions, we thus need a mechanism to detect if the integrity of a requestor's identity has been violated, i.e., if the declared actions of a requestor have changed. Additionally we need a way to detect if an adversary tries to evade detection by executing already declared actions beyond their limited time, i.e., extending the expiration time of the already declared actions.

Liveness and Secure Channel Establishment

To prevent man-in-the-middle attacks and replay attacks, we additionally necessitate mutual liveness checks to be carried out between a requestor and a server before the execution of the requestor's actions. Depending on the specific application, the execution might involve further exchange of session data between the requestor and the server. We would thus want to prevent the session data from being sent as cleartext since the session data might contain sensitive information. The secure channel would additionally need to ensure the integrity of the session data to prevent its modification.

Confidentiality of Actions

A requestor's actions might contain sensitive application data about the system such as the contents of a file or the entries of a database. Therefore even the actions has to be kept secret from anyone other than the requestor's intended recipients when a requestor sends its actions for declaration or execution. To prevent the sensitive data from being leaked during a transmission, we thus require the confidentiality of a requestor's actions so that only intended recipients of a requestor can retrieve its actions. Although this does not stop an adversary with physical access to a requestor from obtaining the requestor's actions, the only thing the adversary can do with stolen actions is essentially to execute the actions on the victim requestor's behalf. Since the actions are already allowed to be executed by the victim requestor, the implication for the stolen actions is thus minimised.

Simplified and Centralised Management

Our desire is that the access control policies of the servers are to be easily managed as the decision for the authorisation of a requestor's actions involves directly the management of these policies. Specifically, a change in the policies shared by several servers should only be applied once, instead of the same changes being applied multiple times at each individual server. This creates a more centralised policy management desirable especially in an organisation with a large number of servers.

Additionally, as a requestor's actions are already specific to a particular server, we will not require a requestor to manage a unique long-term credential for each server. We would thus aim for a requestor to have only a single long-term credential, paired with its actions to form a disposable short-term credential. This, in turn, reduces the number of long-term credentials that a requestor has to manage to only one, regardless of the number of servers present in the system. Since the requestor has to manage its actions anyway regardless of the number of its long-term credentials, our design goal, if fulfilled, eases the credential management of a requestor.

6.1.3 Comparison with Existing Systems

The M2M scenario described in Section 6.1.1 is a very common setting for any organisation running automated tasks and thus it is surprising that existing systems have still yet to address these problems. As our focus is machine authentication, the de facto standard for authenticating machine and establishing secure channels between machines such as SSH (Ylonen, 2006c) and TLS (Dierks and Rescorla, 2008) are still not adequate for our M2M setting. This is because the security of these protocols require the assumption that the credential of a machine, i.e., a private key, being secret and they do not consider the presence of an all-powerful adversary having unrestricted access to the credentials of a machine. Similarly, SSO systems and the research work on SSO systems assume one or more secret information such as passwords (Steiner et al., 1988; Agrawal et al., 2018; Baum et al., 2020; Rawat and Jhanwar, 2020; Y. Zhang et al., 2020) and access tokens (Hardt, 2012; Sakimura et al., 2014; Lockhart and Campbell, 2008), which, if compromised, can lead to impersonation attacks as seen from real-world scenarios (Hanley, 2022; Binder, 2022; Ose, 2022). Additionally, widely deployed SSO systems such as Kerberos (Steiner et al., 1988), OAuth (Hardt, 2012), OIDC (Sakimura et al., 2014), SAML2 (Lockhart and Campbell, 2008) are designed for user authentication. This means that the presence of a user and user interaction, in one form or another, e.g., typing in passwords or completing Multi-Factor Authentication (MFA) challenges, are needed during service access to a machine. These SSO systems are thus not appropriate for

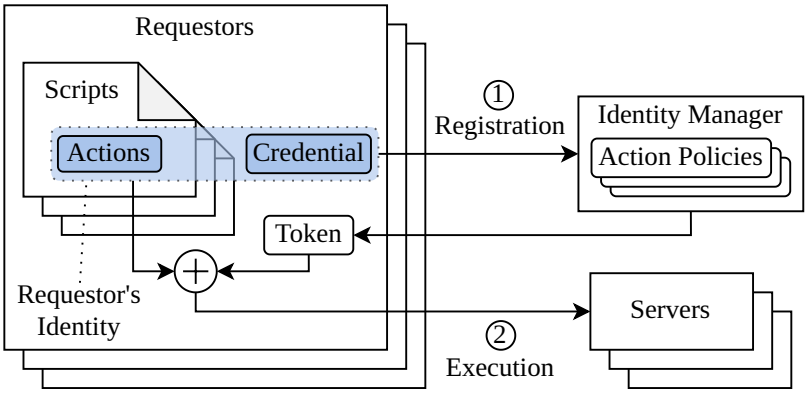


Figure 6.2: An overview of *ActionID*. A requestor's identity comprises of both its actions and its long-term credential. The identity manager is a trusted entity who manages all servers' action policies. A requestor registers its identity ahead of time to get a token for the execution of its actions on a server.

application in a M2M setting where there is no human user involved. We emphasise that our goal here is not to replace existing systems and protocols, but rather, since existing systems do not address the more specific security and management issues that stem from an organisation's internal M2M communication processes as described in Section 6.1.1, we thus propose a new system that leverages this idea of a machine's actions as part of its identity to specifically deal with these issues.

6.2 Overview and Models

We give an overview of the design of *ActionID* in this section. We later provide the system model and the adversary model in detail, which include the assumptions made for the entities in the system and also for the adversaries.

6.2.1 *ActionID* Overview

We propose *ActionID*, a novel M2M scheme for an organisation's network where a requestor's identity comprises of both its credential and actions. A high-level overview of *ActionID* is provided in Figure 6.2. We introduce a trusted third party called the identity manager who centrally stores and manages all *action policies* of servers. Action policies here refer to more than just authorisation policies (who can access what), but also fine-grained or even application-specific conditions for

the execution of actions to happen. For example, some actions should be executed only at a certain day of the week and are only executed for a fixed number of times in a certain time period. As the policies are centrally stored at the identity manager, we have now achieved the last design goal in Section 6.1.2 as changes to the policies shared by several servers are only applied once at the identity manager, instead of being individually applied at each server.

For executing actions, we require that a requestor first declares its actions ahead of time by registering them with the identity manager. If the actions of a requestor comply with the policies of its intended server, the identity manager issues a short-lived token to the requestor to be presented to the server for the execution. Our design not only forces an adversary with compromised credentials to always reveal its intended actions for execution, but also helps the identity manager in monitoring for credential compromise and building up a picture of who does what on the network by having a centralised log of registered actions. This can further facilitate anomaly detection on the logged actions to determine if a requestor's actions deviate from its "normal" actions, which could be an indication of a compromised requestor. The short-lived token limits the time window for action execution. So, even if an adversary's actions comply the policies, the adversary can only execute specific actions only for an allowed duration of time, after which the registration of actions is again needed, limiting the impact of a credential leakage and increasing the chances of detection.

Although the identity manager poses the risk of being a single point of failure, in practice trusted third parties are still being used in organisational networks as trust anchors for the security of organisational systems, despite of these risks. Examples include the Key Distribution Centre in Kerberos (Steiner et al., 1988), and identity providers in OAuth (Hardt, 2012) and OIDC (Sakimura et al., 2014). While a centralised identity manager might raise privacy concerns, privacy of machines from the organisation that owns them is generally undesirable in an organisational network, e.g., a corporate network, where it is, in fact, more preferable

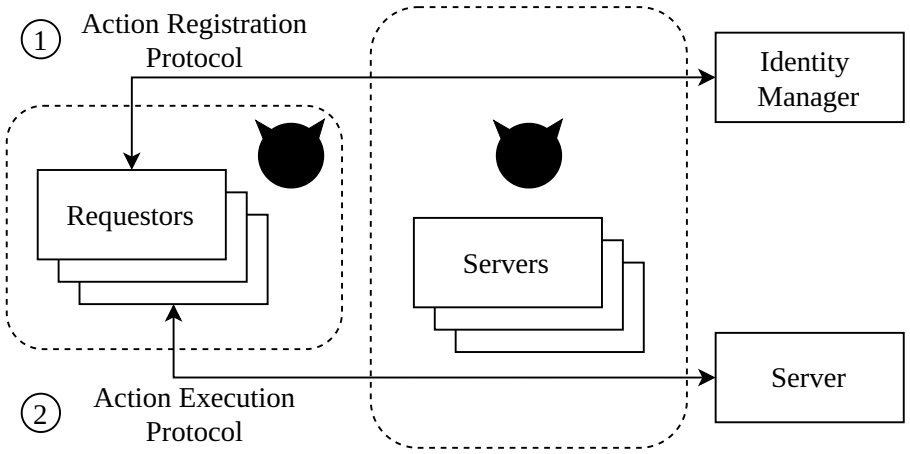


Figure 6.3: An overview of the system and adversary model for *ActionID*. The system model consists of requestors, servers, and the trusted identity manager. The adversary model consists of a malicious requestor and an external adversary. Both adversaries have full control of the communication channel.

for organisations to be aware of who is doing what. Nonetheless, should a more privacy-preserving solution be required, we discuss more on this in Section 9.2.3.

6.2.2 System Model

As depicted in Figure 6.3, our system model consists of three entities: the requestors, the identity manager, and the servers. Each entity possesses a public-private key pair as their long-term credential and has a copy of the public key of the identity manager. All entities communicate in a M2M manner without any human interaction.

In addition to its actions, a requestor possesses a certificate signed by the identity manager and the corresponding private key. The certificate shows that the requestor has previously been registered by the identity manager without the identity manager having to explicitly remember the public key of a requestor.

Each server is previously registered with the identity manager and a copy of the public key of each server is stored by the identity manager. A server is associated with its action policies and trusts that the identity manager enforces the server's policies when issuing a token to a requestor. Only when it is offering its services to a requestor should a server be online.

The identity manager is always online. In addition to managing the public key of each server, the identity manager stores and enforces all action policies of all servers.

We do not restrict the implementation details of such policies, e.g., the encoding or the format. Our only requirement is that the identity manager understands the policies' semantics to perform checks for the compliance of a requestor's actions with the policies of the requested server.

6.2.3 Adversary Model

Also depicted in Figure 6.3, our adversary model consists of two adversaries: an internal adversary called a malicious requestor, and an external adversary. We model both adversaries as Dolev-Yao adversaries (Dolev and Yao, 1983) and we assume that the adversaries are incapable of breaking the underlying cryptographic primitives. We also do not consider cases involving trivial Denial-of-Service (DoS) attacks where the adversaries simply drop all communicated messages.

A malicious requestor additionally possesses its own legitimate certificate issued by the identity manager with the corresponding private key. A malicious requestor can also compromise other requestors and has access to their credentials, i.e., their private keys. A malicious requestor has two goals. First, to access a server with actions not already authorised to compromised requestors under its control. The second is for a server to execute unregistered actions.

We further allow an external adversary to collude with other servers in the system where it has access to the private keys of these servers. The only exception to this is the private key of an honest server who is contacted by a requestor for action execution. An external adversary has two goals. The first is to impersonate as either the identity manager or an honest server to a requestor. The second is to break the confidentiality of a requestor's actions.

6.3 Protocols Description

We introduce two protocols for *ActionID* to illustrate an example of utilising this novel idea of a machine identity associated with its actions, namely, the Action Registration Protocol and the Action Execution Protocol. A requestor runs both protocols in sequence for every action execution. We further assume the

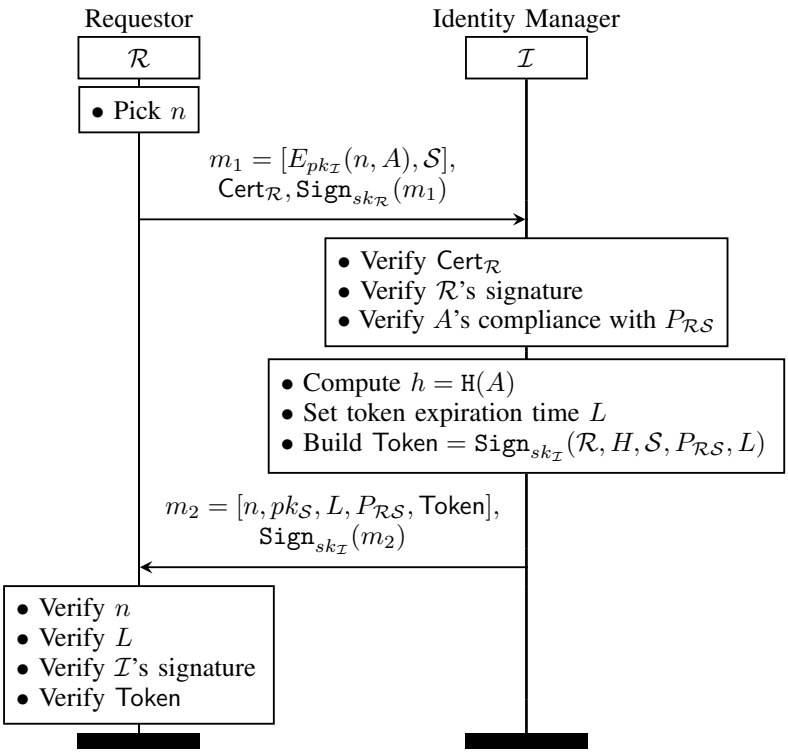


Figure 6.4: The Action Registration Protocol involving a requestor $\mathcal{R}$ and the identity manager $\mathcal{I}$. The requestor initiates the protocol to register its actions and to obtain a token from the identity manager. The token is required for the later execution of the requestor's actions. ',' is used for concatenation.

cryptographic primitives used are secure as per their own security definitions in Section 2.1.

6.3.1 Action Registration Protocol

Figure 6.4 depicts the Action Registration Protocol. A requestor runs this protocol to register its actions to the identity manager and to receive a fresh token on its registered actions.

A requestor initiates the protocol by sending message m_1 containing a nonce n, its actions A, and the identifier of its intended server $\mathcal{S}$. The nonce and the actions are encrypted to, respectively, serve as a liveness challenge for the identity manager and ensure the confidentiality of the actions. Message m_1 is sent together with the requestor's certificate and the requestor's signature on the message.

After verifying the certificate and the signature, the identity manager checks if the requestor's actions comply with the action policies of the requested server,

denoted as $P_{\mathcal{RS}}$. We give an example of how the identity manager completes this task in more detail in Section 6.5.2. If so, the identity manager proceeds to generate a token by first hashing the actions and setting an expiration time for the token. The identity manager then hashes five data items, namely, the expiration time of the token L, the policies of the server $P_{\mathcal{RS}}$, the hashed actions h, and the identifiers of the requestor $\mathcal{R}$ and the server $\mathcal{S}$, and signs the resulting hash to construct the token.

The identity manager further hashes, signs, and sends a reply message m_2 to the requestor, which encompasses the token and the information needed to verify the token, along with the requestor's nonce and the public key of the server. The requestor subsequently verifies the nonce, the expiration time of the token, the signature of the identity manager on message m_2, and the token.

The token is stored by the requestor for action execution and it will not have to register the same actions if the token has not expired. The requestor can simply discard an expired token and request a new token by repeating the Action Registration Protocol.

6.3.2 Action Execution Protocol

Figure 6.5 illustrates the Action Execution Protocol. The purpose for this protocol is for the requestor's registered actions to be executed by the server. The action execution is further secured by the secure channel established at the end of the handshake of the protocol.

The protocol starts with a handshake where the requestor first commits a nonce n_1 and the hash of its actions to the server. These are sent along with the token from the Action Registration Protocol and the requestor's certificate for the server to verify. To prevent offline guessing attacks against the requestor's actions, the hash of the actions is encrypted.

The server first verifies the token and the certificate. If they are valid, the server encrypts and commits another nonce n_2 to the requestor. The server then signs the ciphertext and nonce n_1 to prove the server's liveness to the requestor,

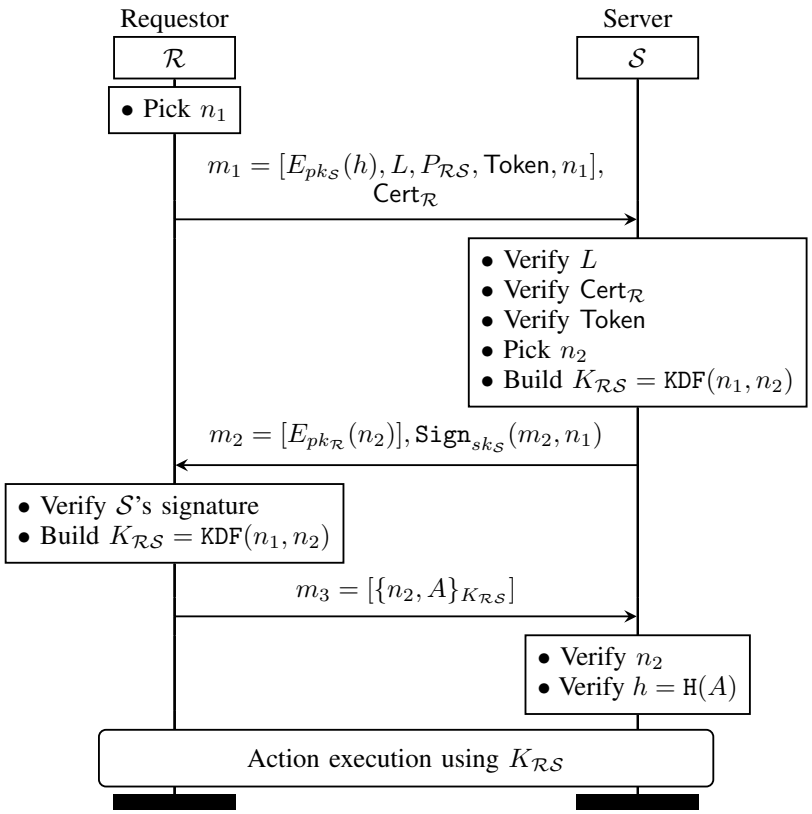


Figure 6.5: The Action Execution Protocol between a requestor $\mathcal{R}$ and a server $\mathcal{S}$. The requestor initiates the handshake where a fresh session key is agreed. The session key is used for establishing a secure channel for the exchange of subsequent messages during the session resulting from the execution of actions. ',' is used for concatenation.

and then computes a fresh session key using a key derivation function $\mathtt{KDF}(\cdot, \cdot)$ with the two nonces as input.

After verifying the signature and decrypting nonce n_2, the requestor computes the session key in the same way as the server. The requestor then reveals its actions with nonce n_2 to the server, both of which are encrypted with the freshly established session key. The server verifies both values and if the verifications passed, this guarantees the server two things: the liveness of the requestor, and that the revealed actions are the same as the registered actions.

Once the handshake is completed, the server executes the verified actions of the requestor. Depending on the application, the session could involve exchanging further messages between the requestor and the server. To preserve the confidentiality and integrity of the session messages, they are to be secured with the established

session key, which forms a secure channel between the requestor and the server. The protocol ends when the session is terminated.

6.4 Security Analysis

We now provide the security guarantees for *ActionID* with respect to the design goals in Section 6.1.2 and we prove these security guarantees by contradiction. For each proof we first exhaustively enumerate the scenarios in which an adversary, either a malicious requestor or an external adversary, can choose to break the guarantee, and prove that all scenarios contradict our assumptions. We emphasise that the underlying cryptographic primitives used in the protocols as described in Section 6.3 are secure as per the definitions in Section 2.1.

Guarantee 6.1 (Mitigate Implications of Credential Leakage)**.** *A malicious requestor in possession of private keys of requestors can get tokens only for actions authorised to the victim requestors according to the action policies and can execute the actions only for a limited period of time.*

Proof. For a malicious requestor to break this guarantee, the adversary must possess a token without a restricted set of actions, i.e., without complying with the action policies, or with no expiration time. There are three ways a malicious requestor can achieve this. (1) First, the adversary can attempt to manipulate the activity of the identity manager to not follow the Action Registration Protocol for issuing tokens. However, in successfully doing so, this breaks our system model where the identity manager is trusted. (2) Second, the adversary can compromise the private key of a requestor whose set of authorised actions contain the adversary's chosen actions, and use that credential to get a token. This allows the adversary to register new actions and continuously renew tokens, and thus the guarantee may seem to be broken. However, as the token is issued only when the chosen actions comply with the policies for the newly compromised requestor, this means the adversary still has to choose from a restricted set of actions, and so the guarantee still stands. (3) Third, to have complete control over the token, the adversary can forge the

token for any arbitrarily chosen actions or expiration time of the token. However, this is only possible with a negligible probability under our assumption of secure cryptographic primitives, specifically under a EUF-CMA secure digital signature scheme.

□

Guarantee 6.2 (Bind Identity to Actions)**.** *A requestor must first register its actions, and once registered, the requestor cannot change the registered actions.*

Proof. A malicious requestor can break this guarantee if (1) it executes the actions without registering, or (2) by changing the registered actions. (1) To successfully execute actions without registration, a malicious requestor needs to present a token for message m_1 in the Action Execution Protocol. The adversary can either forge or replay a token. Forging a token is impossible for the reasons discussed in Guarantee 6.1. If the adversary replays a token, this means that the list of actions associated with the token has already been previously registered. Thus, the first part of the guarantee holds. (2) Even after registering, a malicious requestor might still attempt to cheat by revealing actions that are different from the registered actions in message m_3 of the Action Execution Protocol. To bypass a server's verification for the modified actions, the modified actions have to produce the same hash as the registered actions, which is equivalent to searching for a hash collision. This is only possible with a negligible probability as we require the assumption that secure cryptographic primitives are used, and in this case, a collision-resistant hash function as stated in Section 2.1.5. Thus the second part of the guarantee holds. □

Guarantee 6.3 (Liveness and Secure Channel Establishment)**.** *A requestor and a server mutually check each other for liveness and subsequently establish a fresh session key known only to them for action execution.*

Proof. For an external adversary to break this guarantee, the adversary must establish the session key in the Action Execution Protocol when a requestor contacts its intended server. The adversary has three options: (1) impersonate a server

contacted by a requestor, (2) impersonate the identity manager to a requestor, or (3) impersonate a requestor to an honest server. (1) In the first case where the adversary impersonates an honest server, the adversary has to convince the requestor by forging a valid signature from the honest server in message m_2 of the Action Execution Protocol. (2) The adversary can circumvent this requirement in the second case. The adversary modifies message m_2 of the Action Registration Protocol to change the honest server's public key with the adversary's own public key, allowing the adversary to generate the signature in message m_2 in the Action Execution Protocol with the adversary's own private key. In this case, however, the adversary has to forge the identity manager's signature on message m_2 of the Action Registration Protocol and impersonate the identity manager. The adversary can only succeed in both cases, i.e., case (1) and case (2) only with a negligible probability due to our assumption of secure cryptographic primitives, specifically a EUF-CMA secure signature scheme. (3) In the third case where the adversary impersonates a requestor to an honest server, the adversary has to decrypt the challenge n_2 issued by the honest server in message m_2 of the Action Execution Protocol, which requires the adversary to possess the requestor's private key for the adversary to succeed. The adversary can only either break the underlying encryption scheme, or guess the private key (or the challenge nonce). The former contradicts our assumption of secure cryptographic primitives, specifically a CCA secure encryption scheme as previously stated in Section 2.1.1, and with such an assumption the latter can only occur with a negligible probability. □

Guarantee 6.4 (Confidentiality of Actions)**.** *A requestor's actions must be known only to the identity manager for registration, and only to the intended server for execution.*

Proof. Breaking this guarantee means an external adversary can retrieve a requestor's actions. Since actions (and the corresponding hash) are encrypted in both protocols, the adversary has two methods of achieving this: (1) decrypting the encrypted actions and subsequently launch offline guessing attacks, or (2) deriving

the session key. (1) For the first method, the adversary must decrypt the ciphertext in message m_1 from one of the two protocols. If the adversary chooses to decrypt the encrypted hash from message m_1 of the Action Execution Protocol, the hash is brute-forced to further guess the requestor's actions in an offline manner. However, the first method requires the adversary to be in possession of the private key of the identity manager or the server, and this is only possible if the adversary either breaks the underlying CCA secure encryption scheme or guesses the private key. As we have seen before, the former violates our assumption that secure cryptographic primitives are used while the probability of the latter happening is negligible. (2) For the second method, the adversary must decrypt the actions encrypted in message m_3 of the Action Execution Protocol by deriving the session key. The adversary successfully deriving the session key leads to the violation of our assumptions on secure cryptographic primitives. More precisely, a secure key derivation function or a CCA secure encryption scheme as stated in Section 2.1.6 and Section 2.1.1 respectively. □

6.5 Implementation

To demonstrate the practicality of our scheme, we provide an implementation as a proof-of-concept for *ActionID*. Based on the implementation, we conduct experiments to test the performance of *ActionID* against a well-known protocol: SSH. This section covers the findings for the implementation as well as the experiments.

6.5.1 Overview

We implement the proof-of-concept for *ActionID* in Python and it comprises of two parts: a Python library, `actionid.py`, and a working example for `actionid.py`. Our implementation imagines an application scenario where a requestor routinely uploads and retrieves data to and fro a SQL server. Figure 6.6 illustrates the interactions of the four main functions provided in `actionid.py`. A requestor registers and executes actions via `register_actions` and `access_service` respectively. The identity manager completes the Action Registration Protocol and

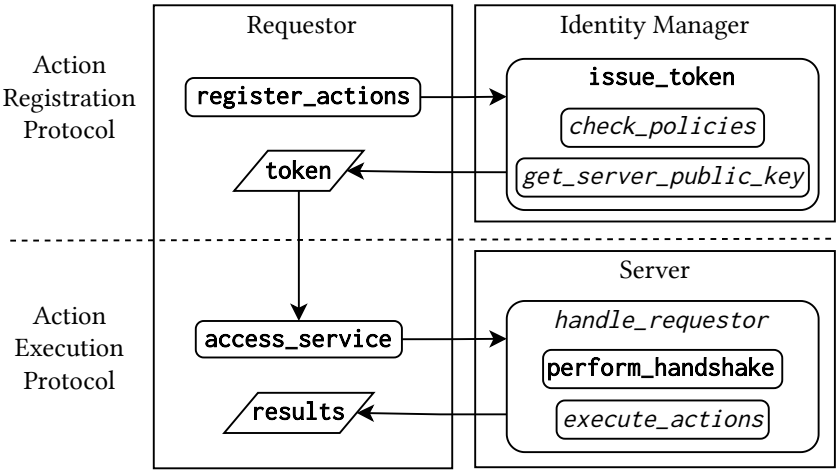


Figure 6.6: An overview of the implementation for *ActionID*. Rounded-corner rectangles are functions with the function's name as the label. Function names in bold are functions provided in `actionid.py` whereas function names in italics are application-specific functions.

```
{
    "any":
    {
        "blocked_actions": ["DROP", "DELETE"]
    }
}
```

Figure 6.7: The action policy encoded in the JSON format for the proof-of-concept. The policy specifies the SQL actions not permitted for requestors under the user group `any`.

issues tokens via `issue_token`, whereas the server performs the handshake of the Action Execution Protocol via `perform_handshake`. To accommodate for various application needs, we further demonstrate that `actionid.py` works with application-specific functions. In our implementation, these functions are `check_policies`, `get_server_public_key`, `handle_requestor`, and `execute_actions`. Our proof-of-concept is available on GitHub[1] for interested readers.

6.5.2 Identity Manager

We provide a more concrete idea on how the identity manager fulfills its responsibilities in the Action Registration Protocol, mainly, how the identity manager (a) checks the compliance of a requestor's actions with the server's action policies

[1] `https://github.com/wilteng/actionid`

(`check_policies`) and (b) stores the public key of servers. (a) The main responsibility of the identity manager, i.e., analysing the compliance of the requestor's actions, requires the consideration of three design choices: (i) the encoding of the server's action policies, (ii) the encoding of the requestor's actions, and (iii) the mechanism for analysing the compliance of the requestor's actions against the action policies. (i) We choose the encoding of a server's action policies to be in the JSON format and as shown in Figure 6.7, it represents a blocklist of the SQL actions that the requestors under the user group `any` are prohibited to execute. (ii) As the server is a SQL database server, the requestor's actions are thus encoded in SQL and should be comprised of syntactically valid SQL statements. (iii) Finally, for the identity manager to analyse a requestor's actions so as to enforce the action policies, the identity manager simply searches for the presence of the prohibited SQL keywords in the string of the requestor's actions, i.e., SQL statements. We acknowledge that static analysis of arbitrary statements for a Turing-complete programming language is undecidable in general. However, as our implementation shows, static analysis with the purpose of access control can be done by methods as simple as searching for forbidden strings. Indeed, static analysis of a requestor's actions can further be implemented using other more complicated parsing methods. (b) If the actions of a requestor comply with the policies, the identity manager computes the token and subsequently retrieves the public key of the requested server (`get_server_public_key`) from its database. Our implementation uses a SQL database for the identity manager for storing and managing the public key of the servers.

6.5.3 Server

We give a more detailed explanation here on how a server handles a connection from a requestor in the Action Execution Protocol. A server first performs the handshake (`perform_handshake`) with a requestor. After a successful handshake, the server receives the registered actions from the requestor and proceeds to execute the actions (`execute_actions`). Similarly, the encoding of the requestor's actions remains an

```
import actionid

# Network configurations
IDM_IP = "127.0.0.1"
IDM_PORT = 8081
SERVER_IP = "127.0.0.1"
SERVER_PORT = 8082
server_id = '81258728'

# Execute actions for data pull
actions =
"""
SELECT * FROM fruits;
"""
token = actionid.register_actions(IDM_IP, IDM_PORT, server_id, actions)
results = actionid.access_service(SERVER_IP, SERVER_PORT, actons, token)
print('\n"fruits" table:' + results)

# Execute actions for data push and pull
actions =
"""
INSERT INTO fruits ("fruit", "colour") VALUES ("grapes", "purple");
SELECT * FROM fruits;
"""
token = actionid.register_actions(IDM_IP, IDM_PORT, server_id, actions)
results = actionid.access_service(SERVER_IP, SERVER_PORT, actions, token)
print('\n"fruits" table:' + results)
```

Figure 6.8: An example of the requestor's script that uses `actionid.py`. The requestor's script calls the `register_actions` and `access_service` functions from `actionid.py` to retrieve from and insert to the database of the SQL server.

important design choice for the server's implementation, as it should recognise and understand the semantics of the requestor's actions to execute them. Since the server is an SQL server in our implementation, the encoding of the requestor's actions should obviously be syntactically valid SQL statements. The action execution ends after the results have been returned to the requestor encrypted with the session key and appended with a Message Authentication Code (MAC). Depending on the application, the action execution can be implemented in alternative ways. For example, both the requestor and the server can choose to keep the connection alive and continue exchanging messages after the handshake. The confidentiality and the integrity of subsequent messages exchanged during this time can still be secured with the session key.

6.5.4 Requestor's Script

Figure 6.8 shows a concrete working example of a requestor's Python script in our implementation. The script shows the requestor completing two sets of

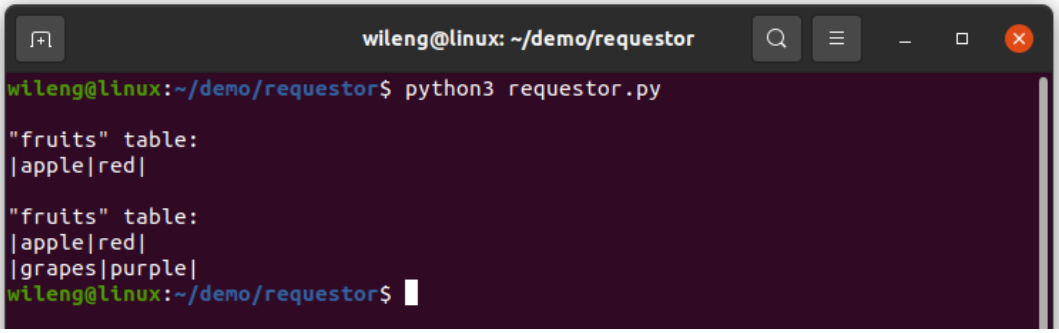


Figure 6.9: Output of the `requestor.py` script that shows the proof-of-concept for *ActionID* successfully executing the SQL actions requested by the requestor.

SQL operations. The result from running the requestor's script is shown in Figure 6.9, which implies a successful execution. Note that since the credential management of the requestor is implicitly handled by *ActionID*, the requestor's script does not specify any kind of credentials, e.g., through statements like `password = "password123"`. This reduces unintentional credentials leakage through the requestor's script, which aligns with our motivation for this chapter. Each task of the requestor only requires three lines of Python statements. As colour-coded in Figure 6.8, the requestor first defines the actions (red statements), then the requestor registers its actions to get a token (blue statements), and lastly the requestor executes its actions (purple statements).

6.5.5 Performance Overhead

To test for the feasibility and performance of the *ActionID*, the implementation is compared with the Secure Shell (SSH) protocol (Ylonen, 2006a) as a performance benchmark. Our experiments measure and compare the performance of the requestors to complete the two protocols of *ActionID* versus the performance of a SSH client to transfer its actions over the SSH protocol.

We conduct the experiments in a network-controlled environment consisting of two individual machines where one acts as a requestor, and another acts as both the identity manager and the server. The requestor machine is equipped with an Intel® Core™ i7-9750H processor and the Windows 10 operating system. The identity

manager/server machine is equipeed with a processor of Intel® Core™ i5-1145G7 and the Ubuntu 20.04 operating system. The processing time for each requestor to complete both protocols is measured. Similarly, the experiments for testing the performance of the SSH protocol are set up in the same network environment using the same machines. The Windows machine acts as a SSH client and the Ubuntu machine acts as a SSH server. The performance of the SSH protocol is similarly measured by the execution time for a SSH client to connect to the SSH server via the `ssh` command.

For interested readers and results reproducibility, we list the parameters and the algorithms of the cryptographic primitives used in the implementation and the experiments. The cryptographic primitives are from the `pyca/cryptography` Python library version 37.0.4. The hash function used is SHA-256, and for asymmetric cryptographic algorithms, RSA with a key size of 2048 bits and an exponent of 65537 is used. For symmetric key algorithms, the block cipher AES-128 in Cipher Block Chaining (CBC) mode[2] is used with a key size of 128 bits and for MAC generation, a Hash-based MAC (HMAC) with a key size of 256 bits is used. The derivation of the session key in the Action Execution Protocol (see Section 6.3.2) is realised using the Concatenation Key Derivation Function (ConcatKDFHash) with the seed being the hash of the two nonces, n_1 and n_2. For experiments involving SSH connections, the authentication of SSH connections is made through a RSA keypair with a key size of 2048 bits generated with the `ssh-keygen` command during experiment setup. While some cryptographic primitives used here are deprecated, such as ConcatKDFHash by Barker et al. (2013) and AES-CBC due to attacks by Irazoqui et al. (2015), we note that the implementation serves only as a proof-of-concept to test the performance of *ActionID*. The cryptographic primitives and the implementation can be readily updated for real-world deployment.

[2]In CBC mode, each plaintext block is operated with the previous ciphertext block using an operation like Exclusive-OR (XOR), where the first block is an initialisation vector (IV). This makes each ciphertext dependent on the previous ones, and any modification of a ciphertext block affects the decryption of that block and the next one(s).

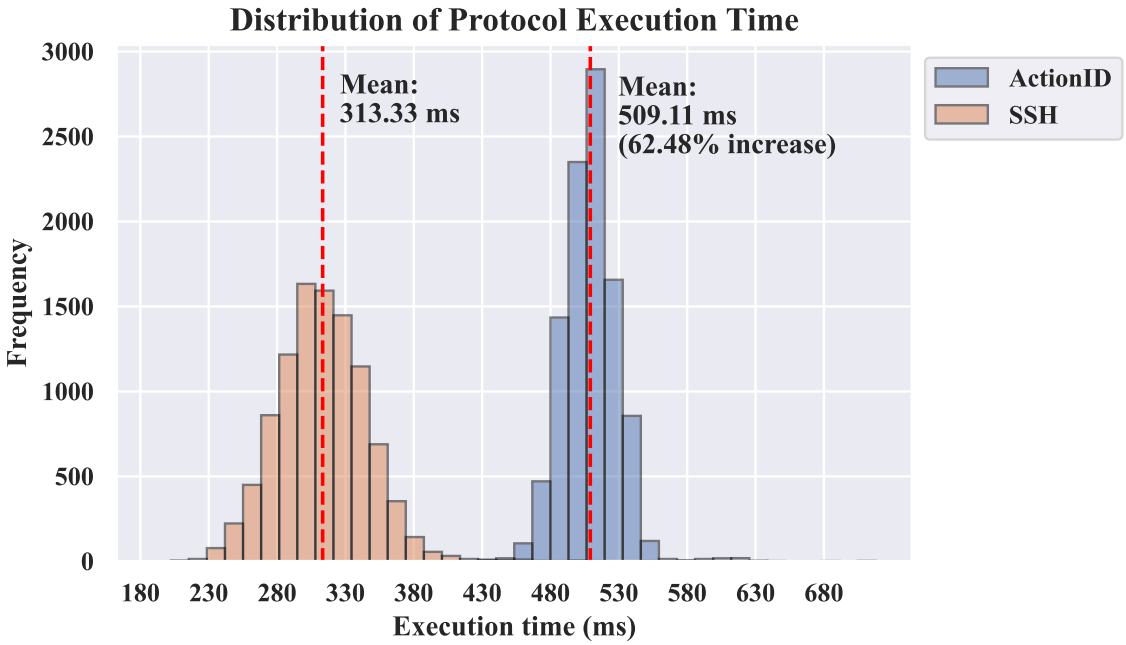


Figure 6.10: The distribution of the time taken by 10000 requestors to execute the two protocols of *ActionID* and the distribution of the time taken by 10000 SSH clients to connect and transmit its actions to a SSH server. The experimental results shows that *ActionID* has a comparable performance with the SSH protocol.

The experiment results are visualised in Figure 6.10. The figure depicts the distribution of the protocol execution time for 10000 requestors and 10000 SSH clients. Note that the generated data take into account the latency of the network but not the actual execution of a requestor's actions, so only the performance of the protocols are tested. The results show that even as a proof-of-concept, the performance of *ActionID* remains comparable and on the same order of magnitude (only 66% overhead, which is within a factor of 10) as the mature and optimised SSH protocol, in spite of *ActionID* requiring more expensive cryptographic primitives, i.e., signature generation and verification.

6.6 Further Considerations

In this section, we discuss two important details pertaining to the design of *ActionID*, namely, the requestor's actions analysis mechanism and the mechanism for token revocation.

6.6.1 Actions Analysis Mechanism

For the identity manager to issue a token to the requestor in the Action Registration Protocol, the compliance of the requestor's actions is required to be checked by the identity manager against the access control policies of the requested server. However, static analysis of arbitrary statements for a Turing-complete programming language is undecidable in general. Even so, analysis of the requestor's actions for the purpose of access control can be implemented in practice by using a blacklist (or whitelist) of commands and objects that the requestors are not allowed to be accessed. By using a blacklist, the identity manager can search for any keyword that is mentioned in the requestor's actions in which the requestor has no permission to access. For example, a requestor's actions can only request for read-only operations and access objects associated with certain identifiers. As shown by the proof-of-concept for the scheme, this proves that the implementation of an actions analysis mechanism is possible.

6.6.2 Token Revocation Mechanism

Short-lived tokens are issued by the identity manager to the requestors for the service access on servers without having the servers to directly contact the identity manager. However, there might be situations where a token issued to a requestor has to be revoked even before the short expiration time for the token is reached for cases such as when a critical yet compromised requestor is involved. In view of this, it is possible to set up a token revocation mechanism similar to Online Certificate Status Protocol (OCSP) (Ekechukwu et al., 2013) where the identity manager publishes the status of each token issued to requestors. This enables a server to check the revocation status of a token for a requestor before allowing the requestor's access for the server's service. Nevertheless, the implementation of such a token revocation mechanism would incur additional communication overhead to both the server and the identity manager as additional messages between the server and the identity manager are needed for every session of requestor's actions execution, which would also directly affect the time needed to complete a session of the Actions Execution Protocol for the requestor and the server. For these

reasons, a token revocation mechanism based on timing is sufficient to achieve the current goals and security guarantees of *ActionID*.

6.7 Chapter Summary

The identity of a machine in the context of protocol execution has traditionally been associated with just the credentials of that machine, i.e., knowledge of a private key, or more generally a secret string. However, this means that anyone obtaining these secrets is able to assume the identity of that machine.

ActionID is a scheme that extends the idea of the identity of a machine to the actions that a machine can take when accessing a remote service in an organisation's network. We have shown that with a trusted third party, we can bind the long-term identity of a machine with a list of actions that allows us to create a "dynamic identity" which limits the otherwise significant consequences of a theft of credentials. It prevents an adversary with stolen credentials from performing arbitrary actions on the server where those credentials are used. It forces the adversary to always declare and register its intended actions to get a service access token, which helps to increase the chances of detection of stolen credentials and their identification.

The use of a trusted third party in *ActionID* also provides a mechanism for the management of user accounts and access control policies of an arbitrary number of clients and servers. These are centrally stored and enforced thereby reducing the chances of the mismanagement of policies, e.g., forgetting to update a particular machine, which, in practice contributes enormously towards strengthening the operational security in an organisation.

Our implementation is in the form of a Python library. We use it to show the ease of integration into existing applications, as well as to demonstrate the practical performance of the scheme when we compare *ActionID* to a mature and optimised protocol such as SSH.

7

Improving Key Authentication in the Signal Protocol

Contents

Instant messaging applications are used extensively for day-to-day communication and have been well-integrated into our everyday lives. In this context,

the Signal protocol is widely regarded as the de facto standard for end-to-end encrypted messages, and has been serving as the underlying protocol for popular instant messaging applications such as Signal's own instant messaging application, WhatsApp (2023), and Facebook Messenger (Facebook, Inc., 2017). In this regard, WhatsApp alone has nearly 3 billion users worldwide in 2023 (Ceci, 2023). One "key" problem (quite literally) in the Signal protocol is key authentication, which is to make sure that the messaging keys used to send messages in the protocol actually belongs to a specific user. The current method of authentication is for the communicating pair of users to compare the fingerprints of their keys in an out-of-band fashion and this requires the active participation of the users. However, usability studies (NEAL, 2017; Vaziripour, Wu, O'Neill, Whitehead, et al., 2017; Vaziripour, Wu, O'Neill, Metro, et al., 2018; Schröder et al., 2016) have shown that the awareness of users of participating in this procedure is quite low. Furthermore, because the user keys change every time a key-ratcheting step is performed, the users would have to manually authenticate their keys frequently which is unlikely to ever be done in practice. While some recent work (Dowling and Hale, 2021; Dowling, Günther, et al., 2022; Barooti et al., 2023) improve on this out-of-band authentication procedure, the proposed solutions ultimately still rely on interaction by users.

At the same time, news of governments coercing companies to disclose user data are getting increasingly common, as shown by the list of government requests received by Signal (2023a). The European Union is currently in the process of passing a new legal framework eIDAS (European Commission, 2023) and Article 45 of eIDAS compels Internet browsers to trust root certificates issued by government chosen certificate authorities (Hoffman-Andrews, 2023; mozilla, 2023). In response, an open letter by global cybersecurity experts (eIDAS 2023 Academics and Civil Society, 2023) condemns that Article 45 "radically expands the ability of governments to surveil both their own citizens and residents across the EU by providing them with the technical means to intercept encrypted web traffic". In the United Kingdom, the Online Safety Act 2023 (UK Parliament, 2023) that allows user messages to be monitored (Guest, 2023) has also received criticisms from cybersecurity experts. The

open letters condemn the law as being able to "undermine end-to-end encryption" (Polk, 2022) by "creating a new power to compel online intermediaries to use *accredited technologies* to conduct mass scanning and surveillance of all citizens on private messaging channels" (Johnson et al., 2022). These policies have the potential domino effect that organisations such as Internet Service Providers and business companies could be lawfully obliged to act as a Man-in-the-Middle (MitM) and break end-to-end encryption to tap into the private communication channels of their users.

In light of these issues, we turn to the existing Signal protocol and build improvements on the key authentication of the protocol as a novel solution for these extreme situations to automate the process of key authentication and to detect active MitM attacks without relying on the interaction of the users. We go a step further by considering a stronger adversary, such as a malicious organisation, that does not only has the power to control the communication channels and act as an active MitM, but also has the ability to clone the state of a client device, thus gaining access to all the client's secret key material, including both long-term and ephemeral secrets.

Much of the existing work on Signal consider a fully untrusted server. This assumption however is often not entirely true in practice as messaging applications are unusable when the client devices are not connected to the server. This means that the server is at a minimum trusted to relay the envelopes between participants and to handle the "correct" key distribution in order for end-to-end encryption to be effective. Our solution makes use of the ever present server to be able to automatically detect the presence of a powerful active MitM in band with the existing communication channels. We do so while also maintaining the existing security properties of the Signal protocol, namely, end-to-end encryption (and authentication) of user messages, perfect forward secrecy, post-compromise security. We summarise our main contributions as follow:

- We propose additions to the Signal protocol that make it resistant to *active* MitM attacks (as opposed to only passive) without requiring user intervention. Our solution will automatically detect the presence of an active MitM attacker

even though he has access to a snapshot of all secrets of a client device, while preserving the existing security properties of the Signal protocol.

- We analyse our new protocol with respect to both the new security properties we provide, and the existing properties the basic Signal protocol provides.
- We implement a proof-of-concept library in Rust of our modified Signal protocol based on the open-source Signal protocol library to demonstrate the practicality of our solution.
- We run experiments and measure the performance characteristics to benchmark our solution against the original Signal protocol.

7.1 System and Adversary Model

In this section we present our system model and adversary model, depicted in Figure 7.1. These include our assumptions regarding the participants of the system, as well as the capabilities of the adversary and its goals. We later prove that our protocols make sure the goals of the adversary are not achieved according to the guarantees provided by our protocols.

7.1.1 System Model

Identical to the underlying communication model for the Signal protocol, our system model represents a typical device-to-device communication model. The system model consists of three parties, a sender (A), a recipient (B), and a server (S). A message (an envelope) sent from the sender to the recipient, will always go via the server, unless dropped by an adversary. While the Signal protocol is often presented as if the server is fully untrusted, we argue that in practice the Signal server is in fact honest-but-curious, since it is expected to forward messages between participants and distribute the correct keys, i.e., not act as a Man-in-the-Middle (MitM), in order to guarantee end-to-end encryption. We assume that the server correctly distributes client keys, computes the required hash values in our modified Signal protocol, and takes action to end the connection if verification fails (as described in

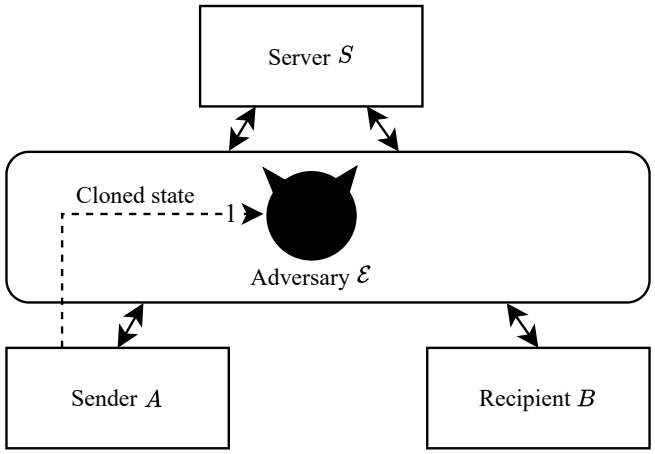


Figure 7.1: Our system and adversary model. The system model is similar to the communication model of the Signal protocol, and it consists of three parties, a sender (A) and a recipient (B), communicating via an honest server S that forwards the envelopes. The adversary model consists of adversary $\mathcal{E}$ that is capable of manipulating the messages on the communication channels between the sender and the server, and between the server and the recipient. The adversary has the additional ability to clone the state of the sender at most one time, thereby accessing the secrets of the sender.

Section 7.2). Beyond this there are no trust assumptions on the server, specifically the server will never be able to access un-encrypted messages.

We assume that the communication channels between the clients and the server are protected with Transport Layer Security (TLS) (Dierks and Rescorla, 2008) (as is often the case in practice). The server is always online, whereas the sender and the recipient need not be. If the recipient is offline, the server stores the message until the message is fetched by the recipient. The sender and the recipient are required to be online when sending a message, or when fetching a message from the server. We further assume that the sender and the recipient can securely delete the ephemeral secrets when such deletions are required as specified by the Signal protocol.

7.1.2 Adversary Model

We consider a powerful adversary $\mathcal{E}$ capable of acting as a MitM to observe the messages exchanged in the protocol. We assume that the adversary has full control of both channels, i.e., the channel between a sender and the server, and the channel between the server and a recipient. We assume that the underlying cryptographic

primitives are secure, and that the adversary acts as a Dolev-Yao attacker, meaning that the adversary can replay, drop, and modify messages on the two channels.

We allow the adversary to act any time after the end of the Registration Protocol. We give the adversary the ability to clone the state of the sender at most once, i.e., without modifying the existing state of the sender. This means that the adversary has access to the secrets of the sender A at the time epoch of compromise, which we denote τ. Specifically, we allow the adversary to obtain the tuple $(x_{i_A}, x_{s_A}, x_{o_A}, x_{b_A}, x_A^\tau, rk^\tau, ck_n^\tau, mk_n^\tau)$, details of which have been explained in Section 2.6.2. Recall that the communication enters a new time epoch whenever an asymmetric ratcheting is completed, i.e., whenever the roles of the sender and the recipient are switched.

The four goals of the adversary are: (1) to remain undetected in the system after sending envelopes impersonating the sender, (2) to retrieve the content of past envelopes at time epochs before τ, (3) to retrieve content of future envelopes sent by the sender A after τ when the sender is eventually online, and (4) to break the confidentiality and authentication of the envelope content in uncompromised epochs t such that $t < \tau$ (before client cloning where perfect forward secrecy is applied) and $t > \tau+1$ (after post-compromise security has established) without cloning the sender.

We remark that while this is a very strong adversary model, perhaps stronger than most practical attackers, and certainly stronger than the models in some previous works (Cremers, Fairoze, et al., 2020; Cremers, Jacomme, et al., 2023; Dowling, Günther, et al., 2022; Barooti et al., 2023) as discussed in Section 3.4.2. Such an adversary could exist in practice and we believe their capabilities should be accounted for. One example of such an adversary could be a company that has installed CA certificates on their employees laptops to perform https deep-packet-inspection (Sherry et al., 2015). The adversary would be able to bypass the protections afforded by TLS and also have an opportunity to clone the client. While arbitrary-time and repeated cloning are possible with such an adversary, we assume that cloning occurs only after the completion of the Registration protocol. This aligns with the standard assumption in cryptographic protocol design that

the adversary is not present during the registration phase to bootstrap the system securely and to provide meaningful security guarantees. We further discuss the impact on the security guarantees of the system if the adversary clones the client repeatedly in Section 8.1.

7.2 Protocol Description

This section presents our changes to the Signal protocol as our novel solution. We first give a high-level overview on how the different protocols in our solution interact with one another throughout the different phases of the original Signal protocol. We then describe each protocol in detail using the algorithms introduced in Section 2.6.2 where we represent each protocol in a message sequence chart, and our changes are highlighted in blue.

7.2.1 Overview

An overview of our solution is shown in Figure 7.2 where our changes are applied at the start of the communication, i.e., in the phases of registration and session establishment. Note that the sequence of registration for the initiator and the responder is insignificant. The Envelope Sending Protocol and the Envelope Fetching Protocol are used to respectively send and receive the first envelope during session establishment (and the later asymmetric ratcheting). Recall from Section 2.6.1 that the established session is extended indefinitely by asymmetric ratcheting, i.e., when the initiator and the responder switch the role of the sender and the recipient of an envelope. The Envelope Sending Protocol and the Envelope Fetching Protocol are similarly applied for the first envelope during asymmetric ratcheting.

Notice that the phases of session establishment and asymmetric ratcheting in our solution are different from the original Signal Protocol. In our solution, the procedure for session establishment and asymmetric ratcheting are unified into one pair of sending and fetching protocols where both phases require the execution of the Envelope Sending Protocol and the Envelope Fetching Protocol. However, the protocol to run session establishment and asymmetric ratcheting

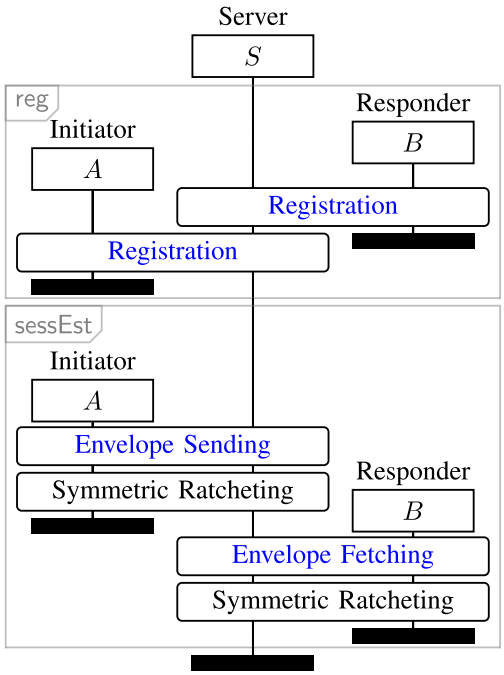


Figure 7.2: An overview of the phases of registration (reg) and session establishment (sessionEst) integrated with our solution. Our modifications are applied in the Registration Protocol, and the first envelope sent and received in the session establishment (and subsequent asymmetric ratcheting) using the Envelope Sending Protocol and the Envelope Fetching protocol respectively. Symmetric ratcheting proceeds identically with the original Signal protocol. Recall from Section 7.1 that due to the asynchronicity of Signal, the communicating pair of clients are not required to be online at the same time.

in Signal are two different protocols since session establishment requires a client to first fetch the prekey bundle of its communicating partner from the server, while asymmetric ratcheting does not.

As the symmetric ratcheting phase of the Signal protocol does not involve the exchange of new key materials, we do not require modification of the protocol in this phase. The symmetric ratcheting phase in our solution is identical with that in the original Signal protocol in which the sender sends subsequent envelopes to the server directly and the server then stores the envelopes until they are fetched by the recipient.

7.2.2 Registration Protocol

Figure 7.3 illustrates the Registration Protocol for a client A to commit its prekey bundle to the server in the registration phase of the Signal protocol. The client

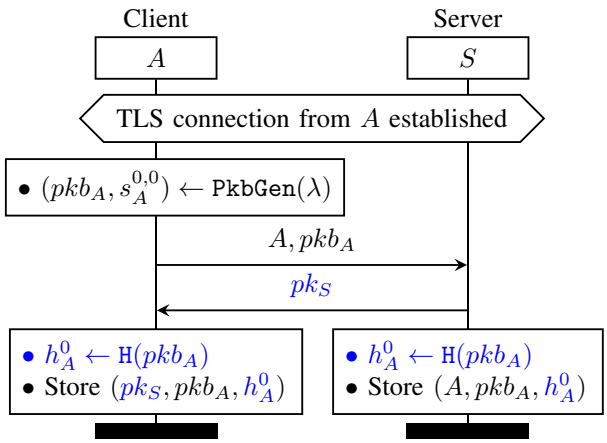


Figure 7.3: The Registration Protocol where a client A generates and sends its prekey bundle to the server S. The client and the server both calculate and store the client's initial keychain computed from the generated public keys. ',' is used for concatenation.

executes the PkbGen algorithm to generate its prekey bundle pkb_A and its initial state $s_A^{0,0}$. After sending the prekey bundle to the server S, the server sends its long-term public key pk_S for use in the later two protocols: the Envelope Sending Protocol and the Envelope Fetching Protocol. Both the client and the server then compute and store the initial *keychain* h_A^0 of the client, which is a hash of the client's prekey bundle. The server stores the client's initial keychain with the client's prekey bundle whereas the client stores its prekey bundle, its initial keychain, and the newly received server's public key. The adversary is assumed to not be present in this protocol, and as per our adversary model, adversary $\mathcal{E}$ does not have write access to modify the newly received server's public key on the client's state but only read access when cloning the client. If an adversary has write access on the client or is present in the Registration Protocol, this would allow the adversary to change the server's public key and replaces with its own public key. This would subsequently enable the adversary to later forge signatures and impersonate as the server without the client noticing. Thus, to meaningfully establish the security guarantees of the system, it is necessary for us to assume that the adversary is not present during the registration phase, which is in line with conventional cryptographic protocol design that assumes the adversary's absence during the registration phase.

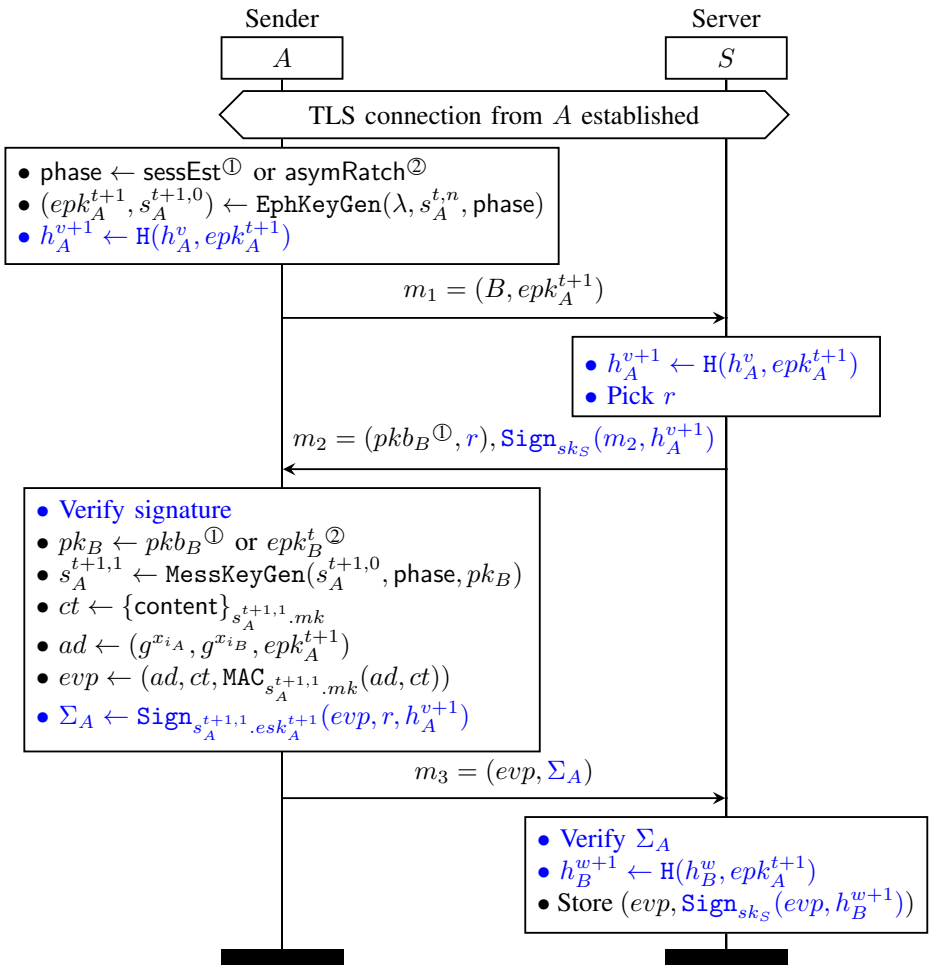


Figure 7.4: The Envelope Sending Protocol in the session establishment phase and the asymmetric ratcheting phase. In this protocol, a sender A sends its freshly generated public keys to the server S and the envelope intended for a recipient B to the server. If the protocol is successful, the server accepts and stores the sender's envelope for the recipient. ①: Parameters for session establishment. ②: Parameters for asymmetric ratcheting. ',' is used for concatenation.

7.2.3 Envelope Sending Protocol

The Envelope Sending Protocol is depicted in Figure 7.4 and the protocol is executed by a sender A only when it sends the first envelope of a new time epoch $t+1$ to a recipient B, in the two phases of the session establishment phase (sessEst) and the asymmetric ratcheting phase (asymRatch). After a successful run of this protocol, the sender sends the envelope intended for the recipient to the server S. The server buffers the envolope for the recipient until the recipient requests to fetch the envelope in the Envelope Fetching Protocol.

The protocol starts with the establishment of TLS connection between the sender and the server, and we assume that the server has authenticated the sender on the TLS layer, e.g., through a bearer token. The sender initiates the protocol by first executing the `EphKeyGen` algorithm, which generates fresh public keys epk_A^{t+1} and the updated state $s_A^{t+1,0}$ of the sender for the new time epoch $t+1$. The public keys are then hashed with the sender's current keychain h_A^v to update the keychain to the new time epoch $v+1$. The sender then sends message m_1 to the server, which comprises of the recipient's identifier and the freshly generated public keys. We use v to denote the time epoch associated with the evolution of the sender A's keychain. The epoch v increases continuously over the entire lifetime of A, regardless of which recipient B (e.g., B_1, B_2, B_3, etc) that A communicates with. That is, v reflects the total number of communication epochs A has participated in across all sessions with all recipients. In contrast, t refers to the time epoch within a single communication session between a specific sender-recipient pair (A, B), e.g., (A, B_1), (A, B_2), (A, B_3), etc.

Once the server receives m_1, the server then updates its copy of the sender's current keychain h_A^v using the received public keys epk_A^{t+1} by hashing both information to generate the new keychain h_A^{v+1}. The server then issues a random nonce r to the sender in message m_2, which additionally includes the prekey bundle pkb_B of the recipient if the phase of the protocol execution is session establishment. The random nonce r serves as a challenge for the sender to test if the sender indeed possesses the corresponding private keys esk_A^{t+1} of epk_A^{t+1}. Message m_2 is sent together with the signature of the server on the message and the sender's keychain for the new epoch using the server's long-term private key sk_S corresponding to the public key pk_S sent in the Registration Protocol. The signature of the server is vital for detecting the presence of adversary $\mathcal{E}$ (see Guarantee 7.1).

The sender first verifies the server's signature after receiving m_2, which assures the sender that the server has the same value of the sender's keychain implying that active Man-in-the-Middle (MitM) attacks have not taken place up until the current time epoch t. The sender proceeds to pick the content of the envelope for the

recipient and generates the ciphertext of the content using the updated message key mk_1^{t+1} contained in the updated state $s_A^{t+1,1}$ returned from running the `MessKeyGen` algorithm. If the phase is session establishment, then the public keys pk_B of the recipient are pkb_B; otherwise pk_B is the recipient's previous public ratchet key epk_B^t. The associated data ad for the communication with the recipient consists of the long term identity public keys $g^{x_{iA}}$ and $g^{x_{iB}}$ of the sender and the recipient respectively, and epk_A^{t+1}. The sender then puts together the first envelope evp for the new time epoch $t+1$ for the recipient, which consists of the associated data, the ciphertext, and a Message Authentication Code (MAC) over both the associated data and the ciphertext under the updated message key mk_1^{t+1}. This operation can alternatively be done using an AEAD scheme. The sender subsequently generates signatures Σ_A on the envelope using the private keys esk_A^{t+1} that the sender has freshly generated at the start of the protocol corresponding to the public keys epk_1^{t+1}. As the last message of the protocol, the sender sends the envelope with Σ_A to the server.

Upon receiving the envelope and Σ_A, the server verifies Σ_A. If the verification passes, then the server has the assurance that the sender of the envelope is the same sender that initiates the protocol. The server then updates its copy of the recipient's current keychain h_B^w to the keychain for the new time epoch $w+1$ by hashing h_B^w with epk_A^{t+1}. The server then generates a signature on the envelope and h_B^{w+1}, and the server stores the envelope and the signature until the envelope is fetched by the recipient in the Envelope Fetching Protocol. Similarly, we use w to indicate the time epoch that the recipient B has participated over its whole lifetime across multiple communication sessions with different instances of senders A (e.g., A_1, A_2, A_3, etc), which might not necessarily equal to the sender's time epoch v. This is in contrast to t, which indicates the time epochs that B shares with a specific sender A in their communication session.

7.2.4 Envelope Fetching Protocol

As portrayed by Figure 7.5, the Envelope Fetching Protocol is for a recipient B to fetch the envelope (evp) sent by a sender A which is buffered at the server

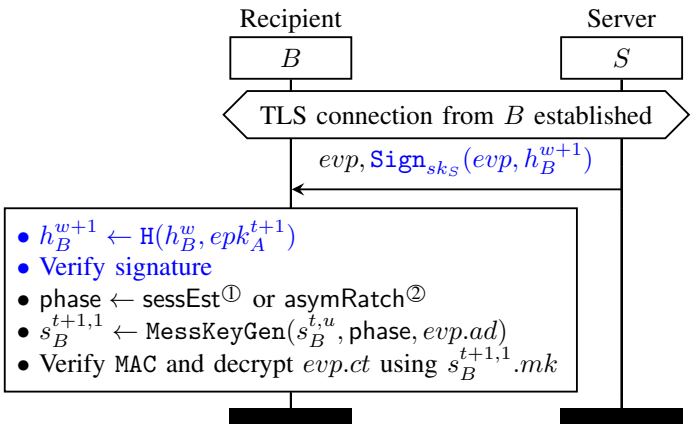


Figure 7.5: The Envelope Fetching Protocol where the server S delivers the buffered envelope from a sender A to a recipient B after the recipient has established a TLS connection with the server to to fetch the envelope. ①: Parameters for session establishment. ②: Parameters for asymmetric ratcheting. ',' is used for concatenation.

S. The protocol starts with the recipient establishing a TLS connection to the server signalling that the recipient is now online and is requesting to fetch the envelope. Similarly, we assume that the recipient is authenticated by the server on the TLS level, e.g., through a bearer token. Upon receiving the request, the server retrieves and sends the recipient the buffered envelope with the server's previously generated signature. After receiving the envelope, the recipient first calculates its keychain h_B^{w+1} for the new time epoch using epk_A^{t+1} from the envelope. With this updated keychain, the recipient verifies the server's signature on the envelope and h_B^{w+1}. The verification of the server's signature assures the recipient that the sender of the envelope has indeed contacted the server before sending the envelope (see Guarantee 7.2). The recipient then proceeds to execute the `MessKeyGen` algorithm to update its state to $s_B^{t+1,1}$ which contains the message key mk for MAC verification and ciphertext decryption of the envelope based on the same phase in which the sender has sent the envelope. Notice that the state $s_B^{t+1,1}$ for the recipient B is updated to the new epoch $t+1$ directly as this is the first envelope for the new epoch.

7.2.5 Comparison with Existing Work

In relation to the existing work on Signal's key authentication and cloning attacks on Signal in Section 3.4, we present our solution for cloning attacks in Signal by

a MitM adversary. Our solution builds directly on top of the protocol without introducing any additional third-party to the communication infrastructure, as opposed to key transparency solutions, and without any user intervention for the authentication ceremony. Our changes for the solution are only applied on the interaction between the server and the client in the protocol, and do not change the interaction between clients or the derivation of the session keys. As such, existing solutions (Marlinspike, 2016; Dowling and Hale, 2021; Lawlor and Lewi, 2023; Yu et al., 2017; Cremers, Fairoze, et al., 2020; Cremers, Jacomme, et al., 2023; Dowling, Günther, et al., 2022; Barooti et al., 2023) can still be integrated seamlessly with our solution. We thus offer a tradeoff and an alternative trust model for applications building on top of Signal: either trust the Signal server but not the users' environment using our solution for the detection of a more powerful active MitM adversary; or for security-savvy users who do in fact perform authentication ceremonies, to not trust the Signal server at all but only recover security against a passive state-cloning adversary (Signal's adversary model for post-compromise security) (Cohn-Gordon, Cremers, and Garratt, 2016).

7.3 Security Analysis

We present the security analysis for our solution in this section. We prove that our solution provides new security guarantees in addition to retaining the existing security guarantees of Signal. Each guarantee is associated with a specific goal of adversary $\mathcal{E}$ as described in Section 7.1. We further give a discussion on how our modified Signal protocol impacts the deniability aspect of the original Signal protocol. Towards the end of this section we give a summary on how the individual guarantees complement one another to form a novel solution that detects the presence of a powerful active Man-in-the-Middle (MitM) adversary and limits the power of the adversary. In the following proofs we use the notation A' to indicate the adversary impersonating A, and m_i to refer to the i-th protocol message. We again refer to Section 2.1 for our assumption of secure crypographic primitives being used in the protocols.

7.3.1 New Guarantees

The two new security guarantees that our solution provides are the detection of active MitM adversary (Guarantee 7.1) and the detection of non-compliance in protocol execution (Guarantee 7.2). The Envelope Sending Protocol provides Guarantee 7.1 whereas the Envelope Fetching Protocol provides Guarantee 7.2. Guarantee 7.1 and Guarantee 7.2 prevent Goal (1) of the adversary as described in the adversary model in Section 7.1.2.

Guarantee 7.1 (Envelope Sending Protocol: Active Adversary Detection)**.** *When A runs the Envelope Sending Protocol at time epoch t_A, the presence of $\mathcal{E}$ is detected, if $\mathcal{E}$ has performed at least one run of the Envelope Sending Protocol impersonating A between time epochs τ and t_A.*

Proof. To break this guarantee, i.e., to remain undetected by both A and S simultaneously in the Envelope Sending Protocol, an active MitM adversary $\mathcal{E}$ has to generate a valid signature on both m_2 and m_3. As the adversary possesses A's keychain h_A^τ from the epoch of compromise τ, the adversary can compute a valid keychain $h_{A'}^{t_{A'}}$ to impersonate A at epoch $t_{A'} > \tau$, using h_A^τ and the public keys $epk_{A'}^{t_{A'}}$ freshly generated by the adversary for m_1. Note that during the time epochs following τ, S's copy of A's keychain is updated to $h_{A'}^{t_{A'}}$ while A's copy of the keychain still remains at h_A^τ. At time epoch $t_A > t_{A'}$ when A initiates the Envelope Sending Protocol, A freshly generates $epk_A^{t_A}$ and a new keychain $h_A^{t_A} = \mathtt{H}(h_A^\tau, epk_A^{t_A})$ for sending m_1. Also note that at time epoch t_A since A and S each has a different copy of A's keychain, the adversary is immediately detected by A if the adversary does not tamper with m_2, or by S if the adversary does not tamper with messages m_1 and m_3.

For the adversary to be undetectable by A requires the adversary to be online at t_A when A runs the protocol for the adversary to change the signature of S in message m_2 on $h_A^{t_A}$ to $h_{A'}^{t_{A'}}$ instead. This then requires the adversary to find a $h_{A'}^{t_{A'}}$ that produces the the same hash as $h_A^{t_A}$, or to forge S's signature. However, if the adversary succeeds in finding such a second pre-image or forging a signature, this

respectively breaks the second pre-image resistance of the underlying hash function or the EUF-CMA security of the underlying signature scheme, which in turn breaks our assumption of secure cryptographic primitives.

If the adversary drops the last message containing the envelope *evp* sent by A during A's run of the Envelope Sending Protocol at epoch t_A, the adversary can send a new envelope evp' with a content chosen by the adversary on behalf of A. However, now the adversary is required to either find a evp' that produces the same hash as *evp*, or to forge a signature Σ_A on evp'. On successfully finding such a second pre-image or forging the signature on evp', this means that the adversary breaks the second pre-image resistance of the underlying hash function or the EUF-CMA security of the underlying signature scheme, which in turn breaks our assumption of secure cryptographic primitives.

□

Guarantee 7.2 (Envelope Fetching Protocol: Non-Compliance Detection)**.** *Every envelope fetched through the Envelope Fetching Protocol has been previously sent to the server through the Envelope Sending Protocol.*

Proof. Breaking this guarantee means that an adversary $\mathcal{E}$ evades detection by not executing the Envelope Sending Protocol with the server S, and instead sending an envolope evp' of the adversary's choosing directly to B in the Envelope Fetching Protocol while impersonating S. To achieve this while also remaining undetectable from B, the adversary must either produce an evp' that generates the same hash as *evp* or produce the signature of S on evp'. As we have seen in Guarantee 7.1, finding such a second pre-image or forging the signature on evp' breaks our assumption that secure cryptographic primitives are used, as this respectively breaks the second pre-image resistance of the underlying hash function and the EUF-CMA security of the underlying digital signature scheme. □

7.3.2 Existing Guarantees

In addition to providing new guarantees, both protocols in our solution maintain the existing properties from the original Signal protocol. These are, perfect forward secrecy (Guarantee 7.3), post compromise security (Guarantee 7.4), and end-to-end encryption (Guarantee 7.5). With the adversary goals described in the adversary model in Section 7.1, Guarantee 7.3 and Guarantee 7.4 prevent Goal (2) and Goal (3) of the adversary respectively whereas Guarantee 7.5 prevents Goal (4).

Guarantee 7.3 (All Protocols: Perfect Forward Secrecy)**.** *Adversary $\mathcal{E}$ is unable to derive the root keys (and thus the chain keys and the message keys) from previous time epochs $t < \tau$.*

Proof. For the adversary to break this guarantee, the adversary has to derive the root key of previous time epochs $t < \tau$, for which the adversary has two options: (1) using messages exchanged between a sender A and a recipient B from previous runs of the Envelope Sending Protocol and Envelope Fetching Protocol during time epochs $t < \tau$, in addition to (2) using the secrets cloned from A at time epoch τ. (1) In the first option, since only the public keys epk_A^t are exchanged in all three messages in the Envelope Sending Protocol during time epochs $t < \tau$, the adversary has to either derive the corresponding private keys esk_A^t from epk_A^t or simply guess esk_A^t. The former breaks the CDH assumption whereas the probability of the latter occurring is negligible. (2) If the goal of the adversary is only to obtain the root key $rk^{\tau-1}$ of time epoch $\tau - 1$, then the adversary requires not only the private ratchet key $x_A^{\tau-1}$ of A at time epoch $\tau - 1$ but also the root key $rk^{\tau-2}$ from time epoch $\tau - 2$. As the adversary obtains only rk^τ and x_A^τ from cloning A at time epoch τ, the adversary is in turn left with only two choices: finding the inverse of the key derivation function to obtain $rk^{\tau-1}$ from rk^τ, or brute force $rk^{\tau-1}$. In successfully achieving the former the adversary breaks the assumption of a secure key derivation function and thus the assumption of secure cryptographic primitives whereas the latter can only occur with a negligible probability. □

Guarantee 7.4 (All Protocols: Post Compromise Security)**.** *If a sender A sends an envelope to a recipient B at time epoch $\tau + 1$ and the sender A receives an envelope from the recipient B at time epoch $\tau + 2$, and the adversary $\mathcal{E}$ is passive at both epochs $\tau + 1$ and $\tau + 2$, then the content of future envelopes in epochs $t \geq \tau + 2$ is only known to A and B. $\mathcal{E}$ can only reveal the content of at most two time epochs, τ and $\tau + 1$.*

Proof. To break this guarantee, the adversary has to be able to reveal the content of future envelopes of time epochs $t \geq \tau + 2$ and this requires the adversary to be able to generate the root key rk^t for time epochs $t \geq \tau + 2$. This means that the adversary must be able to first compute the root key $rk^{\tau+2}$ before the adversary can compute the root key of subsequent time epochs $t > \tau + 2$. While the adversary can still generate the root key $rk^{\tau+1}$ when B switches role to become the sender at time epoch $\tau + 1$ and thus reveal the content of the envelopes sent by B in that time epoch, the adversary still requires the fresh private ratchet key $esk_A^{\tau+2}$ generated by A at the next time epoch $\tau + 2$ in order to generate $rk^{\tau+2}$. Note that at time epoch $\tau + 2$ when A switches back to be the sender, A performs asymmetric ratcheting by generating a fresh ratchet keypair $(epk_A^{\tau+2}, esk_A^{\tau+2})$. Since the adversary is passive when A sends the public ratchet key $epk_A^{\tau+2}$ in message m_1 of the Envelope Sending Protocol, i.e., the adversary does not interfere with the message by injecting its own ratchet keypair, to compute rk^{t_A+2}, the adversary must obtain the private ratchet key $esk_A^{\tau+2}$ corresponding to the $epk_A^{\tau+2}$ sent by A. This leaves the adversary with only two options: to derive $esk_A^{\tau+2}$ from $epk_A^{\tau+2}$, or to guess $esk_A^{\tau+2}$. The former breaks the CDH assumption and thus contradicts our assumption of secure cryptographic primitives whereas the latter can only happen with a negligible probability. □

Guarantee 7.5 (All Protocols: Content Confidentiality and Authentication)**.** *The content of an envelope is only known to the sender A and the recipient B, and the authenticity of the content is preserved. An adversary $\mathcal{E}$ cannot reveal and change the content sent by A unless the adversary uses its cloning ability.*

Proof. For the adversary to break this guarantee, the adversary should be able to decrypt or change the content of the envelope without using its cloning ability. The adversary can either (1) inject the adversary's own chosen envelope in either the Envelope Sending Protocol or the Envelope Fetching Protocol or (2) generate the root key rk^t of a time epoch t that the adversary chooses.

In the first case (1), the adversary can choose to inject an envelope evp' of its own choosing in either (i) the last message m_3 of the Envelope Sending Protocol, or (ii) the message sent to B in the Envelope Fetching Protocol. (i) For the adversary to inject evp' in the last message of Envelope Sending Protocol, the adversary has to generate a valid signature Σ_A on evp'. As previously proved in Guarantee 7.1, this leads to the contradiction in our assumption of secure cryptographic primitives. (ii) Similarly for the adversary to inject evp' into the Envelope Fetching Protocol, the adversary has to generate a valid signature on evp' from the server. As we have established before in Guarantee 7.2, this again leads to a contradiction of our assumption of secure cryptographic primitives.

In the second case (2), to derive the root key rk^t of any time epoch t without having access to the secrets of a sender A via cloning, the adversary has to compute the root key rk^{t-1} at time epoch $t-1$, which itself is derived based on the root key rk^{t-2} at time epoch $t-2$, and this goes on until the first root key rk^1 established in the session establishment phase. This ultimately means that the adversary has to first be able to derive rk^1 and there are only three ways for the adversary to do so. (i) The adversary at time epoch $t = 1$ generates and inserts its own prekey bundle $pkb_{\mathcal{E}}$ in m_2 of the Envelope Sending Protocol sent by A. This again requires the adversary to be able to generate S's signature, and as we have seen many times, this leads to a contradiction of our secure cryptographic primitives assumption. (ii) The adversary chooses to derive or guess the corresponding private keys esk_A^t of A from A's freshly generated public keys epk_A^t sent in m_1 of the Envelope Sending Protocol, or the private keys of B contained in B's initial state $s_B^{0,0}$ corresponding to the public keys in B's prekey bundle sent by the server in m_2 of the Envelope Sending Protocol. From Guarantee 7.3, this breaks the CDH assumption and thus

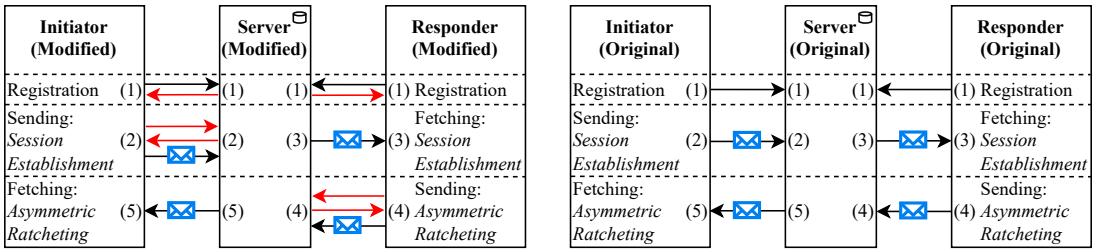
our assumption of secure cryptographic primitives. (iii) Lastly the adversary simply guesses rk^1. With the assumption of secure cryptographic primitives, this can only occur with negligible probability. □

7.3.3 Discussion on Deniability

Before concluding the security analysis, we briefly examine the implications of our protocol modifications on the security guarantee of cryptographic deniability provided by the original Signal protocol (Vatandas et al., 2020). Although our use of a digital signature scheme may appear to undermine the Signal protocol's cryptographic deniability, this security guarantee can be retained through applying a deniable signature scheme, such as those proposed in the work of Jakobsson et al. (1996) and Rivest et al. (2001), as Dowling, Günther, et al. (2022) also suggests. Nevertheless, we note that cryptographic deniability in our adversary model is trivial, as a compromised sender can always plausibly claim that its messages originated from the adversary. Furthermore, deniability is hard to achieve in practice if the server already keeps a log of users, e.g., for logins.

7.3.4 Summary

To summarise, Guarantee 7.5 assures that the adversary $\mathcal{E}$ is unable to reveal the content of the envelopes without using its cloning ability. If the adversary does use its cloning ability, from Guarantee 7.3 and Guarantee 7.4, the adversary can only reveal the content of two time epochs, τ and $\tau + 1$, with the condition that $\mathcal{E}$ is passive in the time epochs $\tau + 1$ and $\tau + 2$. Otherwise, if the adversary is active in time epochs $t_{A'} > \tau$, then from Guarantee 7.1 and Guarantee 7.3 the adversary can only inject its own envelopes in these time epochs until the adversary is detected at time epoch $t_A \geq t_{A'}$ when the cloned sender A runs the Envelope Sending Protocol. All three guarantees, Guarantee 7.1, Guarantee 7.3, Guarantee 7.4, give the adversary that uses its cloning ability two choices: either risk being locked out of the communication, or risk being detected. Finally, Guarantee 7.2 makes sure that all envelopes, including those sent by an active adversary, have to go



(a) Implementation for our modified Signal protocol.

(b) Implementation for the original Signal protocol.

Figure 7.6: An overview of the two versions of our implementation for our modified protocols and the original Signal protocol. In both versions of the implementation, each client performs three actions (registering, sending, fetching) in the three different Signal communication phases (registration, session establishment, asymmetric ratcheting) with the server, who manages a database for storage. The arrows indicate the number of exchanged protocol messages in the respective version of the implementation until an envelope is transferred, and the red arrows indicate the extra interaction between the client and the server in our modified protocols, which is the focus of the experiments.

through both the Envelope Sending Protocol and the Envelope Fetching Protocol before they reach their intended recipient.

7.4 Implementation and Evaluation

To demonstrate the practicality of our solution, we implemented a proof-of-concept that builds on the open-source Signal protocol library (Open Whisper Systems, 2017). This section presents the findings and evaluations on the implementation that forms the basis of our experiments.

7.4.1 Implementation Details

Our implementation is built on top of the open-source Rust Signal library (version `0.32.1`) (Open Whisper Systems, 2017). Our modifications to the Signal protocol integrate seamlessly with the library, and we use the cryptographic primitives already available in the library. We make our implementation available on our git server[1] and only describe its high-level details here.

Figure 7.6 depicts an overview of the two versions of our implementation. Figure 7.6a depicts the implementation of our modified Signal protocol and Figure 7.6b

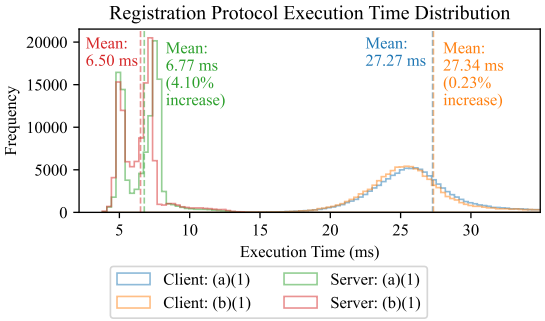
[1] `https://malaria.cs.ox.ac.uk/git/wil/modsignal-experiments`

depicts the implementation of the original Signal protocol. To test the performance of the three different actions of a client throughout the three communication phases of Signal where we have applied our changes, we design the implementation in a way that a client, an initiator and a responder, in both versions goes through, in sequence: (1) registration with the server, (2) sending an envelope to the server, and (3) fetching an envelope from the server. Recall that for an initiator sending an envelope first occurs at session establishment whereas fetching an envelope first occurs at asymmetric ratcheting; for the responder the sending first occurs at asymmetric ratcheting whereas the fetching first occurs at session establishment.

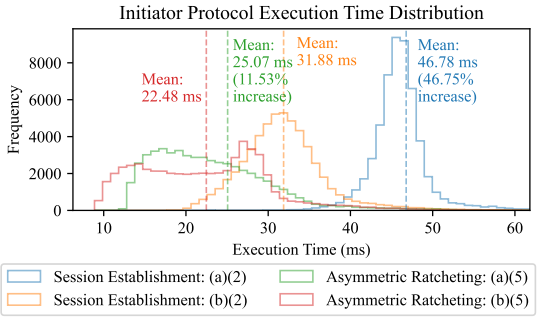
Observe that the arrows for (1), (2), and (3) in Figure 7.6a represent the same number of messages exchanged in the Registration Protocol (Figure 7.3), the Envelope Sending Protocol (Figure 7.4), and the Envelope Fetching Protocol (Figure 7.5) respectively. Notice that the difference between Figure 7.6a and Figure 7.6b is the number of messages exchanged between a client and the server in the protocol until an envelope is sent or received by the client. Also notice that the server in Figure 7.6b only stores and relays the envelopes between clients without performing any other action in the protocol.

7.4.2 Experimental Setup

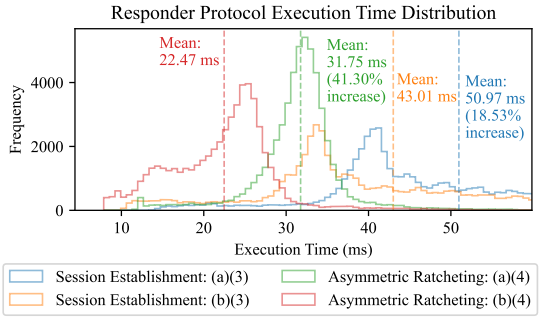
We investigate the performance overhead of the discrepancy in message processing between both versions of implementation in Figure 7.6 experimentally. Specifically our experiment measures the execution time for the protocols in Figure 7.6, i.e., (1) to (5), to be completed by the client and the server in both versions of the implementation where we record the protocol completion time on the client's end and the server's end. We conduct the experiment with n instances of initiators and n instances of responders executing each protocol with a single instance of the server in both versions of the implementation. To make network latency negligible, we execute all client instances and the server on a single machine running Ubuntu 22.04 with Intel® Core™ i5-1145G7.



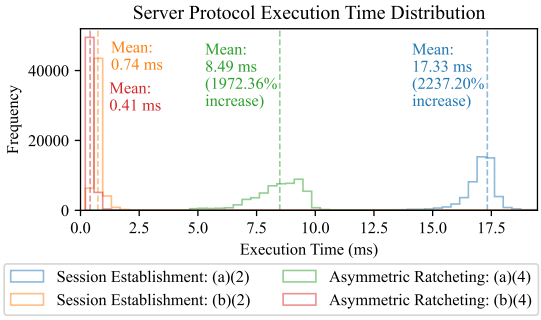
(a) The distribution of the execution time of $n = 110000$ clients ($n = 55000$ initiators and $n = 55000$ responders) for the Registration Protocol with the server.



(b) The distribution of the execution time of $n = 55000$ initiators for session establishment (sending) and asymmetric ratcheting (fetching).



(c) The distribution of the execution time of $n = 55000$ responders for session establishment (fetching) and asymmetric ratcheting (sending).



(d) The distribution of the execution time for the server to complete $n = 55000$ requests for session establishment from the initiators and $n = 55000$ requests for asymmetric ratcheting from the responders.

Figure 7.7: Distributions of execution time for each action performed by the clients (both initiators and responders) and the server in the two versions of the implementation. (a) and (b) refer to Figure 7.6a and Figure 7.6b respectively whereas (1) to (5) refer to the actions (registration, sending, and fetching) taken by the clients with the server in the two subfigures.

7.4.3 Results Analysis

We run an experiment with $n = 55000$ instances of initiators and $n = 55000$ instances of responders with a single instance of the server in both versions of the implementation. The results representing the distribution of the protocol execution time are visualised in Figure 7.7 as histograms. We calculate the difference in mean as the performance overhead in each specific protocol run.

The results in Figure 7.7a show the distribution of the protocol execution time for the registration of both clients (initiators and responders) with the server. The difference in mean between the two implementation versions for the execution time

is 0.06 ms (0.23% increase) on the clients' ends and 0.27 ms (4.10% increase) on the server's end. Both differences in mean are less than 0.5 ms (less than 5%) and they show that our Registration Protocol has an insignificant performance overhead compared to the original Signal's Registration Protocol. This means that a client and the server experience no difference when using our Registration Protocol in terms of processing the messages during registration.

Figure 7.7b depicts the distribution of the protocol execution time for the initiators to send an envelope to the server at session establishment and to receive an envelope from the server at asymmetric ratcheting in both versions of the implementation. At session establishment, the difference in mean is only 14.90 ms (46.75% increase) for sending an envelope to the server, and at asymmetric ratcheting, fetching an envelope from the server observes a difference in mean of only 2.59 ms (11.53% increase). Similarly, Figure 7.7c illustrates the distribution of the protocol execution time for the responders to receive an envelope from the server at session establishment and to send an envelope to the server at asymmetric ratcheting in both versions of the implementation. Fetching an envelope at session establishment sees a difference in mean of just 7.97 ms (18.53% increase) whereas sending an envelope at asymmetric ratcheting sees a difference in mean of just 9.28 ms (41.30% increase). As the differences in mean are all less than 15 ms (less than 50% increase), this demonstrates that there is hardly a noticeable difference on the client-side processing using our protocols. Thus our changes are applicable for real-world deployment without deteriorating the user experience on the clients.

Figure 7.7d illustrates the distribution of the protocol execution time for completing the Envelope Sending Protocol on the server's end. The difference in mean execution time at session establishment is 8.08 ms whereas at asymmetric ratcheting it is 16.59 ms. While the relative percentage increase in the performance overhead of our protocols seems substantially high in both phases (around 2000%), this is primarily because in the original Signal protocol, the server does not perform any cryptographic operations during session establishment or asymmetric ratcheting with the sender. The server only receives and stores the sender's envelopes. In

contrast, our modified protocols require the server to interact cryptographically with the sender during both of these phases. Nonetheless, when comparing absolute values, the differences in mean are still well under 20 ms. This again demonstrates the negligible performance overhead of our modified protocols for the server to handle the requests from the clients. Our modified protocols thus do not affect the usability of the original Signal protocol since a human user can barely notice the 20 ms difference when waiting for the server to complete the sending or the receiving of a message.

Overall, the experimental results indicate that the performance overhead for our protocols is a trivial constant and from that, our solution exhibits a linear time complexity with the number of clients in the system. We can thus conclude that our solution is on the same order as the performance of the original Signal protocol. This is an acceptable trade-off for the novel and stronger security guarantees that our solution provides. Additionally the performance of our solution can still be improved once it is deployed on a real-world application that has a more powerful infrastructure.

We additonally note, as in Section 5.4.3, the experiments here cover only computational overhead from the cryptographic operations. Before closing this section, we take this opportunity to discuss the storage and communication overhead resulting from our modifications to the original Signal protocol. On the client side, our modified protocol requires storing only a single additional hash beyond what is already maintained in the original Signal protocol. The client-side storage overhead is hence of constant complexity, i.e., $O(1)$. On the server side, the system incurs a linear storage overhead, i.e., an $O(n)$ complexity, as the server must store one additional hash per client. However, given that servers supporting messaging applications are typically provisioned to handle significant storage demands, this overhead is expected to be manageable in practice. As for the communication overhead, our protocol modifications introduce only fixed-length fields (namely digital signatures and a nonce) into the existing messages of the original Signal protocol. Consequently, these constitute a constant overhead per message, we therefore do not anticipate a substantial increase in communication overhead, save

for the round-trip time that is dependent on the network conditions where the system is deployed, e.g., network latency and congestion. We revisit these overhead considerations as important aspects for real-world deployments in Section 9.2.2.

7.5 Chapter Summary

The emerging threat of breaking end-to-end encryption to get access to a user's private

ment coercion has the potential to proliferate Man-in-the-Middle (MitM) attacks. Current solutions (Marlinspike, 2016; Marlinspike and Perrin, 2017; Dowling and Hale, 2021; Dowling, Günther, et al., 2022) in instant messaging applications such as Signal and WhatsApp for key authentication to detect MitM attacks depend on a communicating pair of users to compare the fingerprints of their keys in an out-of-band manner. However, the issues with relying on user participation to ensure key authentication present usability difficulties as several studies have shown (NEAL, 2017; Vaziripour, Wu, O'Neill, Whitehead, et al., 2017; Vaziripour, Wu, O'Neill, Metro, et al., 2018; Schröder et al., 2016). Furthermore, this so-called authentication ceremony (Ellison, 2007) is expected to carry out every so often (Dowling and Hale, 2021; Dowling, Günther, et al., 2022) and this again presents practicality issues.

We build our novel solution for these problems directly into the existing Signal protocol without being dependent of user intervention. Our solution allows key authentication to be done in-band with the same communication channel used to send and receive messages between the communicating parties, without requiring any additional entities. We achieve this by making the most of the Signal server, who is always present on the communication channels, and who is expected to be honest in forwarding messages and distributing 'prekey bundles' of the users. We assume the presence of a powerful adversary that is not only capable of performing active MitM attacks, but also gaining one-time access to all secrets on one of the two communicating parties. By embedding a recursively updating key fingerprint of a client as part of the client's identity, our solution allows both the server and the client to keep track of the changes in the client's key fingerprint, thus automating

the detection for the presence of such an adversary. Our security analysis has shown that not only does our solution provide new security guarantees, but does so in such a way that preserves the underlying end-to-end encryption while also keeping the existing properties of the Signal protocol.

We implemented our proof-of-concept of the solution based on the open-source Signal protocol library, which is the underlying library used by the actual messaging applications, and we use this to demonstrate the practicality and integrability of our solution into the library. Based on the implementation we run experiments to determine the feasibility of the performance overhead of our solution as compared to the original Signal protocol. Our experimental results have shown that the performance for our solution has a linear overhead and is thus on the same order as the original Signal protocol.

8 Discussion

Contents

This chapter builds upon the key insights from previous discussions and further explores the practical considerations surrounding the real-world adoption of our solutions. Aligning with the overarching motivation of this thesis, we first examine the security of our solutions against more sophisticated adversaries, including those with persistent adversarial access, even in scenarios where key rotation is absent. We then address the challenges related to trust and availability associated with the involvement of a third party in facilitating communication within our solutions. Finally, before concluding the chapter, we evaluate how credential management is handled by the participants in our solutions.

8.1 Persistent Adversarial Access Without Key Rotation

In this section, we examine how our solutions from Chapter 5, Chapter 6, and Chapter 7 mitigate the risks posed by an adversary with persistent access to a machine's secrets following an initial compromise, even in the absence of key rotation on the compromised machine.

Our adversary model in Chapter 5 assumes that the adversary has only a one-time access to the SSH client's secrets. However, we found that our solution is still able to detect an adversary that has persistent access to a client's state even if the client's long-term keypair for authentication is not rotated. We have previously established in the chapter that if the adversary is active and logs into the server using the stolen credentials, then the adversary is detected. Here we discuss what happens if the adversary stays fully passive on the network after persistently cloning all secret of the clients across multiple sessions. If the adversary clones the client's secrets in epochs τ_0, τ_1, τ_2, etc., the adversary only obtains the corresponding masked master keys $g^{mk^{\tau_0}}$, $g^{mk^{\tau_1}}$, $g^{mk^{\tau_2}}$, etc. Note that in these epochs the adversary must not log into the server while impersonating the client, which will result in a detection of the adversary by the client (from Guarantee 5.1). However, even with the masked master keys and the client's long-term public key, the passive adversary is still unable to derive the cipher key from just observing the protocol messages due to the Computational Diffie-Hellman (CDH) assumption. Thus, the confidentiality and the authenticity of the data exchanged in the session is protected and this shows that our solution is more resilient than we thought. We clarify that we make the assumption of the one-time cloning ability of the adversary in the chapter is to simplify the security analysis. We note that while the original SSH protocol is also secure against an adversary's persistent access since the derived keys are deleted after the session, however without a keypair rotation the adversary still goes undetected if the adversary later launches impersonation attacks, and thus our solution still provides stronger security in this regard.

In Chapter 6, we did not specify a limit on the number of compromises an adversary can perform. Our adversary model assumes that the adversary can compromise any number of clients, while the credential compromise mitigation guarantee, i.e., Guarantee 6.1, remains intact under this assumption. We emphasise that this guarantee holds even if the adversary persists in the system without the requestors rotating their private keys. This is because every action executed by the adversary still requires registration with the identity manager. Consequently, the identity manager can continuously monitor system activity, track who is performing which actions, and facilitate anomaly detection based on the adversary's registered actions. Although a persistent adversary may observe and adapt to the behavior of compromised requestors, potentially complying with identity manager's action policies to evade detection, this does not provide any additional advantage beyond what the victim machine itself possesses. The adversary's actions remain constrained to those already authorised for the compromised machine, as stipulated by Guarantee 6.1. The only caveat for this guarantee is that the identity manager is not compromised, which is the same assumption that we have in the system model.

In a similar manner, our adversary model in Chapter 7 specifies that the adversary is assumed to have only a one-time use of its cloning ability to get access to a device's secrets. However, this does not mean that the strong adversary detection mechanism of our solution does not hold if the adversary can clone a device more than once throughout the long-lived communication session. In fact, our adversary detection guarantee, Guarantee 7.1, still holds even if an adversary is more powerful than our adversary model and has the ability to clone a device more than one time. The only condition here is that the communication channel has been 'healed' (adversary locked out), i.e., Guarantee 7.4 has been established, before each cloning. Recall from Guarantee 7.1 (adversary detection) and Guarantee 7.4 (post-compromise security) that after cloning, the adversary will either be detected if it is active or be locked out if it is passive, respectively. Thus, if a stronger adversary chooses to be locked out of the communication to avoid detection and uses its cloning ability again at a later time epoch, this effectively limits the time epochs of which the adversary

can reveal the content of the envelopes in these time epochs due to the self-healing property of the Signal protocol. So for the stronger adversary to perpetually remain on the communication the adversary has to be active on the channel, which will then trigger detection. Our solution thus makes it possible for this process to repeat with ongoing detection and requiring neither any participation from the users nor any out-of-band channel, which coincides with the motivation of our work in this thesis.

8.2 Trust in- and Availability of a Third Party

In Chapter 6, the identity manager acts as a trusted third-party in the system for issuing tokens as dynamic identities, validating action registrations, and acting as a centralised location for managing the action policies in the system. From Chapter 7, the server acts as a third party for envelope relays and also for key distribution, following the existing system model of Signal. In this section we discuss the considerations on the trust in and the availability of third parties in our proposed solutions for real-world adoption.

Our proposed solutions in Chapter 6 and Chapter 7 utilise a single trusted (and honest-but-curious) third party, and we note that this is a widely accepted and practical assumption that exists in real-world systems such as Single Sign-On (SSO) systems (Hardt, 2012; Sakimura et al., 2014; Lockhart and Campbell, 2008; Steiner et al., 1988) and Signal's end-to-end instant messaging applications. To further improve our solutions against the risk of single-point-of-failure, we can adopt techniques that have been studied and proposed in existing work, such as distributing trust onto multiple entities using threshold cryptography (Agrawal et al., 2018; Baum et al., 2020; Rawat and Jhanwar, 2020; Y. Zhang et al., 2020; Harchol et al., 2018), or integrating key transparency systems (Lawlor and Lewi, 2023; Yu et al., 2017; Malvai et al., 2023; Chase et al., 2019). However, these approaches require additional parties to be introduced into a system, as we have also mentioned previously in Section 7.2.5. Not only will this incur additional resources to maintain the additional parties in the system, but some degree of trust is also placed on the additional parties, and at the same time enlarging the attack surface of the system.

Thus, factors such as trust delegation and maintenance overhead would have to be evaluated carefully when deciding to adopt these approaches. Nonetheless, we presented our solution in this thesis to illustrate their most fundamental form to establish a clear baseline for their effectiveness and feasibility.

The availability of the third parties also remains vital to facilitate the secure communication and to ensure that there is no interruption to the application service. This creates the requirement that the third parties should be resilient against Denial-of-Service (DoS) attacks and that the third parties should be prevented from becoming a bottleneck or also a single-point-of-failure when facing DoS attacks. Fortunately, existing practical technologies have been used to prevent critical infrastructure from failing in the wake of DoS attacks. Load balancers, for example NGINX (Reese, 2008), can be deployed between multiple machines that serve as a third party so that the workload is shared and the performance of the third party does not deteriorate even if the incoming requests from the requestors are high in volume. Similarly, the usage of redundancy servers for the third party is another viable option to prevent the failure of a single machine acting as a third party from stopping the entire communication of the system altogether. A real-world example of this has already been implemented by Signal, which uses geographically distributed servers to additionally optimise the latency of the instant messaging application (Whittaker and Lund, 2023).

8.3 Credential Management

Credential management is another critical aspect to consider when deploying our solutions for real-world applications. In this section, we assess how our solutions handle credential management, on top of the respective novel security guarantees that they deliver.

In our solution from Chapter 5, a client-server pair constantly updates their keys after each login. We thus prevent the credential misuse of a user duplicating and moving the same keys from one SSH client to another in order to log into the same server under the same account from multiple SSH clients. This is because

in our solution, a client who updates the (masked) master key during a login will leave the other master keys stored on other clients desynchronised with the server's master key. Given that this is considered as an 'improper' practice anyway as agreed by industry experts (Goldbery et al., 2024), our solution thus enforces a good key management practice by providing a mechanism to prevent this. We rely on the fact that there is only a one-to-one association between the client's long-term public key and the client's master key. However should this 'improper' practice still be needed for convenience purposes, our solution can be circumvented by implementing policies such as allowing the server to maintain a one-to-many association of one client's long-term public key with multiple master keys. This convenience-security tradeoff should be considered very carefully in the deployment of the system since this also allows an adversary to access the master keys from previous sessions on the server and thus bypassing the detection.

Our solution in Chapter 6 enhances the credential management for both a requestor and an application server. As previously mentioned in Chapter 6, we embed a requestor's actions, which is already application-specific, within a disposable short-term token that forms the requestor's dynamic identity. This design reduces the number of credentials that a requestor must manage down to a single keypair, irrespective of the number of application services accessed. We thus simplify the credential management for the requestor, eliminating the need to manage separate credentials for each application server that it accesses and indirectly minimising the risk of credential exposure in the event of a compromise. On an application server's side, we centralise action policies, which validate a requestor's credential or identity during the requestor's service access. The centralisation streamlines policy management, as any modifications are needed to be applied only once at the identity manager, rather than individually at each affected application server. This approach decreases the likelihood of human error, which can lead to security vulnerabilities, such as misconfigurations that have had real-world implications, as seen in the Capital One incident (Newman, 2019).

In Chapter 7, our solution strengthens the key authentication of the Signal protocol by requiring the client to store only an additional hash chain as part of its credential. As also mentioned previously in Section 3.4.2, this represents an improvement over existing solutions (Barooti et al., 2023; Cremers, Fairoze, et al., 2020; Cremers, Jacomme, et al., 2023), which require a client to keep track of a separate state for each of its communication peers. In contrast, our solution ensures a constant and minimal storage overhead, in spite of the number of communication peers, resulting in a constant storage complexity. On the server's end, our solution requires the server maintaining a database that maps clients to their respective hash chains, necessitating only as many hash chains as there are clients in the system. Given that the server already stores authentication information for each client, our solution introduces only a linear increase in storage requirements. Consequently, we conclude that the credential management of our solution in Chapter 7 does not incur significant additional effort for real-world adoption.

8.4 Chapter Summary

In this chapter, we explored key considerations for the real-world adoption of our solutions. We examined the resilience of our solutions against an even stronger adversary that has persistent adversarial access to the secrets, despite the lack of key rotation after the initial compromise. We additionally analysed potential approaches for extending our solutions to address the compromise of both trusted and honest-but-curious third parties, as well as techniques for ensuring their continued availability. Lastly, we evaluated how credential management is handled by the participants in our systems.

9 Conclusion and Future Work

Contents

9.1 Conclusion

This thesis explores the security of Machine-to-Machine (M2M) communication, i.e., communication between machines that occurs without the presence or active involvement of a human user, through the lens of compromised machine credentials. Given that the secrets that constitute a machine's credentials, and thus the machine's identity, are typically stored in the clear within M2M applications, the work in this thesis focuses on providing security after the compromise of a machine's secrets, rather than preventing the compromise in the first place. By modelling a real-world adversary capable of controlling both the communication channels and the secrets of the machine, the research in this thesis strengthens the security of M2M communication protocols across the three key application domains of M2M

communication as previously set out in Section 1.1 for the scope of the thesis: remote access applications using the Secure Shell (SSH) protocol, automated development workflows within internal organisational networks, and end-to-end encrypted instant messaging applications using the Signal protocol. This final chapter summarises the main findings of the thesis and concludes the thesis by demonstrating how the research questions outlined in Chapter 1, the introduction of the thesis, have been addressed within each of these three M2M application domains.

In Chapter 4, we first establish a stronger adversary model that reflects the practical capabilities of a real-world attacker, encompassing not only the compromise of a machine's credentials, but also full control over the communication channels. This includes the ability to send, delay, modify, intercept, replay, and drop messages. Real-world examples of such adversaries are also presented to contextualise the threat model. Following this, we delineate the design objectives aimed at enhancing the security of M2M protocols in the face of a credential compromise. Notably, our approach shifts the focus from preventing a credential compromise to detecting the adversary's presence post-compromise. We further aim to have our solutions only rely on computationally secure techniques, making assumptions on neither the machine's physical environment, its underlying hardware, any user interaction, nor an out-of-band channel. Finally, we present an overview of the existing cryptographic techniques employed in our research methodology to achieve these design goals. These techniques include associating dynamically changing information with the machine's identity and leveraging both the adversary's temporary passive capabilities and their inability to correctly predict random information when secure cryptographic primitives are used.

Chapter 5 presents our solution to the problem of long-term identity key compromise in the SSH protocol. Even against a more powerful adversary that can compromise the long-term private key of the SSH client, our modifications on the SSH protocol as the solution provides the detection of the presence of such an adversary if the adversary attempts an impersonation (**RQ1**). Our solution provides this detection guarantee by associating the identity of a client-server pair with a

symmetric key that is ratcheted or updated for every session in which the client-server pair participates, in addition to the long-term identity keys of the client and the server while also ensuring that the ratcheted key is masked on the client's storage to prevent further key exposure (**RQ2**). This detection guarantee holds even if the adversary later logs into the SSH server impersonating the client and compromise the server's long-term private key while also attempting to roll back the ratcheted key. Our modifications in the SSH protocol includes the verification step to ensure that the client and the server both holds the updated and synchronised symmetric key as well as the identity key (**RQ3**). This is because a desynchronisation in the ratcheted key indicates that there has been a compromise and an impersonation attempt by the adversary. Our changes to the protocol as the solution also preserves the existing security guarantees of the SSH protocol, namely, mutual entity authentication, session data confidentiality and authentication, and perfect forward secrecy.

We propose a solution in Chapter 6 to mitigate the problem of the leakage of hard-coded credentials in application source code or execution scripts to an adversary in the domain of automated development workflows within internal organisational networks. Our proposed solution comes in the form a framework that associates a machine's or a requestor's identity not only to its credentials, but also to the action sequence (or actions) that it requests to be executed on an application server in the system (**RQ2**). With the help of a trusted third party called the identity manager, our proposed framework requires that every request of action execution on an application server must be registered at the identity manager, which enables the identity manager to build a system-wide view of who-is-doing-what, and that the action execution can only take place with a short-lived token issued by the identity manager (**RQ3**). Our solution provides the security guarantee that the implications of the credential leakage is mitigated as the power of the adversary is contained to only being allowed to execute the same actions with the compromised requestor, and only for a limited period of time, after which the adversary's access is revoked (**RQ1**). Our framework additionally facilitates the detection of the adversary by the

identity manager if there is an anomaly in the actions requested by the adversary from the 'normal' actions requested by the compromised requestor (**RQ1**).

In Chapter 7, we turn our focus to improving the key authentication in the Signal protocol used for end-to-end encrypted instant messaging applications by alleviating the need for the key authentication to be done in an out-of-band fashion that also requires user intervention. Our improvements to the protocol allows the detection of a powerful active Man-in-the-Middle (MitM) adversary that also clones all secrets of one of the communicating client pair, without the need for an out-of-band user-assisted authentication (**RQ1**). We achieve this by associating the identity of a client with a hashchain that reflects the changes in the client's key fingerprint (**RQ2**). We build our solution directly into the Signal protocol by making modifications to the protocol and making good use of the Signal server that is already honest-but-curious to handle key distribution and message relaying. Our modifications to the protocol are designed to detect the presence of a powerful MitM adversary by verifying that server and the client have the same synchronised hashchain when a message is sent (**RQ3**), as a difference in the client's and the server's view of the hashchain would mean that the adversary has sent a message on behalf of the compromised client. We achieve this while simultaneously maintaining the original security guarantees of the Signal protocol, such as post-compromise security, perfect forward secrecy, and end-to-end encryption (and integrity).

We end this section with a note that the research in this thesis has successfully enhanced the security of real-world M2M communication protocols. The work in this thesis have strengthened the post-compromise resilience of M2M communication protocols, even in situations of a total credential compromise, thereby pushing the boundaries of secure M2M communication protocols. We hope that our contributions to the state-of-the-art in the security of M2M communication protocols would inspire many more research to come.

9.2 Future Work

As a closing remark, we outline in this section the possible avenues for future work that extends the research scope of this thesis.

9.2.1 Formal Verification Methods for Security Analyses

The security analyses presented in Chapter 5, Chapter 6, and Chapter 7 are performed through an informal, yet systematic, examination of all potential adversarial strategies that could break the security guarantees. We demonstrated in all cases that the adversarial strategies ultimately contradict the underlying assumptions defined within our system and adversary models. This approach was deliberately chosen to enhance accessibility and clarity for a broader and a more general audience, including readers who may not possess a background in formal verification techniques such as game-based proofs or tool-assisted methods. Rather than applying formal verification methods, we prioritised comprehensibility without compromising rigour. In spite of that, we acknowledge that for a more thorough and formally grounded security analysis, the application of formal verfication methods represents a valuable direction for future work. For example, tools such as ProVerif (Blanchet et al., 2018) can be used to verify the absence of specific vulnerabilities while Tamarin (Meier et al., 2013) offers the potential for the verification of the security guarantees.

9.2.2 Experiments under Real-World Settings

The experimental evaluations presented in Chapter 5, Chapter 6, and Chapter 7 were conducted either on a single machine or within a controlled network environment, involving a limited number of participants to make network latency negligible. This setup allowed for a focused anaylsis of the computational overhead introduced by our proposed solutions from the experimental results, particularly in relation to the required cryptographic operations. While these controlled experiments provide valuable insights into the performance characteristics of the systems, further investigation is necessary to assess their practical viability in real-world scenarios. Specifically, deploying the proposed solutions in actual network environments

and evaluating their behaviour under realistic conditions would offer a more comprehensive understanding of potential operational challenges that we can address. Such future work could involve replicating the same experimental setup in large-scale, real-world systems to examine scalability, performance under high workload, and the systems' resilience in dynamic or unpredictable environments.

9.2.3 Privacy-Preserving Solutions

In Chapter 6, the identity manager handles all incoming action registration requests for application servers from the requestors in the system whereas in Chapter 7, a Signal server receives and stores the end-to-end encrypted envelopes exchanged between a sender and a recipient. In both cases, the identity manager and the Signal server have a global view of system interactions and are capable of inferring communication patterns of the parties in the system, such as who-is-doing-what and who-is-talking-to-whom. While these proposed solutions of ours provide the improved security guarantees on secret compromise, future work could aim to extend these guarantees to include stronger privacy guarantees for the participants of the systems. In the case of the identity manager, privacy-preserving techniques based on Zero-Knowledge Proofs (ZKPs) could be utilised and adapted to allow an action registration from a requestor without revealing the actions. This would also enable the identity manager to verify compliance with the policies of the application servers without learning what the actual actions are, thereby preserving the privacy of the requestor's behaviour. For the Signal server, incorporating anonymity-preserving techniques, such as those applied in mix networks (or mixnets) (D. L. Chaum, 1981) and Signal's Sealed Sender (jlund, 2018) feature, could serve to obfuscate the direct communication link between a sender and a recipient. This would reduce the server's ability from performing traffic analysis and inferring communication relationships, thereby enhancing metadata privacy within the system.

9.2.4 Continuous Adversarial Presence

Throughout this thesis, we focus on an adversary model in which the adversary obtains an one-time access to all of a party's cryptographic secrets in the system and later in Section 8.1 we extend this discussion into the consideration of a more persistent adversary capable of accessing the secrets on multiple, though still interrupted, occassions. An important distinction here is that this model still does not assume continuous unrestricted access of the adversary. As a further contribution to the field, a promising research direction for future work would be to explore the implications of an even stronger adversary model: one in which the adversary has continuous and unrestricted access at all times to all cryptographic secrets of a party (with also the control to communication channels). Investigating the extent to which meaningful security guarantees can still be achieved under such an adversary, if it is even possible at all in the first place, presents a challenging yet valuable line of inquiry, especially in machine-to-machine (M2M) communication scenarios where we do not consider the presence of a user and an out-of-band channel. This adversarial behaviour would reflect even more closely to those of real-world threat actors, such as malicious nation-state adversaries, with the capability to maintain continuous and unbounded access to client devices. Understanding how cryptographic protocols can be designed or adapted to offer resilience against such more powerful adversaries would significantly advance and enhance the robustness of M2M security. If achieving any form of a security guarantee is truly possible in this even stronger adversary model, this could represent the upper bound of adversarial resistance in practical systems and provide arguably the strongest security guarantees for M2M cryptographic protocols.

Bibliography

Abrams, Lawrence (Oct. 2018). *Signal desktop leaves message decryption key in plain sight*. Available: https://www.bleepingcomputer.com/news/security/signal-desktop-leaves-message-decryption-key-in-plain-sight/.

Abrams, Randy (July 2010). *Why steal digital certificates?* Available: https://www.welivesecurity.com/2010/07/22/why-steal-digital-certificates/.

Agrawal, Shashank et al. (2018). "PASTA: Password-based threshold authentication". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2042–2059.

Albrecht, Martin R., Jean Paul Degabriele, et al. (2016). "A surfeit of SSH cipher suites". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, pp. 1480–1491. URL: https://doi.org/10.1145/2976749.2978364.

Albrecht, Martin R., Kenneth G. Paterson, and Gaven J. Watson (2009). "Plaintext recovery attacks against SSH". In: *2009 30th IEEE Symposium on Security and Privacy*, pp. 16–26.

Ali, Yasir and Sean Smith (2004). "Flexible and scalable public key security for SSH". In: *European Public Key Infrastructure Workshop*. Springer, pp. 43–56.

Alwen, Joël, Sandro Coretti, and Yevgeniy Dodis (2019). "The Double Ratchet: Security notions, proofs, and modularization for the Signal protocol". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, pp. 129–158.

AWS (2022). *Creating and using usage plans with API keys*. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html.

Barker, Elaine et al. (2013). *Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography*. Tech. rep. National Institute of Standards and Technology.

Barooti, Khashayar et al. (2023). "On active attack detection in messaging with immediate decryption". In: *Annual International Cryptology Conference*. Springer, pp. 362–395.

Baum, Carsten et al. (2020). "PESTO: Proactively secure distributed Single Sign-On, or how to trust a hacked server". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, pp. 587–606.

Bäumer, Fabian, Marcus Brinkmann, and Jörg Schwenk (2024). "Terrapin attack: Breaking SSH channel integrity by sequence number manipulation". In: *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 7463–7480.

Baushke, M (2022). *RFC 9142: Key exchange (KEX) method updates and recommendations for Secure Shell (SSH)*. Tech. rep.

Bellare, Mihir, Tadayoshi Kohno, and Chanathip Namprempre (2002). "Authenticated encryption in SSH: Provably fixing the SSH Binary Packet Protocol". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 1–11.

— (May 2004). "Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm". In: *ACM Transactions on Information and System Security* 7.2, pp. 206–241. URL: https://doi.org/10.1145/996943.996945.

Bellare, Mihir and Chanathip Namprempre (2000). "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, pp. 531–545.

Bhargavan, Karthikeyan and Gaëtan Leurent (2016). "Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH". In: *Network and Distributed System Security Symposium–NDSS 2016*.

Bienstock, Alexander et al. (2022). "A more complete analysis of the Signal Double Ratchet algorithm". In: *Annual International Cryptology Conference*. Springer, pp. 784–813.

Binder, Matt (2022). *A teen hacked Uber and announced it in the company Slack. Employees thought it was a joke*. Available: https://mashable.com/article/uber-teen-hacker-slack-joke.

Blake-Wilson, Simon, Don Johnson, and Alfred Menezes (1997). "Key agreement protocols and their security analysis". In: *Crytography and Coding*. Ed. by Michael Darnell. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 30–45.

Blanchet, Bruno et al. (2018). "ProVerif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial". In: *Version from* 16, pp. 05–16.

Blazy, Olivier, Angèle Bossuat, et al. (2019). "SAID: Reshaping Signal into an identity-based asynchronous messaging protocol with authenticated ratcheting". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, pp. 294–309.

Blazy, Olivier, Ioana Boureanu, et al. (2023). "How fast do you heal? A taxonomy for Post-Compromise Security in secure-channel establishment". In: *USENIX 2023-The 32nd USENIX Security Symposium*.

Braeken, An (2018). "PUF based authentication protocol for IoT". In: *Symmetry* 10.8, p. 352.

Brands, Stefan and David Chaum (1994). "Distance-bounding protocols". In: *Advances in Cryptology—EUROCRYPT'93: Workshop on the Theory and Application of Cryptographic Techniques Lofthus, Norway, May 23–27, 1993 Proceedings 12*. Springer, pp. 344–359.

Campagna, Matthew (2013). "SEC 4: Elliptic curve Qu-Vanstone implicit certificate scheme (ECQV)". In: *Standards for Efficient Cryptography, Version* 1.

Canetti, Ran et al. (2022). "Universally composable end-to-end secure messaging". In: *Annual International Cryptology Conference*. Springer, pp. 3–33.

Cao, Phuong M et al. (2019). "CAUDIT: Continuous auditing of SSH servers to mitigate brute-force attacks". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 667–682.

Ceci, L. (2023). *Number of unique WhatsApp mobile users worldwide from January 2020 to June 2023 (in millions)*. https://www.statista.com/statistics/1306022/whatsapp-global-unique-users/.

Chase, Melissa et al. (2019). "SEEMless: Secure End-to-End Encrypted Messaging with less trust". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, pp. 1639–1656. URL: https://doi.org/10.1145/3319535.3363202.

Chatterjee, Urbi et al. (2018). "Building PUF based authentication and key exchange protocol for IoT without explicit CRPs in verifier database". In: *IEEE Transactions on Dependable and Secure Computing* 16.3, pp. 424–437.

Chaum, David L. (Feb. 1981). "Untraceable electronic mail, return addresses, and digital pseudonyms". In: *Communications of the ACM* 24.2, pp. 84–90. URL: https://doi.org/10.1145/358549.358563.

Cohn-Gordon, Katriel, Cas Cremers, Benjamin Dowling, et al. (2020). "A formal security analysis of the Signal messaging protocol". In: *Journal of Cryptology* 33, pp. 1914–1983.

Cohn-Gordon, Katriel, Cas Cremers, and Luke Garratt (2016). "On Post-Compromise Security". In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, pp. 164–178.

Cooper, David et al. (2008). *Internet X.509 public key infrastructure certificate and Certificate Revocation List (CRL) profile*. Tech. rep.

Cox, Joseph (Apr. 2018). *Chinese government forces residents to install surveillance app with awful security*. Available: https://www.vice.com/en/article/jingwang-app-no-encryption-china-force-install-urumqi-xinjiang/.

— (July 2019). *China is forcing tourists to install text-stealing malware at its border*. Available: https://www.vice.com/en/article/at-chinese-border-tourists-forced-to-install-a-text-stealing-piece-of-malware/.

Cremers, Cas, Jaiden Fairoze, et al. (2020). "Clone detection in secure messaging: Improving Post-Compromise Security in practice". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1481–1495.

Cremers, Cas and Michèle Feltz (2012). "Beyond eCK: Perfect Forward Secrecy under actor compromise and ephemeral-key reveal". In: *Computer Security – ESORICS 2012*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 734–751.

Cremers, Cas, Charlie Jacomme, and Aurora Naska (2023). "Formal analysis of session-handling in secure messaging: Lifting security from sessions to conversations". In: *USENIX Security*.

Daemen, Joan and Vincent Rijmen (2002). *The design of Rijndael*. 2nd ed. Springer.

daztucker (Sept. 2019). *README.privsep*. Available: https://github.com/openssh/openssh-portable/blob/master/README.privsep.

Denning, Dorothy E and Peter F MacDoran (1996). "Location-based authentication: Grounding cyberspace for better security". In: *Computer Fraud & Security* 1996.2, pp. 12–16.

Dierks, Tim and Eric Rescorla (2008). *The Transport Layer Security (TLS) protocol version 1.2*. Tech. rep.

Diffie, Whitfield and Martin E Hellman (1976). "New directions in cryptography". In: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, pp. 365–390.

Dolev, Danny and Andrew Yao (1983). "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2, pp. 198–208.

Dowling, Benjamin, Felix Günther, and Alexandre Poirrier (2022). "Continuous authentication in secure messaging". In: *European Symposium on Research in Computer Security*. Springer, pp. 361–381.

Dowling, Benjamin and Britta Hale (2021). "Secure messaging authentication against active Man-in-the-Middle attacks". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, pp. 54–70.

eIDAS 2023 Academics and Civil Society (2023). *Joint statement of scientists and NGOs on the EU's proposed eIDAS reform*. `https://eidas-open-letter.org/`.

Ekechukwu, Chikaodili, Dale Lindskog, and Ron Ruhl (2013). "A notary extension for the Online Certificate Status Protocol". In: *2013 International Conference on Social Computing*. IEEE, pp. 1016–1021.

Ellison, Carl (2007). "Ceremony design and analysis". In: *Cryptology EPrint Archive*.

Engelberg, Jerrod (Apr. 2021). *Bash uploader security update*. Available: `https://about.codecov.io/security-update/`.

Esfahani, Alireza et al. (2017). "A lightweight authentication mechanism for M2M communications in Industrial IoT environment". In: *IEEE Internet of Things Journal* 6.1, pp. 288–296.

European Commission (2023). *eIDAS regulation*. `https://digital-strategy.ec.europa.eu/en/policies/eidas-regulation`.

Facebook, Inc. (2017). *Messenger Secret Conversations technical whitepaper*.

Fett, Daniel, Ralf Küsters, and Guido Schmitz (2015). "Spresso: A secure, privacy-respecting Single Sign-On system for the web". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1358–1369.

Fisher, Keith Robert (Mar. 2020). *Update on border searches of electronic devices*. Available: `https://www.americanbar.org/groups/business_law/resources/business-law-today/2020-april/update-on-border-searches-of-electronic-devices/`.

Gao, Lijun et al. (2020). "An efficient secure authentication and key establishment scheme for M2M communication in 6LoWPAN in unattended scenarios". In: *Wireless Personal Communications* 115.2, pp. 1603–1621.

GitHub Docs (n.d.). *Creating a personal access token*. Available: `https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token`.

Goldbery, Sharon, Ann Ming Samborski, and Sebby Lipman (Oct. 2024). *Fearless SSH: Short-lived certificates bring Zero Trust to infrastructure*. Available: https://blog.cloudflare.com/intro-access-for-infrastructure-ssh/.

Goodin, Dan (July 2018). *Stolen certificates from D-Link used to sign password-stealing malware*. Available: https://arstechnica.com/information-technology/2018/07/stolen-certificates-from-d-link-used-to-sign-password-stealing-malware/.

Google Cloud (2022). *Authenticate using API keys*. Available: https://cloud.google.com/docs/authentication/api-keys.

Guest, Peter (2023). *The UK's controversial Online Safety Act is now law*. https://www.wired.co.uk/article/the-uks-controversial-online-safety-act-is-now-law.

Guo, Chengqian et al. (2021). "UPPRESSO: Untraceable and Unlinkable Privacy-PREserving Single Sign-On services". In: *arXiv preprint arXiv:2110.10396*.

Han, Jinguang, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer (2018). "Anonymous Single-Sign-On for n designated services with traceability". In: *European Symposium on Research in Computer Security*. Springer, pp. 470–490.

Han, Jinguang, Liqun Chen, Steve Schneider, Helen Treharne, Stephan Wesemeyer, and Nick Wilson (2019). "Anonymous Single Sign-On with proxy re-verification". In: *IEEE Transactions on Information Forensics and Security* 15, pp. 223–236.

Hanley, Mike (2022). *Security alert: attack campaign involving stolen OAuth user tokens issued to two third-party integrators*. Available: https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/.

Harchol, Yotam, Ittai Abraham, and Benny Pinkas (2018). "Distributed SSH key management with proactive RSA threshold signatures". In: *Applied Cryptography and Network Security*. Ed. by Bart Preneel and Frederik Vercauteren. Cham: Springer International Publishing, pp. 22–43.

Hardt, Dick (2012). *The OAuth 2.0 authorization framework*. Tech. rep.

Harkins, Dan and Dave Carrel (1998). *The Internet Key Exchange (IKE)*. Tech. rep.

Harris, Ben (2006). *RSA key exchange for the Secure Shell (SSH) Transport Layer Protocol*. Tech. rep.

Hoffman-Andrews, Jacob (2023). *Article 45 will roll back web security by 12 years*. https://www.eff.org/deeplinks/2023/11/article-45-will-roll-back-web-security-12-years.

Hussen, Hassen Redwan et al. (2013). "SAKES: Secure authentication and key establishment scheme for M2M communication in the IP-based wireless sensor network (6L0WPAN)". In: *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, pp. 246–251.

IBM (2022). *Creating an IBM Cloud API key*. Available: https://www.ibm.com/docs/en/app-connect/container?topic=servers-creating-cloud-api-key.

Irazoqui, Gorka et al. (2015). "Lucky 13 strikes back". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*.

ASIA CCS '15. Singapore, Republic of Singapore: Association for Computing Machinery, pp. 85–96. URL: https://doi.org/10.1145/2714576.2714625.

ISO (1989). "Information processing systems-open systems interconnection-basic reference model-part 2: Security architecture". In: *International Standard 7498* 2.

Jakobsson, Markus, Kazue Sako, and Russell Impagliazzo (1996). "Designated verifier proofs and their applications". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, pp. 143–154.

jlund (Oct. 2018). *Technology preview: Sealed sender for Signal*. https://signal.org/blog/sealed-sender/.

Johnson, Mark et al. (2022). *Save online speech coalition - letter to michelle donelan - final.pdf*. https://cloud.openrightsgroup.org/nextcloud/s/irGJD4GSRx3d4Mb.

Katz, Jonathan and Yehuda Lindell (2007). *Introduction to modern cryptography: Principles and protocols*. Chapman and hall/CRC.

Kerckhoffs, Auguste (1883). "La cryptographie militaire". In: *Journal des Sciences Militaires* IX, pp. 5–38.

Khalil, Issa, Zuochao Dou, and Abdallah Khreishah (2015). "TPM-based authentication mechanism for Apache Hadoop". In: *International Conference on Security and Privacy in Communication Networks: 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I 10*. Springer, pp. 105–122.

Kost, Edward (Nov. 2024). *What caused the Uber data breach in 2022?* Available: https://www.upguard.com/blog/what-caused-the-uber-data-breach.

Kovacs, Eduard (Mar. 2022). *Thousands of secret keys found in leaked Samsung source code*. Available: https://www.securityweek.com/thousands-secret-keys-found-leaked-samsung-source-code/.

Krawczyk, Hugo (2005). "HMQV: A high-performance secure Diffie-Hellman protocol". In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 546–566.

— (2010). "Cryptographic extraction and key derivation: The HKDF scheme". In: *Annual Cryptology Conference*. Springer, pp. 631–648.

Krawczyk, Hugo and Pasi Eronen (May 2010). *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. URL: https://www.rfc-editor.org/info/rfc5869.

LaMacchia, Brian, Kristin Lauter, and Anton Mityagin (2007). "Stronger security of authenticated key exchange". In: *Provable Security*. Ed. by Willy Susilo, Joseph K. Liu, and Yi Mu. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–16.

Lara, Evangelina et al. (2020). "Lightweight authentication protocol for M2M communications of resource-constrained devices in Industrial Internet of Things". In: *Sensors* 20.2, p. 501.

Lawlor, Sean and Kevin Lewi (2023). *Deploying key transparency at WhatsApp*. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/. Accessed: 2023-08-02.

Lockhart, Hal and B Campbell (2008). "Security Assertion Markup Language (SAML) v2.0 technical overview". In: *OASIS Committee Draft* 2, pp. 94–106.

Loske, Moritz, Lukas Rothe, and Dominik G Gertler (2019). "Context-aware authentication: State-of-the-art evaluation and adaption to the IIoT". In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. IEEE, pp. 64–69.

Malvai, Harjasleen et al. (2023). "Parakeet: Practical key transparency for end-to-end encrypted messaging". In: *Cryptology ePrint Archive*.

Marlinspike, Moxie (2016). *Safety number updates*. https://signal.org/blog/safety-number-updates/.

Marlinspike, Moxie and Trevor Perrin (2016a). *The Double Ratchet algorithm*. https://signal.org/docs/specifications/doubleratchet/.

— (2016b). *The X3DH key agreement protocol*. https://signal.org/docs/specifications/x3dh/.

— (2017). *The Sesame Algorithm: Session management for asynchronous message encryption*. https://signal.org/docs/specifications/sesame/.

McDaniel, Dwayne (Oct. 2022). *Toyota suffered a data breach by accidentally exposing a secret key publicly on GitHub*. Available: https://blog.gitguardian.com/toyota-accidently-exposed-a-secret-key-publicly-on-github-for-five-years/.

Meier, Simon et al. (2013). "The TAMARIN prover for the symbolic analysis of security protocols". In: *International Conference on Computer-Aided Verification*. Springer, pp. 696–701.

Menezes, Alfred J, Paul C Van Oorschot, and Scott A Vanstone (2018). *Handbook of applied cryptography*. CRC Press.

Microsoft (2022). *Use API keys for Azure Cognitive Search authentication*. Available: https://learn.microsoft.com/en-us/azure/search/search-security-api-keys#what-is-an-api-key.

Moriarty, Kathleen et al. (Nov. 2016). *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. URL: https://www.rfc-editor.org/info/rfc8017.

mozilla (2023). *Last chance to fix eIDAS: Secret EU law threatens internet security*. https://last-chance-for-eidas.org/.

NEAL (2017). *Hey Signal! Great encryption needs great authentication*. https://sequoia-pgp.org/blog/2021/06/28/202106-hey-signal-great-encryption-needs-great-authentication/. Accessed: 2023-08-02.

Newman, Lily Hay (Aug. 2019). *Everything we know about the Capital One hacking case so far*. Available: https://www.wired.com/story/capital-one-paige-thompson-case-hacking-spree/.

Nir, Yoav and Adam Langley (2015). *ChaCha20 and Poly1305 for IETF protocols*. Tech. rep.

NIST (July 2008). *The keyed-hash Message Authentication Code (HMAC)*. en. URL: https://csrc.nist.gov/pubs/fips/198-1/final.

— (Feb. 2023). *Digital Signature Standard (DSS)*. en. URL: https://csrc.nist.gov/pubs/fips/186-5/final.

NIST and Quynh Dang (Mar. 2012). *Secure Hash Standard (SHS)*. en. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=910977.

Open Whisper Systems (2017). *libsignal*. `https://github.com/signalapp/libsignal/tree/main`.

OpenSSH (n.d.). *OpenSSH*. Available: `https://www.openssh.com/`.

Ose, Greg (2022). *npm security update: attack campaign using stolen OAuth tokens*. Available: `https://github.blog/2022-05-26-npm-security-update-oauth-tokens/`.

Plaga, Sven et al. (2018). "Secure your SSH keys! Motivation and practical implementation of a HSM-based approach securing private SSH-keys". In: *17th European Conference on Cyber Warfare and Security*, pp. 370–379.

Poireault, Kevin (Sept. 2023). *Chinese hacker steals Microsoft signing key, spies on US government*. Available: `https://www.infosecurity-magazine.com/news/chinese-hacker-steals-microsoft/`.

Polk, Ryan (2022). *70 organizations, cyber security experts, and elected officials sign open letter expressing dangers of the UK's Online Safety Bill*. `https://www.globalencryption.org/2022/11/70-organizations-cyber-security-experts-and-elected-officials-sign-open-letter-expressing-dangers-of-the-uks-online-safety-bill/`.

Rasmussen, Kasper Bonne and Srdjan Capkun (2010). "Realization of RF distance bounding." In: *USENIX Security Symposium*, pp. 389–402.

Rawat, Rachit and Mahabir Prasad Jhanwar (2020). "PAS-TA-U: PASsword-Based Threshold Authentication with Password Update". In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, pp. 25–45.

Reese, Will (2008). "Nginx: The high-performance web server and reverse proxy". In: *Linux Journal* 2008.173, p. 2.

Renuka, KM et al. (2019). "Design of a secure password-based authentication scheme for M2M networks in IoT enabled cyber-physical systems". In: *IEEE Access* 7, pp. 51014–51027.

Rijmen, Vincent and Joan Daemen (2001). "Advanced Encryption Standard". In: *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology* 19, p. 22.

Rivest, Ronald L, Adi Shamir, and Yael Tauman (2001). "How to leak a secret". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, pp. 552–565.

Roy, Lawrence et al. (2022). "Practical privacy-preserving authentication for SSH". In: *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3345–3362.

Rührmair, Ulrich (2022). "Secret-free security: A survey and tutorial". In: *Journal of Cryptographic Engineering* 12.4, pp. 387–412.

Ryan, Keegan et al. (2023). "Passive SSH key compromise via lattices". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2886–2900.

Sakimura, Natsuhiko et al. (2014). "OpenID Connect Core 1.0". In: *The OpenID Foundation*, S3.

Schröder, Svenja et al. (2016). "When Signal hits the fan: On the usability and security of state-of-the-art secure mobile messaging". In: *European Workshop on Usable Security. IEEE*, pp. 1–7.

Sherry, Justine et al. (Aug. 2015). "BlindBox: Deep packet inspection over encrypted traffic". In: *ACM SIGCOMM Computer Communication Review* 45.4, pp. 213–226.

Signal (2023a). *Government requests*. https://signal.org/bigbrother/.

— (2023b). *Technical information*. https://signal.org/docs/.

Song, Dawn Xiaodong, David Wagner, and Xuqing Tian (2001). "Timing analysis of keystrokes and timing attacks on SSH". In: *10th USENIX Security Symposium (USENIX Security 01)*.

Statista (2024). "Internet of things - worldwide". In: Accessed: 2024-08-28. URL: https://www.statista.com/outlook/tmo/internet-of-things/worldwide.

Stebila, Douglas and Jon Green (2009). *Elliptic curve algorithm integration in the Secure Shell Transport Layer*. Tech. rep.

Steiner, Jennifer G., Clifford Neuman, and Jeffrey I. Schiller (1988). "Kerberos: An authentication service for open network systems". In: *USENIX Conference Proceedings*, pp. 191–202.

Teng, Wil Liam and Kasper Rasmussen (2023). "Actions speak louder than passwords: Dynamic identity for Machine-to-Machine communication". In: *Proceedings of the 18th International Conference on Availability, Reliability and Security*. ARES '23. Benevento, Italy: Association for Computing Machinery. URL: https://doi.org/10.1145/3600160.3600165.

— (n.d.). "Attack of the cloned: In-band active Man-in-the-Middle detection for the Signal protocol". In: Currently under submission.

Teng, Wil Liam, David Soler Garcia, and Kasper Rasmussen (n.d.). "How to heal a cracked shell: Post-Compromise Security for long term identity keys in SSH". In: Currently under submission.

Thammarat, Chalee and Chian Techapanupreeda (2021). "A secure authentication and key exchange protocol for M2M communication". In: *2021 9th International Electrical Engineering Congress (iEECON)*. IEEE, pp. 456–459.

Thomas, Elise (Aug. 2018). *Sydney airport seizure of phone and laptop 'alarming', say privacy groups*. Available: https://www.theguardian.com/world/2018/aug/25/sydney-airport-seizure-of-phone-and-laptop-alarming-say-privacy-groups.

Toor, Amar (July 2013). *UK border police can seize and download your phone's data for no reason at all*. Available: https://www.theverge.com/2013/7/15/4524208/uk-border-police-seize-download-mobile-phone-data-under-anti-terror-law.

UK Parliament (2023). *Online Safety Act 2023*. https://bills.parliament.uk/bills/3137.

Vatandas, Nihal et al. (2020). "On the cryptographic deniability of the Signal protocol". In: *International Conference on Applied Cryptography and Network Security*. Springer, pp. 188–209.

Vaziripour, Elham, Justin Wu, Mark O'Neill, Daniel Metro, et al. (2018). "Action needed! Helping users find and complete the authentication ceremony in Signal". In: *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pp. 47–62.

Vaziripour, Elham, Justin Wu, Mark O'Neill, Jordan Whitehead, et al. (2017). "Is that you, Alice? A usability study of the authentication ceremony of secure messaging applications". In: *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pp. 29–47.

VENAFI® (2021). *TLS machine identity management*. New Jersey: John Wiley & Sons, Inc.

Verizon, DBIR (2020). "Data breach investigations report 2020". In: *Computer Fraud & Security* 4, pp. 30059–2.

Wendlandt, Dan and Adrian Perrig (2008). "Perspectives: Improving SSH-style host authentication with multi-path probing". In: *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

WhatsApp (2023). *WhatsApp encryption overview technical white paper*.

Whittaker, Meredith and Joshua Lund (Nov. 2023). *Privacy is priceless, but Signal is expensive*. Available: `https://signal.org/blog/signal-is-expensive/`.

Williams, Stephen C. (2011). "Analysis of the SSH Key Exchange Protocol". en. In: *Cryptography and Coding*. Ed. by Liqun Chen. Vol. 7089. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 356–374. URL: `http://link.springer.com/10.1007/978-3-642-25516-8%5C_22`.

Ylonen, T. (2006a). *RFC4251 The Secure Shell (SSH) Protocol Architecture*. `https://www.rfc-editor.org/rfc/rfc4251.html`.

— (2006b). *RFC4252 The Secure Shell (SSH) Authentication Protocol*. `https://www.rfc-editor.org/rfc/rfc4252.html`.

— (2006c). *RFC4253 The Secure Shell (SSH) Transport Layer Protocol*. `https://www.rfc-editor.org/rfc/rfc4253.html`.

— (2006d). *RFC4254 The Secure Shell (SSH) Connection Protocol*. `https://www.rfc-editor.org/rfc/rfc4254.html`.

Yu, Jiangshan, Mark Ryan, and Cas Cremers (2017). "DECIM: Detecting endpoint compromise in messaging". In: *IEEE Transactions on Information Forensics and Security* 13.1, pp. 106–118.

Zhang, Feng, Aron Kondoro, and Sead Muftic (2012). "Location-based authentication and authorization using smart phones". In: *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, pp. 1285–1292.

Zhang, Yuan et al. (2020). "PROTECT: Efficient password-based threshold Single-Sign-On authentication for mobile users against perpetual leakage". In: *IEEE Transactions on Mobile Computing* 20.6, pp. 2297–2312.

Zhang, Zhenfeng, Yuchen Wang, and Kang Yang (2020). "Strong authentication without tamper-resistant hardware and application to federated identities." In: *NDSS*.

Zhang, Zhiyi et al. (2021). "EL PASSO: Efficient and lightweight privacy-preserving Single Sign On". In: *Proceedings on Privacy Enhancing Technologies* 2021.2, pp. 70–87.

Zhou, Lingli and Zhenfeng Zhang (2010). "Trusted channels with password-based authentication and TPM-based attestation". In: *2010 International Conference on Communications and Mobile Computing*. Vol. 1. IEEE, pp. 223–227.