Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Sub department of Discrete Mathematics

Secure Messaging in Mobile Environments

Sebastian R. Verschoor

Thesis submitted in conformity with
the requirements for the degree of
Master of Science
in
Information Security Technology
at the
Eindhoven University of Technology

Supervisors:
prof. dr. Tanja Lange
prof. dr. Daniel J. Bernstein

FINAL Version

Eindhoven, December 2015

Acknowledgement

This thesis is the result of my graduation project, concluding the Master's program in Information Security Technology at the Technical University of Eindhoven.

I would like to thank those who helped me to get to a successful completion of this project. First of all, I want to thank my supervisor Tanja Lange for encouraging me and guiding me through this project (and many other projects as well). Her feedback and support throughout the project has been invaluable for me. I would also like to thank Dan Bernstein for always taking the time for me and helping me out when I was stuck. Working with Tanja and Dan has taught me an incredible amount and has overall been a great experience. I would also like to thank Jerry den Hartog, without whom I would not have been able to model the protocol in Proverif.

I want to thank Steffen Jaeckel of the LibTom Projects, who has helped me with my first contributions to an open source cryptographic project. He was there to help me when I stumbled upon my first detached head in git.

Lastly, I would like to thank my family and my girlfriend for their moral support, understanding and encouragements over the past few months.

Eindhoven
December 2015

Sebastian R. Verschoor

Abstract

Silent Text, the instant messaging application by the company Silent Circle, provides its users with secure end-to-end encrypted communication on the Blackphone and other smartphones. The underlying protocol, SCimp, has received many extensions during the update to version 2, but has not been subjected to critical review from the cryptographic community. In September 2015, Silent Circle replaced SCimp with the TextSecure protocol in the Silent Phone application. In my master thesis, I analyze both the design and implementation of SCimp by inspection of the documentation and code, and I build a formal model of the protocol using Proverif. Many of the security properties provided by the core protocol are proven in the formal model, however many of the extensions contain vulnerabilities and the implementation contains bugs that affect the overall security. A comparison between SCimp and TextSecure highlights the improvements that the latter has over the former.

Keywords: SCimp, Instant Messaging, Smartphone, ECDHE, AES, CCM mode, Silent Circle, TextSecure, Axolotl, Off the Record, C, Proverif

Contents

List of Figures

List of Tables

List of Code

Chapter 1

Introduction

Silent Circle is an encrypted-communications firm, based in Geneva, Switzerland, founded by Mike Janke, Phil Zimmermann and Jon Callas. Among their security products is an instant messaging application called Silent Text, which is implemented for the iPhone and Android devices. Silent Text is the default messaging app for PrivatOS, which is the operating system running on the Blackphone (both developed by the same company).

The Silent Text application uses the Silent Circle instant messaging protocol (SCimp). SCimp enables users to have a private conversation over the instant message transport protocol XMPP (Jabber). The motivation for developing SCimp in-house was that the then existing protocols did not provide the required features or had unnecessary complexity.

In May 2014, Silent Circle announced the update to SCimp version 2.0, in which the protocol was extended [23]. With the in-house developed "Progressive Encryption", the protocol was designed to be more usable for the asynchronous environment of mobile phones, more cryptographic cipher suites were added and the ability for group messaging was added.

In September 2015, Silent Circle announced the discontinuation of the Silent Text application. Since September 28, SCimp has been replaced with a new secure text messaging protocol, integrated in the Silent Phone application. The new protocol is based upon the TextSecure protocol by Open Whisper Systems.

No academic analysis of SCimp has been done before. To be able to test the validity of the security claims, security proofs against a well-defined attacker model are required. This attacker model should include side-channel attacks, which implies that an analysis of the implementation is required as well.

Furthermore, metadata protection is getting increasingly important in current times of mass surveillance. Although Silent Circle makes no claims on privacy protection built into the protocol, it should be made explicit what metadata can be retrieved from communication over SCimp and if (further) privacy protection is possible with the existing design.

A modern secure communications protocol does not only consist of a way to establish initial key material, but it should also have a systematic way of updating those keys so that old keys can be erased in a secure way. The protocol for updating the keys is referred to as the *ratchet*, after the mechanical device that allows motion in one direction, but prevents motion in the opposite direction.

SCimp has several predecessors and now has gotten a successor as well. A brief analysis of these protocols, together with a comparison against the SCimp protocol, should provide insight in the design decisions behind SCimp and provide a rationale for the switch to a new protocol. I will take known attacks on these protocol from the literature and investigate if they apply to SCimp as well.

1.1 Attacker model

To claim that the protocol is secure, a well-defined attacker model is required in order to specify what the protocol is secure *against*. By defining the goals that adversaries might have and defining their capabilities, it becomes clear what the protocol needs to defend against and which security properties it should provide to the end-users. Throughout this thesis, I will follow the tradition of naming these end-users Alice and Bob, while reserving the name Eve to represent the adversary.

1.1.1 Attacker goals

The attacker goals are closely tied to the security properties of the secure messaging protocol. Table 1.1 lists the different goals that an attacker might have and the corresponding security property that a protocol should provide in order to be considered secure.

Table 1.1: Attacker goals

Attacker goal	Security property
Compromise messages	Confidentiality of messages
Alter sent messages	Integrity of messages
Inject false messages	Authenticity of messages
Identify as another person	Authentication of communication partner
Block communication	Availability of communication
Learn communication metadata	Privacy protection
Prove *what* was said	Deniability of message content
Prove that two persons communicated	Deniability of the conversation
Learn past communication after compromise	Forward secrecy/key erasure
Prolong a successful attack	Future secrecy

Not every attack can be defended against by a secure messaging protocol. It is especially hard to provide availability when an attacker is assumed to be able to block messages on the communications network. Having said that, the protocol should not make it easy for an attacker to block communication.

To protect the privacy of the users, the protocol should not leak metadata about the users' communication, such as who they are communicating with, how many messages they sent and from where. Cell tower data and communication layers below the secure messaging protocol might leak this data as well, but it could be hidden through anonymity tools such as Tor. In that case, the protocol itself should not reveal any metadata.

To provide deniability, it should be impossible for anyone to provide convincing proof to a third party about past communication. To deny that any conversation ever took place is a stronger claim than just denying the precise contents of a message.

Key erasure and future secrecy are properties that ensure some damage control in case that a device or key does get compromised. Key erasure ensures that keys that are currently on the device do not compromise any past communication, so that the impact of a device compromise is minimized. Future secrecy ensures that an attacker that has compromised a key in the past, does not get to prolong his attack indefinitely. This is often achieved by introducing fresh randomness that should be unknown to the adversary.

1.1.2 Attacker capabilities

A base model for the attacker is the Dolev-Yao model [12], in which the attacker has full control over the network. The attacker can listen to, alter, inject and drop any message on the network.

However, real attackers have capabilities beyond control over the network. By inspecting the physical properties of the implementation, they might learn secret information that is on the communication device. This is called a side-channel attack. Device compromises can also

be achieved by low-tech attacks such as a rubber-hose attack or through legal procedures. An attacker is assumed to learn information through side-channels and to get their hands on the device temporarily.

An issue with some existing protocols is that users need to trust in the communications server that is being used. Even if you trust the organization that runs the server, you might not trust the government of the country in which the server is located to protect your privacy. Therefore, the attacker is assumed to have full control over the server that is used for communication.

The last capability that is given to the attacker is to compromise protocol participants themselves. When Alice communicates with Bob, the protocol should provide some protection in case Bob turns out to be a dishonest participant. Basically, the protocol should enforce Bob to play by the rules.

Chapter 2

Silent Circle instant messaging protocol

The company Silent Circle developed the SCimp protocol for ensuring confidential and authenticated communication between two mobile devices. It provides **end-to-end encryption** in such a way that no intermediary—such as the Silent Circle server—needs to be trusted in order for the communication to be secure. SCimp supports **erasure of old keys**, so that a potential device compromise does not leak old messages (which is also known as forward secrecy), by updating the encryption key for every message and rekey every once in a while. The protocol allows for **deniability** of all communication, which means that nobody, including the users Alice and Bob themselves, can provide any proof to convince a third party about what was said over SCimp or even provide any proof that a conversation took place in the first place. The last important property of SCimp is that it provides **future secrecy**: an attacker that compromises key material shared between users, but misses the opportunity to set up a man-in-the-middle attack during rekeying, cannot decrypt future messages. This property is also known as the "self-healing property".

The protocol comes in two versions. SCimp version 2 is in use since May 2014 [23], but the details have only been released in August 2015 [31]. Version 1 has two different modes[1]: ClearText mode and basic Diffie-Hellman (DH) mode. In the first mode, messages are sent unencrypted and not authenticated. Since this provides no security for the communication, I will not further analyze this mode. The second mode is the basic mode of operation for communication: it uses an ephemeral Diffie-Hellman key exchange (explained in Section 2.1) to establish a shared key between two devices. Users do not have long-term keys in SCimp (version 1), so after key negotiation, users need to verify the identity of the other party by confirming a *short authentication string* (SAS) out-of-band. SCimp uses the derived keys to perform authenticated encryption to provide confidentiality and integrity of the messages, as explained in Section 2.3. For enhanced security, the users periodically need to renegotiate their keys, which is explained in Section 2.2.

The main goal of SCimp version 2 is to add some features to enhance usability of the protocol. One problem with the first version is that the users need to do a complete DH exchange before they can send the first message. In the asynchronous environment of mobile devices, this can introduce a big delay. To solve this, Silent Circle invented "Progressive Encryption" (DHv2 mode). Users have a medium-term DH key-pair of which they upload the public key to the Silent Circle server. When Alice wants to send a message to Bob, she downloads that key and derives key material for encryption of the first message, but she also initiated a key negotiation as in basic DH mode. When that key negotiation completes, the users can verify each others identity and confirm that all communication so far has been secure. This mode is explained in Section 2.5. Version two also introduces group messages (explained Section 2.6), which require two more SCimp modes: PubKey mode and Symmetric mode.

[1] also referred to as SCimp methods

Silent Circle also provides some security/usability features to their application by adding functionality in a layer on top of SCimp, called Siren (see Section 2.7). For the SCimp protocol, Siren messages are just normal data messages. However, some of the features that Siren provides do affect the security properties of the protocol. In particular, message signatures in Siren can undermine some deniability of the message and sending files with Siren (using the Silent Circle cloud) is not as secure as it should be.

Each section of this chapter will analyze a component of the SCimp protocol. I will identify which parts ensure which security properties and I identify weaknesses of the protocol design where there are any. The documentation by Silent Circle is not always consistent with the implementation and misses some details. Therefore, my analysis is based upon the implementation, but I will point out the most important differences. The last section of this chapter (Section 2.9) identifies problems with the SCimp protocol that do not fit in any other section.

Message formats

Throughout this chapter, the term *message* can refer to the different layers that are used in the protocol. When the context does not make clear which layer is being referred to, I will name them according to Table 2.1.

Table 2.1: Silent Text message format

User message (text)
Siren message (JSON, base64)
SCimp message (JSON, base64)
XMPP stanza (XML)
TLS (binary)
TCP (binary)

A *User message* is simply the message that the user wants to send, such as "Hi Bob". This message is encapsulated in a Siren message (see Section 2.7) in JSON format. The Siren message is the data part of a SCimp message that is encrypted.

The SCimp message is encoded in JSON, with the binary fields (including the encrypted Siren message) encoded in base64. The SCimp message is encoded as base64 with prefix "`?SCIMP:`" and suffix "`.`". Each SCimp message has a tag that identifies the type of message.

SCimp messages are encapsulated in an XMPP stanza. The XMPP stanzas contain the SCimp message, a unique stanza identifier and the XMPP addresses[2] of both sender and receiver.

The Silent Circle server acts as the XMPP server and is responsible for delivering the XMPP stanzas to the devices. The devices retrieve the messages over a TLS/TCP session with the server. This layer of TLS encryption provides basic security against eavesdropping, but it does not provide end-to-end encryption. In my analysis, compromising the server is within the capability of the attacker, so that the TLS layer cannot be relied upon to provide any security.

2.1 Key negotiation

Alice and Bob are both registered by their XMPP addresses: $A =$ "alice@silentcircle.com/blackphone" and $B =$ "bob@silentcircle.com/android". When Alice wants to communicate with Bob for the first time, she starts the initial key negotiation.

It is not specified how Alice and Bob learn each other's XMPP addresses. Alice needs to know the address (B'), to which she wants to send a message. When Bob receives the first message, he can extract the sender address (A') from the stanza. For the analysis, I assume that these addresses are not authenticated, meaning that Alice does not know if $B' = B$ and Bob does not know if $A' = A$.

[2] also known as Jabber IDs (JIDs)

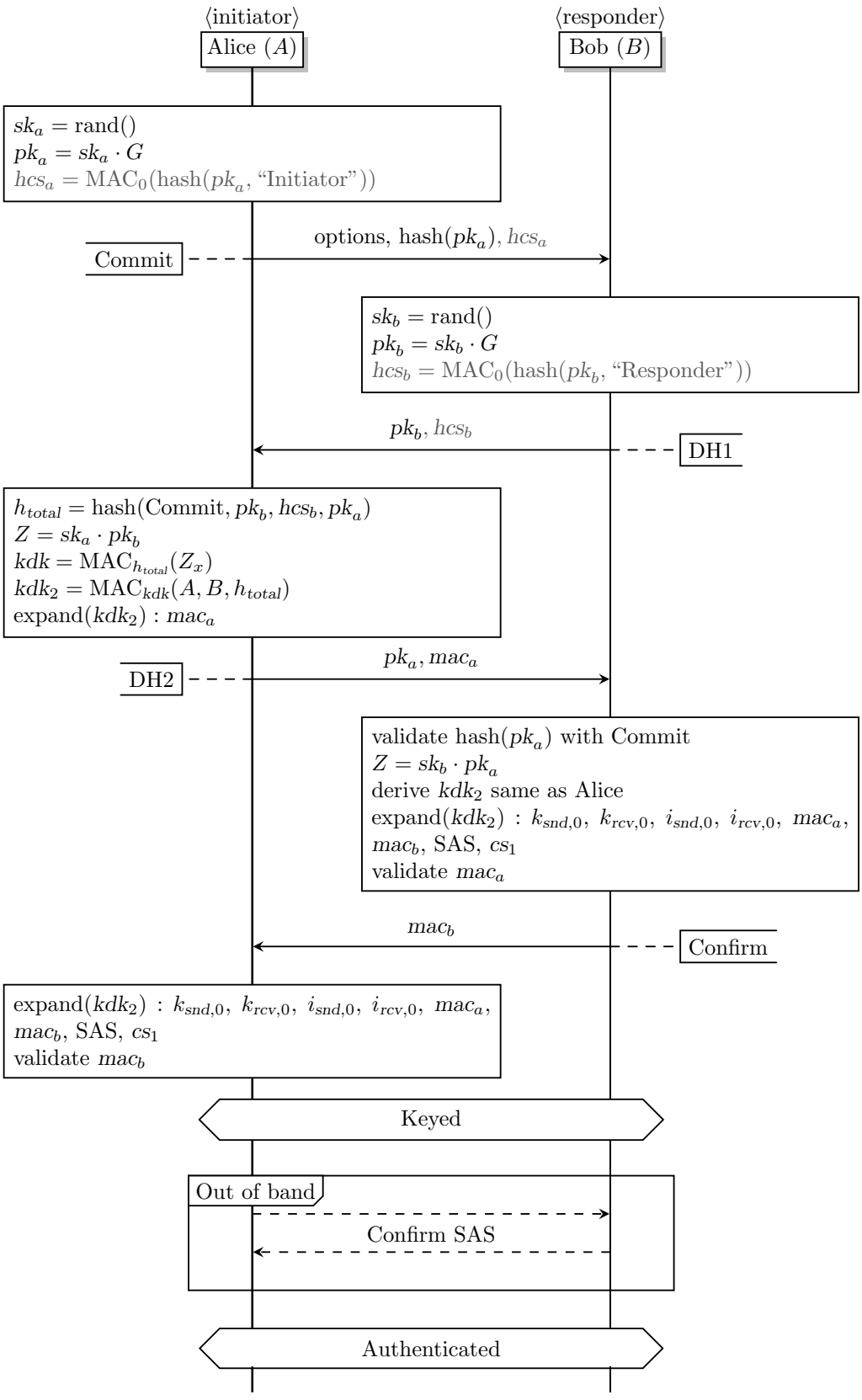

Figure 2.1: SCimp: first key negotiation

The initial key negotiation is basically an *ephemeral elliptic curve Diffie-Hellman* key exchange (ECDHE) with a subsequent confirmation of knowledge of the derived shared secret. It consists of four messages:

1. Alice commits to her public DH key in the **Commit** message.
2. Bob sends his public DH key in the **DH1** message.
3. Alice opens her commitment and confirms knowledge of the shared secret in the **DH2** message.
4. Bob confirms knowledge of the shared secret in the **Confirm** message.

The key negotiation is depicted in Figure 2.1.

2.1.1 Commit

Alice, as initiator, starts by computing an ephemeral secret key for her DH key-pair, which is a random key sk_a. She computes the public key of her DH key-pair:

$$pk_a = sk_a \cdot G, \tag{2.1}$$

where G is a fixed base point on the elliptic curve.[3] Alice commits to point pk_a by computing $\text{hash}(pk_a)$.

The value hcs_a is computed, even though it is superfluous in the first key negotiation.[4] The value is the result of a MAC with zero as the key.

$$hcs_a = \text{MAC}_0(\text{hash}(pk_a \parallel \text{“Initiator”})) \tag{2.2}$$

There are two discrepancies here between the documentation and the implementation. The first is that the value 0 is substituted for cs instead of a random value. With a random value, Bob can ignore hcs_a from everyone he has not communicated with before. It has the advantage that the message does not leak if this is the first key negotiation or not. Indeed, with the implemented version, this information is leaked to an attacker that computes hcs_a using $cs = 0$ and compares the sent value.

The second discrepancy is which value is hashed: according to the documentation the “Initiator” string is not digested: $hcs_a = \text{MAC}_{cs}(\text{hash}(pk_a) \parallel \text{“Initiator”})$. In the implementation, Alice does digest this string. With the documented design, Bob can check the validity of hcs_a before engaging in key generation. He can ignore invalid Commit messages, whereas with the implementation he needs to wait for Alice to open her commitment in message DH2. This makes the protocol less efficient and opens up the possibility for a denial-of-service (DOS) attack, where Eve can trick Bob into opening many SCimp sessions and force Bob to do more work than necessary in order to detect that Eve is not playing by the rules.

In her Commit message, Alice also sends the options she wants to use for the communication. These options have integer values and they determine which cryptographic primitives and security options are going to be used.

Option: version

The `version` identifies the SCimp version, the value can be 1 or 2. For backwards compatibility, SCimp implementations of version 2 still accept version 1.

Table 2.2: SCimp cipher suites

Suite	Hash	MAC/KDF	Cipher	Curve
1	SHA-256	HMAC/SHA-256	AES-128	NIST P384
2	SHA-512/256	HMAC/SHA-512	AES-256	NIST P384
3	SKEIN-512/256	SKEINMAC-512	AES-256	NIST P384
4	-	-	AES-128	-
5	-	-	AES-256	-
6	SKEIN-512/256	SKEINMAC-512	TWOFISH-256	Curve41417

Option: cipherSuite

The `cipherSuite` specifies the cryptographic primitives that are to be used. The possible values are listed in Table 2.2. Cipher suites 4 and 5 are reserved for Symmetric SCimp and cannot be used for DH key negotiation.

The number suffix of the cipher (128 or 256) determines the key length in bits, denoted by l from here on.

Option: sasMethod

The `sasMethod` option defines the method in which the short authentication string is displayed to the user. The number of bits in the SAS depends on the SCimp method as well. The SAS methods are shown in Table 2.3. ZJC11 and NATO are invalid SAS methods for SCimp methods PubKey and Symmetric.

Table 2.3: SCimp SAS methods

Method	Name	Description	SCimp method	Security (bits)
1	ZJC11[5]	4 base32 characters	DH / DHv2	20
2	HEX	hexadecimal	DH / DHv2	20
			PubKey / Symmetric	64
3	NATO	NATO alphabet	DH / DHv2	20
4	PGP	PGP word list	DH / DHv2	16
			PubKey / Symmetric	32

The SAS value is the result of a KDF computation. If the SCimp method is PubKey, then the SAS is computed as a 64 bit value, else it is a 20 bit value.

2.1.2 DH1

Upon receiving the Commit message, Bob checks the options to see if they are compatible with his device or else aborts the protocol. He stores the options and received values hcs_a and $\text{hash}(pk_a)$. Bob continues by generating his own DH key-pair, with the random scalar sk_b and public key pk_b:

$$pk_b = sk_b \cdot G. \tag{2.3}$$

Bob also computes the value hcs_b:

$$hcs_b = \text{MAC}_0(\text{hash}(pk_b \parallel \text{“Responder”})). \tag{2.4}$$

Bob sends his reply (DH1) to Alice, containing the value pk_b and hcs_b.

[3] See [9] for the coordinates of G.
[4] Section 2.2 explains the function of cs
[5] Also known as ZB32 encoding. See [38], section 5.1.6, for the specification of this encoding.

2.1.3 DH2

Alice now has enough information to complete the Diffie-Hellman key exchange. First, Alice checks if the point she received is a valid point on the curve. Next, she computes shared secret Z_x, which is the x-coordinate of point Z on the elliptic curve:

$$Z = sk_a \cdot pk_b. \tag{2.5}$$

From this value, all other key material is derived using a three step process, adapted from the two step process Extract-Expand [7]. An additional Enhance step is added between the steps.

Extract

First, a 256 bit seed h_{total} is computed:

$$h_{total} = \text{hash}(\text{Commit} \parallel \text{DH1} \parallel pk_a). \tag{2.6}$$

Note that the value of pk_a is already mixed in the Commit message as a hash. It appears to be unnecessary to include it in h_{total} again, but this is probably just an implementation error.

A key derivation key kdk is computed from Z_x and h_{total}:

$$kdk = \text{MAC}_{h_{total}}(Z_x). \tag{2.7}$$

Enhance

The reason this step is called enhance is that it mixes in the cached secret, if there was a cached secret shared between the participants. At the first key negotiation, the value `0x00` is used instead.

To derive keys, a *Key Derivation Function* (KDF) is used, which is just a MAC function with specified parameters:

$$\text{KDF}_k(\text{label}, \text{context}, L) = \text{MAC}_k(\texttt{0x00000001} \parallel \text{label} \parallel \texttt{0x00} \parallel \text{context} \parallel L). \tag{2.8}$$

L is a parameter that specifies the output length of the key in bits. The output of the MAC is truncated to the L leftmost bits. The first argument, `0x00000001`, is a counter that is required by NIST Special Publication 800-108 [7]. If you need more bits, this specifies how to extract them. For SCimp, the value is fixed at one because L is always smaller or equal to the MAC output length.

The KDF requires a context parameter that can be the same for all keys that are derived. For unexplained reasons, the protocol computes two values, a context variable:

$$ctx = \text{len}(A) \parallel A \parallel \text{len}(B) \parallel B \parallel h_{total} \tag{2.9}$$

and a session variable, used in later computations:

$$sess = \text{hash}(\text{len}(A) \parallel A \parallel \text{len}(B) \parallel B). \tag{2.10}$$

With these values, the key derivation key can be enhanced.

$$kdk_2 = \text{KDF}'_{kdk}(Z_x, \text{“MasterSecret”}, \text{“SCimp-ENHANCE”} \parallel ctx \parallel \texttt{0x00}, 256) \tag{2.11}$$

Note that the function KDF' differs from the regular KDF function as it has four arguments: the value Z_x is prepended to the digested value in the MAC computation. The value Z_x was already mixed in the value of kdk, so there appears to be no reason to include it again in kdk_2. Since the documentation does not mention this parameter, it appears to be an implementation error and kdk_2 should have been computed with the regular KDF function.

Silent Circle does not explain in its documentation why it is necessary to make the Extract and Enhance steps separate steps. The design could be simplified by including the value cs in the extract step.

Expand

The expand step derives multiple keys from kdk_2, all serving different purposes. This is done by computing the key derivation function KDF with key kdk_2.

For the first key negotiation, only mac_a is derived (and kdk_2 is stored for late use). Otherwise, Alice "expands" the following values, where l is determined by the key size of the symmetric cipher of the cipher suite.

$$k_{snd,0} = \mathrm{KDF}_{kdk_2}(\text{“InitiatorMasterKey”}, ctx, 2l) \tag{2.12}$$

The key $k_{snd,0}$ is the key that Alice uses to encrypt and authenticate messages to Bob.

$$i_{snd,0} = \mathrm{KDF}_{kdk_2}(\text{“InitiatorInitialIndex”}, sess, 64) \tag{2.13}$$

$i_{snd,0}$ is an index that is attached to a message, so that the receiver can identify which key should be used to decrypt the message. This is necessary when messages are dropped or arriving out of order.

In order to receive messages from Bob, Alice also derives a key

$$k_{rcv,0} = \mathrm{KDF}_{kdk_2}(\text{“ResponderMasterKey”}, sess, 2l) \tag{2.14}$$

and index

$$i_{rcv,0} = \mathrm{KDF}_{kdk_2}(\text{“ResponderInitialIndex”}, sess, 64). \tag{2.15}$$

What is remarkable is that there is an asymmetry between the derivation of $k_{snd,0}$ and $k_{rcv,0}$: the former uses ctx and the latter $sess$ as context parameter to the KDF.

Alice proves to Bob that she has been able to complete the DH exchange by sending

$$mac_a = \mathrm{KDF}_{kdk_2}(\text{“InitiatorMACkey”}, ctx, 256) \tag{2.16}$$

to Bob. Symmetrically, Bob authenticates by sending

$$mac_b = \mathrm{KDF}_{kdk_2}(\text{“ResponderMACkey”}, ctx, 256) \tag{2.17}$$

to Alice.

The short authentication string (SAS) is used by the participants to verify each other's identity and to make sure that no man-in-the-middle was present during the key negotiation (see Section 2.1.5).

$$\mathrm{SAS} = \mathrm{KDF}_{kdk_2}(\text{“SAS”}, ctx, 20) \tag{2.18}$$

The cached secret cs_1 is derived to ensure key continuity. The computed value can be used in future key negotiations to authenticate to the other party.

$$cs_1 = \mathrm{KDF}_{kdk_2}(\text{“RetainedSecret”}, ctx, 2l) \tag{2.19}$$

In order to prevent a denial of service attack, the client replaces the stored value of cs with cs_1 only after the other party has authenticated with their mac.

For receiving messages, 16 keys are computed in advance and put in an array. Each receive key is derived from the previous one with the *SCimp ratchet*, which is implemented as follows:

$$k_{x,j+1} = \mathrm{KDF}_{k_{x,j}}(\text{“MessageKey”}, sess \,\|\, i_{x,j}, l), \tag{2.20}$$

where $x \in \{snd, rcv\}$.

The index must be ratcheted forward as well, this is done simply by addition:

$$i_{x,j+1} = i_{x,j} + 1. \tag{2.21}$$

Sending

Alice sends pk_a to Bob to open up the commitment and adds the value mac_a to prove that she was able to complete the DH computation. She can now erase the ephemeral value sk_a from her device.

2.1.4 Confirm

Upon receiving the DH2 message, Bob can validate that the value pk_a that he received corresponds with the one that Alice committed to in her Commit message. After this verification, he can complete the Diffie-Hellman computation:

$$Z = sk_b \cdot pk_a. \tag{2.22}$$

By the same extract/enhance/expand steps, Bob derives all the necessary keys. The only difference with Alice is that Bob refers to the key that Alice uses to send messages to Bob as $k_{rcv,j}$. He checks the computed value of mac_a with the received value. If the values match, Bob is convinced that Alice was able to complete her DH exchange. He stores the required keys, updates his cached secret cs to cs_1 and is ready to communicate with Alice.

Bob now responds with the Confirm message, which consists of only the value mac_b. He can erase all intermediary keys from his device, such as sk_b, Z, kdk and kdk_2.

Upon receiving the Confirm message, Alice compares her computed value of mac_b with the received one. When they match, she updates her cached secret cs to cs_1, deletes all intermediary keys from her device and is ready to communicate with Bob.

2.1.5 Short authentication string

At this stage in the protocol, Alice and Bob have the necessary keys to start sending data and indeed they can. However, it is better for them to verify the identities of their communication partner first. The key exchange so far has been done using only ephemeral keys, so setting up a man-in-the-middle attack is trivial for an adversary with sufficient control over the network. The solution by Silent Circle is the short authentication string (SAS). The method for displaying the SAS to the user and the length of the SAS depends on the `sasMethod` option, see Table 2.3.

Alice and Bob have both computed a SAS value at this stage. They need to set up an out of band connection on which they can authenticate each other before they start comparing their SAS values. According to Silent Circle [36], "[a] phone call would be sufficient for this purpose since confidence building cues such as voice timbre and manner of speech are present." Most options for displaying the SAS to the user are indeed optimized for voice communication. When the users are confident of the identity of they person they are speaking with, they can verify the SAS.

Silent Circle is not specific on what it means to verify the SAS. The fact that this action should result in mutual authentication, suggests that both parties will have to read half of the SAS code that is displayed to them.

Commit

The SAS is a short code. Therefore, the protocol requires the hash commitment, which forces the adversary to select a public key without knowing the key of the other participant. Without the commitment, the adversary could acquire the public keys of the honest participants before searching for corresponding keys that would result in a SAS collision. Such a collision would result in an undetected man-in-the-middle attack. This is displayed in Figure 2.2. The Commit message ensures that an adversary can have only one blind guess for his public key.

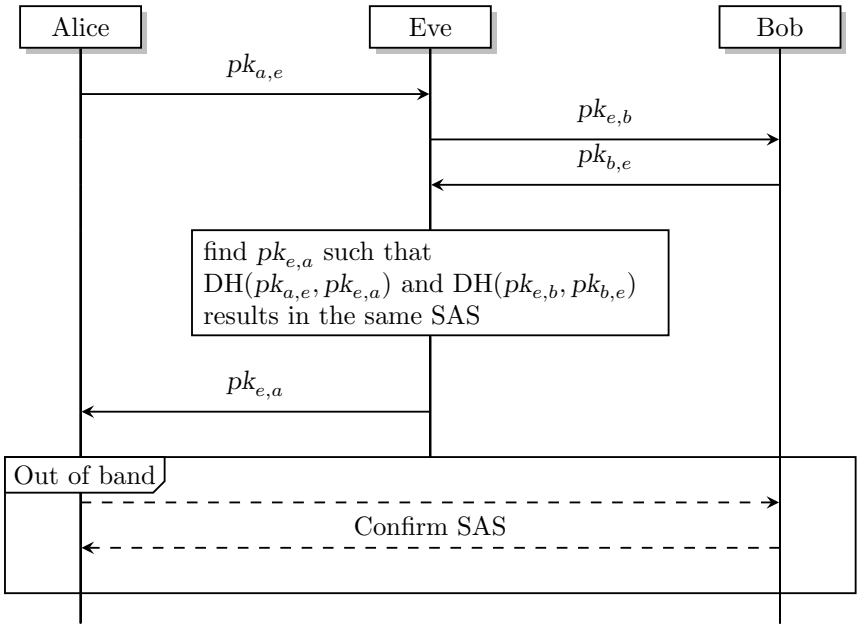

Figure 2.2: Man-in-the-middle attack on key negotiation without commit

2.2 Rekeying

When the participants have negotiated keys before, they share a cached secret (cs), derived from kdk_2. This secret is used for key continuity, meaning that future keys will depend on this value and thus depend upon previous keys. When Alice wants to derive a new key, she initiates the same process as for the initial key negotiation.

The difference with the initial key negotiation is highlighted in Figure 2.3. The values of hcs_a and hcs_b must now be computed with the stored value of cs, instead of 0. Alice must validate the value of hcs_b, sent in message DH1, by computing the value herself. Similar, Bob validates hcs_a after receiving pk_a in DH2. Additionally, the enhance function becomes slightly different from Equation 2.11:

$$kdk_2 = \text{KDF}'_{kdk}(Z_x, \text{“MasterSecret”}, \text{“SCimp-ENHANCE”} \parallel ctx \parallel \boldsymbol{cs}, 256) \tag{2.23}$$

When the received value of hcs is invalid, a warning is issued to the user that the identity of the other party is no longer verified. The protocol continues as if it was the first key negotiation and replaces cs with `0x00`, as in Equation 2.11. That also means that the SAS needs to be confirmed again to ensure that no man-in-the-middle attack is in progress. The reason that the protocol does not abort is that an honest user could lose their copy of the cs, for example, due to a device reset or loss of connection when Bob has updated cs but Alice has not.

Before $k_{rcv,0}$ is overwritten with the newly expanded value, the array of precomputed receive keys is inspected for unused key values. The key with the lowest index is stored as a seed key for later usage, so that messages that arrive out of order (sent before rekeying but received after rekeying) can still be decrypted.

2.2.1 Key erasure

Whenever new keys are negotiated successfully, these keys will depend upon the old keys, but this is a one-way process. The old keys cannot be derived from the new keys. The KDF, MAC and

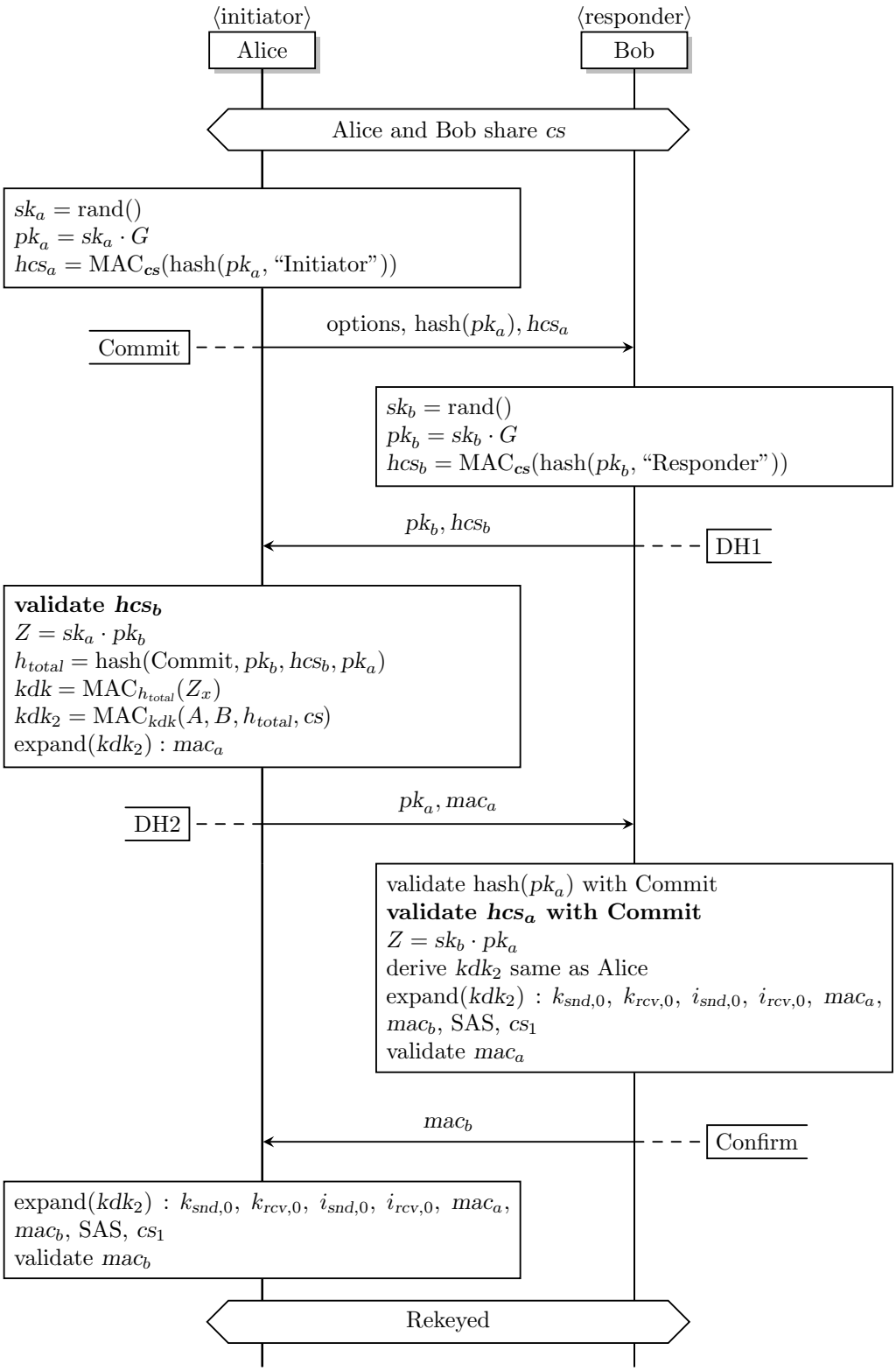

Figure 2.3: SCimp: key negotiation

hash functions are all one-way functions and fresh randomness is introduced in both values of sk. The old keys are erased, meaning that a compromise of key material does not compromise the security of old messages.

There is a problem with the stored seed keys that are kept to be able to decrypt messages from a previous key negotiation. If a message was withheld from the receiver, either by a network failure or by an adversary, the key will remain stored in the array of precomputed receive keys. If rekeying happens before the the key is erased from the array (before 16 messages have been received), that key will be stored as the seed key. This key does not only compromise the key erasure property of the withheld message, but of all messages until rekeying, because the keys for those messages can be derived from the seed key.

2.2.2 Future secrecy

Although it is not documented when keys should be renegotiated, for future secrecy it is important that this happens often. The SCimp ratchet (see Section 2.3) derives each message key directly from the previous one, so when one message key gets compromised, all following message keys are compromised as well, until new keys are negotiated with the rekeying protocol.

An attacker that has compromised the message keys from one participant, but does not have access to either sk_a or sk_b that is used in key renegotiation, will no longer have any knowledge of the freshly generated keys.

On the other hand, compromising the current value of cs is enough for an adversary to set up an undetected man-in-the-middle between the participants. However, as soon as the adversary misses one key negotiation, cs is replaced with a fresh value. The participants will receive a warning that the old values of cs did not match, so they will have to reconfirm the SAS. Additionally, even when an undetected man-in-the-middle attack is in progress, the participants should be able to detect this by reconfirming the SAS after rekeying.

Silent Circle calls the future secrecy property of rekeying the "self-healing property", which is a bit of a misnomer. Only when the adversary misses the first rekeying, will the protocol self-heal. If the adversary has already successfully set up a man-in-the-middle attack in the past and then misses one rekeying, the protocol only detects the error, but does not self-heal. Healing is done by the users who reconfirm the SAS.

The important part for future secrecy is that the adversary misses the key negotiation. When users communicate over the internet, it might be easy for an adversary to intercept all communication, especially if all messages are routed via a single node, such as the Silent Circle server. Because the protocol does not rely on the underlying transportation layers, it should be possible for two participants to do a full key negotiation out of bounds. For example, when they physically meet, they can exchange the key messages using near field communication (NFC) or by scanning QR-codes. This would make it very unlikely that a man-in-the-middle is present, giving the users a guarantee that future communication is secure and giving them the ability to detect whether past communication was secure. I am unaware of such a functionality in Silent Text.

2.3 Sending User messages

When both sender and receiver have derived the keys, they can start sending messages, see Figure 2.4. We assume Alice sends a message to Bob, but the other way around would be the same.

The message is encrypted with AES in CCM-mode (Counter with CBC-MAC) [37]. CCM is a mode that provides Authenticated Encryption with Associated Data (AEAD). That means that besides encrypting and authenticating plaintext, the mode also accepts a header (Associated Data) that is authenticated, but not encrypted. CCM also requires a nonce N that is unique per encryption key k:

$$ct = \text{AES\_CCM}_k^N(\text{header}, pt) \tag{2.24}$$

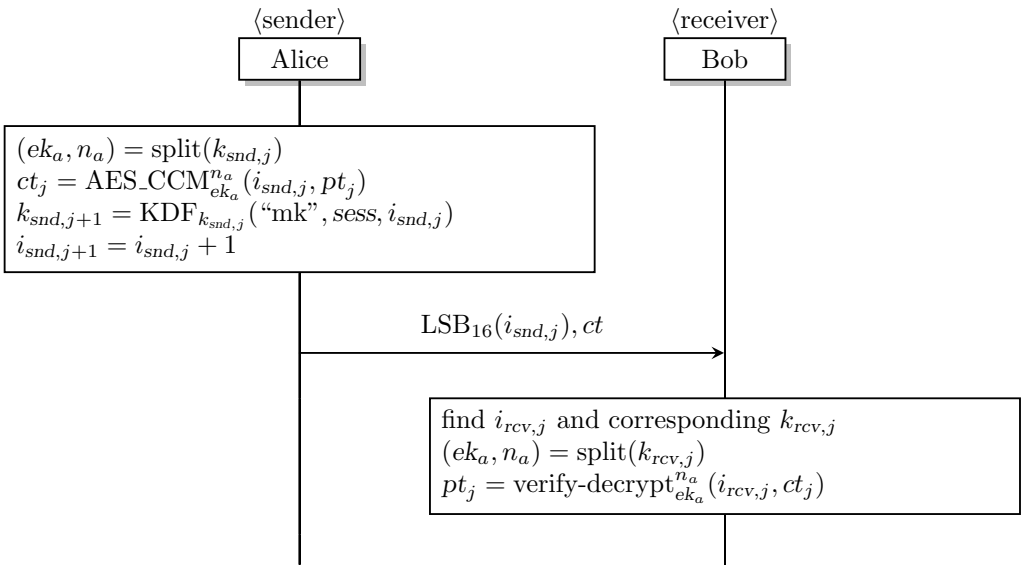

Figure 2.4: SCimp: data exchange

CCM encryption is specified as follows: first compute a CBC-MAC over the nonce (N), header and the plaintext pt, resulting in an authentication tag T. Concatenate $pt \,\|\, T$ and encrypt it in counter mode, using N to initialize the counter, resulting in ciphertext ct.[6]

In case of SCimp, the key k and the nonce N are set to the value of $k_{snd,j}$, split into two equal-sized halves. For the header value, SCimp uses the full 64-bit value of index $i_{snd,j}$. In order to achieve key erasure, the key is updated with every message that is sent. This is simply done by computing the ratchet from equations 2.20 and 2.21. Besides ciphertext ct, Alice also sends the 16 least significant bits of the send index $i_{snd,j}$, which Bob can use to find the correct key for the message.

2.3.1 Receiving

Upon receiving the message, Bob inspects the sent index to retrieve the corresponding key. He does this by inspecting the array of precomputed receive keys and seeking if an index matches. When it does, he copies the stored key for use in decryption and erases the original. When the precomputed array starts to get empty (four or more receive keys have been erased), he forwards the ratchet to fill the array again. Keys older than 16 messages get discarded. If Bob cannot find the key index in the array, he inspects the stack of old seed keys. If the difference in indices is smaller than 32, the key is derived from the seed key. Upon finding the key, Bob can decrypt and verify the message.

2.4 Commit contention

Two participants might try to initiate a key negotiation at the same time. According to the SCimp whitepaper, the protocol will then flag an error and let the application decide what to do. The suggestion they give is to compare the values of the hash commitments and let that comparison decide which participant becomes initiator and which becomes responder.

What actually happens is that when Alice receives an unexpected commit message, she resets her state, issues a warning and continues to process the commit message. This means she will

[6] The actual specification is more convoluted, including authentication of lengths and data encoding instructions. See [13] for further details.

send out the following DH1 message. Bob, on the other hand, has just done the same and sent out his DH1 message. Upon receiving the DH1 message of Alice, he throws a protocol error and resets his protocol context. Resetting the context includes deleting message encryption/decryption keys, ephemeral DH keys and the shared cached secret. When Alice receives Bobs DH1 message, she throws an error as well. Out of order messages will also crash the protocol, but with more delay. After commit contention or a protocol crash, Alice and Bob *must* redo the first key negotiation *including* the SAS confirmation.

Note that when clients receive an out of order keying message (PKStart, Commit, DH1, DH2 or Confirm), the first thing they will do is reset their context and throw away their keys. The only thing that is inspected for this behavior, is the message header. This gives rise to a very simple DOS attack, where the attacker injects just one key message on the network and desynchronizes the clients. This is particularly bad because the identity of the other user is no longer validated. A user that does not carefully read the warning might falsely assume that the identity of their communication partner is authenticated because they have confirmed the SAS in the past.

2.5 Progressive Encryption

The problem with SCimp version 1 is that it requires both participants to be online in order to complete a key exchange. This is not always the case, even if both devices are on. For example, the iPhone puts applications to sleep and disconnects them from the network after a short time when they are not in the foreground. This includes the times the device is in a locked state. The upside is that this reduces battery usage.

The downside is that a background application cannot receive and send the messages required for the key negotiation. If Alice sends a Commit message to Bob, who happens to be "offline", he does not receive the message immediately. Instead, Alice always sends her message to the Silent Circle server, which then sends a push message to Bob (either via Apple Push Notifications (APN) [2] or Google Cloud Messaging (GCM) [15] for iPhones or via GCM for Android devices). When Bob's device is "online", it simply downloads the message from Silent Circle, but when Bob's device is "offline", it will display a notification to Bob, who can put the application in the foreground so that it can receive and send the required messages.

In the delay that was introduced because of this, Alice's device might have gone offline again, introducing more delay when receiving the DH1 message. In total, four key negotiation messages (containing no user input) need to be sent back and forth until Alice can send her first user message. The resulting process gives a very poor user experience.

To solve this problem, Silent Circle invented a technique they named "Progressive Encryption", shown in Figure 2.5. The main idea behind this technique is that users upload a non-ephemeral public key to the Silent Circle server. Whenever Alice wants to send a message to Bob, she downloads Bob's public key from the Silent Circle server. Alice uses that key to complete a Diffie-Hellman key exchange with her own ephemeral key pk_0, from which she derives symmetric key material, just as she would upon completion of a regular key negotiation. She encrypts/authenticates her message with the derived symmetric key. In addition, she also generates an ephemeral key-pair (sk_a, pk_a), as if composing a Commit message. Alice combines pk_0, the ciphertext and the Commit message in one message, labeled PKStart.

Upon receiving the PKStart message, Bob uses pk_0 and the secret key matching his key on the Silent Circle server to derive the symmetric key material and is able to decrypt the ciphertext. Alice and Bob now share key material and can use it to communicate. On the other hand, they also complete the regular key negotiation. Once they have finished that key negotiation, they discard the old key material.

When Alice and Bob have derived the new key material, they can compare their values of the SAS and confirm that no man in the middle was present during any of this. In order to verify that this is also true for the messages that were sent before key negotiation completed, the PKStart message is digested in the value h_{total}.

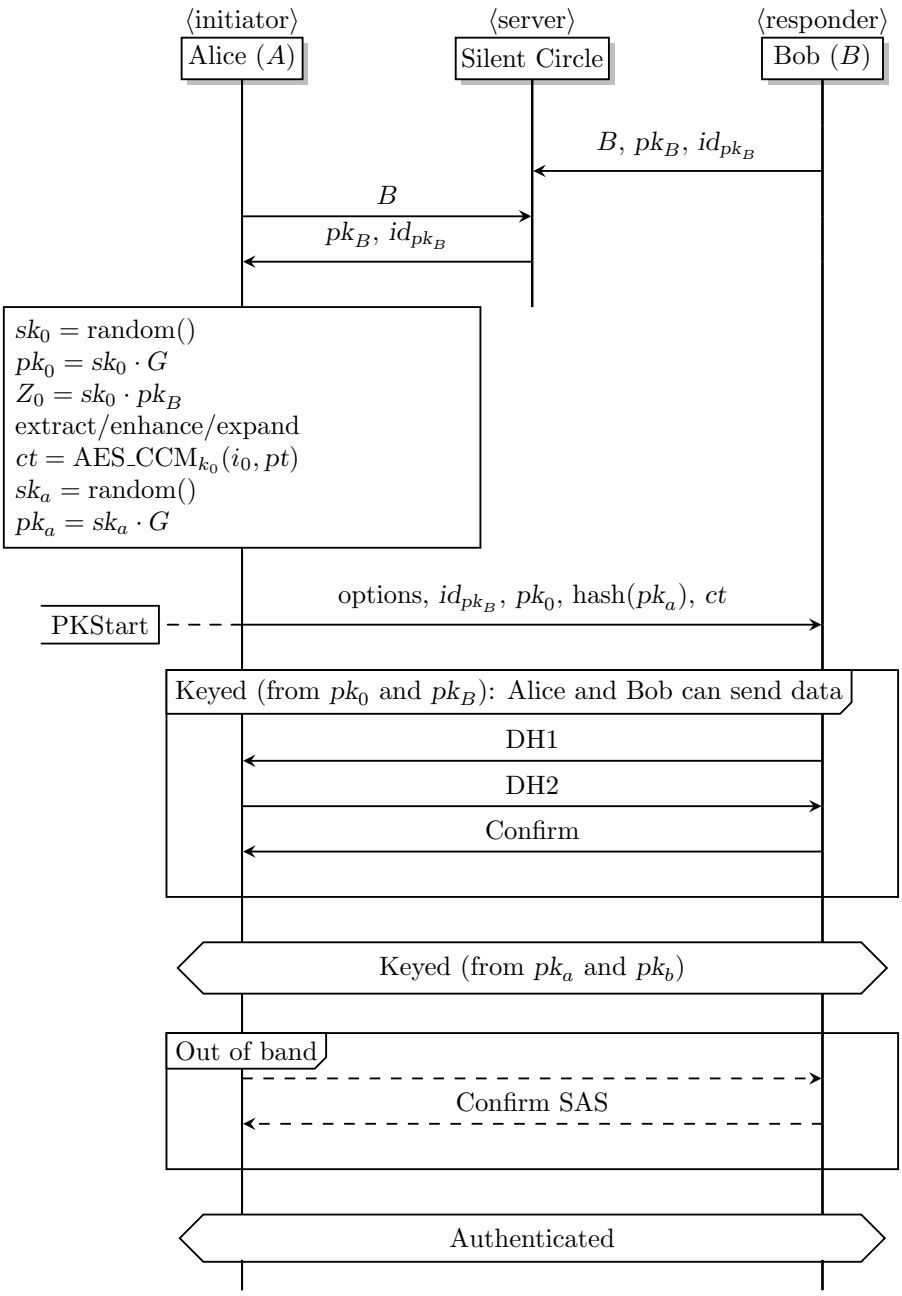

Figure 2.5: SCimp: Progressive Encryption

2.5.1 Public Key

In order for Alice to be able to send a message to Bob, Bob needs to have uploaded his public key pk_B to the Silent Circle Server. Attached to the key that he uploads, is both the owner (B) and a locator id_{pk_B} which is derived from pk_B:

$$id_{pk_B} = \mathrm{KDF}_{pk_B}(\text{“SCKey\_ECC\_Key”}, nonce, 160) \tag{2.25}$$

It is not documented where this *nonce* comes from, but it is very likely a device specific value. However, it is of no importance for the security of the protocol, because the value of the locator is never validated.

A public key also contains a lifetime, indicated by a start and end date, which is set at a *medium term lifetime*, so that it should be updated every 30 days. Public key packets should be self-signed. Additional signatures are allowed. For example, a signature by a previous key of the same user adds a form of key continuity.

2.5.2 PKStart

Alice completes an initial Diffie-Hellman key exchange with Bob's public key, from which she derives the required keys for communication with Bob. This is done in the same extract/enhance/expand process as in SCimp version 1, with the small alteration that the value of h_{total} is set to zero in equations 2.7 and 2.9:

$$kdk_0 = \mathrm{MAC}_0(Z_{0,x}) \tag{2.26}$$

and

$$ctx = \mathrm{len}(A) \parallel A \parallel \mathrm{len}(B) \parallel B \parallel 0. \tag{2.27}$$

The enhance and expand phase are identical. Although the implementation does derive values for mac_a, mac_b, SAS and cs, these values are not used.

In DH mode (with a Commit message instead of PKStart), hcs is included in the Commit and DH1 message, so that an eavesdropper cannot distinguish the first key negotiation from rekeying by inspection of the message.[7]. The PKStart message is easily distinguished from Commit messages, so there is no point in sending a value of hcs_a anymore. The value hcs_b is still sent in the DH1 message, but it can be ignored.

After having sent the PKStart message, Alice and Bob can send more data, similar to how they would send data in a normal situation (they have to keep forwarding the ratchet).

2.5.3 DH1/DH2/Confirm

In parallel to the data messages that Alice and Bob can now send, they should also complete the key negotiation that was initiated by Alice's value of hash pk_a. The rest of the key negotiation is identical to that in SCimp version 1, except for equation 2.6, the computation of h_{total} now becomes:

$$h_{total} = \mathrm{hash}(id_{pk_B} \parallel pk_0 \parallel \mathrm{hash}\, pk_a \parallel ct \parallel pk_b \parallel hcs_b \parallel pk_a) \tag{2.28}$$

where ct is the ciphertext that was sent in the PKStart message.

The presence of id_{pk_B} and pk_0 in h_{total} ensures that these values are mixed into the extract phase of the key derivation. This ensures key continuity, which in turn ensures that when the users confirm the SAS, they also confirm the authenticity and confidentiality of keys derived from $Z_{0,x}$.

[7] This assumes a random value for cs as specified in the documentation, not $cs = 0$ as in the implementation. See also Section 2.1.1.

2.5.4 Authentication

Although progressive encryption benefits the user experience, from a security perspective it introduces a weakness. The keys that are derived from $Z_{0,x}$ are authenticated only retroactively, when the users verify the SAS derived from the key negotiation.

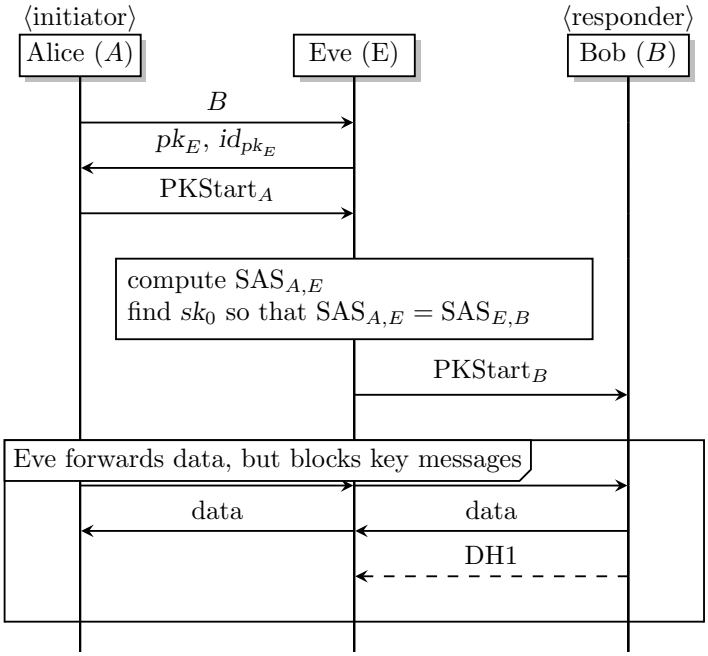

Figure 2.6: SCimp version 2: man-in-the-middle

One might be tempted to verify the SAS that is derived from $Z_{0,x}$, but Figure 2.6 shows why that does not authenticate the other party. The lack of a commitment to the first key of the Diffie-Hellman key exchange gives the attacker the opportunity to find a SAS-collision.

The lower part of Figure 2.6 also illustrates how Eve can maintain a successful man-in-the-middle attack. The trick is to keep forwarding the data (optionally with alterations), but never letting Alice or Bob complete a full key negotiation with her. At that time, Alice and Bob will already send data that is decryptable bye Eve (and requires Eve to re-encrypt to stay unnoticed). If they would complete the key negotiation, they might call each other to verify the new SAS, which no longer matches.

Eve might not go undetected when the users are sufficiently cautious, because they would know that in an uncompromised conversation they should have been able to verify the identity of the other party, after at most four messages have gone back and forth. The protocol does not generate any warning at this point. A paranoid user would probably not want to use the PKStart message anyway, because the authenticity of messages can be guaranteed only after the messages have been sent. At that point, the users might indeed detect that they have been victim of a man-in-the-middle attack, but part of the conversation already took place. For that paranoid user, the Commit message from SCimp version 1 is still available as an alternative in SCimp version 2, with the inconvenience that it somewhat reduces the user experience. I have not been able to test the application and validate that this option is also exposed to the user, nor do I know if the user can inspect which SCimp version is used for each conversation.

2.6 SCimp group conversations

Whereas SCimp version 1 only allowed for one-to-one conversations, SCimp version 2 also enables group conversations. To achieve this goal, SCimp has been extended with two more modes: SCimp PubKey mode and SCimp Symmetric mode. The former is used to set up symmetric keys for members of the group, so that they can have a conversation using the latter mode. Alternatively, Symmetric mode can also be set up with a manually provided initial key.

One group participant sets up the conversation. They will generate a random initial key, from which a symmetric multicast key can be derived. The multicast key will be used for encryption and decryption of every message in the conversation. The initiator encodes the initial key in a Multicast Key Message (which is not a SCimp message). That message can be distributed with a PubKey message (which is a SCimp message) to each group participant. To create a PubKey message, the initiator finds the public key of the receiver and generates a random session key. The Multicast Key Message is symmetrically encrypted with the session key, while the session key itself is asymmetrically encrypted to the public key of the receiver. Both the encrypted message and encrypted session key are then sent to the receiver, who can then decrypt all to retrieve the multicast key. In order to deliver the multicast messages to all group members, SCimp uses the XMPP extension XEP-0033 [17] in the XMPP layer.

2.6.1 SCimp public mode

In order to set up the SCimp Symmetric mode, the users need to have a shared symmetric key. They can set this up with SCimp PubKey mode. This multicast key is encapsulated in a PubData message (similar to how Siren is encapsulated in SCimp), but the PubKey mode of SCimp can in principle send any message.

When Alice initiates the group conversation, she generates a random symmetric key that she encapsulates in a Multicast Key message. After creating the message, Alice now wants to send that message to Bob in a PubKey message. First, she gets Bob's public key from the Silent Circle server. Note that this is the same key used for Progressive Encryption: the public key is used both for Diffie-Hellman key exchange and for encryption. She then generates a random session key:

$$k_{session} = \text{rand}(). \tag{2.29}$$

She derives a 64-bit index from Bob's public key:

$$i_{msg} = \text{hash}(pk_B). \tag{2.30}$$

Similar to sending data, the random session key is split into an encryption key and a nonce:

$$ek, n = \text{split}(k_{session}), \tag{2.31}$$

which are then used to encrypt the message:

$$ct = \text{AES\_CCM}^n_{ek}(i_{msg}, msg). \tag{2.32}$$

Bob needs to be able to decrypt the message in order to learn the symmetric key for the group conversation. Alice sends the key to Bob, asymmetrically encrypted:

$$esk = \text{EC\_encrypt}_{pk_B}(k_{session}). \tag{2.33}$$

Elliptic curve encryption is implemented as ECC Diffie-Hellman encryption. The full PubKey message that Alice sends consists of the label `pubkey`, the protocol version (always 2), a cipher suite (see Table 2.2), a locator for the public key used (id_{pk_B}), the encrypted session key (esk) and the encrypted message (ct).

Multicast key

The contents of the PubKey message in the context of group messages is the multicast key. Each group conversation is identified by its unique thread id. It is not specified how the thread id is generated. A random initial symmetric key k_{init} is generated, with its corresponding locator $id_{k_{init}}$.

A full Multicast Key message consists of the label `multicast_key`, a SCimp version (always 2), a cipher suite (4 or 5, see Table 2.2), the symmetric key (k_{init}), the key locator ($id_{k_{init}}$), the start time of the thread, the thread creator (XMPP address) and the thread ID (id_{thread}).

Man-in-the-middle

An adversary with sufficient control over the network can easily inject her own public key instead of the one from the honest receiver. When the initiator sends the PubKey message, the adversary intercepts and compromises the multicast key. If she wants to remain undetected, she simply re-encrypts the message to the public key of the honest receiver.

The reason that this attack is so trivial, is that there is no SAS (or anything equivalent) for Alice to verify the identity belonging to the public key that she receives. To protect against this attack, it is better not to use the PubKey messages. If she still wants to set up a group conversation, she should set it up using a passphrase. That passphrase can be shared (for example) by setting up a pairwise authenticated SCimp conversation with all group members. Alice will have to do this manually.

It is remarkable that SCimp uses a PubKey message for distributing the random group session keys, instead of distributing the messages with PKStart messages (or use existing SCimp sessions where possible). This solution would have kept the protocol much simpler and more importantly, it gives the users a chance to authenticate each other using the SAS. It would still be susceptible to the man-in-the-middle attack described in Section 2.5.4, but at least that attack is detectable by sufficiently vigilant users.

2.6.2 SCimp symmetric mode

Symmetric conversations are set up by deriving a multicast key k from the initial key k_{init}. This initial key can be received from a PubKey message, but it could also have been derived from a passphrase using the PBKDF-2 algorithm. The following equations list how the key material is derived from the initial key.

$$k = \mathrm{KDF}_{k_{init}}(\text{“SymmetricMasterKey”}, id_{thread}, 2l) \tag{2.34}$$

$$i = \mathrm{KDF}_{k_{init}}(\text{“InitialIndex”}, id_{thread}, 64) \tag{2.35}$$

$$i_{offset} = \mathrm{rand}() \tag{2.36}$$

$$\mathrm{SAS} = \mathrm{KDF}_{k_{init}}(\text{“SAS”}, id_{k_{init}}, 64) \tag{2.37}$$

It is strange that a value for the SAS is computed, even though a method for verifying the SAS was never specified. The SAS is now a 64 bit value, although the verification of the SAS does not necessarily use the full value for verification (see Table 2.3). But most important, the computed SAS value depends only upon the value k_{init}, which renders it useless. Verifying that these values are the same for different parties does not authenticate anything, because an attacker that has intercepted the value can forward it.

To encrypt a message for the group, a participant creates a SCimp Data message. The encryption is similar to that of regular SCimp (see equation 2.24), with the exception that the header value is set to the value of $i \oplus i_{offset}$ and after sending, the value of i_{offset} is incremented by one. The symmetric key k is never updated.

Key erasure

Group messaging does not do anything to ensure key erasure or future secrecy. This is the intended behavior according to the Silent Circle documentation [22]:

> "The process of keeping multiple participants SCIMP contents synchronized would be fraught with errors and trying [to] add the concept of perfect forward security in a shared conversation would seem moot."

I strongly disagree with this statement. Key erasure becomes even more important when one is communicating with multiple users, because the attack surface grows with every device that has access to the keys. For a successful attack, the adversary only needs to compromise one device and retrieve the key. This would compromise all previous group messages.

2.7 Siren

Additional message features are provided by Siren. Siren packets are JSON encoded messages that are the plaintext input (pt) to the SCimp protocol. In addition to sending the actual message, the user can also add some security attributes to the message. For example, Siren could add a "for you eyes only"-tag to a message, which should prevent the other party from copying the message or taking a screenshot of it. Other features are to let the message delete itself after a certain amount of time, to redact a message, to request re-sending of an old message, to request a receipt upon message delivery or to send GPS location data.

None of these features can be ensured by the protocol or by any other cryptographic measure, although it might be enforced by the BlackPhone if these features are integrated in the OS. It cannot be ensured by an application. Therefore, I consider these features to be nothing more than polite requests that may or may not be honored by the other party.

That does not mean that Siren packets do not have impact upon the security of the protocol as a whole. First of all, in SCimp version 2, messages can be signed, which affects the deniability of messages. Some Siren packets, such as message redaction, will even *require* a signature in future versions of SCimp[8]. Secondly, SCimp version 2 allows for files to be sent via the cloud, which also has implications on the security of the protocol.

2.7.1 Signed messages

Silent Circle mentions that it allows for signatures on Siren packets and will even require signatures for certain Siren messages in future requests. It is not clear how this was imagined to be done, since users do not own a long-term public key. The medium-term key has problems of its own, as described in Section 2.6.1.

The documentation [22] mentions that:

> "To prevent a form of denial of service attack at the XMPP level, we require that the burn request originate from the same JID as the message being burned, and in later versions require that the request be signed by the originator."

The reason that a signature is necessary is that a sender might want to redact a message from the Silent Circle server before it has been delivered to the receiver. To do so, the user sends an XMPP stanza to the server containing a plaintext identifier of the earlier message and the plaintext request that the message will not be delivered. Since it is plaintext, anyone observing traffic to the server could fake to send this XMPP stanza.

Without a singature on the redaction message, the sender can plausibly deny sending the message at all. However, when the user has signed a message redaction, they have implicitly admitted that they have sent the message. Instead of being able to deny that the message was sent at all, the user can only deny the *content* of the message. This is a more general issue, but certainly not solved by SCimp.

[8] This was documented, but is no longer true now that SCimp is discontinued.

2.7.2 Cloud storage

A second feature that is encapsulated in Siren packets, is the ability to send files via SCimp. To send files over XMPP would be very inefficient. Instead, the sender uploads their file to the Silent Circle cloud. Before uploading, the files are encrypted using convergent encryption, which encrypts the files using a key that was not generated randomly, but derived from the file contents.

The reason given by Silent Circle [22] for the usage of convergent encryption, is that it reduces storage space on the Cloud service:

> "We do this to avoid duplication for media such as photos and documents. However we purposely added in a device unique salt to prevent third parties from deducing a file's content by checksumming the same data."

Unfortunately, this quote does not reflect what is implemented, because a salt is only added to the file locator and not to its encryption key.

First of all, a device unique salt would prevent third parties from deducing the file's content only if that salt was secret and if it contained enough entropy to be unguessable by a third party. Silent Circle uses a public value, namely the senders bare XMPP address (without resource information, for example "alice@silentcircle.com"). Secondly, when the salt is only added to the file locator, an attacker that can eavesdrop the encrypted data that is downloaded from or uploaded to the server is able to bypass any security that the salt would add if it were secret.

Convergent Encryption

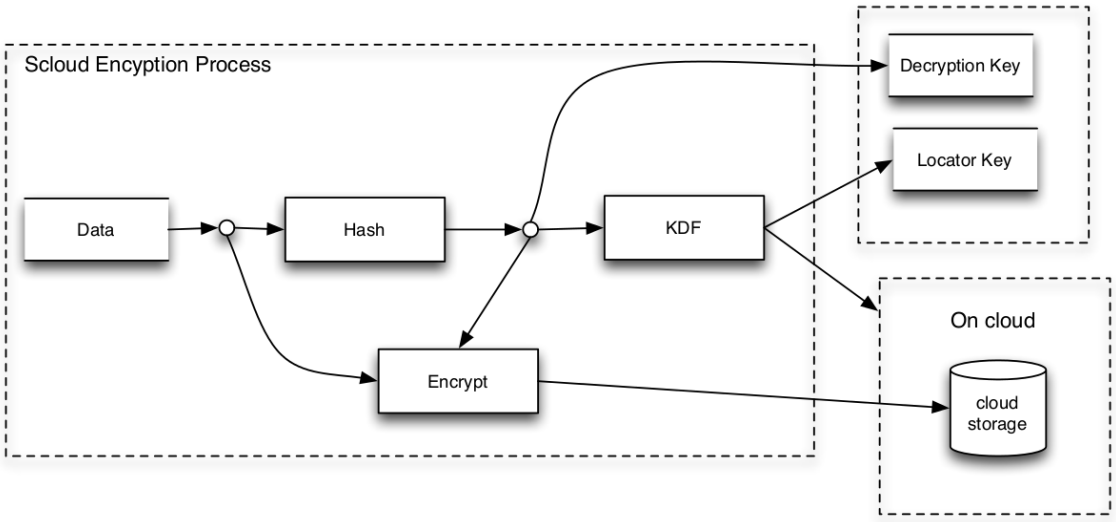

Figure 2.7: SCimp: Convergent encryption (source: [22])

For completeness, in this section I will describe the details of how convergent encryption is implemented by Silent Circle. This is shown in Figure 2.7. When Alice wants to send a file f to Bob, she first derives a key k from the file metadata and data:

$$k = \text{hash}(metadata \,\|\, data) \tag{2.38}$$

The hash-algorithm is always SKEIN256, resulting in a 256 bit value. She continues to compute an identifier for the file (id_f), derived from the computed key and a salt:

$$id_f = \text{KDF}_k(\text{"ScloudLocator"}, salt, 32) \tag{2.39}$$

The KDF always uses HMAC in combination with SHA256. The value for the salt is a device specific value, which is set at the sender's bare XMPP address.

In order to encrypt the data in CBC mode, Alice requires both an encryption key (ek) and an IV (n), which are simply computed by splitting key k in equally sized halves:

$$ek, n = \text{split}(k) \tag{2.40}$$

The encryption itself is straightforward:

$$ct = \text{AES}^{n}_{ek}(metadata \,\|\, data) \tag{2.41}$$

The encryption algorithm is always AES128 in CBC mode. Files bigger than 64kB are cut into chunks, which will each be encrypted with their own key and get their own file locator.

Now Alice can upload the ciphertext ct to the cloud, using the locator id_f as an address for retrieving the file. She encapsulates k, id_f and some file metadata in a Siren packet, which is sent to Bob as a SCimp Data message.

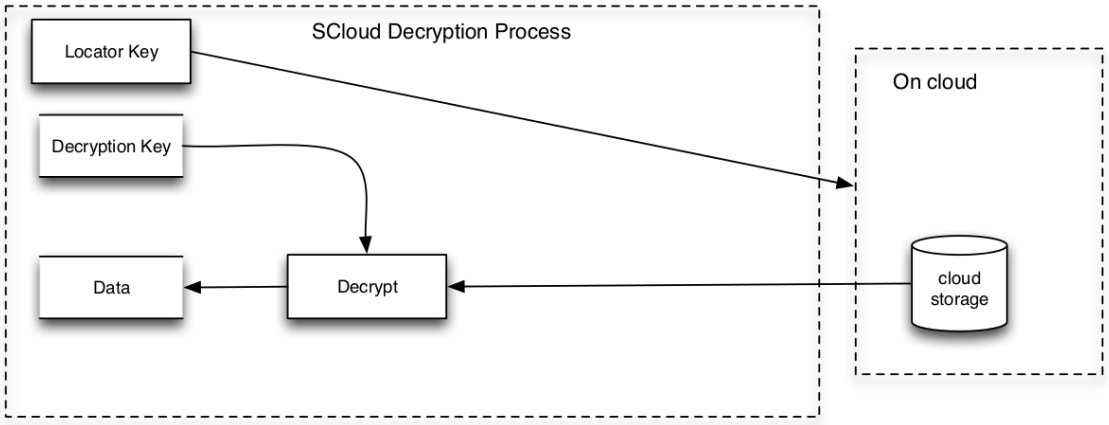

Figure 2.8: SCimp: Convergent decryption (source: [22])

Decryption of convergent encryption is depicted in Figure 2.8. Upon receiving the SCimp message, Bob is able to download the file from the cloud using the file locator. He is able to decrypt the file using the key that Alice has sent him.

Confirmation of a file

A known attack on convergent encryption is the confirmation of a file. I will explain the attack by an example.

Assume that Ed wants to send a secret file to Glenn, but he does not want his employer to know that he sent that file. He also knows that they are monitoring all traffic to and from his device. His employer, afraid that this might happen, has already precomputed a database of all the secret files (t in total), encrypted with convergent encryption. The moment he uploads his file to the cloud, his employer intercepts the data and searches for a match in the database. Using binary search, the employer can complete this search in $\mathcal{O}(\log t)$.

If Silent Circle had added the device specific salt to the encryption key (like they claim in the documentation), the cost of the attack would only have grown if the adversary could not tell who was uploading the file. If for some reason they could not, the adversary would need to add tm encrypted files to the database (with m being the number of employees). Binary search would still make the runtime of the search a very effective $\mathcal{O}(\log tm)$. This does assume that the attacker knows the device specific salt of each device, which is a reasonable assumption when the salt is simply the bare XMPP address.

However, this attack would have been prevented if each device added a secret value instead of a public salt. Ed could then have provided the decryption key per file, without leaking his secret salt.

Learn the remaining information

Another well-know attack on convergent encryption is the "learn-the-remaining-information" attack [39]. This attack applies to a file that contains little entropy. The attacker can use brute-force and convergently encrypt all possible files, until the ciphertext matches that of the uploaded file. Not only has the attacker gained confirmation that this file was uploaded, but they also learned the remaining information that was in the file. A device specific salt would help only a little, because now the attacker has to brute-force the file for each user individually, instead of being able to search all users at the same time.

Again, a secret value instead of a public salt, would have prevented this attack, as long as it contained enough entropy to make a brute-force search unfeasible.

File injection

The above attacks are well known attacks on convergent encryption, which leak metadata or even parts of data of a file. The Silent Text implementation has the additional problem that it allows for an attacker to swap in a different file when they know which file is being uploaded.

When attackers use one of the attacks above to learn which file is being uploaded, that also provides them with the encryption key, because they can derive it from the file. They can use that key to encrypt another file, which could contain malware. When Bob requests the file from the cloud, the attacker substitutes the file with the encrypted malicious file. Bob, who is able to decrypt, falsely assumes that the file came from Alice.

Bob might be able to detect this attack, simply by recomputing the key from the decrypted file and comparing it to the one he used for decryption. When they do not match, he knows that the file he received was not the file uploaded by Alice. At that point the file can be deleted, before it is opened by an application.

Note that authenticated encryption (AE), for example, CCM instead of CBC blockmode, would not have helped prevent this attack, because AE only authenticates the data under the assumption that the encryption key is unknown by the attacker.

Conclusion on convergent encryption

It seems that convergent encryption introduces more problems than it solves. Convergent encryption, as implemented in Silent Text, only saves space when a user sends one file to multiple recipients. When the same file is uploaded by different senders, the file gets stored under different locators, so they are very likely stored as separate files.

A much simpler and more secure solution would be to generate a random key for encryption, encrypt the file in CCM mode, upload the ciphertext to the cloud and send the key in a secure SCimp message to the other party or to multiple parties if the file should be shared.

2.8 Local storage

Sometimes, the user wants to store a local copy of the messages on his device. To prevent data-theft when the device is compromised, the user should encrypt this local message database. The encrypted data from the message cannot be used, because that would require the user to keep all keys in storage, preventing key erasure. The solution is simply to set up a local database, encrypted with its own key.

The local storage key should be stored securely on the device. How to do this, I consider outside the scope of this thesis. However, users should be aware that there is no perfect solution and if they store their messages and the device gets compromised, there is a risk that the attacker learns those messages.

2.9 Problems

In this section I will address two issues that I identified during my analysis of SCimp that did not fit into any of the above sections. The first issue is an identity misbinding attack, which allows an attacker to trick both honest participants into thinking they are talking to her, while they are actually talking to each other. The second problem is a usability problem that is acknowledged by Silent Circle: SCimp has no good way to handle multiple devices that share an XMPP address.

2.9.1 Identity misbinding attack

The identity misbinding attack was first described by Diffie, van Oorschot and Wiener [11]. The original attack describes the scenario where Eve claims to own the public key of Bob and then lets Alice talk to Bob while she thinks she is talking to Eve. SCimp does not have long-term public keys, so the attack does not work directly, so Eve will have to fool Alice and Bob some other way.

Imagine that Alice and Bob are two chess grandmasters. Eve also likes to play chess, but she is not a very good player. Luckily, she has a way to fool both Alice and Bob. She is going to let them play a game of chess over SCimp, in which they will send each other messages containing the moves.

Eve tells Alice that her XMPP address is B, which is actually the address of Bob. Likewise, she convinces Bob that here address is A. They will initiate in a key exchange and derive their keys, at which point they should confirm their SAS.

At this stage, Eve calls them both at the same time, but makes sure that neither party can hear the other. Both phone calls can be over a fully authenticated channel and Alice and Bob are allowed to gain all the "confidence in the identity of the other party using standard human interaction". After all, they really are talking to Eve. Eve just waits for one of the parties to confirm half of the SAS. Assume that Alice talks first. Eve repeats Alice's code to Bob, who will reply with the other half of the code. Eve forwards the code to Alice. Now both grandmasters are convinced that they are connected to Eve, and the chess game begins.

Note that Eve can mount the same attack on both players by simply setting up a separate SCimp connection with both players. She then forwards every message she gets to the other player. This is not the issue with the above scenario: the issue is that Alice is convinced that B is the address of Eve.

2.9.2 Multiple devices

The SCimp protocol has no good way to send messages to a user that uses multiple devices with only one XMPP address.

Assume that Bob owns both an Android device and an iPhone, with both of them linked to his XMPP address. When Alice wants to talk to Bob, she creates a new SCimp context on her device, which she will use to initiate a key negotiation with Bob. She sends an XMPP stanza, containing her Commit message, to the Silent Circle XMPP server. The server sees Bob's XMPP address as recipient and it sends the message to the first of Bob's device that it sees online. Assume that it is the Android device, then Alice and Bob can complete the key negotiation and set up a session between Alice's device and Bob's Android device.

When Alice sends a message to the Bob, the XMPP server will just send the message to whatever device it sees online first. If that device is the iPhone, Bob cannot decrypt the message, because the keys that belong to the SCimp context are stored on the Android device and not on the iPhone. Bob's iPhone will display an error and send back a resend request.

SCimp has no robust method for handling the problem. Instead, Alice will have to store a SCimp context on her device for both of Bob's devices. When encrypting a message, she simply picks the context that she used to decrypt the last message she received from Bob. If that context turned out to be wrong, hopefully Bob will send a resend request from the correct context so she can resend the message, re-encrypted in the correct context. Because the communication is asynchronous, this might take a long time. The user experience can get especially bad if Bob

switched to his Android device before the resent message was received: he will again be unable to decrypt the message and another context switch is required.

This problem is not a specific problem of SCimp, but one that affects many other secure messaging protocols as well. A possible solution would be not to allow switching between devices, but this would give a poor user experience, possibly worse than the current solution for multiple devices.

Chapter 3

Formal verification

In order to find flaws in the protocol, I created a model of the protocol in Proverif. Proverif is an automated cryptographic protocol verifier, using the Dolev-Yao model [12] as a model for the attacker. That means that the attacker is assumed to have full control over the network: he can read, modify, block and inject any message sent on the network. For many SCimp security properties, such as future secrecy and key erasure, a stronger attacker is needed that can compromise some (but not all) secrets. Proverif allows us to model this by defining a compromised user that publishes some secrets on a public channel, so that they become available to the Dolev-Yao attacker.

For building the models, I used typed Proverif (version 1.90). Proverif assumes that cryptographic primitives are perfect, for example: an attacker is unable to learn any information on ciphertext when he does not have the encryption key and hash functions are one-way, without the possibility of a collision. Proverif does not have a concept of lengths, so a 16 bit secret is considered as strong as a 256 bit secret. For many protocols, the solution is simply to choose your secrets large enough so that the conclusions about the model represent the conclusions about the actual protocol. However, SCimp has the low-entropy SAS, which could lead to attacks that Proverif is unable to find. When evaluating Proverif results, this needs to be taken into account.

The optimal model for SCimp combines all different components (such as key negotiation and sending messages) in one model and queries for all the security properties that I want to know about. Unfortunately, this would result in a model that is too complex for Proverif to handle. For example, Proverif can verify the integrity and confidentiality of a message and it can check if two processes are equivalent from the perspective of an attacker, but it cannot do both in the same model.

In this chapter, I will describe the models that I built for the different components of SCimp and explain the different security properties that are proven per model. In Section 3.1 I will explain how I modeled the (cryptographic) primitives in Proverif. In Section 3.2, I will explain how I modeled SCimp in Proverif. For each protocol component, I will argue why the model reflects the actual protocol and I will interpret the Proverif results. The full Proverif models are published online [30].

3.1 Used primitives

Not every security property could be checked in a single file, so the full model of SCimp has been split up into several files. However, each file uses the same cryptographic primitives and operations, which are used in every file. The common parts of the SCimp model are described in this section.

Although SCimp version 2 allows for messages that have public key signatures on them, these do not have any function in the protocol. By not modeling these, I have kept the models a bit simpler, at the risk that Proverif finds an attack that is not possible in a protocol with signatures.

3.1.1 Types

The following Proverif types were used.

```
type mac_key.
type secret_key.
type nonce.
type point.
type scalar.
type identity.
```

Listing 3.1: Proverif: types

SCimp was implemented in C, with most of the above types being implemented simply as bitstrings. As such, type conversion is often implicit in the C code. In typed Proverif syntax conversion between types must be made explicit, which is done with the following functions:

```
fun mk2bs(mac_key)     : bitstring      [data, typeConverter].
fun bs2mk(bitstring)   : mac_key        [data, typeConverter].
fun sk2bs(secret_key)  : bitstring      [data, typeConverter].
fun bs2sk(bitstring)   : secret_key     [data, typeConverter].
fun pt2bs(point)       : bitstring      [data, typeConverter].
fun bs2n (bitstring)   : nonce          [data, typeConverter].
fun sk2mk(secret_key)  : mac_key        [data, typeConverter].
fun mk2sk(mac_key)     : secret_key     [data, typeConverter].
```

Listing 3.2: Proverif: type conversions

The `data` annotation ensures that the adversary can freely convert back and forth between types. The `typeConverter` keyword allows the prover to optimize the type conversions away after it has verified correctness of the types and compiled the code.

```
free ch:channel.
free sync:channel [private].
```

Listing 3.3: Proverif: channels

The channel `ch` represents the public channel over which SCimp is executed. This is most likely the internet.

The channel `sync` represents a channel on which the protocol participants negotiate a new key, which is missed by the adversary. This should cause the protocol to self-heal, so that even an adversary that had compromised previous key material is no longer able to compromise the security of future communication.

Later on, I dynamically define the `phone` channel, which is an authenticated channel between honest participants. The adversary can eavesdrop this channel but is unable to inject messages, which is achieved by publishing everything on this channel to the public `ch` channel. This corresponds to a phone conversation between participants, where they confirm the SAS: the adversary can hear the SAS, but cannot inject/change a message without being detected.

3.1.2 Constants

Several constant values are used in the protocol. They are public values and are modelled in Proverif by using the `const` keyword.

```
const Null            : bitstring [data].
const OK              : bitstring [data].

const InitStr         : bitstring [data]. (* "Initiator" *)
const RespStr         : bitstring [data]. (* "Responder" *)
const MasterStr       : bitstring [data]. (* "MasterSecret" *)
const AlgId           : bitstring [data]. (* "SCimp-ENHANCE" *)
const InitMasterLabel : bitstring [data]. (* "InitiatorMasterKey" *)
const RespMasterLabel : bitstring [data]. (* "ResponderMasterKey" *)
const InitMACLabel    : bitstring [data]. (* "InitiatorMACkey" *)
```

```
11 const RespMACLabel     : bitstring [data]. (* "ResponderMACkey" *)
12 const SasLabel         : bitstring [data]. (* "SAS" *)
13 const CsLabel          : bitstring [data]. (* "RetainedSecret" *)
14 const InitIndexLabel   : bitstring [data]. (* "InitiatorInitialIndex" *)
15 const RespIndexLabel   : bitstring [data]. (* "ResponderInitialIndex" *)
16 const MsgKeyLabel      : bitstring [data]. (* "MessageKey" *)
17
18 const Compromised : identity [data].
```

Listing 3.4: Proverif: SCimp constants

The actual values for the labels are not important, the only thing that matters for Proverif is that each constant has its own distinct value.

The Compromised identity is that of the attacker. When one of the protocol participants identity is set to this value, they will publish all knowledge they have. By publishing all knowledge on a public channel, the built-in Proverif adversary can participate in the protocol with honest users.

3.1.3 Functions

Several helper functions are required for Proverif:

```
1  fun increment(bitstring) : bitstring [data].
2
3  fun splitFst(bitstring) : bitstring.
4  fun splitSnd(bitstring) : bitstring.
5  reduc forall x:bitstring;
6        unsplit(splitFst(x), splitSnd(x)) = x.
7
8  fun getCS(identity, identity) : bitstring [private].
9  equation forall x:identity, y:identity;
10          getCS(x, y) = getCS(y, x).
```

Listing 3.5: Proverif: helper functions

The increment function allows Proverif to increment values. The `data` keyword ensures the adversary can also revert the operation: decrement. The construction for split ensures that you can split a bitstring into two pieces, while the unsplit function allows you to combine back the two to get back the original bitstring. Proverif has no concept of length of the bitstrings, so as far as proverif is concerned, splitting off sixteen bits is the same as splitting the bitstring in equal sized parts.

The function `getCS` is a private function, that retrieves the value of the cached secret shared by two users. The advantage of encoding this as a function, as opposed to generating a new random value for the participants on the fly, is that the returned value is a fixed constant for two participants. The equation ensures the commutative property of *cs*. By making the function private, the adversary cannot derive this value from the identities.

3.1.4 Cryptographic functions

Things get slightly more exciting with the Proverif implementation of the cryptographic primitives.

```
1 fun hash(bitstring) : bitstring.
2
3 fun mac(mac_key, bitstring) : bitstring.
4
5 reduc forall key:mac_key, context:bitstring, label:bitstring;
6       kdf(key, label, context) = mac(key, (label, context)).
```

Listing 3.6: Proverif: cryptographic primitives

There is an important consequence of the simple implementation of the above primitives. As far as the Proverif prover is concerned, the hash function behaves as a random oracle, revealing

nothing on the digested value and never colliding. An attacker will not be able to guess the hash without knowledge of the value to be hashed.

The `kdf` function is the same as the `mac` function and can be used interchangeably in the Proverif code (just as in the real code). The only difference is that it is missing the L parameter. It would not make sense to include it, because it represents a length and Proverif is length-agnostic.

Symmetric encryption

The naive implementation of symmetric encryption in Proverif would be to build an encrypt function and reduce the resulting ciphertext to the message with the corresponding decrypt function. In that implementation, Proverif would not understand how to decrypt ciphertext that was encrypted with another key. Effectively, this naive implementation implements authenticated encryption.

The CCM blockmode, used in SCimp, is more complex, as it also allows for additional data to be authenticated but not encrypted. I have modelled this by first implementing (non authenticated) encryption/decryption in `sym_enc` and `sym_dec`. This represents the Counter-mode encryption in CCM. The CBC-MAC part of CCM is implemented simply as a `mac`. These primitives are combined as mac-then-encrypt, like they are in CCM mode, in the sub-processes `aead_enc` and `aead_dec`.

```
1  fun sym_enc(secret_key, nonce, bitstring) : bitstring.
2  fun sym_dec(secret_key, nonce, bitstring) : bitstring.
3  equation forall k:secret_key, n:nonce, m:bitstring;
4           sym_dec(k, n, sym_enc(k, n, m)) = m.
5  equation forall k:secret_key, n:nonce, m:bitstring;
6           sym_enc(k, n, sym_dec(k, n, m)) = m.
7
8  letfun aead_enc(k:secret_key, n:nonce, header:bitstring, plaintext:bitstring) =
9         let tag = mac(sk2mk(k), (n, header, plaintext)) in
10        sym_enc(k, n, (plaintext, tag)).
11 letfun aead_dec(k:secret_key, n:nonce, header:bitstring, ciphertext:bitstring) =
12        let (plaintext:bitstring, tag:bitstring) = sym_dec(k, n, ciphertext) in
13        let (=tag) = mac(sk2mk(k), (n, header, plaintext)) in
14        plaintext.
```

Listing 3.7: Proverif: AEAD encryption

Note that the sub-process `aead_dec` checks the value of the tag. If the tag mismatches, the process halts.

The downside of this implementation is that the prover is not as powerful in checking equations as it is when checking reductions. I believe that the results of Proverif are more meaningful when they are about an accurate model of the protocol, which is why I have chosen to implement the more complex model for AEAD.

Diffie-Hellman

The last cryptographic primitive used in SCimp that I have modelled in Proverif is a Diffie-Hellman key exchange. The functions are labeled in the additive structure that is common for elliptic curve groups, even though Proverif does not know about the underlying group structure.

```
1  const Base : point [data].
2  fun mult(scalar, point) : point.
3
4  equation forall x:scalar, y:scalar;
5           mult(x, mult(y, Base)) = mult(y, mult(x, Base)).
```

Listing 3.8: Proverif: Diffie-Hellman

3.2 SCimp model

Now that the cryptographic primitives are defined, the actual protocol can be implemented.

3.2.1 First key negotiation

The full Proverif model for the first key negotiation can be found in `scimp_first_key_neg.pv`. At the first key negotiation, the participants do not share a cached secret (cs), which means they rely upon the confirmation of the SAS to authenticate and verify each others identity. After confirmation of the SAS, the protocol should guarantee the following to the users:

1. The confidentiality of the derived keys and indices.
2. The authenticity of the derived keys and indices.
3. The authenticity of the identity of the other party.

Queries

```
1  (* Queries for confidentiality *)
2  free ksndInitFlag, krcvInitFlag, isndInitFlag, ircvInitFlag,
3       ksndRespFlag, krcvRespFlag, isndRespFlag, ircvRespFlag,
4       cs1InitFlag, cs1RespFlag : bitstring [private].
5
6  query attacker(ksndInitFlag); attacker(krcvInitFlag);
7        attacker(isndInitFlag); attacker(ircvInitFlag);
8        attacker(ksndRespFlag); attacker(krcvRespFlag);
9        attacker(isndRespFlag); attacker(ircvRespFlag);
10       attacker(cs1InitFlag);  attacker(cs1RespFlag).
11
12 (* Queries for authenticity *)
13 event beginInit(identity, identity, bitstring, bitstring, bitstring, bitstring,
       bitstring).
14 event acceptInit(identity, identity, bitstring, bitstring, bitstring, bitstring,
       bitstring).
15 event beginResp(identity, identity, bitstring, bitstring, bitstring, bitstring,
       bitstring).
16 event acceptResp(identity, identity, bitstring, bitstring, bitstring, bitstring,
       bitstring).
17
18 query x:identity, y:identity, ki:bitstring, kr:bitstring,
19       ii:bitstring, ir:bitstring, cs:bitstring;
20       inj-event(acceptInit(x, y, ki, kr, ii, ir, cs))
21         ==> inj-event(beginInit(x, y, ki, kr, ii, ir, cs)).
22 query x:identity, y:identity, ki:bitstring, kr:bitstring,
23       ii:bitstring, ir:bitstring, cs:bitstring;
24       inj-event(acceptResp(x, y, ki, kr, ii, ir, cs))
25         ==> inj-event(beginResp(x, y, ki, kr, ii, ir, cs)).
```

Listing 3.9: Proverif: SCimp first key negotiation (queries)

The queries for confidentiality are encoded as queries for reachability in Proverif. This means that an adversary cannot "reach" (or compute) these values. Note that Proverif can only prove reachability of free variables declared at the top level or variables declared in a process with the `new` keyword. In case of SCimp, the variables of which I want to prove the secrecy are computed using the `kdf`, so Proverif is unable to prove the secrecy of these variables directly.

Instead, I use a standard Proverif trick. I declare free (private) flag variables per variable and per protocol participant. At the end of the process, the honest participant publishes the flag variables, but encrypted symmetrically with the actual variable of which I want to prove the secrecy. The only way for an attacker to get the value of the flag variable, is to decrypt that message, but for that he needs to know the value of the actual variable. This way, the confidentiality of the flag variable is equivalent to that of the actual variable.

The queries for authenticity are encoded in Proverif as correspondence assertions. Correspondence assertions should be read as *if event e has been executed, then event e' has been previously executed*. The arguments to these events allow relationships between the arguments of the events

to be studied. From the perspective of the adversary, a process with events is indistinguishable from a process without events.

The events and queries that are encoded in Listing 3.9 should be read as follows:

- Line 13 (`beginInit`) records the belief of the initiator that she (first parameter) has initiated a protocol run with the other party (second parameter), resulting in authenticated key material (other parameters).
- Line 14 (`acceptInit`) records the belief of the responder that he (second parameter) has just completed a protocol run with the initiator (first parameter), which resulted in the authenticated key material (other parameters).

The correspondence between these events is encoded in the query on Lines 18–21. The keyword `inj-event` ensures that this is a one-to-one correspondence. This ensures that no replay attack is possible. This correspondence verifies the identity of the initiator to the responder and proves the authenticity of the corresponding key material from initiator to responder.

The events `beginResp` and `acceptResp`, with the corresponding query, symmetrically verify the authenticity of the responder to the initiator.

Process

```
1  let processInitiator(init:identity, resp:identity, phone:channel) =
2    (* Commit *)
3    new ski : scalar;
4    let pki    = mult(ski, Base) in
5    let hpki   = hash(pt2bs(pki)) in
6    let hcsi   = mac(bs2mk(Null), (hpki, InitStr)) in
7    let commit = (hpki, hcsi) in
8    out(ch, commit);
9
10   (* DH1 *)
11   in(ch, dh1:bitstring);
12   let (pkr:point, hcsr:bitstring) = dh1 in
13
14   (* DH2 *)
15   let z       = mult(ski, pkr) in
16   let htotal  = hash((commit, dh1, pki)) in
17   let kdk     = bs2mk(mac(bs2mk(htotal), pt2bs(z))) in
18   let context = (init, resp, htotal) in
19   let sessId  = hash((init, resp)) in
20   let kdk2    = bs2mk(mac(kdk, (MasterStr, AlgId, context, Null))) in
21   let maci    = kdf(kdk2, InitMACLabel,    context) in
22   out(ch, (pki, maci));
23
24   (* Confirm *)
25   in(ch, macr:bitstring);
26   let ksnd    = kdf(kdk2, InitMasterLabel, context) in
27   let krcv    = kdf(kdk2, RespMasterLabel, sessId) in
28   let (=macr) = kdf(kdk2, RespMACLabel,    context) in
29   let sas     = kdf(kdk2, SasLabel,        context) in
30   let cs1     = kdf(kdk2, CsLabel,         context) in
31   let isnd    = kdf(kdk2, InitIndexLabel,  sessId) in
32   let ircv    = kdf(kdk2, RespIndexLabel,  sessId) in
33
34   (* Start verification of responder identity *)
35   event beginInit(init, resp, ksnd, krcv, isnd, ircv, cs1);
36
37   (* Confirm the SAS *)
38   out(phone, sas);
39   in(phone, (=sas, ok:bitstring));
40
41   (* Check that the conversation was not with the Compromised party *)
42   if resp <> Compromised then
```

```
43
44    (* Accept the responder identity and corresponding key material *)
45    event acceptResp(init, resp, ksnd, krcv, isnd, ircv, cs1);
46
47    (* Publish secret values to test secrecy of generated key material *)
48    out(ch, sym_enc(bs2sk(ksnd), bs2n(Null), ksndInitFlag));
49    out(ch, sym_enc(bs2sk(krcv), bs2n(Null), krcvInitFlag));
50    out(ch, sym_enc(bs2sk(isnd), bs2n(Null), isndInitFlag));
51    out(ch, sym_enc(bs2sk(ircv), bs2n(Null), ircvInitFlag));
52    out(ch, sym_enc(bs2sk(cs1),  bs2n(Null), cs1InitFlag)).
```

Listing 3.10: Proverif: SCimp first key negotiation (initiator)

The Proverif process is modelled as closely as possible to the SCimp process as specified by the C implementation. A few specific constructs have been introduced in the Proverif code, in order to verify the queries.

The `beginInit` event is executed as soon as the required key material is derived, but just before the SAS is confirmed. The `acceptResp` event is executed just after the SAS confirmation, meaning that the initiator accepts the responder.

The method for confirmation of the SAS is never specified. It is modelled here as both parties saying it to each other. The reason that the `OK` constant is tagged along the second message, is so that the initiator does not get confused and accepts the SAS that he just sent himself. Of course, this is not likely to happen in a real phone conversation.

Lines 48–52 publish the encrypted flag values (as described in Section 3.2.1). Note that it only does so when the conversation did not involve the Compromised agent. Honest participants will complete the conversation with the Compromised agent. The reason for this is that an attacker can negotiate keys with an honest participant and then publish the derived keys. The confidentiality of the keys is trivially broken when that happens, so I only check for confidentiality when two honest participants have negotiated the keys.

```
1  let processResponder(init:identity, resp:identity, phone:channel) =
2    (* Commit *)
3    in(ch, commit:bitstring);
4    let (hpki:bitstring, hcsi:bitstring) = commit in
5
6    (* DH1 *)
7    new skr : scalar;
8    let pkr  = mult(skr, Base) in
9    let hpkr = hash(pt2bs(pkr)) in
10   let hcsr = mac(bs2mk(Null), (hpkr, RespStr)) in
11   let dh1  = (pkr, hcsr) in
12   out(ch, dh1);
13
14   (* DH2 *)
15   in(ch, (pki:point, maci:bitstring));
16   let (=hpki) = hash(pt2bs(pki)) in
17
18   (* Confirm *)
19   let z       = mult(skr, pki) in
20   let htotal  = hash((commit, dh1, pki)) in
21   let kdk     = bs2mk(mac(bs2mk(htotal), pt2bs(z))) in
22   let context = (init, resp, htotal) in
23   let sessId  = hash((init, resp)) in
24   let kdk2    = bs2mk(mac(kdk, (MasterStr, AlgId, context, Null))) in
25   let ksnd    = kdf(kdk2, RespMasterLabel, sessId) in
26   let krcv    = kdf(kdk2, InitMasterLabel, context) in
27   let sas     = kdf(kdk2, SasLabel,        context) in
28   let cs1     = kdf(kdk2, CsLabel,         context) in
29   let (=maci) = kdf(kdk2, InitMACLabel,    context) in
30   let macr    = kdf(kdk2, RespMACLabel,    context) in
31   let isnd    = kdf(kdk2, RespIndexLabel,  sessId) in
32   let ircv    = kdf(kdk2, InitIndexLabel,  sessId) in
33   out(ch, macr);
34
```

```
35   (* Start verification of the initiator identity *)
36   event beginResp(init, resp, krcv, ksnd, ircv, isnd, cs1);
37
38   (* Confirm the SAS *)
39   in(phone, =sas);
40   out(phone, (sas, OK));
41
42   (* Check that the conversation was not with the Compromised party *)
43   if init <> Compromised then
44
45   (* Accept the initiator identity and corresponding key material *)
46   event acceptInit(init, resp, krcv, ksnd, ircv, isnd, cs1);
47
48   (* Publish secret values to test secrecy of generated key material *)
49   out(ch, sym_enc(bs2sk(ksnd), bs2n(Null), ksndRespFlag));
50   out(ch, sym_enc(bs2sk(krcv), bs2n(Null), krcvRespFlag));
51   out(ch, sym_enc(bs2sk(isnd), bs2n(Null), isndRespFlag));
52   out(ch, sym_enc(bs2sk(ircv), bs2n(Null), ircvRespFlag));
53   out(ch, sym_enc(bs2sk(cs1),  bs2n(Null), cs1RespFlag)).
```

Listing 3.11: Proverif: SCimp first key negotiation (responder)

The protocol description for the responder contains the same Proverif constructs for proving confidentiality and authenticity as the protocol description for the initiator. The protocol actions are simply the actions as specified by SCimp.

```
1  process
2    !
3    in(ch, (init:identity, resp:identity));
4    new phone : channel;
5    (! in(phone, x:bitstring); out(ch, x)) |
6
7    if init = Compromised then (
8      out(ch, phone);
9      processResponder(init, resp, phone)
10   ) else if resp = Compromised then (
11     out(ch, phone);
12     processInitiator(init, resp, phone)
13   ) else (
14     processInitiator(init, resp, phone) |
15     processResponder(init, resp, phone)
16   )
```

Listing 3.12: Proverif: SCimp first key negotiation (main)

Replication of a process is indicated by the exclamation mark (`!`). `!P` represents the infinite composition `P | P | ...`, which is used to indicate that the process can be run in arbitrarily many parallel sessions. In Listing 3.12, the full process is replicated, in order to let a protocol participant engage in key negotiations with arbitrarily many others, both in the role of initiator and responder.

In the real SCimp, participants should only have to run the first key negotiation protocol once per peer. After that, they share a cached secret *cs* and they engage in the re-keying protocol when they want to derive new keys. There is no effective way to encode this in Proverif.

However, when Proverif is able to prove the queries for arbitrarily many first key negotiations, that implies that those queries are also proven for just one first key negotiation. On the other hand, if Proverif finds an attack trace for multiple first key negotiations, this might be a false positive that does not apply to just one first key negotiation.

Most protocols have fixed roles assigned for participants, such as server and client. Because SCimp is a protocol between equivalent users, both of them can be the initiator of the protocol. This is encoded by letting the adversary set any value for the initiator and responder.

The next protocol step is to create a new phone channel, which is an authenticated, but non-secret channel. This is modelled by publishing everything that is put on the phone line to the public channel `ch`.

Result

Proverif is able to prove the correctness of all the queries.

I would have liked to add one more function to this model, which is to send data messages (or even rekeying) after key negotiation, but before the SAS confirmation. The protocol should ensure the confidentiality and authenticity of these messages, if the following SAS confirmation is successful. Unfortunately, Proverif does not allow sequential composition (`;`) after replication (`!`). That means that the model could not verify the SAS after sending/receiving arbitrary many data messages. Therefore, I have left this function out of the model.

3.2.2 Re-keying

When Alice and Bob already share a cached secret (*cs*), they no longer need to verify the SAS to authenticate. Instead, they rely on the value of *cs* to authenticate to each other and update their key material accordingly.

The full Proverif model can be found in `scimp_key_neg.pv`.

Queries

The queries are the same as in the first key negotiation, with the addition that I query the confidentiality and authenticity of the old value of *cs*.

Process

```
let processInitiator(init:identity, resp:identity, cs:bitstring) =
  (* Commit *)
  new ski : scalar;
  let pki    = mult(ski, Base) in
  let hpki   = hash(pt2bs(pki)) in
  let hcsi   = mac(bs2mk(cs), (hpki, InitStr)) in
  let commit = (hpki, hcsi) in
  out(ch, commit);

  (* DH1 *)
  in(ch, dh1:bitstring);
  let (pkr:point, hcsr:bitstring) = dh1 in
  if hcsr = mac(bs2mk(cs), (hash(pt2bs(pkr)), RespStr)) then

  (* DH2 *)
  let z       = mult(ski, pkr) in
  let htotal  = hash((commit, dh1, pki)) in
  let kdk     = bs2mk(mac(bs2mk(htotal), pt2bs(z))) in
  let context = (init, resp, htotal) in
  let sessId  = hash((init, resp)) in
  let kdk2    = bs2mk(mac(kdk, (MasterStr, AlgId, context, cs))) in
  let ksnd    = kdf(kdk2, InitMasterLabel, context) in
  let krcv    = kdf(kdk2, RespMasterLabel, sessId) in
  let sas     = kdf(kdk2, SasLabel,        context) in
  let cs1     = kdf(kdk2, CsLabel,         context) in
  let maci    = kdf(kdk2, InitMACLabel,    context) in
  let macr    = kdf(kdk2, RespMACLabel,    context) in
  let isnd    = kdf(kdk2, InitIndexLabel,  sessId) in
  let ircv    = kdf(kdk2, RespIndexLabel,  sessId) in
  event beginInit(init, resp, ksnd, krcv, isnd, ircv, cs1, cs);
  out(ch, (pki, maci));

  (* Confirm *)
  in(ch, =macr);

  (* Accept the responder identity and corresponding key material *)
  event acceptResp(init, resp, ksnd, krcv, isnd, ircv, cs1, cs);

```

```
39    (* Publish secret values to test secrecy of generated key material *)
40    out(ch, sym_enc(bs2sk(ksnd), bs2n(Null), ksndInitFlag));
41    out(ch, sym_enc(bs2sk(krcv), bs2n(Null), krcvInitFlag));
42    out(ch, sym_enc(bs2sk(isnd), bs2n(Null), isndInitFlag));
43    out(ch, sym_enc(bs2sk(ircv), bs2n(Null), ircvInitFlag));
44    out(ch, sym_enc(bs2sk(cs1),  bs2n(Null), cs1InitFlag));
45    out(ch, sym_enc(bs2sk(cs),   bs2n(Null), csInitFlag)).
```

Listing 3.13: Proverif, SCimp rekeying (initiator)

There is no need to check if the other party was compromised before running the `acceptResp` event and publishing the flags, because a protocol is never initiated with a Compromised party.[1]

During rekeying, the moment of authentication is when the values `maci` and `macr` are validated. This explains the location of the `beginInit` and `acceptResp` events. A closer comparison with the actual C implementation shows that many values are *expanded* only after the macr was received.

After rekeying, the values of the cached secret can be updated to the value `cs1`. The precise moment is indicated by the `accept` event. Unfortunately, Proverif has no method for updating values.

The responder process is similar to the initiator process:

```
1   let processResponder(init:identity, resp:identity, cs:bitstring) =
2     (* Commit *)
3     in(ch, commit:bitstring);
4     let (hpki:bitstring, hcsi:bitstring) = commit in
5
6     (* DH1 *)
7     new skr : scalar;
8     let pkr  = mult(skr, Base) in
9     let hpkr = hash(pt2bs(pkr)) in
10    let hcsr = mac(bs2mk(cs), (hpkr, RespStr)) in
11    let dh1  = (pkr, hcsr) in
12    out(ch, dh1);
13
14    (* DH2 *)
15    in(ch, (pki:point, maci:bitstring));
16    let (=hpki) = hash(pt2bs(pki)) in
17
18    (* Confirm *)
19    let z       = mult(skr, pki) in
20    let htotal  = hash((commit, dh1, pki)) in
21    let kdk     = bs2mk(mac(bs2mk(htotal), pt2bs(z))) in
22    let context = (init, resp, htotal) in
23    let sessId  = hash((init, resp)) in
24    let kdk2    = bs2mk(mac(kdk, (MasterStr, AlgId, context, cs))) in
25    let ksnd    = kdf(kdk2, RespMasterLabel, sessId) in
26    let krcv    = kdf(kdk2, InitMasterLabel, context) in
27    let sas     = kdf(kdk2, SasLabel,        context) in
28    let cs1     = kdf(kdk2, CsLabel,         context) in
29    let isnd    = kdf(kdk2, RespIndexLabel,  sessId) in
30    let ircv    = kdf(kdk2, InitIndexLabel,  sessId) in
31    let (=maci) = kdf(kdk2, InitMACLabel,    context) in
32    event acceptInit(init, resp, krcv, ksnd, ircv, isnd, cs1, cs);
33    let macr    = kdf(kdk2, RespMACLabel,    context) in
34    event beginResp(init, resp, krcv, ksnd, ircv, isnd, cs1, cs);
35    out(ch, macr);
36
37    (* Publish secret values to test secrecy of generated key material *)
38    out(ch, sym_enc(bs2sk(ksnd), bs2n(Null), ksndRespFlag));
39    out(ch, sym_enc(bs2sk(krcv), bs2n(Null), krcvRespFlag));
40    out(ch, sym_enc(bs2sk(isnd), bs2n(Null), isndRespFlag));
41    out(ch, sym_enc(bs2sk(ircv), bs2n(Null), ircvRespFlag));
42    out(ch, sym_enc(bs2sk(cs1),  bs2n(Null), cs1RespFlag));
43    out(ch, sym_enc(bs2sk(cs),   bs2n(Null), csRespFlag)).
```

[1] This was required in the first key negotiation, because that contained freshly generated phone channel.

Listing 3.14: Proverif: SCimp Rekeying (responder)

Note the location of the events `acceptInit` and `beginResp`, which shows that the responder only provides his proof for authentication after he has already accepted the authentication of the initiator.

```
process
  !
  in(ch, (init:identity, resp:identity));
  let cs = getCS(init, resp) in

  if init = Compromised || resp = Compromised then (
    out(ch, cs)
  ) else (
    processInitiator(init, resp, cs) |
    processResponder(init, resp, cs)
  )
```

Listing 3.15: Proverif: SCimp Rekeying (main)

The main process is slightly simpler than that of the first key negotiation. Instead of generating a new private phone channel, a *cs* value is derived from the identities. When one of the identities is the compromised agent, that value of *cs* is published on `ch`. For now, I am only interested in the security of the protocol when the protocol is run with uncompromised participants.[2]

Result

Proverif is able to verify the correctness of all queries. That means that when the value *cs* is not compromised, the attacker has no way of compromising the rekeying protocol.

3.2.3 Sending data

When sending messages (data) with SCimp, there are several properties that must be proved. First of all, both the message and key must be kept confidential and authentic. Furthermore, key erasure should ensure that a compromise of keys does not compromise the confidentiality of keys or messages in the past. Lastly, I would like to prove that deniability holds for sent messages. That last property can be proven with the Proverif notion of *equivalence*, which cannot be combined with any other queries. Therefore, two Proverif documents were created to test all properties.

One thing that is missing from the Proverif files is sending arbitrary many messages. The problem is that Proverif has no notion of state and is therefore unable to update the value of variables. Specifically, it is unable to update the key value. I have modelled the sending of a single message, after which I prove the confidentiality and authenticity of the new key. The updated key (assuming it was not compromised) could be fed back into the model, also ensuring the confidentiality and authenticity of subsequent messages.

Another thing that is missing from this model is the actual deleting of keys. Proverif does not have a notion of device memory. The solution is to leak only the information that is stored in the device memory at the time of leakage. When messages arrive in order, this model would be a very accurate reflection of reality. Unfortunately, out-of-order messages happen frequently in a mobile environment, meaning that my model is only an inconclusive approximation of the actual implementation. Therefore Proverif misses the problem that SCimp has with keys that remain stored, which was described in Section 2.2.1.

Key erasure

[2] See Section 3.2.5 for a security proof of backwards secrecy when *cs* is compromised.

```
(* Key material (established at key negotiation) *)
free k0 : bitstring [private].
free i0 : bitstring.
free sessionId : bitstring.
query attacker(k0).

(* Future key material *)
free k1SndFlag, k1RcvFlag : bitstring [private].
query attacker(k1SndFlag) phase 0;
      attacker(k1RcvFlag) phase 0.

(* Secrecy of the message *)
free msg : bitstring [private].
noninterf msg.

(* Queries for authenticity *)
event sendMessage(bitstring, bitstring, bitstring, bitstring, bitstring).
event receiveMessage(bitstring, bitstring, bitstring, bitstring, bitstring).
query k0:bitstring, i0:bitstring, k1:bitstring, i1:bitstring, m:bitstring;
      event(receiveMessage(k0, i0, k1, i1, m))
        ==> event(sendMessage(k0, i0, k1, i1, m)).
```

Listing 3.16: Proverif: SCimp data (queries)

The shared encryption key `k0` comes from key negotiation and is thus assumed to be private. The sessionId is derived from the public XMPP addresses of the participants and is thus public. Although the index `i0` is proven to be private initially, it is only authenticated in the ciphertext and never encrypted. Although Proverif can prove that the index stays confidential, this is due to some assumptions that Proverif makes from the way that I defined encryption, which does not necessarily correspond with the actual AEAD mode that is used in SCimp. To prevent such subtle reasoning on interpretation of the result, I have made the index value public. If the key and message stay confidential when using a public index, then surely it stays confidential with a (partially) private index.

For the confidentiality of future key material, the *phase* keyword is introduced. Phases can be inserted in Proverif processes and function as a synchronizing event for parallel processes, starting from phase 0. I introduced phase 1 (see Listing 3.17), in which the key is leaked. The prover checks that the key remains confidential until the leak happens.

Confidentiality of the messages requires strong secrecy, as denoted by Proverif's notion of non-interference. Assertion of this query proves that the value of the message does not affect the observable behavior of the protocol. In other words, this means that the message does not need to have much entropy in order to remain confidential.

For authenticity, no injective events are required, because the protocol runs only once before the keys are updated.

```
let processSender(ksnd:bitstring, isnd:bitstring) =
  let key = bs2sk(splitFst(ksnd)) in
  let nonce = bs2n(splitSnd(ksnd)) in
  let ct = aead_enc(key, nonce, isnd, msg) in
  let ksnd1 = kdf(bs2mk(ksnd), msgKeyLabel, (sessionId, isnd)) in
  let isnd1 = increment(isnd) in
  out(ch, sym_enc(bs2sk(ksnd1), bs2n(Null), k1SndFlag));
  event sendMessage(ksnd, isnd, ksnd1, isnd1, msg);
  let isnd_pub = splitSnd(isnd) in
  out(ch, (isnd_pub, ct));
  out(ch, isnd1);

  phase 1;
  out(ch, ksnd1).

let processReceiver(krcv:bitstring, ircv:bitstring) =
  let ircv_pub = splitSnd(ircv) in
  in(ch, (=ircv_pub, ct:bitstring));
  let key = bs2sk(splitFst(krcv)) in
```

```
20    let nonce = bs2n(splitSnd(krcv)) in
21    let pt = aead_dec(key, nonce, ircv, ct) in
22    let krcv1 = kdf(bs2mk(krcv), msgKeyLabel, (sessionId, ircv)) in
23    let ircv1 = increment(ircv) in
24    out(ch, sym_enc(bs2sk(krcv1), bs2n(Null), k1RcvFlag));
25    out(ch, ircv1);
26
27    event receiveMessage(krcv, ircv, krcv1, ircv1, pt);
28
29    phase 1;
30    out(ch, krcv1).
31
32  process
33    processSender(key0, index0) | processReceiver(key0, index0)
```

Listing 3.17: Proverif: SCimp data (process)

The process for sending the data is straightforward. In phase 0, the ciphertext is encrypted and decrypted and the keys get updated. After sending the message, the index is also leaked, because it is assumed to be a public value. Only in phase 1 does the key leak.

Deniability

Deniability is encoded as an equivalence between two processes. In the first process, the sender creates and sends a message, which then gets leaked by the receiver by publishing the key. In the second process, the receiver simulates the process by creating a fake message and sending it to himself, after which he leaks the value again. To satisfy deniability, a third party should not be able to distinguish between the two processes.

```
1   let processSender(ksnd:bitstring, isnd:bitstring) =
2     new msg : bitstring;
3     let key = bs2sk(splitFst(ksnd)) in
4     let n = bs2n(splitSnd(ksnd)) in
5     let ct = aead_enc(key, n, isnd, msg) in
6     let isnd_pub = splitSnd(isnd) in
7     out(ch, (isnd_pub, ct)).
8
9   let processFakeSender(krcv:bitstring, ircv:bitstring) =
10    new fakemsg : bitstring;
11    let key = bs2sk(splitFst(krcv)) in
12    let n = bs2n(splitSnd(krcv)) in
13    let fakect = aead_enc(key, n, ircv, fakemsg) in
14    let isnd_pub = splitSnd(ircv) in
15    out(ch, (isnd_pub, fakect)).
16
17  let processReceiverReveal(krcv:bitstring, ircv:bitstring) =
18    let ircv_pub = splitSnd(ircv) in
19    in(ch, (=ircv_pub, ct:bitstring));
20    (* Reveal the message by publishing the key *)
21    out(ch, (krcv, ircv)).
22
23  equivalence
24    ! in(ch, (k:bitstring, i:bitstring));
25      processSender(k, i) |
26      processReceiverReveal(k, i)
27    ! in(ch, (k:bitstring, i:bitstring));
28      processFakeSender(k, i) |
29      processReceiverReveal(k, i)
```

Listing 3.18: Proverif: SCimp data (deniability)

Result

Proverif is able to verify all queries on the secrecy and authenticity of the messages. That also means that the attacker is unable to learn old keys or messages from compromised keys.

The model on deniability is trivially proven by Proverif: the process for the sender and the fake sender only differ by the names of variables.

3.2.4 Progressive Encryption

Progressive Encryption, introduced in SCimp version 2, introduces an extension to the first key negotiation. This allows for a message to be sent alongside the first key negotiation message. To keep the Proverif model simple enough for the tool to be able to prove all queries, I have only modelled this first message. All data messages after the first message (PKStart) should be as secure as that first message, for the same reasons explained in Section 3.2.3.

In addition to confidentiality and authenticity queries for the keys that are derived from the ephemeral key exchange, I also added those queries for the keys derived from the responders medium term public key. For the message, I added one more query.

```
1  free msg0InitFlag, msg0RespFlag : bitstring [private].
2
3  query attacker(msg0InitFlag); attacker(msg0RespFlag).
4
5  query attacker(new msg0).
```

Listing 3.19: Proverif: SCimp Progressive Encryption (message query)

Listing 3.19 shows there are two queries to be asked about the message that is sent in the PKStart message (`msg0`). The queries in line 3 check for a flag, that is published encrypted with the value `msg0`, however, only after an honest participant has authenticated the other person. This query tests the claim that if you have verified the SAS with the other party, that proves that all communication so far has been confidential and authenticated. The query in line 5 simply checks if an attacker can get the value of the first sent message.

```
1  (* Role of the server (handing out keys) *)
2  let processGetKey =
3    in(ch, resp:identity);
4    get keys(=resp, pkResp, locResp) in
5    out(ch, (pkResp, locResp)).
6
7  (* Role of the server (key registration) *)
8  let processKeyRegistration =
9    in(ch, (id:identity, pubkey:point, locator:bitstring));
10   insert keys(id, pubkey, locator).
```

Listing 3.20: Proverif: SCimp Progressive Encryption (server)

The server processes are straightforward. Note that communication with the server goes over a public channel. Additionally, the key registration process simply adds the public key for any value that it receives, meaning that an attacker can simply overwrite the public keys of an honest participant with his own public key.

This design probably does not reflect the actual process for key registration and retrieval in the actual application, where there is some protection against these attacks at the server side. However, the reason for not doing any server side checks in the model is that the protocol should protect the secrecy and integrity of user messages even when the server is corrupted. Any checks that I would have added in the model would imply that the protocol users need to trust the server[3] that they use to communicate. This model requires no trust in the communication server.

```
1  let processInitiator(init:identity, resp:identity, phone:channel) =
2    (* Get responder public key *)
3    out(ch, resp);
4    in(ch, (pkResp:point, locResp:bitstring));
5
6    (* PKStart *)
7    new msg0 : bitstring;
```

[3] and the government of the country in which those servers are placed

```
  new ski0 : scalar;
  let pki0     = mult(ski0, Base) in
  let z0       = mult(ski0, pkResp) in
  let kdk0     = bs2mk(mac(bs2mk(Null), pt2bs(z0))) in
  let context0 = (init, resp) in
  let sessId0  = hash(context0) in
  let kdk20    = bs2mk(mac(kdk0, (MasterStr, AlgId, context0, Null))) in
  let ksnd0    = kdf(kdk20, InitMasterLabel, context0) in
  let krcv0    = kdf(kdk20, RespMasterLabel, sessId0) in
  let isnd0    = kdf(kdk20, InitIndexLabel,  sessId0) in
  let ircv0    = kdf(kdk20, RespIndexLabel,  sessId0) in
  let k0       = bs2sk(splitFst(ksnd0)) in
  let n0       = bs2n(splitSnd(ksnd0)) in
  let ct0      = aead_enc(k0, n0, isnd0, msg0) in

  new ski : scalar;
  let pki     = mult(ski, Base) in
  let hpki    = hash(pt2bs(pki)) in
  let pkstart = (locResp, pki0, hpki, ct0) in
  out(ch, pkstart);

  ...
```

Listing 3.21: Proverif: SCimp Progressive Encryption (initiator)

The process for the initiator starts of by retrieving a public key over a public channel, then creating the PKStart message. After this, the protocol continues as it did for the first key negotiation protocol (see Section 3.2.1).

```
let processResponder(init:identity, resp:identity, phone:channel) =
  (* Register public key *)
  new skr0 : scalar;
  let pkr0 = mult(skr0, Base) in
  let locr0 = kdf(pt2mk(pkr0), LocatorLabel, id2bs(resp)) in
  out(ch, (resp, pkr0, locr0));

  (* PKStart *)
  in(ch, pkstart:bitstring);
  let (=locr0, pki0:point, hpki:bitstring, ct:bitstring) = pkstart in
  let z0       = mult(skr0, pki0) in
  let kdk0     = bs2mk(mac(bs2mk(Null), pt2bs(z0))) in
  let context0 = (init, resp) in
  let sessId0  = hash(context0) in
  let kdk20    = bs2mk(mac(kdk0, (MasterStr, AlgId, context0, Null))) in
  let ksnd0    = kdf(kdk20, RespMasterLabel, sessId0) in
  let krcv0    = kdf(kdk20, InitMasterLabel, context0) in
  let isnd0    = kdf(kdk20, RespIndexLabel,  sessId0) in
  let ircv0    = kdf(kdk20, InitIndexLabel,  sessId0) in
  let k0       = bs2sk(splitFst(krcv0)) in
  let n0       = bs2n(splitSnd(krcv0)) in
  let pt0      = aead_dec(k0, n0, ircv0, ct) in

...
```

Listing 3.22: Proverif: SCimp Progressive Encryption (responder)

The responder first registers his key with the server, then receives the PKStart message and continues as in the first key negotiation.

```
process
  !
  in(ch, (init:identity, resp:identity));
  new phone : channel;
  (! in(phone, x:bitstring); out(ch, x)) |

  ! processKeyRegistration |
  ! processGetKey |
```

```
9   if init = Compromised then (
10    out(ch, phone);
11    processResponder(init, resp, phone)
12  ) else if resp = Compromised then (
13    out(ch, phone);
14    processInitiator(init, resp, phone)
15  ) else (
16    processInitiator(init, resp, phone) |
17    processResponder(init, resp, phone)
18  )
```

Listing 3.23: Proverif: SCimp Progressive Encryption (main)

The main process is the same as for the first key negotiation, with the additional server processes for key registration and key retrieval.

The full Proverif model has one alteration to the one presented in this section, which is that it contains two different processes for the initiator: one when talking with a Compromised agent and one when talking with an honest agent. The responder process is split up in two similarly. The reason is that this allows me to provide hints to the prover, so that the prover can achieve a significant speedup.

Result

Proverif finds an attack on retrieving `msg0`. Upon investigation of the attack trace, it turns out that it is a really simple attack: the attacker injects her own public key when the initiator asks the server for the correct key and is thus able to decrypt. This corresponds to the man-in-the-middle attack described in Section 2.5.4.

However, Proverif is able to prove that the attacker cannot get the flags `msg0InitFlag` and `msg0RespFlag`. This implies that although the attacker can compromise `msg0`, the users will detect that they have been compromised once they try to verify the SAS.

All other queries for confidentiality and authenticity for the key material also hold. For these values as well, the protocol can only guarantee these properties after the users have confirmed the SAS.

Before I can conclude that the model proves that SCimp Progressive Encryption is indeed secure, I need to look at the differences between the model and reality. Progressive Encryption contains two Diffie-Hellman key exchanges, but only one SAS validation. Key continuity ensures that this validates both key exchanges, by digesting the full PKStart message in the context value. What is not modelled, is what happens when key negotiation messages arrive out of order. As I said in Section 2.4, when this happens, the client throws away all key material. When it happens after a PKStart message, there is no longer any way for clients to verify the confidentiality and authenticity of the messages sent up to then.

3.2.5 Backwards secrecy

The SCimp protocol promises to provide backwards secrecy when key material was compromised. An attacker that has compromised the k_{snd} and k_{rcv} keys, can perform a man-in-the-middle attack on the messages. However, when the devices rekey, they update those keys with additional randomness from the ephemeral Diffie-Hellman key exchange, so that the attacker can no longer compromise new messages. However, if the attacker has also compromised the value of cs, she can execute a man-in-the-middle attack on the rekeying protocol, thereby compromising subsequent key material as well. What I want to test in the following models are two ways in which this attack is mitigated.

The Proverif models for both methods require just some minor modifications to the model for rekeying (see Section 3.2.2), so I will not write the listings of the model here.

Attacker misses rekeying

The first mitigation happens when the attacker misses the Diffie-Hellman key exchange in the rekeying protocol, thereby missing the opportunity to inject her own keys and prolonging the man-in-the-middle attack. With Proverif, I want to prove that the self-healing property works as intended. When the attacker misses the first rekeying, the protocol should self-heal without any action required from the user.

The only change in Proverif is that the messages DH1 and DH2 are sent over the authenticated `sync` channel, instead of the public channel `ch`. The attacker can eavesdrop on this channel, but cannot inject his own DH exchange message.

Users reconfirm the SAS

The second mitigation happens when the users reconfirm the SAS after rekeying. Under uncompromised circumstances, this is not necessary because of the key continuation properties of the protocol, but when they have been victim of a man-in-the-middle during rekeying, they should be able to detect this by comparing their SAS values.

The Proverif model is simply a combination of the models for the first key negotiation (the users confirm the SAS) and the rekeying model (the users share a value *cs* beforehand). The model tests a subtly different security claim than mere detection of a man-in-the-middle: it checks that Eve is unable to successfully set up a man-in-the-middle attack during the *first* rekeying, if Alice and Bob reconfirm the SAS afterwards.

This last model does require another side note: if an attacker is able to create a SAS collision, the users gain nothing by reconfirming the SAS. Consider an attacker (Eve) that has compromised *cs* and attacks the rekeying protocol by setting up a man-in-the-middle. First, she sets up a connection with Alice, using the *cs*. She inspects the resulting SAS value. She then keeps rekeying with Bob until she has found a SAS collision. With just 16 bits of entropy in the SAS, Eve expects to do 2^{15} of these rekeyings before she finds such a collision.

According to Proverif, finding a collision among SAS values is impossible. Proverif does has a notion for values with low entropy: *weaksecret*. By declaring a weaksecret, Proverif validates that no offline brute-force attack is possible. It should be encoded that SAS is a *weaksecret*, but instead of being vulnerable to an offline attack, it is vulnerable to an online attack requiring interaction with an honest user device. Unfortunately, I found no way to model this in Proverif.

Result

Proverif is able to verify that, when *cs* is compromised, the protocol does self-heal and that the newly computed key material is both confidential and authentic, if the attacker misses the first opportunity to inject his man-in-the-middle attack.

Proverif is also able to prove the same when Alice and Bob reconfirm the SAS after rekeying. The assumption that Proverif makes is that SAS collisions are impossible, which is not necessarily true for the low-entropy SAS values used in the actual protocol.

Chapter 4

Implementation details

Security protocols are three-line programs that people still manage to get wrong.

—Roger Needham

Although security protocols can often be abstracted to just the key negotiation messages, real world implementations that have to run on a physical machine require more than three lines of code. The SCimp implementation is written in C and should be suited to run on both Android and iPhone mobile devices. A cross-platform library called *libsccrypto* contains the code for both SCimp and SCloud.

The heart of SCimp is implemented in a few files: `SCimp.c` (1219 sloc[1]), `SCimpProtocol.c` (2970 sloc), `SCimp.h` (451 sloc) and `SCimpPriv.h` (227 sloc). This code will call cryptographic functions, provided by the LibTomCrypt library, wrapped in the custom Silent Circle functions (in `SCcrypto.c` and `SCccm.c`). To convert the data to messages suitable for communication with the SCimp message format, two serializers are provided in `SCimpProtocolFmtJSON.c` and `SCimpProtocolFmtXML.c`. For the communication to be secure, this large code-base needs to remain bug-free. To put it into the words of one of the Silent Circle co-founders:

> *All bugs are security bugs* —Jon Callas [18].

There are many ways in which a bug in the code can affect the security of the communication. The encryption primitives could be implemented wrong and leak information to an adversary through side-channels, which I analyze in Section 4.3. A bug could also crash the application, leading to a possible denial of service attack or even compromise the integrity and confidentiality of the protocol. I further investigate this in Section 4.4. Section 4.6 concludes this chapter by analyzing non security critical bugs, which could also be classified as *style issues*. I explicitly mention them because I believe that these bugs hinder the legibility and maintainability of the overall software.

4.1 Basic structure of the implementation

The basic flow of a SCimp conversation is shown in Figure 4.1. Each conversation is managed by a separate `SCimpContext` "object" (actually a structure, because C has no objects). It can be created with the function `SCimp.c::SCimpNew`, as shown in point 1. Various properties can then be set, such as a choice for the cipher suite and the SAS method. Setting the event handler is non-optional and turns out to be very important for the functionality and security of the protocol. This handler will be responsible for many types of events, the most important ones being errors, warnings and sending packets.

[1] source lines of code, not counting comments and empty lines

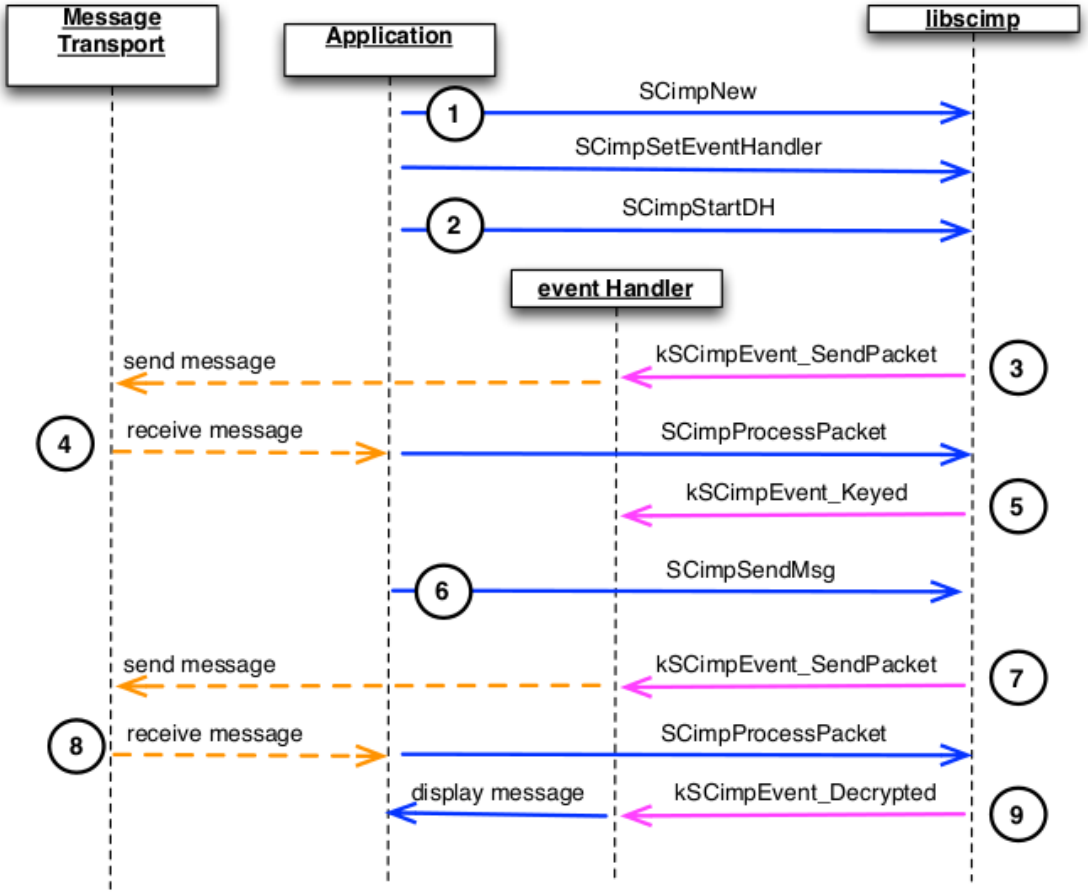

Figure 4.1: SCimp message flow (source: [21])

The `SCimpContext` structure contains a few fields worth mentioning here. First of all, it holds a `state` variable of type `SCimpState`. It indicates the current position in the protocol. For example, when the devices `state` is `kSCimpState_DH2`, the role of the device is initiator and it has just sent a DH2 message. Internally, SCimp has a function `sProcessTransition` that manages the `state`, by referring to the state machine, implemented in `SCIMP_state_table`. Each table entry consists of four items:

1. the current state
2. the transition that was called
3. the next state
4. the function that handles the transition

A `SCimpContext` also holds handlers for serializing and deserializing the data. The default handler serializes C structures to a JSON format.

At point 2 of Figure 4.1, the initiator starts the first key negotiation by calling `SCimpStartDH` (or for Progressive Encryption: `SCimpStartPublicKey`). Points 3–5 show the basic structure of how the first key negotiation is further handled. Internally, when `SCimpProcessPacket` is called, the deserialize handler processes the message and triggers a transition, based upon the message tag. The functions that are then called by the state machine are responsible for putting all the data in the right place to be able to call the cryptographic primitives. After the *keyed* event has been triggered, the participants can send data, as shown in points 6–9.

Besides handing the messages, a `SCimpContext` is also responsible for managing key material, including storing keys for out of order messages. It also holds on to a queue of transitions (necessary when running with multiple threads), the role of the current device (initiator or responder), et cetera. Basically, it is just a structure that holds the entire administration that is required for running SCimp.

Most functions have roughly the same structure. They return an error code (`kSCLError_NoErr` indicates no error), and all input and output is passed as a parameter, accompanied by a length parameter if necessary. Usually, the first parameter is the `SCimpContext`. A label `done` marks the end of the function, after which some memory cleanup can be done before returning the error code.

4.2 CCM Encryption

The design of SCimp contains the CCM blockmode, standing for Counter mode with CBC-MAC. Counter mode is used for encryption/decryption of the data, while the CBC-MAC authenticates the data, by computing an authentication tag over the plaintext and additional data—or header. This blockmode is implemented in LibTomCrypt [35][33], according to the NIST specification [13]. This implementation contained an error leading to a timing side-channel vulnerability.

According to the NIST specification of CCM, the authentication tag is part of the ciphertext. In order to decrypt, this full ciphertext must be decrypted, resulting in a "plaintext" tag. The tag must then be recomputed from the plaintext and compared with the decrypted value. However, upon decryption in the LibTom implementation, the plaintext is computed and a tag is computed from the header and (decrypted) plaintext. This is then re-encrypted, so that the caller of the function must check the correctness of the tag.

The NIST specification specifies that "only the error message INVALID is returned" when the decryption-verification fails. In that case "the payload P and the MAC T shall not be revealed" and "the implementation shall ensure that an unauthorized party cannot distinguish whether the error message results from [invalid message format] or from [authentication failure], for example, from the timing of the error message."

In the case of SCimp, the `ccm_memory` wrapper function in `SCccm.c` does indeed compare the tag value after decryption. The comparison is done with a call to `memcmp`, which takes a variable amount of time (see also Section 4.3).

I proposed several fixes to the LibTomCrypt developers, for example removing the tag parameter from the function and letting it be part of the ciphertext parameter (as it should be according to the specification). Verification of authentication can then be done inside the function `ccm_memory`. Of course, this would break compatibility with the existing implementation, which is not what they wanted. Instead, I came up with a quickfix[2] that is a bit of a compromise.

```
1   if (direction == CCM_ENCRYPT) {
2      /* store the TAG */
3      for (x = 0; x < 16 && x < *taglen; x++) {
4          tag[x] = PAD[x] ^ CTRPAD[x];
5      }
6      *taglen = x;
7   } else { /* direction == CCM_DECRYPT */
8      /* decrypt the tag */
9      for (x = 0; x < 16 && x < *taglen; x++) {
10        ptTag[x] = tag[x] ^ CTRPAD[x];
11     }
12     *taglen = x;
13
14     err = XMEM_NEQ(ptTag, PAD, *taglen);
15  }
```

Listing 4.1: LibTomCrypt: CCM quickfix

The function `ccm_memory` does both encryption and decryption, depending on the direction parameter. At the end of the `ccm_memory` function, the `PAD` array holds the value of the authentication tag, computed over the plaintext and header, `CTRPAD` is the encryption/decryption block from the Counter mode and `tag` is the function parameter. In case of decryption, the `tag` is decrypted and compared against the computed `PAD` with the `XMEM_NEQ` function.

Upon inspection of this function, I found another bug. The function did indeed do a constant time comparison of the input arrays, by computing the exclusive or of the complete array per byte and combining the results with the or-operator. The problem was that they did not convert this byte into a single bit output before outputting the result. By investigating the exact value of the returned value, an attacker might learn something about the secret data that was compared.

I proposed a simple fix, by folding the resulting byte (`ret`) into one single bit.[3]

```
1   [...]
2
3   ret |= ret >> 4;
4   ret |= ret >> 2;
5   ret |= ret >> 1;
6   ret &= 1;
7
8   return ret;
```

Listing 4.2: LibTomCrypt: `XMEM_NEQ`

The developers also asked me to have a look at timing side channel attacks in other encrypted authentication methods. I politely refused, because it is outside the scope of this thesis. One thing to note, however, is that modes that work by *encrypt-then-mac* are not vulnerable to the attack that was described in this section, because a good implementation will only decrypt data that is authenticated.[4]

4.3 Side-channels

A side-channel to software implementations of a cryptosystem happens when an attacker can learn secret information from observing the physical implementation. For example, an attacker

[2] commit 25af184cd59b1769c0588678362adb5fd41a50ed
[3] commit 75b114517a3f8db2075a45b0af87d4d74778ad66
[4] See also: `http://thoughtcrime.org/blog/the-cryptographic-doom-principle/`.

might observe how long a device takes to perform a certain cryptographic operation. If the timing depends upon secret data, some information of that secret leaks to an attacker measuring the time taken.

In this section, I looked for software bugs that might lead to a timing attack. I did not analyze the LibTomCrypt library for other side-channel vulnerabilities, such as cache-based side-channel attacks [8], because I considered this out of scope of the project.

4.3.1 Comparison of secrets

Every comparison of secret data should be done in constant time. I have shown in the previous section that after CCM decryption, the verification was done by calling `memcmp`, which does non-constant time comparison, because it sequentially compares byte-by-byte and returns as soon as the first difference was encountered This is not the only place in the code where this happens, because almost all secret values are compared with this same function, leading to side-channel attacks.

Lines 121–125 of file `SCpubTypes.h` define a macro `CMP`, that is used to compare secret values. It is defined as:

```
1 #define CMP(b1, b2, length)                    \
2 (memcmp((void *)(b1), (void *)(b2), (length)) == 0)
3
4 #define CMP2(b1, l1, b2, l2)                    \
5 (((l1) == (l2)) && (memcmp((void *)(b1), (void *)(b2), (l1)) == 0))
```

Listing 4.3: SCpubTypes.h::CMP, SCpubTypes.h::CMP2

The macro opens up a side-channel in the following places in the code.

1. when validating hcs_b, sent in message DH1 (`SCimpProtocol.c::2254`);
2. when validating hcs_a, sent in message DH2 (`SCimpProtocol.c::2318`);
3. when validating mac_a, sent in message DH2 (`SCimpProtocol.c::2336`);
4. when validating mac_b, sent in message Confirm (`SCimpProtocol.c::2371`);

The macro is also used to validate pk_a, but because that value is not secret, it is not a problem.

When the *mac* value is invalid, the protocol is simply aborted and the key material is discarded. Because the *mac* depends upon ephemeral key material, the next value of the *mac* will be totally different. It is therefore unlikely that an attacker is able to exploit this side-channel.

The value of *hcs* depends upon the long-term secret value *cs* and not upon fresh random values, meaning that it might leak through a timing attack. The attacker can fix a value for pk_b and send a guess for hcs_b in DH1 and time how long it takes to receive DH2. When the value of hcs_b is wrong, the honest initiator will get a warning, but the protocol will continue and the device will reply with DH2.

For the attack to succeed, a few things are required. First of all, the attacker has to trick the victim into initiating the protocol repeatedly. I have not found a way to do this remotely. Secondly, the warning might give away that the attack is in progress, at which point the device might decide[5] that the value *cs* is no longer trusted and that the SAS should be confirmed to re-authenticated the other person. If that happens, it does not matter if the attacker learns *cs*. Thirdly, the attacker needs to make sure that the value of *cs* does not get erased on the side of the honest initiator, which will happen if she sends any other message than Confirm. The honest initiator might still erase her value of *cs*, but that depends upon the error handler.

A timing attack on hcs_a is also unlikely, because this value gets sent together with mac_a in message DH2. The chance of a correct guess for mac_a is negligible, so the honest responder will abort the protocol. For the attacker, that means that there will be no reply from which to measure timing.

[5] depending on the handlers, see Section 4.6.2

Of course, these timing attacks are assumed to be done over the network. If an attacker can get their hands on the victims device, it might be easier to exploit the timing vulnerabilities. For example, re-initiating the rekeying protocol might be trivial.

4.3.2 Fix

Even though exploiting the existing timing channels is non-trivial, to fix the vulnerabilities is very easy. Simply use a constant time comparison, preferably by reusing the `XMEM_NEQ` function in LibTomCrypt (see Listing 4.2).

4.4 Software bugs

The code contains a few other bugs as well.

4.4.1 Transition queue race condition

The library is designed to work with multiple threads running in parallel. In order to keep these threads from competing for the same resources, the `SCimpContext` structure has a `pthread_mutex_t` value (`mp`), that gets locked every time a SCimp transition is triggered:

```
1  SCLError scTriggerSCimpTransition(SCimpContextRef ctx,
2                                    SCimpTransition trans, SCimpMsg* msg)
3  {
4      SCLError      err          = kSCLError_NoErr;
5
6
7      if(pthread_mutex_trylock(&ctx->mp) == EBUSY)
8      {
9          err = sQueueTransition(&ctx->transQueue, trans, msg);
10         return err;
11     }
12
13     err = sProcessTransition(ctx, trans, msg);
14
15     while(ctx->transQueue.count > 0)
16     {
17         TransItem  item;
18
19         err = sDeQueueTransition(&ctx->transQueue, &item);
20         if(err == kSCLError_NoErr)
21         {
22             sProcessTransition(ctx, item.trans, item.msg);
23         }
24     }
25
26     pthread_mutex_unlock(&ctx->mp);
27
28     return err;
29 }
```

This mutex ensures that no more than one thread is able to process a transition at the same time. However, this implementation is missing a mutex on the queue itself. The thread holding the mutex might unlock it before the new transition was enqueued, with the result that it will not be processed. The thread that holds the mutex `mp` might try to dequeue the `ctx->transQueue`, while another thread is not yet finished with enqueuing a new item, or two threads might try to enqueue to the same queue slot at the same time, resulting in corrupted data. This bug is known as a race condition and can lead to non-deterministic behavior.

Fix

To fix this bug, the blocking function `pthread_mutex_lock` should be used instead of the non-blocking `pthread_mutex_trylock`. If this costs too much in terms of performance, an additional mutex could lock access to the queue. The latter solution will probably turn out moderately complex, because it should prevent the active thread from unlocking the `mp` mutex before the item was enqueued.

4.4.2 Delete receiving keys

Upon receiving a data message, the receiver tries to retrieve the key by using the (plaintext) value $\mathrm{LSB}(i_{snd,j})$ in the function call `sGetKeyforMessage`. When that function finds the key in the queue of receive keys, it copies the key to a buffer and overwrites the queue item with zeros. It continues to verify-decrypt the message, displaying it to the user and finally clear the copied key buffer as well.

This gives the opportunity for an attacker to prevent a denial of service attack, by injecting random messages to the receiver, but with a value for the index that was expected. Since the index value is simply incremented with every message, the attacker can repeat the attack for as long as he likes, thereby forwarding the key of the receiver arbitrarily far, disabling further communication between the honest participants.

Fix

Only after the message was verified successfully, should the key value be deleted.

4.4.3 Verify memory allocation

In order to allocate memory, C has the function `malloc`, which is used on a few places in the SCimp implementation. More often, the code calls the macro `XMALLOC`, which is defined either in the file `src/main/sccrypto/SCPubTypes.h`, the file `src/main/tommath/tommath.h` or in `src/main/tomcrypt/headers/tomcrypt_custom.h`, depending on the order of imports. In any case, the macro simply gets reduced to `malloc`. A call to `malloc` can fail and return `NULL`, which should be caught and handled appropriately. The same applies for other memory allocating functions such as `calloc` and `realloc`.

For this reason, `SCpubTypes.h` defines the `CKNULL` macro in lines 127–129:

```
1 #define CKNULL(_p) if(IsNull(_p)) {\
2 err = kSCLError_OutOfMemory; \
3 goto done; }
```

More often than not, this check is omitted after a memory allocation.[6] This can lead to the software crashing, but I am not aware of any way this can compromise the security of the application.

Fix

The fix is simple: search for all memory allocations in the code and add the `CKNULL` macro where necessary.

4.4.4 Checking error codes

Something similar happens with the `CKERR` macro, which should be called after every SCimp function that might return an error (which includes almost every function).

[6] I was not the only one to notice: see `https://github.com/SilentCircle/silent-text/pull/2`, which catches (only) a few of the lines with this problem.

```
1 #define CKERR  if((err != kSCLError_NoErr)) {\
2 STATUS_LOG("ERROR %d  %s:%d \n",  err, __FILE__, __LINE__); \
3 goto done; }
```

The structure of this macro implies that every call to such a possibly failing function should store the result in a variable named `err`, and should define a flag at the end of the function named `done`. One could argue that this is a fragile structure, but it appears to work throughout the code. Except of course, in the cases where this call to the macro `CKERR` is forgotten, so that the error is not caught and the code continues to run. The consequences for the security of the implementation is highly dependent on the context. I have found a few omissions of `CKERR`, but I have not been able to escalate these bugs into security exploits.

Fix

Again, the fix is simple: search for the string "`err =`" and add `CKERR` where necessary.

4.5 State machine

As I have explained in Section 4.1, the SCimp implementation defines a state machine that determines what will happen when certain transitions are triggered. The state machine is defined in `SCIMP_state_table` and displayed in diagram 4.2. In this section, I will explain the state machine, indicate some problems that it has and look at a few ways how the implementation bypasses the state machine all together.

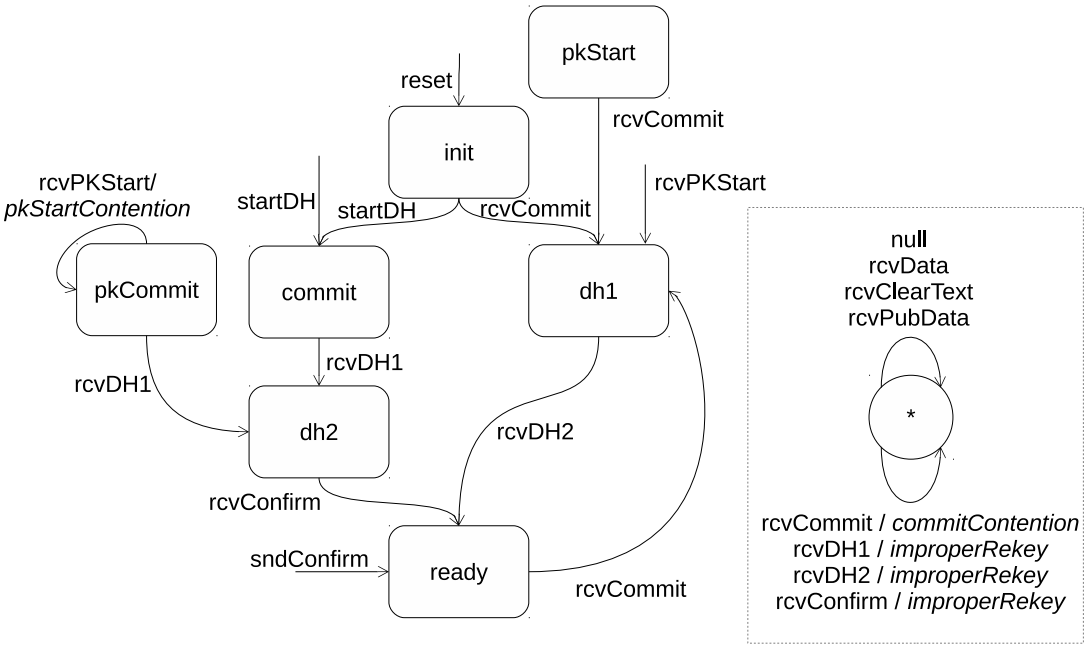

Figure 4.2: SCimp state diagram, as defined by `SCIMP_state_table`

In the diagram, the rectangular blocks define the states, while labeled arrows define the transitions between states. The label defines both the transition that is triggered and the function that handles the transition. For example, the *rcvPKStart/pkStartContention* label says that when the device is in state *pkCommit* and receives a *rcvPKStart* transition, the *pkStartContention* handles the processing of the PKStart message, after which the state will be *pkCommit*. A one-word label means the transition and function are roughly named the same. For example, *rcvDH2* is triggered by transition `kSCimpTrans_RCV_DH2` and is handled by `sDoRcv_DH2`.

Arrows that do not start at a state represent events that can be handled from any state, except for states that already handle that transition. For example, the *rcvPKStart* transition goes to the *dh1* state, except when the current state is *pkCommit*.

Finally, the part on the bottom right defines transitions that can be triggered from any state and that do not alter the state.

4.5.1 State machine bugs

There are a few things wrong with this state machine. First of all, SCimp defines a state *pkInit* that does not even occur in the table. It is strange that there are three states for Progressive Encryption, even when there is only one message that differs from keying in SCimp version 1. According to the code comments, the states represent the following phases in the protocol:

pkInit *pkStart* info ready, waiting for first send

pkStart *pkStart* sent, waiting for DH1

pkCommit *pkStart* was received, sent DH1

The *pkStartContention* should happen when the device has just sent the PKStart message and then receives a PKStart message, so when it is in state *pkStart*. Upon inspection of the state machine, it turns out that contention on PKStart happens in the *pkCommit* state. In the *pkStart* state, the state machine is showing that it expects a Commit message. It appears as if the *pkCommit* state should have been named *pkStart*, *pkStart* should have been *pkInit* and *pkInit* should not exist.

What is remarkable is that no state transition leads to either the *pkStart* or *pkCommit*. It turns out that the state machine is bypassed, see section 4.5.2.

A second issue is that the *sndConfirm* transition seems redundant, because the *rcvDH2* handler should just handle anything that needs to happen when the confirm message is sent. The code comment at the *sndConfirm* handler suggests that this is a pseudo state, used to inform the user of keying when it is ready. I think this could and should have been done in the *rcvDH2* handler.

The *commitContention* handler is called when the *rcvCommit* transition is triggered in any state other than *init*, *PKStart* or *ready*. Real commit contention can only happen when the sender is in the *commit* state. This indicates that the *commitContention* handler does not actually handle commit contention. It actually resets the `SCimpContext` and thereby bypasses the state machine.

A minor style issue is that the *startDH* transition from the *init* state is redundant, because the *startDH* transition will have the same result from any state.

4.5.2 State machine bypass

The described bugs might suggest that this state machine does not work at all. The fact that the application can actually send messages suggests otherwise. The reason is simple, the state machine is ignored often and many functions are called directly, which should only be called from a state machine handler. In other places in the code, the `state` variable is even modified directly.

It already begins with creating a new `SCimpContext`. The `SCimpNew` function resets the context, including setting the `state` to the value *init*. After resetting, the function `scEventTransition` is called, which tricks the event handler into thinking that an event transition has taken place. There is no reason not to use the *reset* transition of the state machine.

Approximately the same problem occurs with the handlers *commitContention*, *improperRekey* and *pkStartContention*. By the definition of the state machine, we expect these handlers not to alter the state. The *commitContention* handler first resets the `SCimpContext`, effectively setting the `state` to *init*, then goes on to call the *rcvCommit* handler (bypassing the state machine) and finally sets the `state` variable to *dh1*. The *pkStartContention* handler follows the same pattern.

The *improperRekey* handler has its own confusing implementation. First, it triggers the warning *protocolError* (apparently not an error) and then goes on to reset the `SCimpContext`.

The irony is that all this bypassing of the state machine means that the state machine must set `NO_NEW_STATE` for the next state, otherwise it would undo everything that was bypassed by the handler.

4.5.3 Fix

The state machine appears to be a good idea, taken from the OTR documentation. However, it should be the foundation of the message handling implementation and not implemented as an afterthought, as is in SCimp. The result is that the state machine more often gets in the way of the programmer than that it helps him to keep the code clean. Bypassing the state machine can then feel more natural for the programmer, even if it results in hacks such as the *sndConfirm* transition.

Unfortunately, there is no easy fix for this bug, as it would require changes to the entire structure underlying the code itself. Many functions will have to be rewritten. It might be easier to simply strip out the state machine completely and directly interact with the handling functions. From a security perspective, this easier fix is not ideal, since a well implemented state machine makes it much easier to analyze the code.

4.6 Style issues

In this section, I will look at some issues that affect the code, but which are not necessarily bugs. I argue that these style issues will make it very difficult to understand and thus maintain the code. These style issues could lead to someone making a change in the code that results in a bug, because they misunderstood someone. Or someone interacting with the code might have some incorrect assumptions about the function they are using and will trigger some unexpected or undefined behavior. In the long term, these style issues could lead to application failure or even security vulnerabilities.

4.6.1 MAC length in the serializer

When inspecting the calls to the CCM decryption function, something stands out:

```
1  err = CCM_Decrypt_Mem(        scSCimpCipherAlgorithm(ctx->cipherSuite),
2                        key,    scSCimpCipherBits(ctx->cipherSuite)/4,
3                        msgIndex,       sizeof(msgIndex),
4                        m->msg,         m->msgLen,
5                        m->tag,         sizeof(m->tag),
6                        &PT,            &PTLen); CKERR;
```

Listing 4.4: SCimp: call to `CCM_Decrypt_Mem`

The length of the authentication tag is specified by the length of the tag received in the message. In other words: the sender of the message—or an attacker—can set this length to any value they want! Actually, an attacker can *not* set the length. The problem is that any length other than 8 bytes will not be parsed correctly by the message deserializer. Libscimp will protest and return a `kSCLError_Corrupt` error. I think it is unwise to let the serializer be responsible for any security features. Instead, the `tagSize` parameter should be a constant value.

4.6.2 Warning/error handlers

One of the responsibilities of the event handler is to handle warnings and errors. I think that errors are an important part of how the protocol works and they should be handled correctly by the protocol implementation itself. The only function that I could find that actually did something with the error (instead of just aborting and passing the function along), is the function `sProcessTransition`, although it does not update the state when an error has occurred.

Much more sophisticated error handling is required. For example, when an attacker injects out of order key negotiation messages, the current approach is just to reset the context. This could include the *cs*, which makes a denial of service attack trivial. A simple error handling strategy would be to reset the `SCimpContext` without deleting the *cs*. Only then (if still necessary), should the error be sent to the event handler.

Sometimes the errors do not get passed on to the error handler, but the error simply gets ignored. For example, the function `sDoStartDH` never assigns to the `err` variable. Both contention handlers never check the error of the handlers they call.

4.6.3 Superfluous computations

Some values are computed even when they are not necessary. Removing these computations could speed up the application.

hcs in first key negotiation

In the first key negotiation, it is not necessary to compute the value hcs_a. According to the documentation, the value is still computed, so that an eavesdropper cannot distinguish the first key negotiation from rekeying. However, in the first key negotiation, the value *cs* is substituted with 0, making the value hcs_a public. Now it is trivial for an eavesdropper to distinguish a first key negotiation from rekeying, by computing hcs_a with $cs = 0$ and comparing it with the value in the Commit message. Generating a random value (as is documented but not implemented) could introduce a timing attack, unless a random value was generated for every rekeying as well, which decreases performance. If the PKStart message is sent, the fact that it is the first key negotiation is given away immediately. A solution is to leave out the computations of the *hcs* values in the first key negotiation.

One other thing becomes noticable in the implementation. The initiator always checks the value of hcs_b, by using $cs = 0$ when it is the first key negotiation. The responder only verifies hcs_a when rekeying, but does the comparison anyway. If a timing attack (on distinguishing the first key negotiation from rekeying) is no problem, than these checks can be left out.

mac_a computed twice

The initiator computes mac_a twice, once before sending the DH2 message and once when receiving the Confirm message. The second computation is redundant and can be eliminated.

Message index

The message index makes it possible to handle out-of-order messages. It is derived from kdk_2, using the KDF. In Section 3.2.3, I have shown that the protocol is secure when this value is made public. This indicates that the value might as well be initialized at zero, speeding up the implementation.

4.6.4 KDF vs MAC

Code can be eliminated where multiple calls to the MAC functions (init, update and final) can be replaced by a single cal to the KDF (`sComputeKDF`). This happens for the computation of kdk_2, *cs*, $k_{snd,j}$ and $k_{rcv,j}$ (for $j > 0$).

The prototype of the KDF function itself could be simplified as well. The L parameter is passed twice to `sComputeKDF`: once as the `hashLen` (in bits) and once as the `outLen` (in bytes). One would expect the equation $\lceil \texttt{hashLen}/8 \rceil = \texttt{outLen}$ to always be true, but in reality it is more often false (see also Section 4.6.5). The reason that this does not break the application is that both sender and receiver of messages use the same erroneous implementation to derive keys.

4.6.5 Inconsistencies

The following code fragment illustrates a problem with length parameters that propagates throughout the entire code-base:

```
1 unsigned long        ctxStrLen = 0;
2 size_t             kdkLen;
3 int                  keyLen = scSCimpCipherBits(ctx->cipherSuite);
```

Listing 4.5: SCimpProtocol.c::sComputeKdk2 (lines 1136–1139)

The values `ctxStrLen` and `kdkLen` represent byte lengths, while `keyLen` represents a length in bits. The types do not match and are not computed consistently. For example, `ctxStrLen` is computed as a `size_t` but (implicitly) converted to an unsigned long. The length of key k_{snd} is actually $2 * \texttt{keyLen}$, so to convert to bytes, one needs to divide by four. This confusing design has the consequence that the computation of the KDF (equation 2.8) is often done wrong, with the value of L not actually matching the hash length. This might also have to do with the fact that the KDF function is sometimes computed with the function `sComputeKDF`, while other times it is bypassed and the MAC functions `MAC_init`, `MAC_update` and `MAC_final` are used directly.

The LibTomCrypt function `ccm_memory` has the strange property that it silently downgrades the length of the authentication tag to the largest valid value of 16 bytes. The wrapper functions, that are used for SCimp (see `SCccm.c`), always pass the value 32. To add to the confusion, these wrappers distinguish a value `tagLen` and `tagSize`, with the following cryptic comment in the decryption wrapper:

```
1     // This will only compare as many bytes of the tag as you specify in tagSize
2     // we need to be careful with CCM to not leak key information, an easy way to
          do
3     // that is to only export half the hash.
4
5     if((memcmp(T,tag,tagSize) != 0))
6         RETERR(kSCLError_CorruptData);
```

I could not find any literature about CCM mode leaking key information. Nor is there a hash in sight to export. The value tagSize turns out to be 8 bytes in both encryption and decryption, which is only half as big as the tag that is actually computed in both encryption and decryption. According to the NIST specification [13]: "Larger values of *Tlen* provide greater authentication assurance [...]. The performance tradeoff is that larger values of *Tlen* require more bandwidth/storage". An overhead of 8 bytes on a message that is encoded in XML seems negligible to me. Even if it is not, the correct parameter of 8 bytes should be passed to the function `ccm_memory`—and not 32 bytes silently lowered to 16.

The length variables are not the only thing that make the code analysis confusing. Many inconsistencies exist between names as well. A pointer to a `SCimpContext` is called a `SCimpContextRef`, while a pointer to a `SCimpMsg` is called a `SCimpMsgPtr`. To add to the confusion, it seems an arbitrary choice whether to use this pointer type or to use the C pointer syntax (`*`). All the SCimp version 1 code calls the `SCimpContext` parameter that gets passed are called `ctx`, while code that was introduced with version 2 refers to it by the name `scimp`.

None of these issues are necessarily bugs, but they make understanding and (I can only guess) maintaining the code more difficult than necessary. It is almost inevitable that in the long term, this code base will lead to application bugs. It might already have been the reason for past bugs.

4.6.6 Code comment/documentation

The code has little documentation. The existing documentation in the form of the whitepaper [36] is outdated and inconsistent, both with itself and with the actual code of either version. The messaging ecosystem document [22] is a very good high-level description of SCimp and the context in which it runs, but lacks the details necessary for analysis of the protocol.

As an example: SCimp version 2 implemented a sync mode for several functions, without any explanation. It turns out to have nothing to do with the protocol itself, but as Vinnie Moscaritolo

explained to me: “sync mode is something we added to the API recently to allow android apps to work a bit better. The android environment couldn’t handle callbacks through the JNI very well, so we made a mode that worked without callbacks.” In my opinion, this is something that needs to be documented.

Another approach is to write the code in such a clear and understandable way that comments are no longer necessary. This is what TextSecure does[7]: “Each class should include a class-level Javadoc comment above the class declaration, describing what the overall purpose of the class is. This should be the **only** comment included in the code.”

[7] `https://github.com/WhisperSystems/RedPhone/wiki/Code-style-Guidelines`

Chapter 5

Comparison to other IM protocols

The design for the SCimp protocol took many ideas from then existing protocols and combined them in a protocol that was suited for mobile messaging. Now that SCimp has been discontinued and replaced with Silent Phone[1], the underlying secure messaging protocol has been changed as well to that of the TextSecure messaging application.

The SCimp designers have drawn many ideas from:

1. ZRTP [38][6]
2. Off the Record (OTR) [4][1]
3. Secure Short Messaging Service (SSMS) [3]
4. Cryptocat [19]

One can recognize the influences of each of these protocols in the design and implementation of SCimp, but most of all from Secure SMS. SCimp is basically an implementation of SSMS with some minor alterations. In its turn, SSMS draws many ideas from OTR, with the main difference that SSMS and SCimp are optimized for a mobile environment. This chapter is structured chronologically. It will first compare SCimp to the protocols that preceded it and then to the protocol that replaced it. In Section 5.1, I compare SCimp to the OTR protocol. In Section 5.2, I enlist the differences between SCimp and SSMS. Finally, in Section 5.3, I compare SCimp with the TextSecure protocol.

5.1 Off the Record

Off the Record messaging (OTR) was introduced in 2004 [4] as an alternative to PGP. The authors compared the privacy that is provided by the then commonly used systems such as PGP, with the privacy that is expected from real-world social communication. In a real-world private conversation—that was not compromised in the first place with, for example, a hidden recorder—, there should be no way to compromise the conversation later. Furthermore, nobody is able to obtain any evidence about what Alice has said, other than what Bob claims she said. OTR identified this as two problems with a conversation using PGP: a key compromise would leak messages from the past and message signatures would not allow their repudiation.

Their solution was the OTR instant messaging protocol. In the protocol, both Alice and Bob have a long term signing key-pair, of which they publish the public key. They initiate communication with a non-authenticated, signed and ephemeral Diffie-Hellman key exchange to derive the initial key material. They can then mutually authenticate over the unauthenticated channel using a shared secret. Once they have set up the channel, they can exchange messages.

[1] `https://web.archive.org/web/20150923095110/https://support.silentcircle.com/customer/en/portal/articles/2118889-what-protocols-are-used-for-voice-and-text-communication-in-the-new-app-`

Each message is encrypted with *malleable encryption* and authenticated with a MAC, to ensure deniability. To make sure the key gets updated regularly, the users send a fresh random DH public key along with each message, which is used to generate new key material, at which point the old keys can be erased.

In their 2004 paper, the authors of OTR already identified one problem with this protocol in an asynchronous communication environment: it requires a full Diffie-Hellman key exchange before the first message can actually be sent. Several other properties of the protocol make it less suitable for secure mobile communication. On the other hand, OTR is still widely used, it is a secure protocol and is the foundation of later protocols such as SCimp and TextSecure. Therefore, I will provide a short description of the protocol in subsections 5.1.1 and 5.1.2. In subsection 5.1.3, I will compare OTR to SCimp.

5.1.1 Authenticated Key Exchange

The initial authenticated key exchange (AKE) in OTR has changed between versions of the protocol. The reason is that the first version—signed Diffie-Hellman key exchange—was subject to the *identity misbinding attack* [11], as pointed out in [10]. To defeat the identity misbinding attack, the AKE was changed to a SIGMA variant [20]. In this section, I will describe the SIGMA variant of the AKE in OTR.

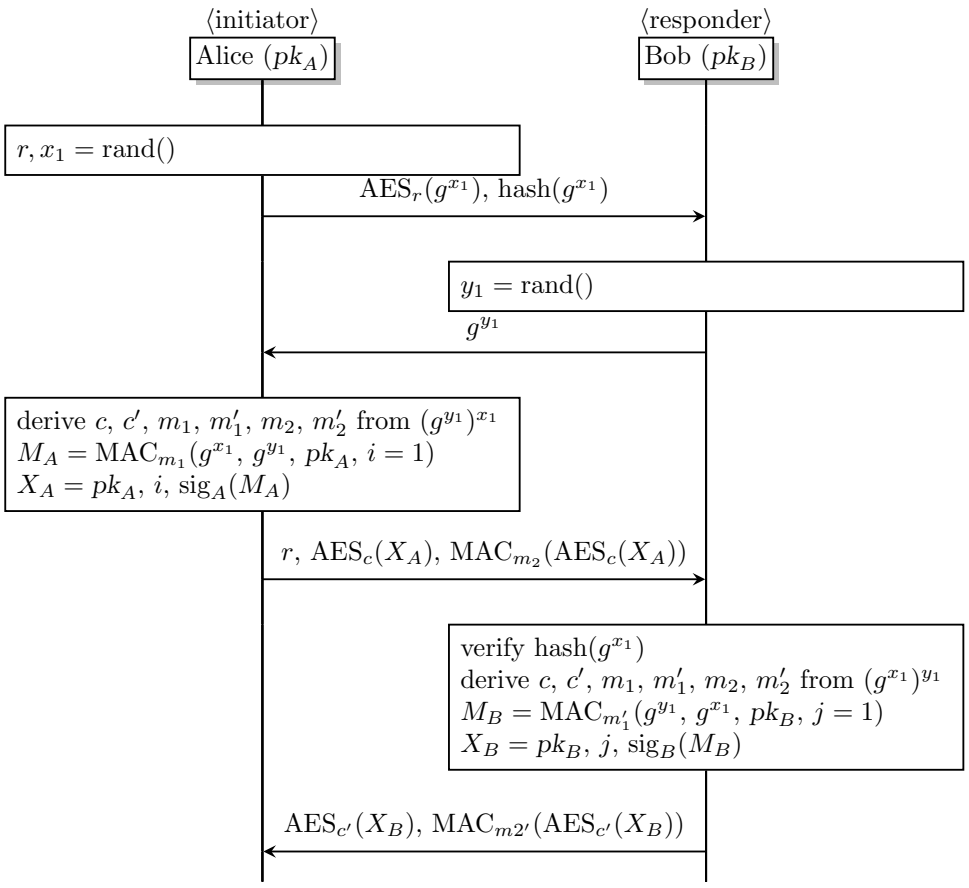

Figure 5.1: Off the Record: Authenticated Key Exchange

In Figure 5.1, AES encryption is done with 128-bit AES in Counter mode, using 0 as the initial counter. Hashes are performed by SHA256, MACs are computed with SHA256-HMAC, sometimes

truncated to fewer bits. The generator g is a fixed value and exponentiations are done modulo a fixed 1536-bit prime.

The first message is a hash commitment by Alice to her ephemeral public key g^{x_1}. The encryption with r is there because otherwise the third message would grow to large for some IM protocols. It also avoids message amplification atacks. The commitment to g^{x_1} is given by $\text{hash}(g^{x_1})$.

In the second message, Bob sends his ephemeral key g^{y_1}. Upon receipt, Alice can use it to complete the ephemeral Diffie-Hellman by computing the shared secret $(g^{y_1})^{x_1}$, from which she derives more key material.

In the third message, Alice reveals g^{x_1} by sending r. She also proves knowledge of the shared secret, by computing a MAC (M_A) over the ephemeral keys, her long-term public key and her ephemeral key id ($i = 1$), using the shared secret. To prove knowledge of her long-term secret key sk_A, she provides a signature on M_A. She sends her long-term public key, her key id and the signature to Bob, encrypted under a key derived from the shared secret so that the long-term key is not leaked to an eavesdropper. Finally, Alice sends a MAC to authenticate the ciphertext.

Upon receipt, Bob opens and verifies the commitment to g^{x_1}, from which he also computes the shared secret $(g^{x_1})^{y_1}$. He validates the values Alice sent and creates the fourth message, which authenticates Bob to Alice.

User authentication

The practical drawback of this protocol is the underlying assumption that Alice and Bob know each others public keys beforehand, so that they can verify the values in X_A and X_B. If they do not know each others keys beforehand, Eve can perform an undetected man-in-the-middle attack by injecting her own key instead. Until Alice has verified that the public key of the other party indeed belongs to the Bob, the channel is said to be unauthenticated. The same goes for Bob and Alice's public key.

The first version of OTR solved this problem by asking the users to verify the fingerprint of the public keys out-of-band, much like SCimp does with the SAS, but with a long fingerprint instead of a user friendly short code. To improve user experience, later OTR versions offer an alternative to compare a secret that is only known by the honest participants. This secret is mixed with a session id and the public keys of Alice and Bob and compared using a zero-knowledge method: the socialist millionaires protocol (SMP) [16][5]. Because the protocol reveals only if the compared values are equal or not, the protocol can be executed over the unauthenticated channel.

Key erasure

The AKE protocol allows users to erase the ephemeral DH keys x_1 and y_1 as soon as they have derived the shared secret. All transferred messages consist of ephemeral key values or values that are encrypted and authenticated with ephemeral keys.

Deniability

The AKE should not leak the metadata that the conversation between Alice and Bob ever took place. In the third message, Alice signed (a MAC of) the ephemeral keys and her own public key. The ephemeral key in the second message could have come from anybody, so this signature only reveals that Alice has initiated in an AKE, but not with whom. Bob might reveal y_1 as an attempt to prove that she contacted him, but Alice will quickly point out that Bob could have gotten that value from anyone that she initiated the AKE with and that was willing to leak their value. The same applies to Bob, where his signature reveals that he responded in an AKE, but not with whom.

5.1.2 Exchanging data

Once the channel has been set up with the AKE, the participants can exchange data, as shown in Figure 5.2.

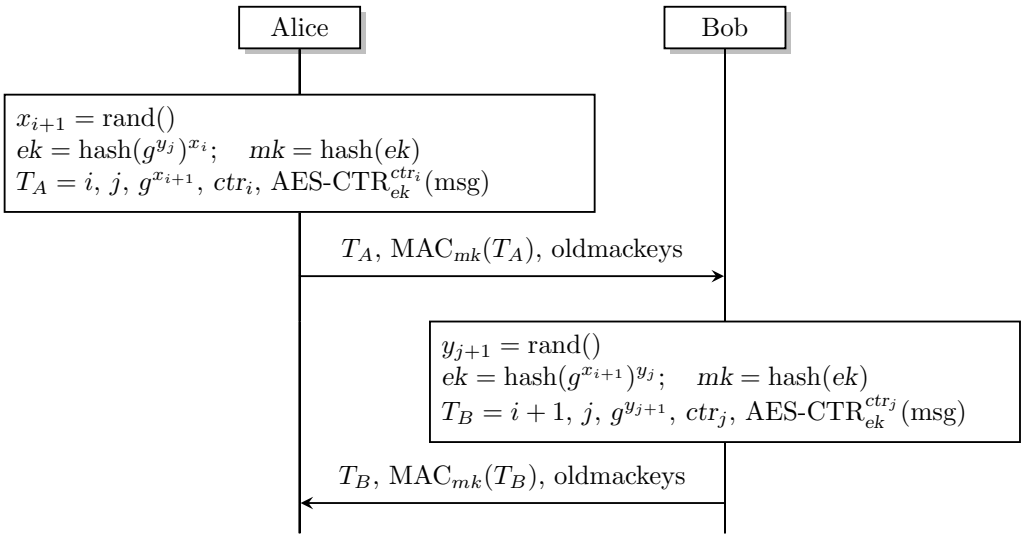

Figure 5.2: Off the Record: Data exchange

In order to encrypt and authenticate a message, Alice derives an encryption key (ek) from the shared secret, a MAC key (mk) from ek and increments the counter (ctr) for use as the lower 16 bytes used in counter mode. The encrypted value is embedded in T_A, which also contains the ephemeral key indices that were used to compute the shared secret and a fresh ephemeral public key. T_A is sent together with a MAC that authenticates it.

Upon receiving the message, Bob inspects the key indices (i and j), to derive the shared secret that is required for verifying and decrypting the message. To encrypt a new message, he uses the last keys that Alice confirmed ($g^{x_{i+1}}$ and g^{y_j}) to compute a new shared secret, from which he derives the required ek and mk. The counter ctr_j can be reset to 1.

Bob needs to keep a lot of keys in memory in order to be able to decrypt all messages, including the ones that arrive out of order. He needs to hold on to $ek_{i,j}$ for decrypting more new messages, to $ek_{i-1,j-1}$ for decrypting old messages, to $ek_{i+1,j}$ for encrypting messages and to y_{j+1} for completing the next DH exchange.

Deniability

Besides the confidentiality of the messages, OTR provides deniable integrity of messages, by computing $\text{MAC}_{mk}(T_A)$. The value mk is known by both Alice and Bob. Since both of them could have created the message, neither can provide a proof that the other one sent that message. But, in an honest conversation, Bob knows that he did not generate the MAC, which means Alice must have generated it. This proves the integrity of the message to Bob.

OTR goes one step further, by publishing the old MAC keys and using malleable encryption. When Eve eavesdrops on a conversation and can guess the plaintext, she can alter the ciphertext to reflect a change in the plaintext. Later on, when she eavesdropped the corresponding MAC key, she can also forge a MAC on the changed ciphertext. Assume that Alice is confronted with a full transcript of the conversation. Without the published oldmackeys, she could deny it by claiming that Bob could have made it up. With the public MAC keys, she can claim that anybody could have made it up.

It is not entirely clear when old MAC keys can be published. In [4], the authors claim that Alice "can be sure that all messages authenticated with $[ek_{i-1,j-1}]$ have been received" when she receives a message encrypted with $ek_{i+1,j+1}$. However, when Alice has sent multiple messages signed with $mk_{i-1,j-1}$, there is no reason to assume that all messages have arrived. She only knows that Bob has discarded all key material before i, *because it has been implemented that way.* The protocol does not make any guarantees, so this has simply been a design decision.

5.1.3 Comparison to SCimp

OTR and SCimp aim to achieve the same goals, but in a different environment. OTR was designed for instant messaging protocols, which usually run on personal computers that are connected during the entire conversation. SCimp was designed to work in a mobile environment, using limited computing power and devices that are not online most of the time. I will first enlist the similarities:

- Both protocols use the same SIGMA structure (with commitment to g^x) for the key exchange. This means that both protocols need to complete the DH exchange before a message can be sent.
- Both protocols use a MAC and malleable encryption to ensure deniability.
- Both OTR and SCimp have found a way to let the users authenticate to each other without having to educate them on fingerprints. SCimp uses the SAS, whereas OTR has an option to use the SMP.
- Neither OTR nor SCimp has a clear idea of when old keys can be erased. Messages that arrive too much out-of-order are simply discarded.

One can argue that the PKStart message refutes the first point for SCimp. However, messages sent with PKStart cannot be proven secure until a full DH exchange has taken place and the SAS was confirmed.

There are also important differences between the protocols.

- SCimp uses only ephemeral keys. In theory, one can transfer authenticated OTR public keys to a different device and the values can still be trusted. With SCimp, users only authenticate a session, which is not transferable to other devices.
- OTR rolls forward the key with additional randomness in every reply message. In SCimp, there does not exist a well-defined point in time for when renegotiating the keys must be done.
- OTR does not update the key when sending multiple messages without receiving a reply. SCimp updates the key with every sent message.
- OTR uses regular Diffie-Hellman over a 1536-bit prime Field. SCimp uses ECDH, resulting in shorter messages and requiring less computational power.
- SCimp uses AES in CCM mode, which uses the same key for encryption and authentication. The MAC keys cannot be published, because they would reveal the plaintext.
- SCimp (version 2) can immediately send a user message, whereas OTR needs four keying messages.
- SCimp (version 2) can send group messages.

SCimp is slightly better suited for mobile than OTR, because it uses elliptic curves, reducing the size of keys and messages and requiring less computational power. OTR does not rely on the group underlying the Diffie-Hellman exchange, so it should be possible to build a version of OTR that is based on ECC. Except for ECC, SCimp version 1 has no real advantage over OTR. Version 2 of the protocol has some usability improvements for mobile environments over OTR, but they come at the price of lowered security, as show in Chapter 2.

5.2 Secure SMS

Secure SMS was created by Gary Belvin as his master project for the John Hopkins University [3]. Its aim is to provide a more efficient communications protocol, suited for transport over SMS messages. SCimp version 1 is simply an implementation of this protocol, but sending the messages over XMPP instead of SMS. For example, many equations in the SCimp documentation [36] have been taken directly from this paper.[2] For the reader who wants to understand SCimp, I recommend reading Gary Belvin's well written thesis before looking at the Silent Circle documentation.

Because SCimp and SSMS are so similar, I will only list the differences between the protocols:

- SSMS was designed for sending over SMS, SCimp for XMPP
- The design of SSMS provides a recommended rekey frequency (10-100 messages) and a lower bound (every 2^{40} messages) for security. SCimp provides no information on the rekey frequency.
- SSMS options allow to specify cryptographic primitives individually, which SCimp simplifies by supporting only a few combinations, defined by the cipher suites.
- SSMS uses AES in EAX mode, SCimp uses CCM.
- SSMS derives MAC keys for computing mac_a and mac_b, while SCimp simply compares the derived keys.

Notably, Gary's master thesis describes future work for a real-world implementation, containing four issues:

1. Locally stored data need to be protected.
2. A secure life cycle model should be deployed to allow bug patching.
3. The protocol needs a way to distinguish devices.
4. The protocol can run out of sync when one device updates cs when sending the Confirm message, but the other party does not receive it.

Although Silent Circle has provided a solution for the first two issues, the last two issues remain a problem, even in SCimp version 2.

5.3 TextSecure

In September 2015[3], Silent Circle announced that they will no longer support SCimp and that they switched to a different protocol. That new protocol is TextSecure. The protocol aims for the same security goals as SCimp, but has some advantages. The new protocol is integrated in the Silent Phone application. The protocol is originally implemented in the Signal application (formerly TextSecure on Android), by Open Whisper Systems.

The TextSecure is currently at version 3, so I will analyze that version in this section. There is no documentation for this version, so my analysis is based upon the code found in [28] and [27].

TextSecure no longer supports messaging via SMS.[4] This means that the initial key exchange is always performed using PreKey messages, which are somewhat similar to the PKStart message in SCimp [25]. For a more detailed analysis, see [14]. Note that it analyzes TextSecure version 2, so their analysis might contain some discrepancies with my description.

[2] Sometimes the Silent Circle documentation even copied the equation numbering and it contains equations that do not apply to SCimp.

[3] `https://web.archive.org/web/20151020135735/https://support.silentcircle.com/customer/en/portal/articles/2118859-can-the-new-app-talk-to-the-standalone-silent-text-app-`

[4] `https://web.archive.org/web/20150308141628/https://whispersystems.org/blog/goodbye-encrypted-sms/`

5.3.1 Key negotiation: Triple(?) Diffie-Hellman

Initial key negotiation in TextSecure version 2 was done using a triple Diffie-Hellman key exchange (3DH), using both the long-term key and an ephemeral key of both users. Version 3 of the protocol replaces the ephemeral key of Bob with a signed non-ephemeral key and optionally adds an ephemeral key, so technically it should be called a Quadruple-but-sometimes-Triple Diffie-Hellman key exchange. I argue in this section that this decision—to make the ephemeral key optional—has weakened the protocol.

Figure 5.3 shows version 3 of the initial key "negotiation".[5] Alice and Bob have a long-term public key pk_A and pk_B that identifies them. In addition, Bob has a long-term PreKey y_B that he signed with the private part of his long-term private key (sk_B).

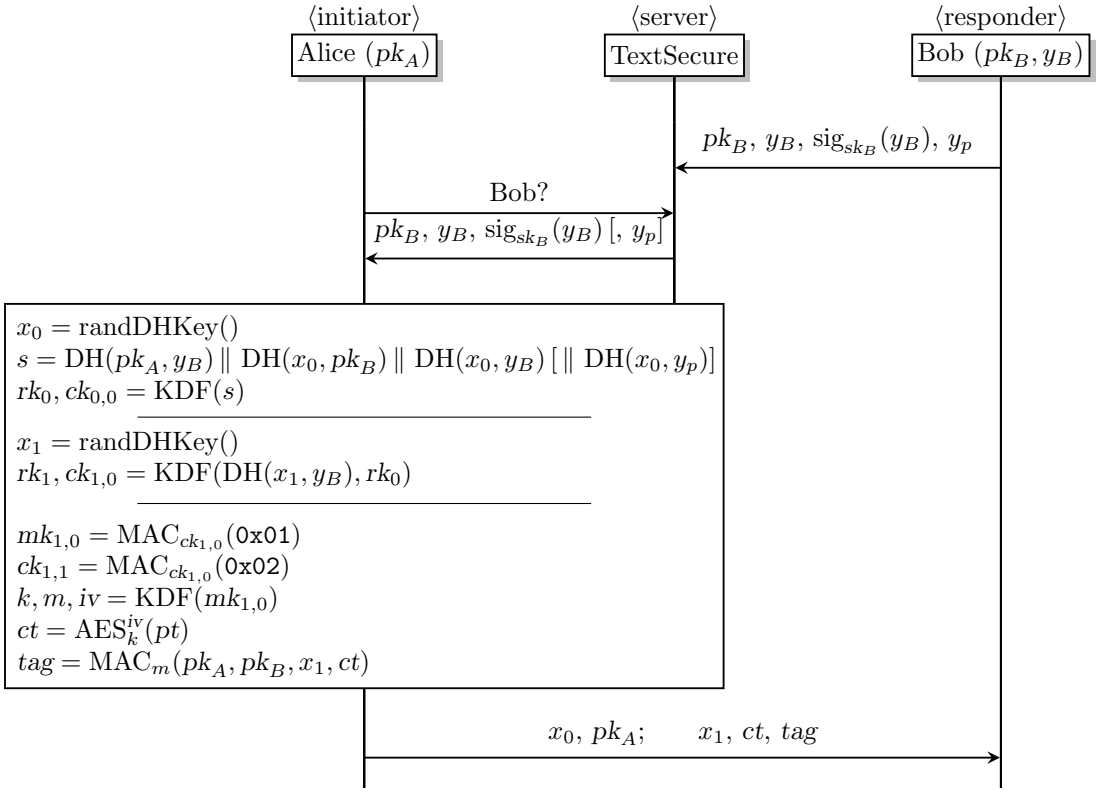

Figure 5.3: TextSecure version 3

When Bob wants to communicate using TextSecure, he first needs to register with the TextSecure server. He generates and sends his long-term public key pk_B, a signed public DH key y_B and 100 one-time ephemeral keys y_p (also called PreKeys). When Alice wants to talk to Bob, she first queries the server for his key material. The server returns his long-term key material and a one-time PreKey y_p, which is then removed from the server. The one-time PreKey is optional. For example, it might be omitted when more than 100 requests have been issued to the server before Bob was able to upload fresh PreKeys.

Alice will now have to derive many keys in order to initialize the Axolotl ratchet. She starts by generating her own random ephemeral DH key x_0 (x_0 represents the public part of her DH keypair, but of course she knows the secret part as well). She derives a shared secret s by concatenating four Diffie-Hellman key exchanges—or three when no one-time key was provided—which is then

[5] The horizontal lines in Alice computations and space in the last message of the figure are explained in Section 5.3.2.

used to derive the initial root key (rk_0) and chain key ($ck_{0,0}$). She generates another ephemeral DH key (x_1) and computes its DH shared secret with y_p. From that shared secret and rk_0 she derives the next root key rk_1 and the next chain key $ck_{1,0}$. From $ck_{1,0}$, she derives a message key $mk_{1,0}$ and the next chain key ($ck_{1,1}$). From $mk_{1,0}$ she can derive the key material for encryption (done with 256 bit AES in CBC-mode) and MAC-authentication (HMAC-SHA256) of the message. She now sends everything to Bob that he needs for verification and decryption of the message: her long-term public key, her 2 public ephemeral keys, the ciphertext and the authentication tag.

Figure 5.3 actually shows a slightly simplified version of the initial key negotiation. Keys y_B and y_p have an identifier that Alice also needs to send to Bob, so that he can find their corresponding private keys. Messages also contain counters for the number of messages sent per key (both initialized at 0), see also Section 5.3.2.

Bob uploads the public key of 100 ephemeral PreKeys y_p to the server, ready to be downloaded by other clients, including Eve. This gives Eve two advantages in retrieving the corresponding private key in TextSecure, compared to regular 3DH exchange with ephemeral keys. First of all, Bob needs to hold on to the private part of the PreKeys until it they are used in a key negotiation. Effectively, Bob always has the private keys corresponding to *some* ephemeral keys on his device, which might at any time be compromised by Eve. A second advantage to Eve is that she can download the PreKeys from the server and start an offline brute-force attack, giving her more time to complete the search. In TextSecure version 3, the situation is worse because Bob might only use non-ephemeral keys, which have a longer window in which they can be compromised.

Identity misbinding attack

In TextSecure, you can verify the identity of the other party by verifying the fingerprint of their public key. This verification needs to happen out-of-band, but can happen before the actual TextSecure communication takes place. For example, a user might print his fingerprint on their business-card. A user study on OTR [32] has shown that the method of verifying fingerprints out of band provides usability problems for some end users. But one should remember that it is always *possible* to verify the identity of your communication partner.

Even if the users take precaution and verify the public key, the initial key negotiation remains subject to an identity misbinding attack. The attack works when Eve pretends to own Bob's long term key and provides this to Alice. She can let Alice think she is talking to her, while she is actually talking to Bob. The details of the attack can be found in [14].

No ephemeral key from Bob

Normal 3DH key exchange uses one long-term key and one ephemeral key of both users. In [26], the rational for this design is presented: no signatures are required (which would complicate the implementation and enlarge the message size), the ephemeral part of the handshake is still there to enable key erasure and the deniability of the overall protocol is improved. This last point is achieved because Eve is able to take Bob's public key, her own DH key and generate an ephemeral key for Bob. With this she can simulate the entire key negotiation and thus an entire conversation. For this simulation, she does not need to have had any conversation with Bob before. Bob can thus deny having had a conversation with Eve or even with anyone else. Also note, that it is no longer required to publish the old MAC keys and that this protocol no longer relies on the notion of malleable encryption in order to ensure deniability.

However, these advantages disappear with version 3 of the key negotiation, in which Alice can start a conversation with Bob without using Bob's ephemeral one-time key, but using his signed PreKey instead. First of all, the implementation will have to implement signatures again, making it more complex and more likely to contain implementation bugs. Second and more important, when Alice (or the server) does not include y_p, none of the three DH exchanges is fully ephemeral. When Bob's device gets compromised, he might leak the secrets of both his long-term key-pair and his signed prekey, which compromises the first message.

Eve can also prove that Alice has contacted Bob from the fact that one of the DH exchanges contains two long-term keys. First, she records the initial message from Alice to Bob and presents this first recording as evidence to a judge. The judge then asks (or forces) Bob to provide his device in order to learn the private keys corresponding to pk_B and y_B, from which she can derive the necessary key material for the first key message. Because the judge is able to verify the message tag, she is convinced that the derived key material is correct. Eve could not have simulated that message, even if it included a one-time prekey y_p.

Wildcard attack

The wildcard attack on 3DH was first described by Tarr in [34], in the context of a capability system, which is a generalization of an authentication system. For such a system to be well-behaved it should (at least) protect against a wildcard attack, which is an attack that could happen in case of a key compromise.

The wildcard attack can best be explained by an example. Assume that Alice and Bob want to derive a shared key by doing a single (non-ephemeral) DH key exchange: Alice sends public key pk_A (derived from sk_A) to Bob, who replies with pk_B. Alice learns the shared secret by computing $\mathrm{DH}(sk_A, pk_B)$, Bob by computing $\mathrm{DH}(sk_B, pk_A)$. But now assume that Bob gets compromised and leaks sk_B to Eve. Eve can then talk to Alice while pretending to be Bob, by computing $\mathrm{DH}(sk_B pk_A)$. However, she can also talk to Bob and pretend to be Alice by computing $\mathrm{DH}(sk_B, pk_A)$. In fact, when she talks to Bob, she can pretend to be anybody of whom she knows the public key. We say that non-ephemeral DH is subject to a *wildcard attack*, where sk_B acts as a wildcard, because it allows the attacker to impersonate anybody.

Triple DH is also subject to a wildcard attack, but with the private key sk_b corresponding to the ephemeral public DH key pk_b as the wildcard. Assume that Eve wants to impersonate Alice in a conversation with Bob. She generates her own ephemeral DH key-pair (sk_e, pk_e) and engages in the 3DH key negotiation, using the long term public key of Alice pk_A, of which she does not know the corresponding private key. She sends the pair pk_e, pk_A to Bob and he replies with the pair pk_b, pk_B. She can complete the 3DH computation if she is able to derive sk_b from pk_b: $\mathrm{DH}(sk_b, pk_A) \parallel \mathrm{DH}(sk_e, pk_B) \parallel \mathrm{DH}(sk_e, pk_b)$. Symmetrically, Alice's private ephemeral key acts as a wildcard.

It should not be easy to compromise an ephemeral key. According to the discrete log assumption, it should be infeasible to derive sk_b from pk_b, but this assumes that sk_b has enough entropy. This might not be the case, for example, when it was generated with a weak PRNG: the key is then vulnerable to a brute-force attack. Another strategy to compromise sk_b is to extract it from the devices memory. Ephemeral keys are usually more difficult to compromise, because they live in memory for a shorter time and thus have a smaller window in time in which they might be stolen from the device. As stated before, it might be slightly easier to compromise ephemeral keys in the TextSecure implementation, because the private keys can be downloaded from the server.

5.3.2 Sending data: Axolotl ratchet

Just as in OTR and SCimp, the keys used for encryption and authentication need to be updated often to be able to erase old keys. For this reason, TextSecure uses the Axolotl ratchet, for which a more detailed description can be found in [24] and [29]. In Axolotl, just as in OTR, Alice adds a fresh public DH key to her encrypted and authenticated message, but only when Bob just sent her a fresh public DH key—in other words, only when she *replies*. When she does not reply (she sends an additional message), just as in SCimp, she derives a new key from old key material to encrypt her next message.

One of the elegant parts of the TextSecure design, is that the ratchet is already embedded in the first message. It is already shown in Figure 5.3. Alice needs to complete three phases to send her first message. In the first phase she sets up the initial root key rk_0 from the initial 3DH exchange. In the second phase she derives a new root key rk_i and chain key $ck_{i,0}$ from the previous root key rk_{i-1} and a Diffie-Hellman exchange between a fresh key x_i and the latest key she received from

Bob y_i. In the third phase she derives a new chain key $ck_{i,j}$ and message key $mk_{i,j-1}$ from the previous chain key $ck_{i,j-1}$. When Alice replies to a message from Bob, she executes phases two and three. When she sends additional messages before she gets a reply, she only executes phase three. The message format does not change much either: the left half of the message in Figure 5.3 is only sent in the first message. All subsequent messages take the shape of the right half.

One advantage over OTR is that the fresh random DH keys can be used immediately to derive the key material for the current message, whereas in OTR they had to be confirmed by the other party first. This is achieved by mixing in the old keys in the derivation of the new keys. The consequence is that less keys need to be stored in memory.

Maximum key erasure

One clear advantage of Axolotl over both OTR and SCimp is that a device compromise at any point minimizes the amount of key material that is leaked. Axolotl achieves this by distinguishing chain keys from message keys and by sending two counters with every message: the total message count c_p for the previous root key and a message counter c for the current root key.

Alice's perspective of the Axolotl ratchet is displayed in Figure 5.4, in which time flows down on the vertical axis. It depicts the situation where: Alice initializes and sends two messages; Bob replies; Bob sends two more messages, but only the second arrives and only after Alice has sent her reply. The arrows between the keys show how keys are derived from each other, while the dashed arrows show the messages that are being sent. These contain the counters c_p and c, ephemeral keys x and y and the message (encrypted and with authentication tag).

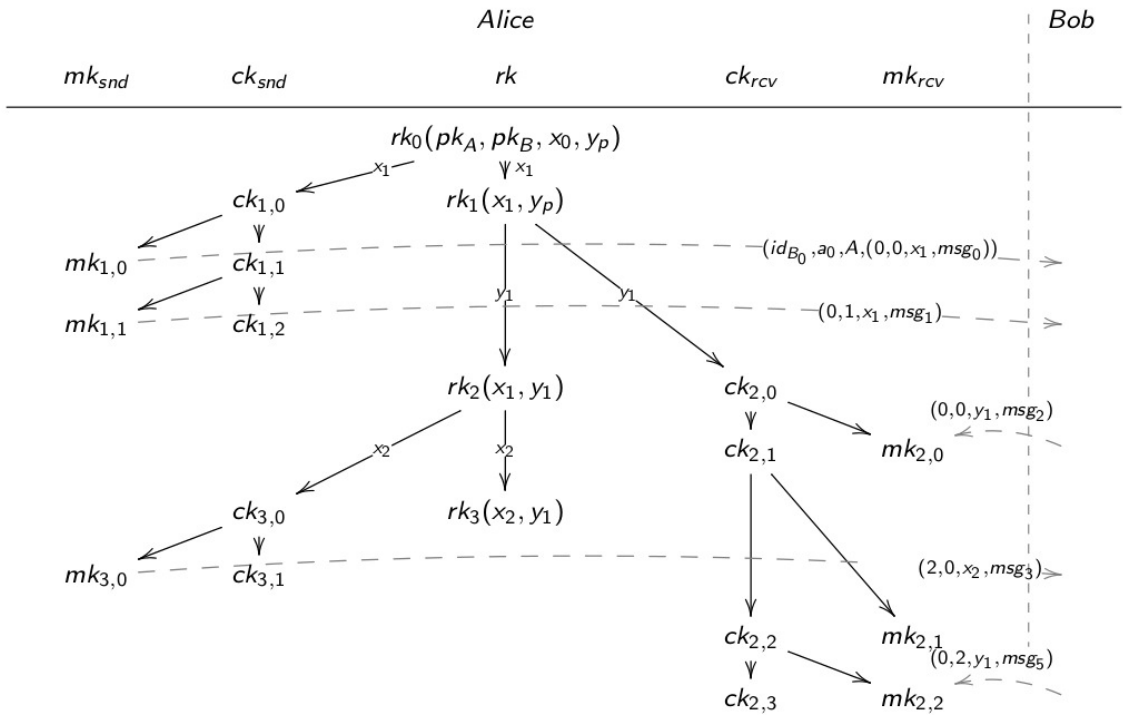

Figure 5.4: Axolotl key management

At any time, Alice needs to know only the latest root key rk, only her latest ephemeral secret key x_i and only the latest key in both the sending chain ck_s and in the receiving chain ck_r. When Bob sends a new ephemeral key y_i, she checks the value c_p that he sent to see if she missed any messages, for which she will then derive the required message keys. These keys will not be updated and need to stay in memory for a longer time, but that is not a problem, because no other keys can

be derived from them. She can discard a value *mk* after she verified and decrypted the message, or after a certain time when the message is not expected to be delivered anymore.

By sending along how many messages were sent, Axolotl is able to precisely determine which keys need to be remembered in order to decrypt messages that might arrive out of order. By keeping these keys separated from the chain keys, a compromise of the key reveals one message at most. Axolotl derives new root keys from the old ones, so that the derived key material can be used directly. These mechanisms ensure that the maximum number of keys can be erased and that a device compromise results in minimal loss of confidentiality.

5.3.3 Comparison to SCimp

TextSecure aims for the same goals as SCimp, but has different means of arriving at those goals. There are similarities...

- Both SCimp and TextSecure can forward the keys before a reply is sent by the other, unlike OTR.
- Both SCimp and TextSecure use ECC.

...and there are differences:

- TextSecure is more precise when it comes to key erasure.
- TextSecure uses public keys, while SCimp is completely ephemeral. In TextSecure, the user needs to confirm the other parties. This allows for an identity misbinding attack, depending on how Alice learns Bob's public key.
- A TextSecure user can validate the other parties identity before sending the first message, SCimp can only ensure identity after the SAS is confirmed.
- TextSecures initial message kicks off the Axolotl ratchet immediately and is immediately secure. After the PKStart message in SCimp, three key negotiation messages are required before any security can be guaranteed.
- The TextSecure code is cleaner.

I admit that the last point is very subjective, but I think that it needs to be mentioned. While I have dedicated an entire chapter in this thesis to the implementation bugs in SCimp, I have not found a single bug in the TextSecure protocol implementation.

The precision of the Axolotl ratchet and its ability to be secure from the start on make TextSecure a more suitable protocol for a secure mobile communication protocol than SCimp.

The protocol is no silver bullet. The wildcard attack could be a problem in some situations and should be looked into. TextSecure could benefit from implementing the SMP protocol to verify the identity of the other party, which mitigated the identity misbinding attack in OTR. Finally, the change in the protocol to version 2 appears a step back from version 2, because Bob cannot erase the keys used in key negotiation and it removes some deniability of the conversation.

Chapter 6

Conclusions

The Silent Circle instant messaging protocol is a security protocol that ensures confidential and authenticated communication over the XMPP protocol. It provides deniable end-to-end encryption for which the end-users do not need to trust any intermediary party. Users set up a secure communications channel with a completely ephemeral key negotiation protocol. To verify the identity of the communication partner on this channel, the users verify a short authentication string on another authenticated channel. By updating the key material, via rekeying and with the SCimp ratchet, old encryption keys can be erased, so that compromises of the current key material do not compromise the secrecy of old messages.

The first version of SCimp had a few problems that are solved in the second version. That version can send user data with the first SCimp message, so that the communication can start immediately. The solution has the problem that that no guarantees about the security of those messages can be made until the SAS is confirmed much later. A man-in-the-middle can stretch the period in which authentication is not possible to an arbitrarily long time.

The second version of SCimp also introduces group messages, but requires trust in the server to distribute the encryption keys to the group. Only group conversations that have been set up with a sufficiently strong password that was communicated out-of-bounds can be considered confidential, authenticated and deniable. SCimp does not allow key erasure or provide future secrecy for group conversations.

The second version of SCimp introduced a way to communicate files via the Silent Circle cloud. Because it uses convergent encryption and no proper handling of the salt, this is vulnerable to file confirmation attacks and learn-the-remaining-information attacks. Because the implementation does not validate that the key used to decrypt actually belongs to the file that was decrypted, an attacker can inject arbitrary files. Checking if the used key was correct can prevent the file injection attack.

SCimp does not protect against an identity misbinding attack where Eve tricks both Alice and Bob by letting them talk to each other while they believe they are each talking to Eve.

SCimp does not provide a user-friendly method for handling multiple devices that share the same XMPP address.

The formal model that I built in Proverif proves that many security claims by Silent Circle on SCimp are indeed true. It should be noted that Proverif is somewhat limited in its power to model the more subtle security requirements of SCimp, which leaves open the possibility that SCimp is vulnerable to an attack that is not detected by Proverif.

The actual implementation of SCimp has several bugs, some of which are significant security vulnerabilities. I consider the impact of these bugs to be less than the earlier described problems with the SCimp design, for the simple reason that bugs can be patched.

A comparison of SCimp with contemporary security protocols shows that SCimp (version 1) is simply an implementation of Secure SMS. SCimp does some things different than Off-the-Record, but it is not immediately clear that these differences lead to an improvement for security. Version 2 of SCimp has some clear improvements for usability, but can be considered a step back for

security.

TextSecure, the protocol that replaces SCimp, can immediately send user messages, of which the secrecy and integrity can be verified immediately. The Axolotl ratchet allows for more precise key erasure and more precise rekeying. The TextSecure protocol does have some problems, because it is still vulnerable to an identity misbinding attack and a wildcard attack. The latest TextSecure version loses some deniability by including a non-ephemeral Diffie-Hellman exchange in the initial handshake. However, because the impact of these attacks is much lower than the above attacks on SCimp (and because TextSecure has much cleaner code), I consider the TextSecure protocol an improvement over SCimp.

Bibliography

[1] Chris Alexander and Ian Goldberg. Improved user authentication in off-the-record messaging. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, WPES '07, pages 41–47, New York, NY, USA, 2007. ACM. 58

[2] Apple Inc. Local and Remote Notification Programming Guide, October 2015. `https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/RemoteNotificationsPG.pdf` Accessed: 2015-26-10. 16

[3] Gary Belvin. A Secure Text Messaging Protocol. Master's thesis, John Hopkins University, May 2011. `https://eprint.iacr.org/2014/036`. 58, 63

[4] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-Record Communication, or, Why Not to Use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM. 58, 62

[5] Fabrice Boudot, Berry Schoenmakers, and Jacques Traoré. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 111(1–2):23 – 36, 2001. Coding and Cryptology. 60

[6] Riccardo Bresciani. The ZRTP Protocol, Analysis on the Diffie-Hellman Mode. Technical report, Trinity College Dublin, June 2009. `https://www.cs.tcd.ie/publications/tech-reports/reports.09/TCD-CS-2009-13.pdf` Accessed: 2015-10-27. 58

[7] Lily Chen. NIST Special Publication 800-108: Recommendation for Key Derivation Using Pseudorandom Functions (revised). Technical report, NIST, October 2009. 9

[8] Daniel J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/papers.html#cachetiming` Accessed: 2015-12-02, April 2005. 49

[9] Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to`. Accessed: 2015-25-10. 8

[10] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Secure Off-the-record Messaging. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, pages 81–89, New York, NY, USA, 2005. ACM. 59

[11] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992. 26, 59

[12] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, March 1983. 2, 28

[13] Morris Dworkin. NIST Special Publication 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. Technical report, NIST, May 2004. 15, 47, 56

[14] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How Secure is TextSecure? Cryptology ePrint Archive, Report 2014/904, November 2014. http://eprint.iacr.org/2014/904. 63, 65

[15] Google. Cloud Messaging, August 2015. https://developers.google.com/cloud-messaging/ Accessed: 2015-26-10. 16

[16] Markus Jakobsson and Moti Yung. Proving without knowing: On oblivious, agnostic and blindfolded provers. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 186–200. Springer Berlin Heidelberg, 1996. 60

[17] Joe Hildebrandt and Peter Saint-Andre. XEP-0033: Extended Stanza Addressing. Technical report, XMPP Standards Foundation, September 2004. https://xmpp.org/extensions/xep-0033.html Accessed: 2015-10-26. 20

[18] Jon Callas. Building Hardware We Are Proud Of, November 2015. https://web.archive.org/web/20151024104841/http://hardwear.io/wp-content/uploads/2015/10/Building-Hardware-We-Are-Proud-Of-by-Jon-Callas.pdf Accessed: 2015-10-24. 45

[19] Nadim Kobeissi. Cryptocat. https://crypto.cat Accessed: 2015-12-02. 58

[20] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols. In *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science* (Dan Boneh, Series Editor), pages 400–425. Springer Berlin Heidelberg, 2003. 59

[21] Vinnie Moscaritolo. Silent Circle Instant Messaging Protocol - libscimp API guide, November 2012. https://github.com/SilentCircle/silent-text/blob/master/Documentation/ Accessed: 2015-10-21 (commit bee6f4955252995e07b761ecd40bf68e64d809f1). 46

[22] Vinnie Moscaritolo. Silent Circle Messaging Ecosystem, September 2014. https://github.com/SilentCircle/silent-text/blob/master/Documentation/ Accessed: 2015-10-21 (commit bee6f4955252995e07b761ecd40bf68e64d809f1). 22, 23, 24, 56

[23] Vinnie Moscaritolo. Silent Text 2.0: The next generation of private messaging, May 2014. https://web.archive.org/web/20150506152939/https://blog.silentcircle.com/silent-text-2-0-the-next-generation-of-private-messaging/ Accessed: 2014-05-22. 1, 4

[24] Moxie Marlinspike. Advanced cryptographic ratcheting, November 2013. https://web.archive.org/web/20150930055656/https://whispersystems.org/blog/advanced-ratcheting/ Accessed: 2015-09-30. 66

[25] Moxie Marlinspike. Forward Secrecy for Asynchronous Messages, August 2013. https://web.archive.org/web/20150906204213/https://whispersystems.org/blog/asynchronous-security/ Accessed: 2015-09-06. 63

[26] Moxie Marlinspike. Simplifying OTR deniability, July 2013. https://web.archive.org/web/20151011185006/https://whispersystems.org/blog/simplifying-otr-deniability/ Accessed: 2015-10-11. 65

[27] Open Whisper Systems. Axolotl Library for Android, September 2015. https://github.com/WhisperSystems/libaxolotl-java Accessed: 2015-10-24 (commit 01bc1eb37be2113f78392df4bed93ff173aee98e). 63

[28] Open Whisper Systems. TextSecure, October 2015. https://github.com/WhisperSystems/TextSecure/ Accessed: 2015-10-24 (commit 474922493f0020f182d8497ba5ea447f6eb79f0e). 63

[29] Trevor Perrin. Axolotl Ratchet, July 2015. https://github.com/trevp/axolotl/wiki Accessed: 2015-10-21 (commit 6fa4a516b01327d736df1f52014d8b561a18189a). 66

[30] Sebastian R. Verschoor. SCimp Proverif models, December 2015. https://github.com/sebastianv89/scimp-proverif Accessed: 2015-12-02 (commit d0b56f0c6e7b6ff2e0a10cdd3f76fb79abb7c000). 28

[31] Silent Circle. Encrypted text messaging, August 2015. https://github.com/SilentCircle/silent-text Accessed: 2015-10-24 (commit bee6f4955252995e07b761ecd40bf68e64d809f1). 4

[32] Ryan Stedman, Kayo Yoshida, and Ian Goldberg. A User Study of Off-the-record Messaging. In *Proceedings of the 4th Symposium on Usable Privacy and Security*, SOUPS '08, pages 95–104, New York, NY, USA, 2008. ACM. 65

[33] Steffen Jaeckel. libtom/libtomcrypt, September 2015. https://github.com/libtom/libtomcrypt Accessed: 2015-10-27 (commit 16f397d55c9f4971a66a7ce9d87d0305ab45eaa7). 47

[34] Dominic Tarr. Designing a Secret Handshake: Authenticated Key Exchange as a Capability System, July 2015. https://web.archive.org/web/20150630160251/https://dominictarr.github.io/secret-handshake-paper/shs.pdf Accessed: 2015-06-30. 66

[35] Tom St Denis. LibTom. http://libtom.net Accessed: 2015-10-27. 47

[36] Vinnie Moscaritolo, Gary Belvin and Phil Zimmermann. Silent Circle Instant Messaging Protocol - Protocol Specification, December 2012. https://github.com/SilentCircle/silent-text/tree/master/Documentation Accessed: 2015-10-21 (commit bee6f4955252995e07b761ecd40bf68e64d809f1). 11, 56, 63

[37] Doug Whiting, Russell Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). RFC 3610, RFC Editor, September 2003. 14

[38] Philip Zimmerman, Alan Johnston, and Jon Callas. ZRTP: Media Path Key Agreement for Unicast Secure RTP. RFC 6189, RFC Editor, April 2011. 8, 58

[39] Zooko Wilcox-O'Hearn. Attacks on Convergent Encryption. Technical report, Tahoe-LAFS, March 2008. https://web.archive.org/web/20151027101054/https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html Accessed: 2015-10-27. 25