EINDHOVEN UNIVERSITY OF TECHNOLOGY

(In-)Secure messaging with the Silent Circle instant messaging protocol

Citation for published version (APA):
Verschoor, S. R., & Lange, T. (2016). *(In-)Secure messaging with the Silent Circle instant messaging protocol*. (Cryptology ePrint Archive; Vol. 2016/703). IACR. http://eprint.iacr.org/2016/703

Document status and date:
Published: 01/01/2016

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

(In-)Secure messaging with the Silent Circle instant messaging protocol

Sebastian R. Verschoor
David R. Chariton School of Computer Science and Institute for Quantum Computing, University of Waterloo
srverschoor@uwaterloo.ca

Tanja Lange
Department of Mathematics and Computer Science, Technische Universiteit Eindhoven
tanja@hyperelliptic.org

ABSTRACT

Silent Text, the instant messaging application by the company Silent Circle, provides its users with 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 this paper, we analyze both the design and implementation of SCimp by inspection of the documentation (to the extent it exists) and code. Many of the security properties of SCimp version 1 are found to be secure, however many of the extensions contain vulnerabilities and the implementation contains bugs that affect the overall security.

These problems were fed back to the SCimp maintainers and some bugs were fixed in the code base. In September 2015, Silent Circle replaced SCimp with a new protocol based on the Signal Protocol.

Keywords

SCimp, Silent Circle, Instant Messaging Protocol

1. INTRODUCTION

Blackphone by Silent Circle is an Android-based smart phone running the Silent OS operating system, with current versions (Blackphone 2) selling at a hefty 799 USD. The phone promises its users higher security and privacy: "Blackphone puts privacy first". The Blackphone 1, costing more than 600 USD, was rolled out in summer 2014 and came with service subscriptions for encrypted phone (Silent Phone) and chat (Silent Text). Silent Circle's "Products and Solutions" page [25] advertised Silent Text as

> Share unlimited encrypted texts on any Silent OS, iOS or Android device. Use the burn feature to set messages to automatically self-destruct after a set time period. Silent Text also offers secure file transfers of up to 100MB and voice memo functionality.

Silent Text was available for free on Apple's App store (since 2012) and on Google Play (since 2013) but users needed to buy a subscription for using the servers, with prices starting at 9.95 USD per month. Given these prices one could expect some security analysis of the services, but only after an intense online discussion, started by Zooko Wilcox-O'Hearn's open letter to Phil Zimmermann and Jon Callas [31], did Silent Circle post source code for Silent Text for iOS on GitHub [27] allowing inspection of their code base. We are not aware of any public analysis of SCimp.

1.1 History of SCimp

Silent Text uses XMPP for transportation and adds a crypto layer to generate shared keys between communicating parties. Messages are passed (and temporarily stored if necessary) by Silent Circle's server. Instead of deploying OTR, the Off-The-Record protocol [6] by Borisov, Goldberg and Brewer, Silent Circle started from SecureSMS, introduced in Belvin's masters' thesis [3], and adapted it to the XMPP setting. The resulting protocol by Moscaritolo, Belvin, and Zimmermann from 2012 is called SCimp: "Silent Circle instant messaging protocol". SCimp is described on Silent Circle's web page [24] and in a white paper [21]; some source code and explanations are available on GitHub. In 2014 Silent Text 2.0 was rolled out [20] "to improve the security and refine the user experience of our customers". This update changes how parties provide keys, most importantly it allows Alice to start a communication with an offline Bob, and added functionalities for group messaging and encrypted file storage.

1.2 Advertised security properties

SCimp is designed to provide **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 rekeying 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** through key continuity: 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 advertised as the "self-healing property".

1.3 Results

SCimp version 2 was in use since May 2014 [20], but the details have only been released in August 2015 [27], and then only upon our request. Our paper shows that the update introduced new vulnerabilities, most remarkably that a man-in-the middle attacker Eve can arbitrarily delay authentication between Alice and Bob and thus remain undetected. Eve can also easily interrupt communications and force a

new key exchange, which she then can man-in-the-middle. This interruption also destroys the advertised self-healing property of key continuity (version 1 and 2).

Due to the insufficient level of documentation (or close to none in the case of SCimp version 2), code review was necessary, though tedious. The implementation of SCimp provided on GitHub had several implementation errors, including cryptographic errors. The SCimp version 2 implementation defines a state machine that leaves the protocol vulnerable to an undetectable man-in-the-middle attack. In addition, the way the state machine is often bypassed in the code leaves the implementation vulnerable to another man-in-the-middle attack.

1.4 Disclosure and updates

Throughout this study we have communicated with Jon Callas and Vinnie Moscaritolo from Silent Circle. In September 2015, Silent Circle announced [26] that they will stop Silent Text on September 28 and change to a messaging application integrated with an updated Silent Phone app which would be based on the Signal Protocol [13]. Jon Callas confirmed that we may say that our analysis motivated their decision to switch. The switch takes care of most of our concerns reported in Section 3 and Section 5, and possibly (but not necessarily) Section 4.

2. PROTOCOL DESCRIPTION

This section presents the SCimp protocol and explains the design rationale. We base our analysis on the Silent Circle documentation provided in [21] and [19]. A comparison with the code—showing several inconsistencies—is given in Section 5. The reason for the high level of detail in the protocol description is that the official documentation lacks the required details for proper cryptographic analysis.

SCimp version 1 has two modes of which one, basic Diffie-Hellman (DH) mode, provides encryption. Users do not have certified long-term keys. The protocol uses an ephemeral DH key exchange (explained in Section 2.1) to establish a shared key between two devices. After key negotiation, users need to verify the identity of the other party by confirming a *short authentication string* (SAS) out-of-band. Sessions are not time limited, so for enhanced security, the users periodically need to renegotiate their keys, which is explained in Section 2.2. SCimp uses the derived keys to perform authenticated encryption to provide confidentiality and integrity of the messages, as explained in Section 2.3.

The main goal of SCimp version 2 is 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 introduced "Progressive Encryption" (DHv2 mode). In this mode the Silent Circle server also serves as a key server; users have a medium-term DH key pair of which they upload the public key to the server. When Alice wants to send a message to Bob, she downloads his key and derives key material for encryption of the first message, but she also initiates a parallel key negotiation as in basic DH mode. When that key negotiation completes, both users verify each others identities and thereby confirm that all communication so far has been secure. This mode is explained in Section 2.4. Version 2 also introduces group messages (explained Section 2.5), which require two more SCimp modes: PubKey mode and Symmetric mode.

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". Without loss of generality, we assume that Alice initiates the key negotiation with Bob.

It is not specified how Alice and Bob learn each others 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, we assume that these addresses are not authenticated, meaning that Alice does not know if $B' = B$ and Bob does not know if $A' = A$.

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 1, with some details omitted for brevity.

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 $pk_a = sk_a G$, where G is a fixed base point on the elliptic curve.[1] Alice commits to point pk_a by computing $\text{hash}(pk_a)$, where hash is a hash function.

If Alice and Bob have negotiated keys before, they share a cached secret (cs). They use this value in subsequent key negotiations in order to authenticate to each other. When they do not yet have a shared cs, they substitute cs with a random value and skip the corresponding validations and they authenticate later (see Section 2.1.5).

Alice proves knowledge of cs by sending hcs_a: a Message Authentication Code (MAC) using cs as the key. Alice also sends protocol options (in plaintext), such as the cryptographic cipher suite and the method for verifying the SAS.

2.1.2. DH1. Upon receiving the Commit message, Bob validates the correctness of the received value hcs_a (or ignores it if he shares no cs with Alice) and stores $\text{hash}(pk_a)$ for later validation. If hcs_a is incorrect, the application gives a warning and continues the protocol. He generates his own DH key pair, with the random scalar sk_b and public key $pk_b = sk_b G$. Bob sends his reply (DH1) to Alice, containing the value pk_b and hcs_b.

2.1.3. DH2. After validating hcs_b (or ignoring it in the first key negotiation), Alice now has enough information to complete the DH key exchange. First, she checks if the point

[1]The curve details do not matter for this analysis, version 1 uses NIST P-384.

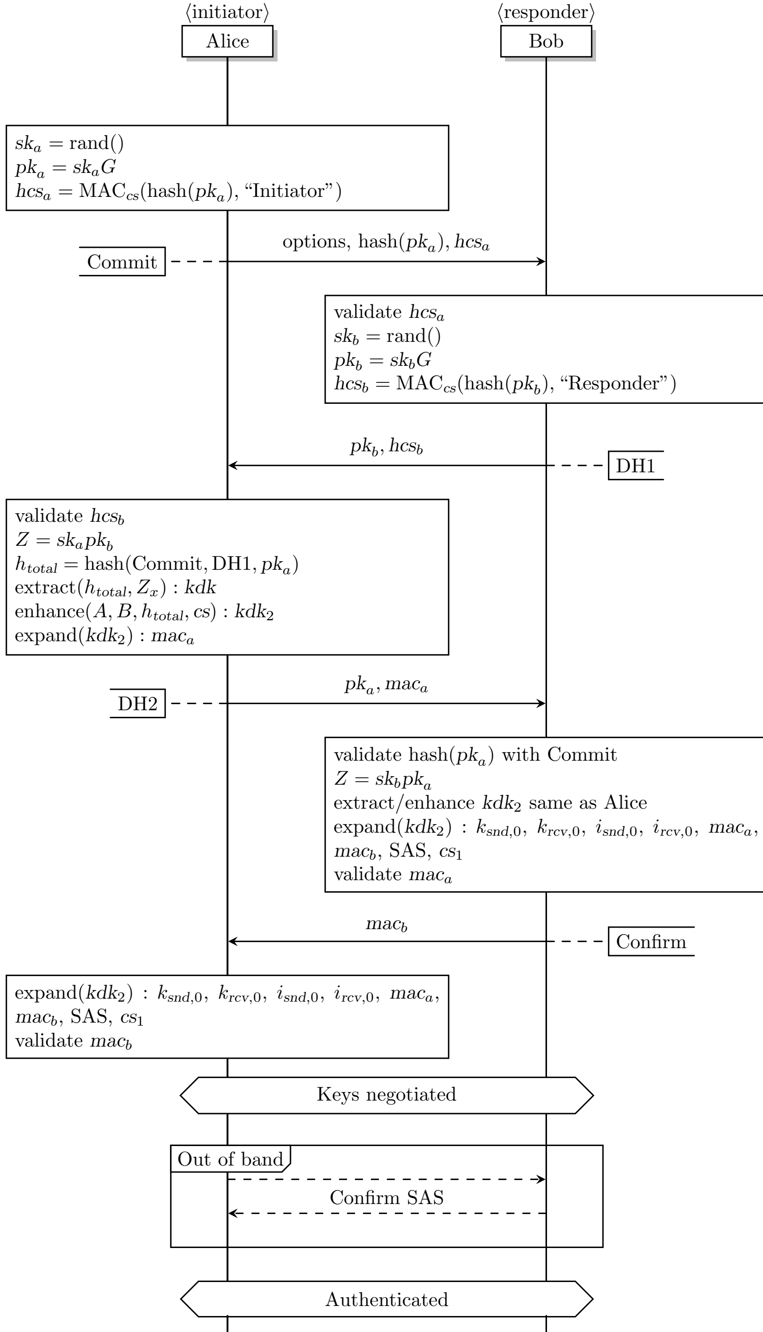

Figure 1: SCimp: key negotiation

she received is a valid point on the curve. Next, she computes shared secret Z_x, which is the x-coordinate of point $Z = sk_a pk_b$ on the elliptic curve.

From this value, all other key material is derived using a three-step process. It is adapted from a standard two-step process for creating a *Key Derivation Function* (KDF): Extract-Expand [9, 17]. An additional Enhance step is added between the steps.

Extract/Enhance/Expand. Keys are derived inside the extract/enhance/expand steps using a MAC function that is labeled KDF.[2] Let $\text{KDF}_k(\text{label}, \text{context}, L) = \text{MAC}_k(\texttt{0x00000001} \,\|\, \text{label} \,\|\, \texttt{0x00} \,\|\, \text{context} \,\|\, L)$, where L 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 SP 800-108 [9]; if more bits would need to be extracted, this counter would be incremented. The KDF sets a label to describe what the key is used for and a context parameter that can be the same for all keys that are derived. In the extract step, a seed h_{total} is computed and a key derivation key is derived: $kdk = \text{MAC}_{h_{total}}(Z_x)$. For unexplained reasons, the protocol computes two context values, a context variable $ctx = \text{len}(A) \,\|\, A \,\|\, \text{len}(B) \,\|\, B \,\|\, h_{total}$ and a session variable $sess = \text{hash}(\text{len}(A) \,\|\, A \,\|\, \text{len}(B) \,\|\, B)$.

The enhance step is just another extract step: it mixes in the cached secret cs, if such a value is shared between the users: $kdk_2 = \text{KDF}_{kdk}(\text{"MasterSecret"}, \text{"SCimp-ENHANCE"} \,\|\, ctx \,\|\, cs, 256)$.

The expand step derives multiple keys from kdk_2, all serving different purposes. This is done by computing the KDF with key kdk_2. Before the DH2 message, only $mac_a = \text{KDF}_{kdk_2}(\text{"InitiatorMACkey"}, ctx, 256)$ is derived (and kdk_2 is stored for later use).

Alice sends pk_a, to open up the commitment, and 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 validates that pk_a matches the commitment $\text{hash}(pk_a)$ and is valid. After this verification, he completes the DH computation $Z = sk_b pk_a$.

By the same extract/enhance/expand steps, Bob derives all the necessary keys. Bob expands kdk_2 into all keys that are required for further communication. The keys $k_{snd,0}$ and $k_{rcv,0}$ are used to send and receive messages (see also Section 2.3). Bob sets $k_{snd,0} = \text{KDF}_{kdk_2}(\text{"ResponderMasterKey"}, ctx, 2l)$, where l is determined by the key size of the symmetric cipher of the cipher suite. Bob uses $k_{snd,0}$ to encrypt and authenticate messages to Alice. Bob sets $k_{rcv,0} = \text{KDF}_{kdk_2}(\text{"InitiatorMasterKey"}, sess, 2l)$ to decrypt and validate messages from Alice. Alice computes the same values, but of course swaps the *snd* and *rcv* labels.

Note the earlier mentioned asymmetry in the derivation of $k_{snd,0}$ and $k_{rcv,0}$: without justification, the former uses ctx and the latter $sess$ as context parameter to the KDF.

To identify which key should be used to decrypt the message, an index is attached to each user message. Bob initializes $i_{rcv,0} = \text{KDF}_{kdk_2}(\text{"InitiatorInitialIndex"}, sess, 64)$ and $i_{snd,0} = \text{KDF}_{kdk_2}(\text{"ResponderInitialIndex"}, sess, 64)$. The indices are necessary when messages are dropped or are arriving out of order.

The cached secret $cs_1 = \text{KDF}_{kdk_2}(\text{"RetainedSecret"}, ctx, 2l)$ is derived to ensure key continuity. The computed value can be used in future key negotiations to authenticate to the other party. In order to prevent a denial of service (DOS) attack, the client replaces the stored value of cs with cs_1 only after the *mac* of the other party is verified.

After key generation, Bob compares the computed value of mac_a with the received value. If the values match, Bob is

[2] The standards suggest that instead of just the expand function, the entire extract/enhance/expand construction constitutes the KDF and it should have been named accordingly. Throughout this paper, we use the Silent Circle notation in order to match the specification.

convinced that Alice was able to complete her side of the 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, consisting of $mac_b = \text{KDF}_{kdk_2}(\text{“ResponderMACkey”}, ctx, 256)$. He can erase all intermediary keys from his device, such as sk_b, Z, kdk and kdk_2.

Upon receiving the Confirm message, Alice expands the rest of the key values from kdk_2 and 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 start messaging. 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 (sent in the Commit message) which determines the length and how the SAS is compared by Alice and Bob.

Alice and Bob both know $\text{SAS} = \text{KDF}_{kdk_2}(\text{“SAS”}, ctx, 20)$ 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 [21] “[a] phone call would be sufficient for this purpose since confidence building cues such as voice timbre and manner of speech are present.”

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. However, without very precise instructions, the users might verify using a subtly different method and not detect that they are the victim of a man-in-the-middle attack.[3]

2.1.6. Reasons for the commit value. The SAS is a short code, typically 20 bits. Therefore, the protocol requires the hash commitment, which forces the adversary to select a public key without knowing the key of the other party. Without commitments, the adversary could acquire the public keys of the honest parties 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. The Commit message ensures that an adversary can have only one blind guess for his public key.

[3]For example, assume that Eve sets up a man-in-the-middle attack on the SCimp connection of Alice and Bob. Further assume that Eve has some control over the voice channel that they use to verify the SAS (not unlikely in case of a phone call). Eve lets Alice and Bob talk until they are convinced they are talking to each other but degrades the quality. She then sends both participants a message over SCimp to speak out the *full SAS*. When Alice says the SAS, Eve (temporarily) mutes the audio for Bob. Eve confirms to Alice that the SAS is correct, which she can do with a message over SCimp, or maybe she has a recording of Bob saying “OK”. She can repeat the trick with Bob. More ways to circumvent SAS authentication are described in [23].

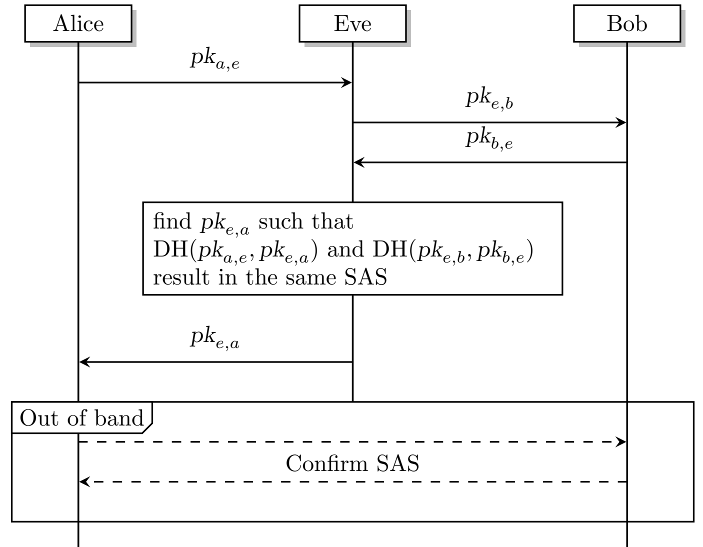

Figure 2: Man-in-the-middle attack on (hypothetical) key negotiation without commit value

2.2 Rekeying

If 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.

If 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. 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.

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

The protocol specification provides only partial information to detect how many (if any) messages have not arrived yet when user messages may arrive out of order. See Section 3.5 for how this is handled in the implementation and how this affects key erasure.

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" requires a new SAS verification.

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-band. For example, when they physically meet, they could 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. We are unaware of such a functionality in Silent Text or Blackphone.

2.2.3. Commit contention. Two participants might try to initiate a key negotiation at the same time. According to the SCimp white paper, 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.

2.3 Sending User messages

When both sender and receiver have derived the keys, they can start sending messages, see Figure 3. Messages are encrypted with AES in CCM-mode (Counter with CBC-MAC) [30]: $ct = \text{AES_CCM}_k^N(header, pt)$ providing 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.

CCM encryption is specified as follows: first compute a CBC-MAC over N, the header and 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.[4]

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

[4]The actual specification is more convoluted, including authentication of lengths and data encoding instructions. See [12] for further details.

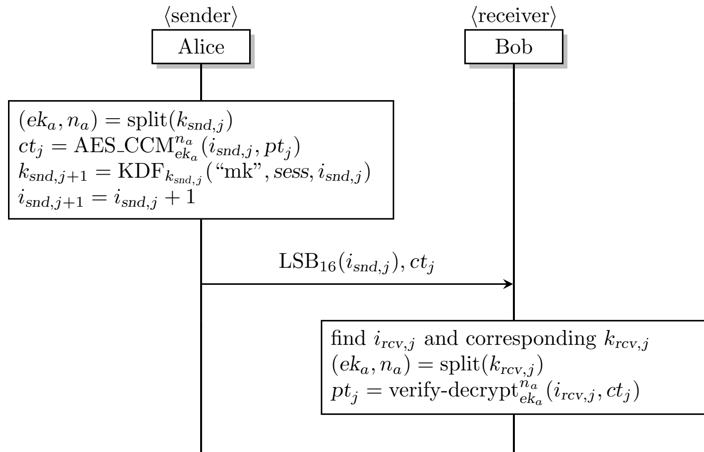

Figure 3: SCimp: data exchange

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.

Each send/receive key is derived from the previous one with the *SCimp ratchet*, which is implemented as follows: $k_{x,j+1} = \text{KDF}_{k_{x,j}}(\text{"MessageKey"}, sess \| i_{x,j}, l)$, where $x \in \{snd, rcv\}$.

The index must be ratcheted forward as well, this is done simply by addition: $i_{x,j+1} = i_{x,j} + 1$.

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. The specification does not state how many old keys to keep in memory. We note that these stored keys not only compromise the secrecy of the messages that have not been received yet, but also that of all subsequent messages, because the keys corresponding to those messages can be derived from the stored keys.

2.4 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) [14] 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.

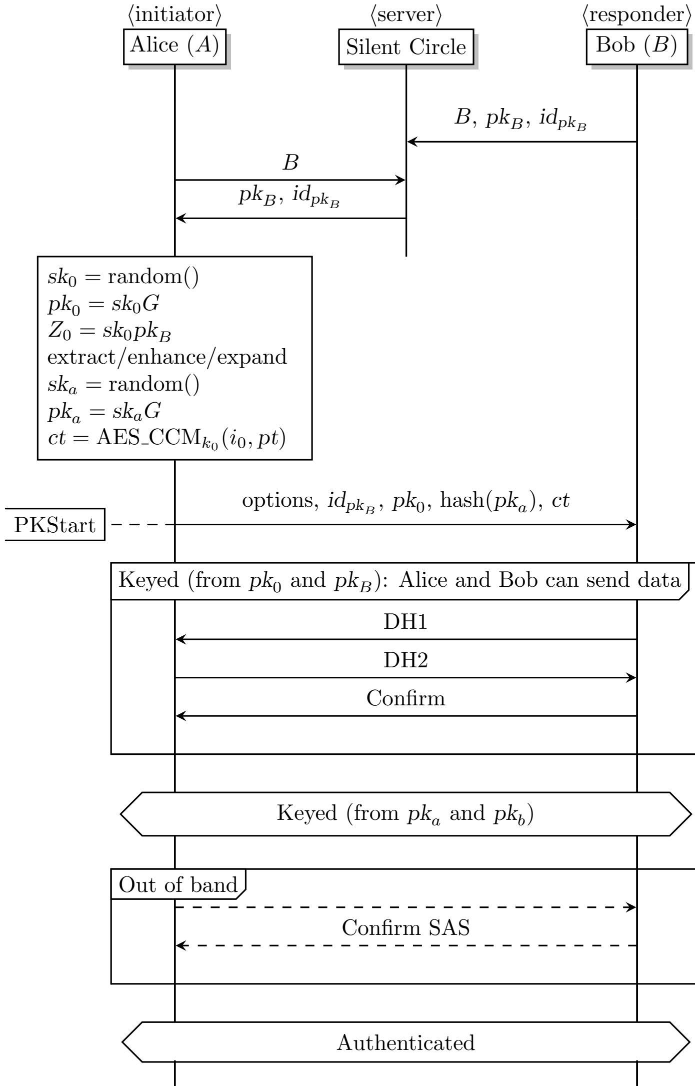

Figure 4: SCimp: Progressive Encryption

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 4. 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 DH 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 pt with the derived symmetric key and can send the ciphertext ct in the first message. In addition, she also generates an ephemeral key pair (sk_a, pk_a) as if composing a Commit message. Alice combines pk_0, ct, and the Commit message in one message, labeled PKStart.

Once he gets online, Bob uses pk_0 and his medium-term secret key sk_B to compute $sk_B pk_0$ and to decrypt the ciphertexts sent so far. Alice and Bob now share key material and can use it to communicate. But, they should also complete the regular key negotiation. Once they have finished that key negotiation, they discard the old key material.

Only when Alice and Bob have derived the new key material can they 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}.

2.4.1. 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 $id_{pk_B} = \text{KDF}_{pk_B}(\text{“SCKey_ECC_Key”}, nonce, 160)$. 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.

2.4.2. PKStart. Alice completes an initial DH 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 key negotiation, with the small alteration that the value of h_{total} is set to zero in $kdk_0 = \text{MAC}_0(Z_{0,x})$ and $ctx = \text{len}(A) \parallel A \parallel \text{len}(B) \parallel B \parallel 0$.

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. 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).

2.4.3. 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 Figure 1, except for the computation of $h_{total} = \text{hash}(id_{pk_B} \parallel pk_0 \parallel \text{hash}(pk_a) \parallel ct \parallel pk_b \parallel hcs_b \parallel pk_a)$, where ct is the ciphertext from 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 $Z_{0,x}$.

2.5 SCimp group conversations

While SCimp version 1 only allowed for one-to-one conversations, SCimp version 2 also enables group conversations by introducing 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.

In order to deliver the multicast messages to all group members, SCimp uses the XMPP extension XEP-0033 [15] in the XMPP layer.

2.5.1. SCimp PubKey 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, see Section 4), 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 *msg*. She sends 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 DH key exchange and for (ElGamal) encryption. She then generates a random session key $k_{session} = \text{rand}()$ which she splits into $(ek, n) = \text{split}(k_{session})$, an encryption key and a nonce. She derives a 64-bit index from Bob's public key $i_{msg} = \text{hash}(pk_B)$ encrypts the message: $ct = \text{AES_CCM}^n_{ek}(i_{msg}, msg)$. Alice sends the key to Bob, asymmetrically encrypted: $esk = \text{EC_encrypt}_{pk_B}(k_{session})$.

The full PubKey message that Alice sends consists of the label "pubkey", the protocol version (always 2), a cipher suite, a locator for the public key used (id_{pk_B}), the encrypted session key (esk) and the encrypted message (ct).

2.5.2. Multicast key. The content of the PubKey message in the context of group messages is the initial 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 $id_{k_{init}}$.

The full Multicast Key message *msg* consists of the label "multicast_key", the protocol version (always 2), a cipher suite, 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}).

2.5.3. SCimp Symmetric mode. Symmetric conversations are set up by deriving a multicast key k from the initial key k_{init} as $k = \text{KDF}_{k_{init}}(\text{"SymmetricMasterKey"}, id_{thread}, 2l)$, $i = \text{KDF}_{k_{init}}(\text{"InitialIndex"}, id_{thread}, 64)$, $i_{offset} = \text{rand}()$, and $\text{SAS} = \text{KDF}_{k_{init}}(\text{"SAS"}, id_{k_{init}}, 64)$.

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. 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: an attacker can just replay it.

To encrypt a message for the group, a participant creates a SCimp Data message. The encryption is similar to that of regular SCimp, 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 group key k is never updated.

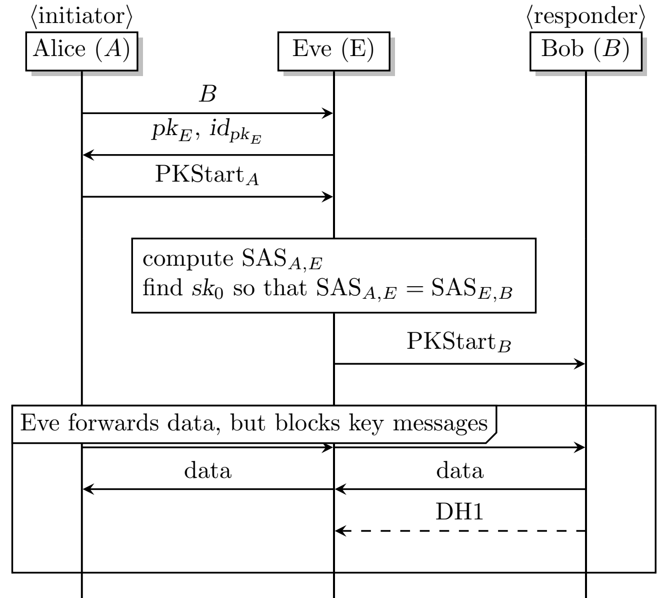

Figure 5: SCimp version 2: man-in-the-middle

2.5.4. 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 [19]: "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."

We 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.

3. RESULTS

This section shows the main vulnerabilities found in SCimp. We base our analysis on the Silent Circle documentation, but because of the lack of details we also considered the SCimp implementation [27].

3.1 Persistent man-in-the-middle attack on SCimp version 2

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.

One might be tempted to verify the SAS that is derived from $Z_{0,x}$, but Figure 5 shows why that does not authenticate the other party. The lack of a commitment to the first key of the DH key exchange gives the attacker the opportunity to find a SAS-collision.

The lower part of Figure 5 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 by 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 vigilant 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 vigilant user, the Commit message from SCimp version 1 is still available in the protocol layer as an alternative in SCimp version 2, but we are unaware if such a functionality is accessible through the user interface. Furthermore, there are no clear indications of what method is used for an incoming message.

3.2 Man-in-the-middle on SCimp PubKey mode

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, but no protocol support for this is provided in SCimp.

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 3.1, but at least that attack is detectable by sufficiently vigilant users.

3.3 Desynchronizing clients

Silent Circle only documents how progressive encryption should be done in the context of a fresh conversation. It remains undefined how Bob should handle a PKStart message from Alice when he has already set up a SCimp session with her (and they share a *cs*). It turns out that Eve can use the unspecified behaviour in order to desychronize the parties, which she can leverage into a full undetected man-in-the-middle attack.

By inspection of the code (see Section 5.2 for details) we found that the implementation will blindly accept any incoming PKStart message, *delete cs* and initiate a fresh protocol. Only the plaintext message header is inspected before triggering this behaviour. By deleting *cs*, the key continuity is destroyed and the identity of the other party is no longer validated. This gives rise to a trivial man-in-the-middle attack: Eve waits until Alice and Bob have set up a SCimp session and have verified the SAS. She then sends Alice a PKStart message, pretending it originates from Bob. She could use (for example) an automatically handled Siren message receipt (see Section 4) as user message so that Alice does not even detect that any message was sent at all. Alice continues with the key negotiation with Eve—as if it was the first key negotiation with Bob—and she will never even receive a warning that she has rekeyed from scratch. Eve completes the same process with Bob and effectively she has set up a man-in-the-middle attack without any of the parties noticing. Eve can go one step further and repeat the process with one of the parties until the SAS of both sessions matches. This way she avoids detection even in the unlikely event that a vigilant Alice and Bob somehow decide to revalidate their already confirmed SAS.

Version 2 of the protocol introduced another similar bug. When a client receives an out of order keying message (DH1, DH2 or Confirm), a general protocol error warning is given and *cs is deleted*. Only the plaintext message header is inspected before triggering this behaviour. The given warning does not hint at any malicious activity. Eve can leverage this to a man-in-the-middle attack as well. When Alice wants to send her next message it will probably be a PKStart message, for which she needs Bob's medium-term key (that was also deleted during the session reset). Eve injects her own public key and completes the key negotiation. Vigilant users can protect themselves from this attack by rekeying and revalidating the SAS after every protocol error warning they get. It is unlikely that they will, especially considering that Eve can inject arbirtrarily many invalid messages to trigger protocol errors until Alice and Bob grow tired of revalidating every single time.

3.3.1. Commit contention. According to the specification, clients should gracefully handle Commit contention (see Section 2.2.3).

What actually happens in the implementation is that when Alice receives an unexpected Commit message (possibly but not necessarily during key negotiation), she resets her state, deletes *cs*, issues a protocol contention warning and continues to process the Commit message. This means she will reply with a DH1 message that Bob does not expect, so it will trigger the above described bug and Bob resets his session. Out of order messages after Commit contention will also crash the protocol, but with more delay. Alice and Bob *must* redo the first key negotation *including* the SAS confirmation, losing all benefit from key continuation.

3.4 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. For example, this attack applies to the Signal Protocol [13]. 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 her 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 attack would match the chess-grandmaster attack described in [5] or the mafia-fraud attack described in [10]. However, this is not the issue with the above scenario: the issue is that Alice is convinced that B is the address of Eve and Alice and Bob are communicating directly without any message passing from Eve.

3.5 Out of order messages

The documentation lacks details on how to handle out of order messages. The implementation has two separate mechanisms in place in order to handle these messages: a few receive keys are stored to handle messages that are not too old and seed keys are stored in case messages arrive after rekeying.

More precisely, Bob computes 16 symmetric message receive keys k_{rcv} in advance and puts them in an array. When he receives a message, he looks up the corresponding key, 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.[5] 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.

After successful rekeying, Bob has derived new message keys. 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.

The problem with the stored seed keys is that the key will remain stored in the array of precomputed receive keys, if a message was withheld from the receiver, either by a network failure or by an adversary. If rekeying happens before 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.

[5]Keys are deleted before verifying the message authenticity, see Section 5.5.2.

3.6 First key negotiation leakage

The documentation states that a random value should be used for cs, when Alice and Bob do not yet share a cached secret. With a random value, Bob can ignore hcs_a from everyone he has not communicated with before (and parties should be careful not to include the value in the computation of h_{total}). Substituting a random value for cs has the advantage that the message does not leak if this is the first key negotiation or not.

However, in the implementation, the value cs is not chosen randomly, but set to 0. The implemented version leaks whether the key negotiation is an initial key negotiation or a rekeying. It is leaked in the message content to an attacker that computes hcs_a using $cs = 0$ and compares the sent value. This means an attacker can easily see that a new key negotiation has started without existing cs. This gives a confirmation that the DOS attack of Section 3.3 was successful in erasing cs.

3.7 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 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 first device before the resend 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.

4. SIREN

Silent Circle also provides some security/usability features to their application by adding functionality in a layer on top of SCimp, called Siren. 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.

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 data.

None of these features can be ensured by the protocol or by any other cryptographic measure in an application, although it might be enforced by the Blackphone if these features are integrated in the OS. We 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 SCimp. Secondly, SCimp version 2 allows for files to be sent via the cloud, which also has implications on the security of the protocol. We do not know if Siren (or a similar layer) is added on top of the Signal Protocol layer in the Silent Phone application, but if it is then the findings of this section apply to the current application as well.

4.1 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 3.2.

The documentation [19] 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.

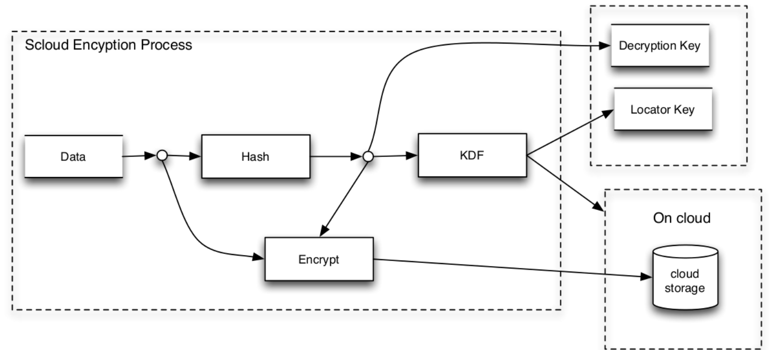

Figure 6: SCimp: Convergent encryption (source: [19])

Without a signature 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 one solved by SCimp.

4.2 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.

Silent Circle claims [19] that convergent encryption is used because 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.

Furthermore, 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 on 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.

4.2.1. Convergent Encryption. For completeness, in this section we will describe the details of how convergent encryption is implemented by Silent Circle. This is shown in Figure 6. When Alice wants to send a file f to Bob, she first derives a key k from the file metadata and data as $k = \text{hash}(metadata \,\|\, data)$.

The hash-algorithm is 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 as $id_f = \text{KDF}_k(\text{“SCloudLocator”}, salt, 32)$.

The KDF always uses HMAC in combination with SHA256.

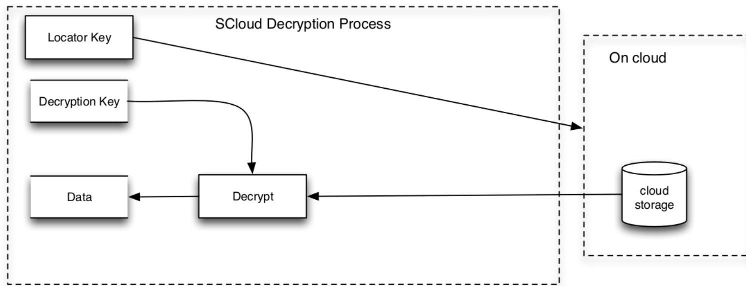

Figure 7: SCimp: Convergent decryption (source: [19])

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: $(ek, n) = \text{split}(k)$. She encrypts as $ct = \text{AES}_{ek}^{n}(metadata \,\|\, data)$. The encryption algorithm is 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.

Alice uploads 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.

Decryption of convergent encryption is depicted in Figure 7. 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.

4.2.2. Confirmation of a file. A known attack on convergent encryption is the confirmation of a file:

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 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 for the employer would only have grown if multiple users are suspected. For m suspicious employees they would need to add tm encrypted files to the database. 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.

4.2.3. Learn the remaining information. Another well-know attack on convergent encryption is the "learn-the-remaining-information" attack [32]. This attack applies to files that contain little entropy. The attacker can just 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 files 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.

4.2.4. 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 attacker can 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, they also learn 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 can detect this attack 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, for example, CCM instead of CBC blockmode, would not have helped prevent this attack, because it only authenticates the data under the assumption that the encryption key is unknown by the attacker.

4.2.5. 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. We are not aware of other implementations of convergent encryption omitting the easy check of file correctness.

5. IMPLEMENTATION DETAILS

Due to the insufficient level of documentation, code review was necessary, though tedious. The implementation of SCimp provided on GitHub had several implementation errors, including cryptographic errors. We report the results of our code review in this section. 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.

There are many ways in which the described bugs affect the security of the communication. Some bugs compromise the integrity and confidentiality of the protocol, or simply crash the application, leading to a possible DOS attack. More subtly, some encryption primitives are implemented

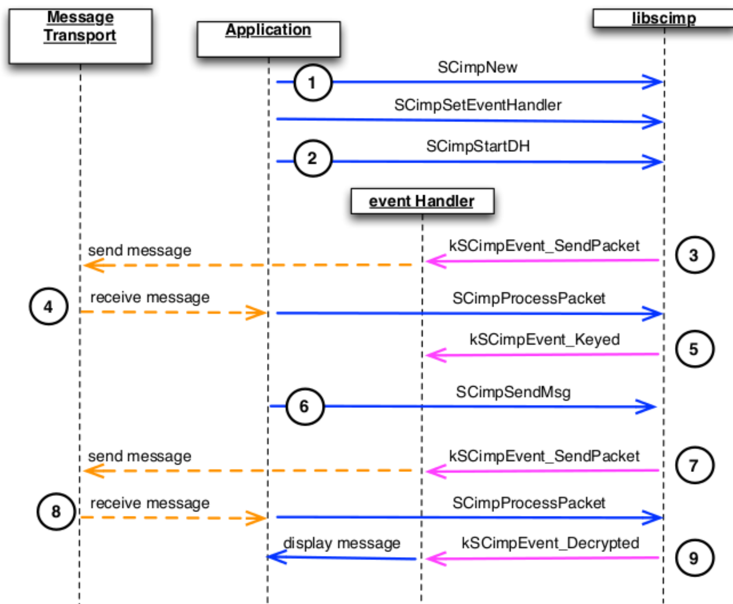

Figure 8: SCimp message flow (source: [18])

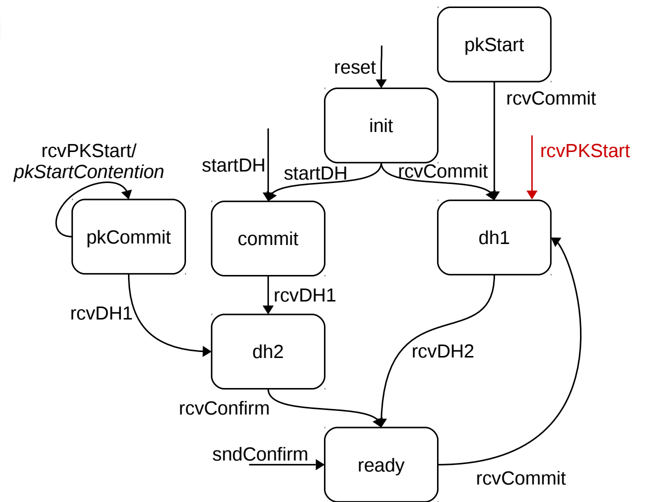

Figure 9: SCimp state diagram, as defined by `SCIMP_state_table`

wrong and leak information to an adversary through side-channels.

We conclude this section by analyzing non security-critical bugs, which could also be classified as *style issues*. We explicitly mention them because they hinder the legibility and maintainability of the overall software. To put it into the words of Jon Callas, one of the Silent Circle co-founders: "*All bugs are security bugs*" [8].

5.1 High level overview

The heart of SCimp is implemented in a few files: `SCimp.c` (1219 sloc[6]), `SCimpProtocol.c` (2970 sloc), `SCimp.h` (451 sloc) and `SCimpPriv.h` (227 sloc). This code calls 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` (default) and `SCimpProtocolFmtXML.c`.

The basic flow of a SCimp conversation is shown in Figure 8. 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 such as a choice for the cipher suite and the SAS method are set. Setting the event handler is non-optional and turns out to be very important for the functionality and security of the protocol. This handler is responsible for many types of events, the most important ones being errors, warnings and sending packets.

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. A `SCimpContext` also holds a reference to the used serializer.

At point 2 of Figure 8, 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.

5.2 State machine

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 9. This section explains the state machine, indicates some problems that it has, and looks at a few ways how the implementation bypasses the state machine all together.

In the diagram, the 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, *rcvDH2* triggers the transition `kSCimpTrans_RCV_DH2` and is handled by `sDoRcv_DH2`. The double label at the *pkCommit* state means that the transition *rcvPKStart* is handled by *pkStartContention*. 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*.

[6]Source lines of code, not counting comments and empty lines.

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

5.2.1. State machine bugs. The biggest issue with the state machine is highlighted in red. The *rcvPKStart* transition can be triggered by an incoming message and is independent of the current state. This leads to the man-in-the-middle attack described in Section 3.3.

There are a few other 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 5.2.2.

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. We 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*. It is not displayed because it can be triggered in any current state and does not affect the next state, so it is unrelated to the rest of the state machine. 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, triggering the bug described in Section 3.3.1.

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.

5.2.2. 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`, which leads to the second man-in-the-middle attack described in Section 3.3.

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.

Fix. The state machine appears to be a good idea, taken from the OTR documentation/code. 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.

5.3 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 [28, 16], according to the NIST specification [12]. 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.

NIST 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, but this comparison is done with a call to `memcmp`, which does a byte-by-byte comparison and returns as soon as the first difference was encountered (see also Section 5.4) thereby leaking the first difference.

We 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`. Since this would break compatibility with the existing implementation we came up with a quickfix as a compromise which was integrated upstream.[7]

Listing 1: LibTomCrypt: CCM quickfix

```
if (direction == CCM_ENCRYPT) {
  /* store the TAG */
  for (x = 0; x < 16 && x < *taglen; x++) {
    tag[x] = PAD[x] ^ CTRPAD[x];
  }
  *taglen = x;
} else { /* direction == CCM_DECRYPT */
  /* decrypt the tag */
  for (x = 0; x < 16 && x < *taglen; x++) {
   ptTag[x] = tag[x] ^ CTRPAD[x];
  }
  *taglen = x;

  err = XMEM_NEQ(ptTag, PAD, *taglen);
}
```

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, we found another bug. The function indeed did 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 learns the error locations and might learn something about the secret data that was compared. We proposed a simple fix, by folding the resulting byte (`ret`) into one single bit.[8]

[7] commit 25af184cd59b1769c0588678362adb5fd41a50ed
[8] commit 75b114517a3f8db2075a45b0af87d4d74778ad66

Listing 2: LibTomCrypt: `XMEM_NEQ`

```
[...]
ret |= ret >> 4;
ret |= ret >> 2;
ret |= ret >> 1;
ret &= 1;

return ret;
```

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.[9]

5.4 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 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, we look for software bugs that might lead to a timing attack. We did not analyze the LibTomCrypt library for other side-channel vulnerabilities, such as cache-based side-channel attacks [4], because there were enough timing attacks.

5.4.1. Comparison of secrets. Every comparison of secret data should be done in constant time. We already mentioned that after CCM decryption, the verification is 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.

`SCpubTypes.h` defines a macro `CMP`, that is used to compare secret values.

Listing 3: SCpubTypes.h (line 121–125)

```
#define CMP(b1, b2, length) \
(memcmp((void *)(b1), (void *)(b2), (length)) == 0)

#define CMP2(b1, l1, b2, l2) \
(((l1) == (l2)) && (memcmp((void *)(b1),
  (void *)(b2), (l1)) == 0))
```

The macro opens up a side-channel when validating hcs_b in DH1 (`SCimpProtocol.c::2254`); when validating hcs_a in DH2 (`SCimpProtocol.c::2318`); when validating mac_a in DH2 (`SCimpProtocol.c::2336`); when validating mac_b in Confirm (`SCimpProtocol.c::2371`); and in validating pk_a.

The leakage is not an issue for *mac* and pk_a – all key material is discarded. However, *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, the victim needs to initiate the protocol repeatedly and there might be warnings, at

[9] See also: http://thoughtcrime.org/blog/the-cryptographic-doom-principle/.

which point the device might decide[10] that the value *cs* is no longer trusted and that the SAS should be confirmed to re-authenticate the other person. If that happens, it does not matter if the attacker learns *cs*. The attacker also 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.

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.

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 2).

5.5 Software bugs

The code contains a few other bugs as well.

5.5.1. 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.

Listing 4: SCimpProtocol.c (line 3516–3544)

```
SCLError scTriggerSCimpTransition(
  SCimpContextRef ctx, SCimpTransition trans,
  SCimpMsg* msg)
{
  SCLError err = kSCLError_NoErr;

  if(pthread_mutex_trylock(&ctx->mp) == EBUSY)
  {
    err = sQueueTransition(&ctx->transQueue,
          trans, msg);
    return err;
  }

  err = sProcessTransition(ctx, trans, msg);

  while(ctx->transQueue.count > 0)
  {
    TransItem  item;

    err = sDeQueueTransition(&ctx->transQueue,
          &item);
    if(err == kSCLError_NoErr)
    {
      sProcessTransition(ctx, item.trans,
      item.msg);
    }
  }

  pthread_mutex_unlock(&ctx->mp);

  return err;
}
```

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. 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.

5.5.2. 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 clears the copied key buffer as well.

This gives the opportunity for an attacker to perform a DOS 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.

5.5.3. Mixing label in *hcs*. A discrepancy between code and spec is which value is hashed: according to the documentation the "Initiator" string is not digested: $hcs_a = \mathrm{MAC}_{cs}(\mathrm{hash}(pk_a) \,\|\, \text{“Initiator”})$. In the implementation, Alice does digest this string as $\mathrm{hash}(pk_a \,\|\, \text{“Initiator”})$. 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 DOS attack, where Eve can trick Bob into opening many SCimp sessions.

5.5.4. 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.

[10] This depends on the handlers, see Section 5.6.2.

Listing 5: SCpubTypes.h (line 127–129)

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

More often than not, this check is omitted after a memory allocation.[11] This can lead to the software crashing, but we are 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.

5.5.5. 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).

Listing 6: SCpubTypes.h (line 73–75)

```
#define CKERR  if((err != kSCLError_NoErr)) {\
STATUS_LOG("ERROR %d  %s:%d \n",
            err, __FILE__, __LINE__); \
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 consequence for the security of the implementation is highly dependent on the context. We have found a few omissions of `CKERR`, but we have not been able to escalate these bugs into security exploits.

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

5.6 Style issues

In this section, we look at some issues that affect the code, but which are not necessarily bugs. 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.

5.6.1. MAC length in the serializer. When inspecting the calls to the CCM decryption function, something stands out.

Listing 7: SCimpProtocol.c (line 2076–2081)

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

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. We think it is unwise to let the serializer be responsible for any security features. Instead, the `tagSize` parameter should be a constant value.

5.6.2. Warning/error handlers. One of the responsibilities of the event handler is to handle warnings and errors. We 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 we 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 including cs, which even leads to the man-in-the-middle attack described in Section 3.3. A simple error handling strategy would be to reset the `SCimpContext` without deleting 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.

5.6.3. *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.

5.6.4. 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.

5.6.5. Message index. The message index makes it possible to handle out-of-order messages. It is derived from kdk_2, using the KDF. In our Proverif analysis we showed that the protocol is secure when this value is made public. This in-

[11] We were not the only ones to notice: see https://github.com/SilentCircle/silent-text/pull/2, which catches (only) a few of the lines with this problem.

dicates that the value might as well be initialized at zero, speeding up the implementation.

5.6.6. KDF vs MAC. Silent Circle uses a standard extract-expand construction for deriving keys. Instead of naming this as their KDF, they decided to relabel a MAC function with fixed labels as a KDF, while it should have been called `expand`. Their additional enhance "step" is not really a different step, it is really just a part of the extract step.

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.

The confusing specification of the KDF lead to another implementation bug. $kdk_2 = \mathrm{KDF}'_{kdk}(Z_x, \text{“MasterSecret”}, \text{“SCimp-ENHANCE”} \parallel ctx \parallel \texttt{0x00}, 256)$. 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.

5.6.7. KDF implementation. Code could be eliminated where multiple calls to the MAC functions (init, update and final) could be replaced by a single call 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 5.6.8). 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.

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

Listing 8: SCimpProtocol.c (line 1136–1138)

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

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 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:

Listing 9: SCccm.c (line 216–221)

```
// This will only compare as many bytes of
// the tag as you specify in tagSize
// we need to be careful with CCM to not
// leak key information, an easy way to do
// that is to only export half the hash.

if((memcmp(T,tag,tagSize) != 0))
  RETERR(kSCLError_CorruptData);
```

We 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 [12]: "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 us. 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.

5.6.9. Naming inconsistencies. 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 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.

5.6.10. Code comment/documentation. The code has little documentation. The existing documentation in the form of the whitepaper [21] is outdated and inconsistent, both with itself and with the actual code of either version. The messaging ecosystem document [19] is a 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 us: "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 our opinion, this is something that needs to be documented.

6. COMPARISON

SCimp is an implementation of SecureSMS [3] in XMPP. The SecureSMS protocol took many ideas from existing protocols (most notably ZRTP [33] and OTR [6, 1]) and combined them to be suitable for a mobile environment. Now

Table 1: Comparison of Secure IM protocols
✓ means the protocol provides the feature, × means it does not, and ↯ means it provides it but is insecure.

	OTR	SCimp (v1/v2)	Signal
Data in first message	×	×/↯	✓
Forward secrecy	✓	✓/✓	✓
Preshare public keys	✓	×/×	✓
Rekey each reply	✓	×/×	✓
Ratchet each message	×	✓/✓	✓
Elliptic curve crypto	×	✓/✓	✓

that SCimp has been discontinued, Silent Circle wrote their own implementation[12] of the Signal Protocol [13]. The results of a comparison between the protocols is given in Table 1.

Although SCimp version 2 can send user data in the first message, we have shown that this is insecure. Signal improves the forward secrecy by hashing the message keys of the ratchet keys. As SCimp is a SAS based protocol, users cannot share public keys before initiating communication. Note that elliptic-curve crypto does not necessarily mean that the protocol is more secure, but its lower computational requirements and smaller keysizes make it more suitable for a mobile environment.

We refer the interested reader to [29] for a more detailed explanation of these results and to [13] for a detailed analysis of Signal.

7. FORMAL VERIFICATION

The ProVerif models of SCimp version 1 and Progressive Encryption are available on GitHub:

github.com/sebastianv89/scimp-proverif

SCimp version 1 shares some structure with ZRTP, of which the secrecy was modelled in ProVerif in [7]. We go beyond that model, by formalizing the SAS confirmation instead of having to rely on an ad-hoc argument that the SAS prevents a man-in-the-middle attack. In addition, our models cover authenticity, forward secrecy, future secrecy and deniability of the protocol and we model Progressive Encryption. ProVerif confirms that SCimp version 1 is secure and finds the man-in-the-middle attack on Progressive Encryption described in Section 3.1.

8. CONCLUSIONS

This paper found several weaknesses in the SCimp protocol and implementation. Despite the high price tag of Blackphone and Silent Text, the quality of crypto and code looks worse than for OTR or Signal, which is yet another confirmation of the importance of open code and detailed security analysis.

9. ACKNOWLEDGMENTS

This work was done while Verschoor was a student at TU/e. A much extended version can be found in his masters thesis [29]. This work was supported in part by the European Commission through H2020-ICT-645421 ECRYPT-NET. This work was supported in part by Canada's NSERC CREATE Program. IQC is supported in part by the Government of Canada and the Province of Ontario.

[12] We did not inspect the Silent Circle implementation, just the original implementation [22] by Open Whisper Systems.

10. REFERENCES

[1] C. Alexander and I. 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.

[2] Apple Inc. Local and Remote Notification Programming Guide, 2015. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/RemoteNotificationsPG.pdf.

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

[4] D. J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/papers.html#cachetiming, 2005.

[5] T. Beth and Y. Desmedt. Identification tokens – or: Solving the chess grandmaster problem. In *CRYPTO*, volume 537 of *LNCS*, pages 169–177. Springer, 1990.

[6] N. Borisov, I. Goldberg, and E. Brewer. Off-the-Record Communication, or, Why Not to Use PGP. In *WPES '04*, pages 77–84. ACM, 2004.

[7] R. Bresciani and A. Butterfield. A formal security proof for the ZRTP protocol. In *Proceedings of the 4th International Conference for Internet Technology and Secured Transactions, ICITST 2009, London, UK, November 9-12, 2009*, pages 1–6, 2009.

[8] J. Callas. Building Hardware We Are Proud Of, 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.

[9] L. Chen. NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions, 2009.

[10] Y. Desmedt, C. Goutier, and S. Bengio. Special uses and abuses of the Fiat-Shamir passport protocol. In *CRYPTO*, volume 293 of *LNCS*, pages 21–39. Springer, 1987.

[11] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[12] M. Dworkin. NIST SP 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, 2004.

[13] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is TextSecure? In *EuroS&P*, pages 457–472. IEEE, 2016.

[14] Google. Cloud Messaging, 2015. https://developers.google.com/cloud-messaging/.

[15] J. Hildebrandt and P. Saint-Andre. XEP-0033: Extended Stanza Addressing. Technical report, XMPP Standards Foundation, September 2004. https://xmpp.org/extensions/xep-0033.html.

[16] S. Jaeckel. libtom/libtomcrypt, 2015. https://github.com/libtom/libtomcrypt (commit 16f397d55c9f4971a66a7ce9d87d0305ab45eaa7).

[17] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function

(HKDF). RFC 5869, 2010.
https://www.rfc-editor.org/rfc/rfc5869.txt.

[18] V. Moscaritolo. Silent Circle Instant Messaging Protocol - libscimp API guide, 2012. https://github.com/SilentCircle/silent-text/blob/master/Documentation/ (commit bee6f4955252995e07b761ecd40bf68e64d809f1).

[19] V. Moscaritolo. Silent Circle Messaging Ecosystem, 2014. https://github.com/SilentCircle/silent-text/blob/master/Documentation/ (commit bee6f4955252995e07b761ecd40bf68e64d809f1).

[20] V. Moscaritolo. Silent Text 2.0: The next generation of private messaging, 2014. https://web.archive.org/web/20150506152939/https://blog.silentcircle.com/silent-text-2-0-the-next-generation-of-private-messaging/.

[21] V. Moscaritolo, G. Belvin, and P. Zimmermann. Silent Circle Instant Messaging Protocol - Protocol Specification, 2012. https://github.com/SilentCircle/silent-text/tree/master/Documentation (commit bee6f4955252995e07b761ecd40bf68e64d809f1).

[22] Open Whisper Systems. Signal Protocol Library for Java/Android, September 2015. https://github.com/WhisperSystems/libsignal-protocol-java (commit 01bc1eb37be2113f78392df4bed93ff173aee98e).

[23] M. Shirvanian and N. Saxena. Wiretapping via mimicry: Short voice imitation man-in-the-middle attacks on crypto phones. In *ACM-CCS*, pages 868–879, 2014.

[24] Silent Circle. SCIMP | Silent Circle. https://web.archive.org/web/20150718145410/https://silentcircle.com/scimp-protocol.

[25] Silent Circle. Software: Communicate privately on Silent OS, iOS, and Android. https://web.archive.org/web/20150816133352/https://www.silentcircle.com/products-and-solutions/software/.

[26] Silent Circle. What is Silent Phone? https://support.silentcircle.com/customer/en/portal/articles/2118686-what-is-silent-phone-.

[27] Silent Circle. Encrypted text messaging, 2015. https://github.com/SilentCircle/silent-text (commit bee6f4955252995e07b761ecd40bf68e64d809f1).

[28] T. St Denis. LibTom. http://libtom.net.

[29] S. R. Verschoor. Secure messaging in mobile environments. Master thesis, Technische Universiteit Eindhoven, 2015. http://alexandria.tue.nl/extra1/afstversl/wsk-i/Verschoor_2016.pdf.

[30] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610, 2003.

[31] Z. Wilcox-O'Hearn. Open letter to Phil Zimmermann and Jon Callas of Silent Circle, re: Silent Mail shutdown. http://lists.randombit.net/pipermail/cryptography/2013-August/004982.html.

[32] Z. Wilcox-O'Hearn. Attacks on Convergent Encryption. Technical report, Tahoe-LAFS, 2008. https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html.

[33] P. Zimmerman, A. Johnston, and J. Callas. ZRTP: Media Path Key Agreement for Unicast Secure RTP. RFC 6189, RFC Editor, April 2011.