Implementing Clone Detection in Signal's Double Ratchet Algorithm

KYLEN ADAMS, University of Minnesota - Twin Cities
QUINTEN NORTON, University of Minnesota - Twin Cities
ZANE SMITH, University of Minnesota - Twin Cities

ABSTRACT

This paper will discuss signal's double ratchet algorithm and it's implementation to our own chat client. It will also discuss the security concerns around cloning and how we implement a protocol to the double ratchet algorithm to detect clones. First we discuss what the double ratchet is and how it works in depth. Then, we talk about the different ways clones can attack this algorithm and why this is a concern. Next, we talk about our code and how we implemented the double ratchet and the clone detection protocol. Finally, we share our results and how our code reacts to certain cases involving a clone in the messaging chat client.

Additional Key Words and Phrases: Clone Detection, KDF chains, Double Ratchet, Signal, Implementation

ACM Reference Format:
Kylen Adams, Quinten Norton, and Zane Smith. 2021. Implementing Clone Detection in Signal's Double Ratchet Algorithm. 1, 1 (May 2021), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Signal protocol is a popular cryptographic protocol among messaging apps. Applications such as Facebook Messenger, Skype, WhatsApp, and others all use the Signal protocol, either as their default protocol, or as an additional security oriented option. Hence, there is reason to scrutinize the strength of the protocol, due to its wide usage. We will talk about one security concern, cloning and finding a way to detect such clones. We will first discuss some necessary machinery from the protocol for the sake of our topic.

2 THE SIGNAL PROTOCOL

The Signal protocol is comprised of two main elements. Namely, the triple Diffie-Hellman key exchange (X3DH), and the Double Ratchet algorithm. For the sake of our research, the Double Ratchet algorithm is the important feature.

2.1 KDF Chains

Each "ratchet" of the Double-Ratchet algorithm are built upon key derivation function (KDF) chains. A KDF takes an input chain key along with some other input (a bit string), and outputs both the next input chain key, along with an output encryption/decryption key. These key derivation functions are pseudo-random functions. This awards two important properties: An attacker has negligible probability of distinguishing its output from random output, and it is deterministic. Hence, we can build a "chain" by simply using the output chain key as the input key for the next call of our KDF. This way, if Alice and Bob want to generate identical keys they can, so long as they agree on an initial chain key and the input for each key they generate. We then have a basis for the Symmetric Ratchet.

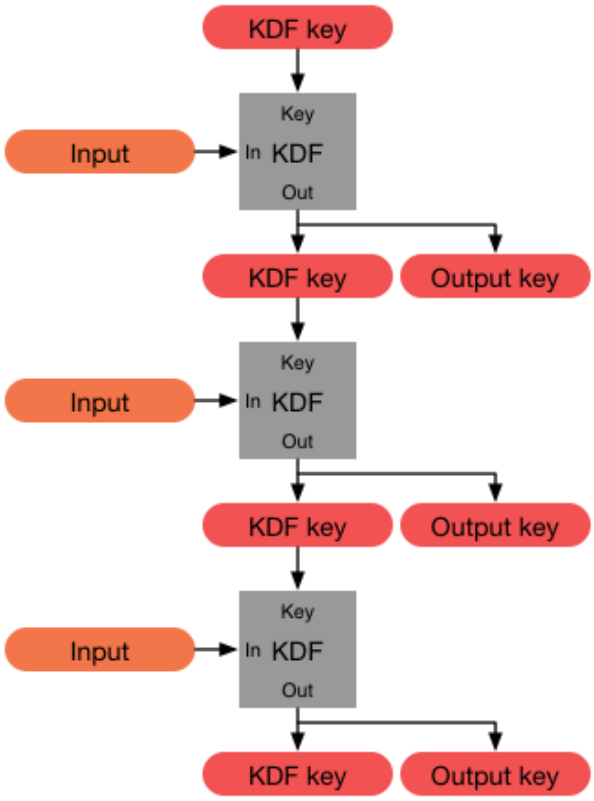

Fig. 1. Key Derivation Function Chain [1]

2.2 The Symmetric Ratchet

Say Alice and Bob would like to have a conversation. Through the use of KDF chains, Alice and Bob can each generate identical keys, for which they can use to encrypt and decrypt their messages. In fact, in the Symmetric Ratchet both Alice and Bob will create a "send" chain, and a "receive" chain. In order for this scheme to work, Alice's send chain must be in sync with Bob's receive chain, and Bob's send chain must be in sync with Alice's receive chain. This

Authors' addresses: Kylen Adams, adam1200@umn.edu, University of Minnesota - Twin Cities; Quinten Norton, norto318@umn.edu, University of Minnesota - Twin Cities; Zane Smith, smit9474@umn.edu, University of Minnesota - Twin Cities.

XXXX-XXXX/2021/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

way, when one party wants to send a message to the other (say Alice wants to send to Bob), then Alice will simply generate a key from her send chain. Then she will use this key to encrypt her message, and send it to Bob. Bob can then generate a key from his receive chain, to get an identical key which he can use for decryption. Of course, this only works so long as Bob's receive chain is indeed in sync with Alice's send chain. For this to happen, Alice and Bob must agree on an initial chain key, as well as the bit string input for each run of their KDF.

2.3 The Diffie-Hellman Ratchet

To agree on an initial chain key, the Signal Protocol specifies the second ratchet of the Double-Ratchet algorithm, the Diffie-Hellman Ratchet. This ratchet is yet again a KDF chain, known as the "root" chain. In this chain, the output keys serve not as encryption and decryption keys, but rather initial chain keys for Alice and Bob to use for their sending and receiving chains. Of course, this begs the question, how do Alice and Bob agree on an initial root chain key? For this, they each generate an identical key via a triple Diffie-Hellman key exchange, the second staple of the Signal Protocol. However, this is not the only key exchange performed in the Diffie-Hellman ratchet. In fact, each bit-string input is the result of a Diffie-Hellman key exchange performed by Alice and Bob. This provides the root-chain break-in security. If an attacker recovers the root chain key at any point, they still need to crack the input Diffie-Hellman key in order to generate the same keys that Alice and Bob do. With this mechanism in mind, we can discuss holistically the Double-Ratchet Algorithm.

2.4 The Double Ratchet Algorithm

Now that we have the necessary machinery, let us explicitly lay out the Double-Ratchet Algorithm: Suppose Alice would like to communicate with Bob. Then,

(1) Alice and Bob will perform a triple Diffie-Hellman key exchange to each produce an identical key, $ROOT_0$.
(2) Alice and Bob will then perform a Diffie-Hellman key exchange to generate a secret identical key $DH1$.
(3) Alice and Bob will each calculate the result $ROOT_1, CK1 = KDF(ROOT_0, DH1)$, where KDF is a predetermined pseudo-random function.
(4) Alice and Bob will repeat step 2 to generate $DH2$, and then repeat step 3 to compute $ROOT_2, CK2 = KDF(ROOT_1, DH_2)$.
(5) Alice will use $CK1$ as her initial send chain key, $ALICE\_SEND_0$, and $CK2$ as her initial receive chain key $ALICE\_RECEIVE_0$.
(6) Bob will use $CK2$ as his initial send chain key BOB_SEND_0, and $CK1$ as his initial receive chain key $BOB\_RECEIVE_0$.
(7) If Alice wants to send a message to Bob, she will calculate $ALICE\_SEND_1, KEY1 = KDF(ALICE\_SEND_0, inp)$ where inp is some predetermined input. Alice can then encrypt her message with $KEY1$. Bob will then compute $BOB\_RECEIVE_1$, $KEY1 = KDF(BOB\_RECEIVE_0, inp)$ and decrypt the message with $KEY1$.
(8) Bob can send a message to Alice with a similar process.

In principal, Alice and Bob can use each of their chains for the remainder of their conversation. Each time one wants to send a message to the other, they use their current chain key to generate an encryption key, and the other can generate an identical decryption key. In practice, Alice and Bob reinitialize their send-receive chains quite frequently, if not every message (using a similar process to steps 2-4). Doing this ensures that if Alice or Bob's current chain key is compromised, it can not be used indefinitely, as another Diffie-Hellman key exchange will take place, rendering the recovered key useless for any future messages.

3 MOTIVATION

Now that we have built the necessary machinery, we can discuss the strengths, and weaknesses of the Signal Protocol. First, we have two properties that can be achieved with the protocol:

(1) Forward Secrecy: An attacker that recovers the state of a KDF chain at some time t cannot recover any messages sent prior to t, except with negligible probability.
(2) Break-In Security: An attacker that recovers the state of a KDF chain at some time t cannot recover any future messages, except with negligible probability, provided that Alice and Bob reinitialize their symmetric ratchet.

It seems as though if the Signal protocol is implemented wisely, an attacker that breaks the current state (chain key, output key) of a KDF chain (which is no trivial feat if sufficient sized keys are used!) really cannot recover *any* messages besides the current one.

Of course, applications are free to implement as they may, and perhaps they choose not to reinitialize the symmetric ratchet on every message, but rather at a different chosen frequency. In either case, we expect that there is a pretty strong notion of post-compromise security. However, the protocol does not exist in a vacuum, and there are usability concerns to be had. Namely, how can an app handle state loss, while balancing usability and security?

Say Alice sends a message to Bob. Alice will then generate a new chain key and output key, and encrypt her message and send it to Bob. Say, however, that Alice's device dies before her new chain key can be written to memory. What happens the next time that Alice would like to send a message to Bob? If Alice and Bob do not reinitialize their symmetric ratchets, then Alice will regenerate the same key that she used to encrypt her last message with. Bob may still have this old key, and may still be able to decrypt Alice's message, but using old keys threatens our notion of forward secrecy.

As another example, say Alice gets a new device entirely. Then, the states of both of her ratchets may be lost entirely. If she wants to continue her conversation with Bob, Alice and Bob will have to reinitialize, even though Bob's protocol dictates that he should keep using his functioning ratchets.

Clearly, these usability concerns are of great importance to an application. State-loss may not be as infrequent as one would think, and dealing with it leads to a crossroads in security vs. usability. However, as the paper that we studied shows, there is in fact a way to remedy this dichotomy. It sacrifices no usability features observed in popular applications, while preventing a genre of attacks dubbed as "cloning attacks." So, we will now discuss the cloning problem.

4 CLONING PROBLEM

The Cloning Problem refers to a particularly hard to defend against form of attack called a "Clone Attack" wherein the attacker has managed to create a perfect copy of one of the users devices at a particular point in time. The Double Ratchet algorithm is able to handle this form of attack and offer a form of post-compromise security, however doing so requires a perfect state of synchronicity is maintained between the two parties. There are several other contemporary solutions to the problem of detecting that a user has been cloned.

One such approach developed by Yu et al. [5] was given the title "Detecting Endpoint Compromise In Messaging (DECIM)". In this approach a reliable third party is introduced for the role of a log maintainer in charge of appending data to a write-only log, and publishing digests of it upon request. This log is then used to by users to detect that they have been cloned in a future epoch where the cloner cannot act as a MITM against the cloned party. The figure below from their paper illustrates the logging protocol.

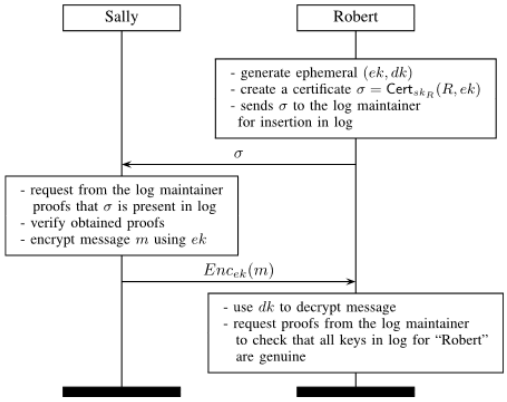

Fig. 2. The DECIM Logging Protocol [5]

Several other state of the art methods for clone detection that do not require a third party were developed by Milner, Cremers, Yu, and Ryan [4] and elaborated on in their 28 page publication cataloguing the foundations, design principles, and applications of automatically detecting secret misuse. These range from counter and hashing based methods to commitment based methods. For example one such method is to maintain "rolling nonces in a hash chain". IE, when Alice sends an authenticated message she sends a new nonce and a hash chain of the the previous nonces used by both parties, and Bob can check that it matches his own to ensure a clone hasn't acted on his part. This is Example 3 in their work and is illustrated with the diagram in Figure 3.

5 IMPLEMENTATION

To deviate further from the original paper [3], we sought out a way to implement their proposed clone detection protocol to the double ratchet algorithm. We used the programming language python to implement the algorithm and protocol. To start explaining our implementation, we used the python pip package installer to install

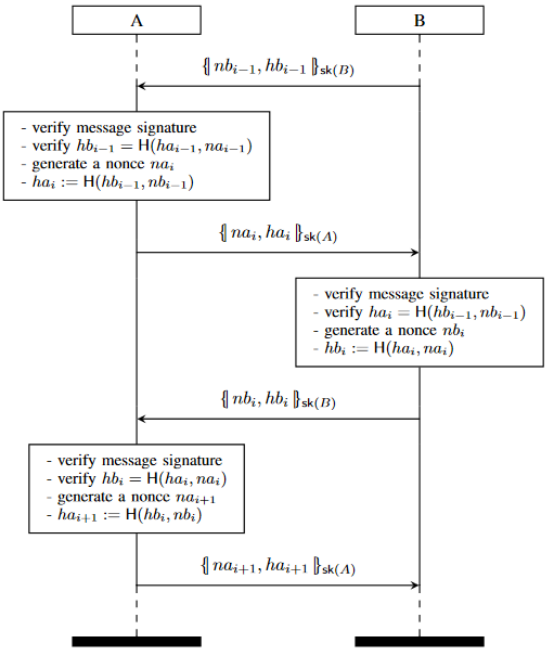

Fig. 3. The Rolling Nonce Hash Chain Protocol [4]

the packages pycryptodome and cryptology. By executing these commands in bash.

```
$ pip install pycryptodome
$ pip install cryptology
```

Here we use cryptology to implement the key derivations and HKDF. We use pycryptodome to implement AES, SHA256, and HMAC. With these algorithms implemented we can proceed.

We also wrote three important helper functions in the implementation.

(1) B64: A function we implemented to help decode byte-arrays to readable text that gets displayed in terminal.
(2) Pad/Unpad: A function dedicated to padding a message with the length of the message along with filler bytes. This is used for adjusting messages to the correct length so that AES can input a fixed length message. Unpad just undoes this process of padding.
(3) HKDF: This is a function for our KDF chain, this is the function which makes calls to derive a key.

5.1 Client Class

Next, we created a client class where all of the personal client-side aspects reside. This is where we make characters like Bob, Alice, and more. Here we initialize the client with four private keys, their name, and a boolean value signifying if that client is initializing the conversation with another.

5.2 H3DH

We then create the H3DH key exchange function between two clients. In this function we have provide two options providing if the client calling this function is flagged as the initializer of the conversation. If so, we preform three iterations of the Diffie-Hellman key exchange and develop a shared-key *sk*. We use the HKDF function to combine all the previous key exchanges from the three Diffie-Hellman exhanges to create the shared-key. The process is the same if the client calling the function is not the initializer, but uses different created keys from the client class.

5.3 Diffie-Hellman Ratchet Class

Next, we create one of the two ratchets in the double ratchet algorithm the Diffie-Hellman ratchet. Here we create and initialize the ratchet. We initialize this class with a generated key, and a state. This initial state is modified in our client class, when a client is labeled as the “initializer" it makes the state 1, if not, 0.

We then implement a function called “next". This function is how we “turn" the ratchet to produce a new root key. We start by implementing a conditional statement that checks if the state is less than 2. If so, it uses the initial key generated in the initialization and preforms a Diffie-Hellman key exchange with the inputed key. Then iterates the state up 1 and outputs this exchanged key. If the state is not less than 2, the state changes to 0, a new inital key gets generated and preforms a Diffie-Hellman key exchange with the inputed key. Then iterates the state up 1 and outputs this exchanged key. This effectively creates new root keys so it can be utilized in the symmetric ratchet.

5.4 Symmetric Ratchet Class

For our next and final ratchet, we create a symmetric ratchet class. Here we initialize the state as an inputed key. We also create another version of the next function we used in the Diffie-Hellman ratchet to “turn" the symmetric ratchet deriving a new key and iv. To create this function the function has an inputed root key from the Diffie-Hellman ratchet, this is then added to our state where this is the input for the HKDF function we call to derive the key. We use this output from the HKDF function to update the state with the first 32 bits, then we output the derived key from the next 32 bits and an iv from the remaining amount. This effectively creates new keys and iv’s used later for sending and receiving messages.

5.5 Initializing the Ratchets

Now, we need to start both ratchets between two clients messaging each other. To do so, We create a function called init_root which uses the clients shared key as an input to make a call to the symmetric ratchet. This starts the symmetric ratchet with the root key being the shared key. Then we define a function called init_send, this function takes an inputed client public key and calls the clients Diffie-Hellman ratchet’s next function with this public key as an input. This output is then stored as “inp" and then calls the clients symmetric ratchet’s next function and uses inp as an input for it. This is now initialized as the clients send ratchet. Finally, we create a function called init_recv which preforms the same exact process as init_send. But in the end this initializes the clients receive ratchet.

6 CLONE DETECTION PROTOCOL

Here, we will describe the protocol for detecting clones that was described in the original paper [3].

(1) First, they took the root key generated from the Diffie-Hellman ratchet and sent it through a hash function to create what they call an epoch key. In our implementation we used SHA256 as our hash function. We also have this epoch key calculated and tied to the client from the Diffie-Hellman ratchet.
(2) Next, they provided counters for each client. One for how many messages they sent and another for how many messages they received. In our implmentation we initialize these counters in the client class and have they tied to each client individually.
(3) Then, when a client wants to send a message they append their send counter to the message and encrypts it using keys derived from the symmetric ratchet to obtain a cipher text. The receiving client can then derive the send counter from this cipher text by decoding it. In our implementation we used AES to encrypt.
(4) Finally, with this cipher text we derived, they then use a MAC with the cipher text as an input and the epoch key as the key. This is then sent to the other client to verify. In our implementation we used HMAC for the MAC.

How we detect if a clone is present, is if the receiving client fails to derive the MAC with the epoch key or if the send counter is less than the other clients receive counter. In the original paper [3] they proved you needed both of these check cases to provide an accurate way to detect clones. [2] We can see this protocol in figure 1.

6.1 Implementing the Clone Detection Protocol

To implement this protocol we create a send and receive function, where these functions preform the ratcheting steps, sending encrypted messages, receiving and decrypting messages, and providing the steps necessary to protecting against clones. First, this function takes two clients, the one sending and one receiving and the message being sent. We initialize the ratchets from 4.5, with the respective public key from the initializer of the two. Then we proceed to sending a message, we iterate the send counter and append it to the message being sent. We obtain the key and iv from the symmetric send ratchet next function. We then call AES with this key and iv encrypting the message appended with the send counter to receive a ciphetext. We also calculate the MAC-cipher by calling HMAC with the epoch key from the sending client and as an input the ciphertext. We then send the ciphertext and the MAC-cipher to the receiver.

In the receive function, we use HMAC with the receive client’s epoch key and the cipher text as an input this output will be called the receive-MAC-cipher. Next, we provide a conditional statement if the send counter (received from the cipher text) is less than the receive counter and if the the MAC-cipher is the same as receive-MAC-cipher then we may proceed, if not then this detects a clone. To proceed, we iterate the receive counter and obtain a key and iv from the symmetric receive ratchet next function. Then we use AES

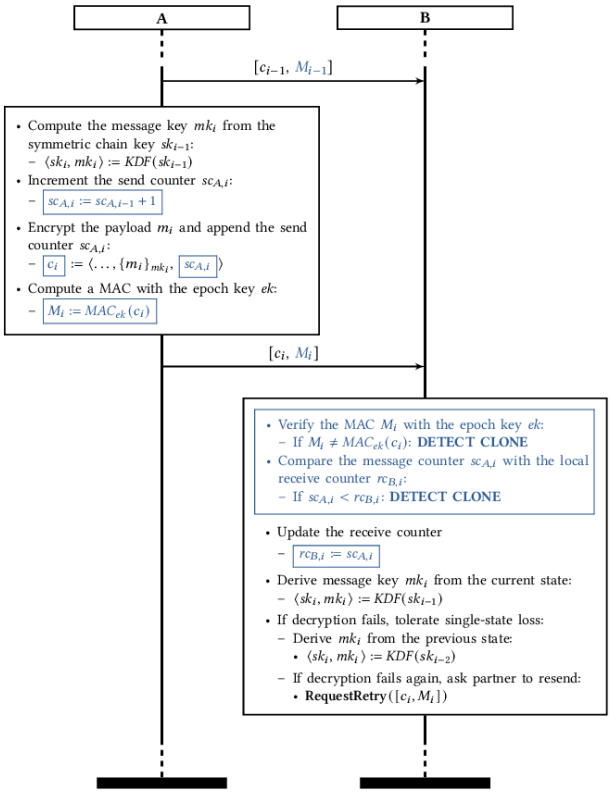

Fig. 4. Clone detection protocol [3]

with this key and iv to decrypt the ciphertext provided to obtain the original message.

7 CONCLUSION

To see our implementations in practice, we devised some ways to test if our implementation detects a clone. We first test to see if we can start a conversation between two clients Bob and Alice. We start this by creating the profiles "Bob" and "Alice" by using the client class. Then we preform the X3DH function with each client to produce a shared key. Next, we initialize each root chain (int_root) on each client. Then, we initialize each send and receive chain (init_send and init_recv). Now we have initialized the ratchets on each client to start sending and receiving messages. Then, we provide a conversation between Bob and Alice, where Bob and Alice send a couple of messages each. In our test, we can see Bob receives and decodes Alice's messages and vice-versa.

We then devise a test to clone Bob labeled "Evan" where Evan represents a single-state-loss event. In this test Evan communicates with Alice for a couple of messages while Bob is offline. Once Bob comes back on to message Alice our implementation outputs a clone has been detected since Bob's send counter is less than Alice's receive counter. This covers the case for when the send counter is less than the other clients receive counter.

Finally, we create a test for when Bob goes through a total-state-loss. Evan, Bob's clone goes through an entire initialization with Alice and provides a conversation with Alice all while Bob is offline. Once, Bob comes back online he sends a message to Alice. Our implementation detects a clone since Bob cannot derive the MAC with the epoch key, since Alice went through a different initialization with Evan. This satisfies the case where the receiving client fails to derive the MAC with the epoch key.

In conclusion, our implementation works in providing a messaging system with the double ratchet algorithm between two clients. While also detecting clones with single-state-loss and total-state-loss. This implementation proves one can devise a messaging system while protecting against clones, without giving up security for usability.

REFERENCES

[1] [n. d.]. Specifications » The Double Ratchet Algorithm. https://www.signal.org/docs/specifications/doubleratchet/
[2] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. 11476 (2019), 129–158. https://doi.org/10.1007/978-3-030-17653-2_5
[3] C. Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. 2020. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020).
[4] Kevin Milner, Cas Cremers, Jiangshan Yu, and Mark Ryan. 2017. Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Applications. Cryptology ePrint Archive, Report 2017/234. https://eprint.iacr.org/2017/234.
[5] Jiangshan Yu, Mark Ryan, and Cas Cremers. 2017. DECIM: Detecting Endpoint Compromise In Messaging. *IEEE Transactions on Information Forensics and Security* PP (08 2017), 1–1. https://doi.org/10.1109/TIFS.2017.2738609