CS6431 Project Report: Implementation of Message Franking on `libsignal` and Analysis

Wei-Kai Lin (wl572), Yue Guo (yg393)

May 24, 2018

1 Introduction

In the scenario of end-to-end messaging, a receiver may want to report to a trusted third party that the sender has sent abused message. Note that the server typically needs to verify the report is authentic, i.e., the sender indeed sent such message. However, as the encryption is end-to-end, the server could not know the message had the receiver not reported. Message franking [2, 1] considers such scenario and proposes schemes that realizes 1) the security of end-to-end authenticate encryption, and 2) the feature of verifying an authentic report. For the sake of exploration, for each message, we say that the message is sent from Sender to the receiver in a *sending phase*, where both Sender and Receiver are two clients; then, in a *reporting phase*, Receiver (w.r.t. the message) may want to report to the trusted Server that the message is abused.

In this project, we implement message franking in the open source library `libsignal`, which is an open-source end-to-end encryption protocol. The implemented message franking scheme is the Committing Encrypt-then-PRF (CEP) that is proposed in Grubbs et al.[2]. We include the pseudocode of CEP in Algorithm 1.

2 Implementation of Message Franking

Adding message franking to an existing end-to-end encrypted messaging protocol incurs interface change on both client and server side. The interface of the sender (`encrypt`) doesn't change, which takes as input a plaintext although outputs a cipher message in a new format. However, the interface of the receiver (`decrypt`), after takes as input a cipher message, outputs not only a plaintext but also an opening or *proof*, where the proof is needed whenever the receiver wants to report the plaintext to the server. Finally, to verify a pair of plaintext and proof, in our implementation, the server *tags and forwards* the cipher message in the sending phase, and thus it is necessary to use the new format of the cipher message to efficiently compute the tag. In the following, we describe such interface and its usage, and then, some implementation details are shown regarding the security and efficiency.

Algorithm 1 Scheme of Committing Encrypt-then-PRF (CEP). In our implementation, the encrypt/decrypt algorithm (E, D) is AES-CTR; symmetric key K and initial vector N are derived using HKDF from the shared secret of each message in `libsignal`; H is the header of each message, which will be elaborated in Section 4.2; M and C are plaintext/ciphertext respectively.

```
procedure $\text{CEP-}\textsc{Enc}_K^N(H, M)$
    $P \leftarrow E_K(N, 0^\ell || 0^\ell || M)$
    $P_0 \leftarrow P[0:\ell], P_1 \leftarrow P[\ell:2\ell], C \leftarrow P[2\ell:]$
    $\text{com} \leftarrow \text{HMAC}_{P_0}(H||M)$
    $T \leftarrow \text{HMAC}_{P_1}(\text{com})$
    return $(C||T, \text{com})$
procedure $\text{CEP-}\textsc{Dec}_K^N(H, C||T, \text{com})$
    $P \leftarrow D_K(N, 0^\ell || 0^\ell || C)$
    $P_0 \leftarrow P[0:\ell], P_1 \leftarrow P[\ell:2\ell], M \leftarrow P[2\ell:]$
    $\text{com}' \leftarrow \text{HMAC}_{P_0}(H||M)$
    $T' \leftarrow \text{HMAC}_{P_1}(\text{com}')$
    if $T \neq T'$ or $\text{com} \neq \text{com}'$ then
        return $\perp$
    return $(M, P_0)$
procedure $\text{CEP-}\textsc{Verify}(H, M, K_f, \text{com})$
    $\text{com}' \leftarrow \text{HMAC}_{K_f}(H||M)$
    if $\text{com} \neq \text{com}'$ then return 0
    return 1
```

2.1 New Interfaces of Library

We implement Committing Encrypt-then-PRF on `libsignal-protocol-javascript`, the Signal Protocol library for JavaScript. The following interface works directly in JavaScript, but it is straightforward to implement them on the library for Java or for C given that it was already implemented using Protocol Buffers[1].

Client-side Interface: Output of `decryptWhisperMessage`. In this paragraph, we describe the change of return type of `decryptWhisperMessage`, the function that handling decryption. The interface of session-building and encryption are unchanged. We defer the usage of the new interface to Section 2.2.

The function call to decrypt a cipher message works as follows, where `ciphertext` is the object returned by the encryption, and `plaintext` is an array of bytes.

```
sessionCipher.decryptWhisperMessage(ciphertext.body, "binary").then(function (plaintext) {
    console.log(plaintext);
});
```

[1] See https://developers.google.com/protocol-buffers/.

```
{
    header: ArrayBuffer,         // metadata, returned for HMAC verification
    body: ArrayBuffer,           // the original plaintext
    commitKey: ArrayBuffer(32),  // commitKey and commitment, to verify, check
    commitment: ArrayBuffer(32)  // HMAC(commitKey, concat of (header, body)) == commitment
}
```

Table 1: Evidence Structure

However, with message franking implemented, we augment the plaintext into a structure. In the context of reporting an abuse, we call it “evidence.” Note that such returned structure doesn’t include a server generated tag, which depends on how does server verify the `commitment`. It is defined in the following paragraph.

Server-side Interface: Cipher Message. In scheme of CEP, the server needs to “know” the `commitment` of each cipher message in order to verify the plaintext. In the implementation, the server computes a tag of `commitment` using its secret key, and then forwards both cipher message and tag to the receiver. We modify the format of cipher messages as follows, where fields `mac`, `commitment` and `tag` are modified or added for the purpose of message franking. Upon forwarding

```
{
    type: Unit8,
    body: ArrayBuffer, concatenation of
        version: 1 byte,
        message: serialized Protocol Buffer of
            ephemeralKey: bytes,
            counter: uint32,
            previousCounter: uint32,
            ciphertext: bytes
        mac: 32 bytes,                      // 8 bytes in original format
        commitment: 32 bytes                // new entry
    registrationId: Uint32,
    tag: ArrayBuffer(32)                    // new entry, write by Server
}
```

Table 2: Cipher Message Structure

a cipher message, the server shall use its secret key to compute a tag from `commitment`, and then write the tag to the `tag` field. The following is our sample code.

```
function signMessage(cipher) {
    var com = getCommitment(cipher);
    return calcHMAC(secretKey, com).then(function (mac) {
        cipher.tag = mac;
        return cipher;
    });
}
```

To report a message, the client of Receiver has to store `tag` in the cipher message, as well as `Evidence` returned from `decrypt`. In the following sample code, the server accepts directly the structure of `Evidence` and the `tag`, and then verifies both `tag` and the `commitment` in `Evidence`. Ideally, such server should be part of the protocol even though it is not included in client-side library `libsingal-protocol`. Also note that a client application shall maintain every pair of `evidence, tag` rather than storing data inside `libsingal-protocol`.

```
function reportAbuse(evidence, tag){
    var macInput = new Uint8Array(evidence.header.byteLength + evidence.body.byteLength);
    macInput.set(new Uint8Array(evidence.header));
    macInput.set(new Uint8Array(evidence.body), evidence.header.byteLength);
    return Promise.all([
        verifyMAC(evidence.commitment, secretKey, tag, 32),
        verifyMAC(macInput, evidence.commitKey, evidence.commitment, 32)
    ]);
}
```

2.2 Usage

The procedures to generate keys and to encrypt are identical to the original procedures, and hence we show only decryption. Compared to the original procedures, the only difference is that decryption returns a structure of Evidence, and that the tag of the server is also returned to the application.

```
var address = new libsignal.SignalProtocolAddress(sender.identifier, sender.keyId);
var sessionCipher = new libsignal.SessionCipher(rcver.store, address);
return sessionCipher.decryptPreKeyWhisperMessage(cipher.body, "binary").then(function (evidence) {
    return [evidence, cipher.tag];
});
```

To report an abused message, it suffices to send `evidence, cipher.tag` to Server. It is straightforward and omitted here.

3 Performance

In this section, we test the running time of modified implementation of encryption and decryption algorithm on client, and the signing time of server for different sizes of messages. By comparison with the running time of encryption and decryption in the original implementation, we claim that adding the message franking (also called abuse report below) functionality to `libsignal` library introduces only very small and totally acceptable time overhead.

Our implementation is based on the open-source project `libsignal-protocol.js` of version v1.3.0. The original `libsignal` library only deals with the logic at client side. At the beginning, to test the performance in a more practical environment, we tried to set up the client and server using the implementation of `Signal`, the open-source messaging application based on `libsignal` protocol. However, setting up the user identity in `Signal` requires SMS message validation, which is a little bit tedious and irrelevant to our goal. So finally we choose to evaluate the performance of simulation in a widely-used browser. As the whole process is simulated in one process in browser rather than different processes in native environment, the absolute value of running time might not be of that

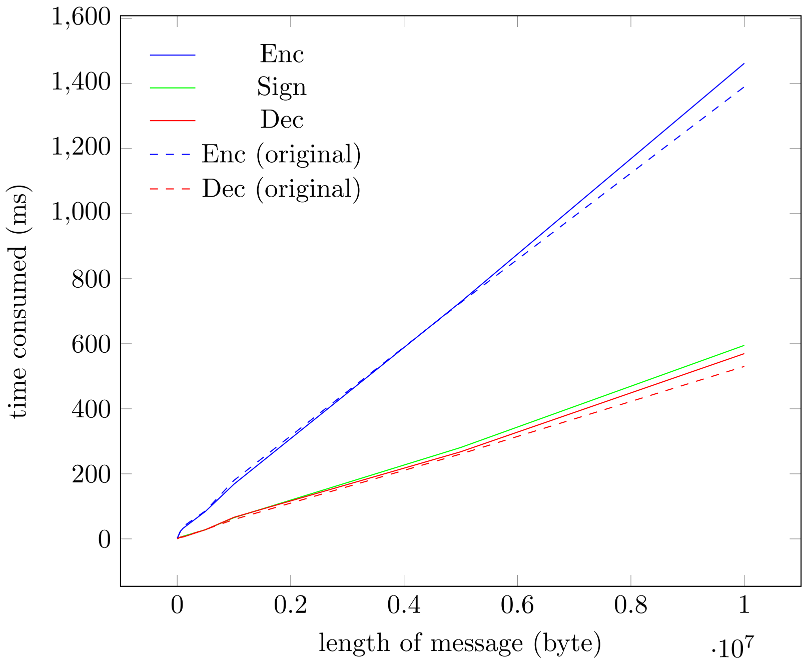

Figure 1: Simulation time of Signal with report.

valuable reference. Therefore we focus on the comparison between the time consumption of original implementation and our modification.

Test environment. We test our simulation in Google Chrome browser of version 66.0.3359.170, with embedded JavaScript of version V8 6.6.346.32. The Chrome browser runs on Windows 10 operating system, on a commodity desktop computer with Intel Core i7-6700 @ 3.40 Hz CPU and 16.0 GB RAM memory.

Although the physical RAM is large enough to handle longer message encryption, the memory space we can use is restricted by the sandbox environment of Chrome browser. In our experiment, the limitation of size of plaintext is between $50 \sim 100$ million bytes. For larger plaintext message, the page will throw error. If we implement the application (the client and the server) in local environment, we can implement the encryption for larger messages by occupying more memory. Besides, even in the browser terminal, if the user wants to send message or file of large size, we can always extend the library with large message slicing, streaming and different symmetric encryption scheme suitable for files.

To evaluate the time overhead, we choose message size from 1000 to $10,000,000$ bytes. For each data point, we run the whole process of two cases (original implementation and our modification with message franking), each for 50 times. We measure the time consumption of each encryption, signing and decryption process and calculate the average. The running results are shown in Figure 1.

From the graph, we can see that in both cases (with or without abuse report functionality), the encryption time is more than twice of decryption time. The time of calculating commitment on the sender side and the time of validating the commitment on the receiver side is around 5% of the original time consumption. Adding the abuse report functionality won't significantly impact the performance.

In addition to time overhead, adding abuse report functionality also introduces constant space overhead for each message, as the receiver client now need to store a commitment with each plaintext message.

4 Discussion

In this section, we discuss two of the major issues we encountered in the process of implementation. One is whether the participation of server in each message sending is necessary or not, the other one is which meta data the message authentication process should check. We explain the existence of these two problems and propose some preliminary ideas.

4.1 Server-in-the-Middle is Necessary

Many of the existing message franking schemes, including Facebook implementation and our work, requires that each message from the sender to the receiver be relayed by the server in the middle. It seems a little bit unsatisfactory from the privacy perspective, as the server can immediately know when and to whom the sender wants to talk. Even though the server cannot decrypt the ciphertext, it still leaks private information and might lead to higher vulnerability to side channel attacks. So we want to ask here: *can we get rid of the server between the sender and the receiver while having the good abuse report functionality?*

It looks like just a choice of design. Message relaying through the server is not really necessary to achieve the abuse report functionality. For example, each pair of user can generate public-secret key pairs, exchange own public keys, and sign the ciphertext with undeniable signature using the corresponding secret key when sending messages to each other. The server only stores the public keys of all users. If one user wants to accuse someone of abusing, it can just send the signed message to the server, and server can verify if the message is sent by the accused user.

However, all designs of this kind has an obvious drawback: the server does not know any more information about the message than any other third-party, all of the evidence are in the receiver's control. If the receiver can prove to the server that a message is indeed sent by the sender, it can also prove to any other untrusted ones. It causes privacy concern on the sender side, as the sender might not want to disclose to any third party that it has signed on some message. Actually, passing the message through the server is just choosing server as a trusted third party, as the receiver can maliciously disclose any message to the server and prove that the message is indeed sent from the sender, even if the message is not an abuse message.

Take all these into consideration, we finally choose the strategy of our current implementation: the server signs and relays the commitment of ciphertext. The only extra thing the server needs to store is its own secret key. The idea is: if I want a messaging application supports abuse reporting, I need to trust some third party; if I have to trust some third party, I choose to trust the server.

4.2 Version Number Verification

Message authentication code (MAC) is an important tool to verify the integrity of the message and the meta information. In `libsignal`, the MAC is a component of cipher message as shown in Table 2. We carefully investigate the original implementation of libsignal, find that it MACs the whole body but without the `type` field in the header of the message.

The `type` field indicates the type of the current message:

- `type` = 3: PreKeyMessage, the initial message of a new session.
- `type` = 1: any following message in a session that has already been built.

These two categories of messages are wrapped in different formats. The PreKeyMessage has an additional header including more public identities of the sender, e.g. signed public key. We argue that the `type` field should also be MAC-ed, otherwise there will be a security hole here. If the `type` is not included in the integrity check, an adversary can perform an man-in-the-middle attack by changing the `type` value. For example, changing `type` from 1 to 3 will force the receiver to decrypt a ordinary message as a PreKeyMessage. Due to the time limit, we are following the original implementation of `libsignal` now. We plan to include the type information into the MAC in the next step.

5 Conclusion

To conclude, in this work, we realize the message franking functionality to `libsignal` library based on [2]. We describe the high level idea and important details in our implementation, and also give performance analysis. The time overhead and space overhead are both minimal, indicating that we don't need to sacrifice too much performance to get the message franking scheme. The issues emerging in our implementation are also discussed and we propose our opinion on how to handle them.

For the next step, we are planning to release publicly[2] both the augmented library and the corresponding sample code. We believe our contribution will enhance our own every-day messaging softwares.

References

[1] Messenger secret conversations technical whitepaper. https://fbnewsroomus.files.wordpress.com/2016/07/secret_conversations_whitepaper-1.pdf.

[2] For the time being, we have a demo at https://github.com/RippleLeaf/libsignal-protocol-javascript.

[2] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *Advances in Cryptology – CRYPTO 2017*, Lecture Notes in Computer Science, pages 66–97. Springer, Cham.