Implementing Signal

Yi Tang, Yevgeniy Dodis
New York University

August, 2019

Abstract

The *Signal protocol* is an open source secure messaging protocol that provides end-to-end authenticated encryption with many appealing security advantages. The protocol is extensively deployed and secures the daily communication of billions of users via popular messaging applications such as Signal (originally TextSecure), WhatsApp, Google Allo, Facebook Messenger, Skype, etc.

[ACD19] formally analyzes the *double ratchet algorithm*, which is the core component of the Signal protocol. The paper proposes a decomposition of the double ratchet algorithm into multiple generic cryptographic modules. The modularization enables customization of the algorithm using different instances of the modules. As a result, if certain cryptographic structure currently used in the Signal protocol turned out to be unsafe, then it could be replaced with other safe alternatives and the security of Signal protocol could be recovered. More interestingly, the modularization naturally leads to *post-quantum* variants of the Signal protocol by employing quantum-safe module instances.

The modularization in [ACD19] decomposes the Signal protocol into four modules, namely the *Continuous Key Agreement* (CKA), *PRF-PRNG* (PRGF), PRG and *Authenticated Encryption with Associated Data* (AEAD). The paper also describes how to construct CKA from *Key Encapsulation Mechanism* (KEM) and PRGF from HKDF. Besides, there are well known constructions of HKDF from HMAC (e.g. see [KE10]), AEAD from SKE along with HMAC (e.g. see [McG08]), and PRG from PRF (trivial by definition). Following the modularization and the constructions, the implementation has a hierarchical software structure as illustrated in Figure 1. The implementation is presented as a C library.

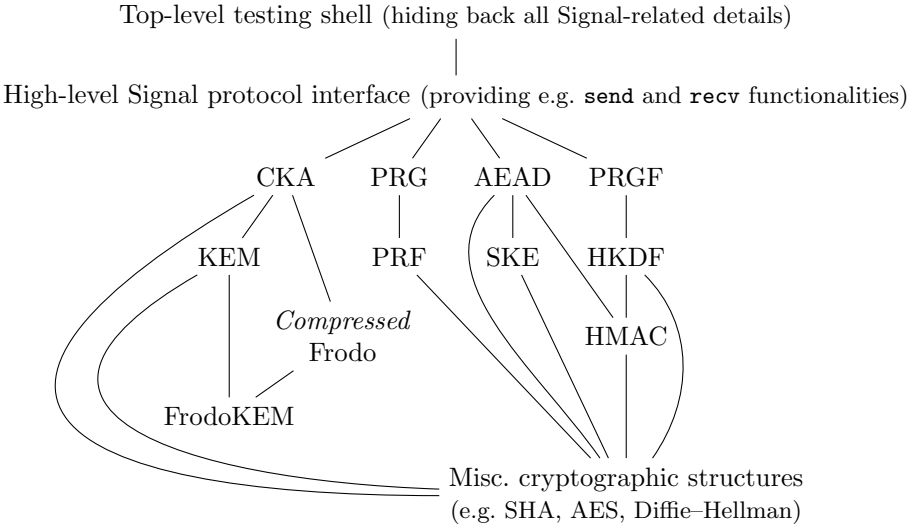

Figure 1: The software hierarchy of the implementation. *FrodoKEM* (`http://frodokem.org/`) is a post-quantum KEM scheme whose security derives from the *learning with error* problem. The miscellaneous cryptographic structures have dependency on library *Botan* (`https://botan.randombit.net/`).

To exemplify, by using the first configuration in the following list, one can reconstruct the original Signal protocol; the second configuration leads to a post-quantum variant of the Signal protocol; and the third configuration brings about an improvement to the second.

- CKA: (compressed) Diffie–Hellman with Curve25519
- PRGF: [HKDF-based]
  - HKDF: SHA-256
- PRG: [PRF-based]
  - PRF: SHA-256
- AEAD: AES-128-SIV

- CKA: [KEM-based]
  - KEM: Frodo640
- PRGF: [HKDF-based]
  - HKDF: SHA-256
- PRG: [PRF-based]
  - PRF: SHA-256
- AEAD: AES-128-SIV

- CKA: (compressed) Frodo640
- PRGF: [HKDF-based]
  - HKDF: SHA-256
- PRG: [PRF-based]
  - PRF: SHA-256
- AEAD: AES-128-SIV

Note that in the first and third configuration the CKA instances are not constructed from KEM and are referred to as some "compressed" scheme. It is known that KEM can be generically constructed from PKE, but there are often more efficient schemes if one directly construct CKA from PKE and apply certain compression. However currently a generic way to perform such compression is not yet found. In addition, it is worth mentioning that the implementation of the *compressed* Frodo module used in the third configuration is a challenging task from the perspective of coding. The FrodoKEM library is not highly modularized as it does not expect customization. Therefore in the implementation there exists a middleware FrodoKEM toolbox, which might be found useful by those about to customize FrodoKEM.

As a primitive benchmarking, the time and space performances of the three example Signal variants are compared. The results are plotted in Figure 2.

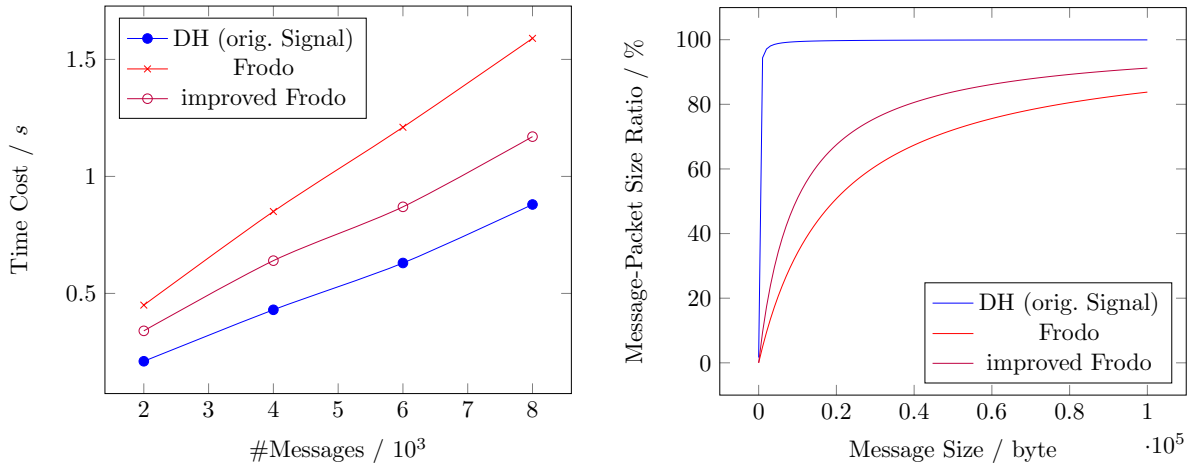

Figure 2: Left: comparison of time performances of example Signal variants under random asynchronous messaging test. Right: comparison of message-packet size ratio of example Signal variants.

The codes as well as the description of the implementation can be found at the project homepage: `https://cims.nyu.edu/~yt1433/signal.html`. Besides FrodoKEM, we are also planning to adapt other post-quantum schemes such as NewHope, CRYSTALS-Kyber and SIKE, and trying to seek their compressed versions like of Diffie–Hellman and FrodoKEM.

References

[ACD19] Joël Alwen, Sandro Coretti and Yevgeniy Dodis, "The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol", `https://cims.nyu.edu/~dodis/ps/signal.pdf`.

[KE10] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)" (RFC 5869), `https://tools.ietf.org/html/rfc5869`.

[McG08] D. McGrew, "An Interface and Algorithms for Authenticated Encryption" (RFC 5116), `https://tools.ietf.org/html/rfc5116`.

Appendix A: Code Demo

C Code for a toy *asynchronous* communication:

```
int main() {
    // initialize
    init();

    // send first message from sender A
    size_t s = send(A, "Hello World");

    // asynchronously send & receive
    size_t s1 = send(A, "message no.1, A to B");
    recv(s1);
    size_t s2 = send(B, "message no.2, B to A");
        // Note that B should receive at least one message
        // (but not necessarily the first message) before sending.
    size_t s3 = send(A, "message no.3, A to B");
    recv(s3);
    recv(s2);
    size_t s4 = send(B, "message no.4, B to A");
    recv(s);
    recv(s4);

    // cleanup
    free_all();

    return 0;
}
```

Built-in command line visualization:

```
                Alice                                   Bob

--> message 0: (1, 1 | 0) Hello World
--> message 1: (1, 2 | 0) message no.1, A to B
                                  --> decrypt 1: (1, 2 | 0) message no.1, A to B
                                  <-- message 2: (2, 1 | 0) message no.2, B to A
--> message 3: (1, 3 | 0) message no.3, A to B
                                  --> decrypt 3: (1, 3 | 0) message no.3, A to B
<-- decrypt 2: (2, 1 | 0) message no.2, B to A
                                  <-- message 4: (2, 2 | 0) message no.4, B to A
                                  --> decrypt 0: (1, 1 | 0) Hello World
<-- decrypt 4: (2, 2 | 0) message no.4, B to A
```

The module instances are specified at compile time. Note that code is totally generic as it does not depend on specific choice of module instances. And the decryption under asynchronous communication succeeds no matter what module instances are used.