CS 528 Project – Signal Secure Messaging Protocol

Raushan Pandey (pandey72@purdue.edu)
Jacob White (white570@purdue.edu)

May 6, 2022

1 Introduction

Signal [1] is an end-to-end encrypted (E2EE) and open-source messaging protocol which aims to preserve the privacy and security of communications without inherently relying on a *trusted* third party's servers to preserve the confidentiality and integrity of communications data. While Signal Messenger LLC maintains their own open-source messaging application [2] that itself uses the Signal protocol, many other E2EE messaging applications have incorporated their own closed-source implementations of the Signal protocol, including WhatsApp [3], Facebook Messenger [4], Skype [5], and Android Messages [6].

Because the Signal protocol acts as a de-facto standard for many of the secure E2EE messaging applications that are commonly used today, the security of the protocol deserves further analysis. See Sections 2 and 3 for more details regarding the cryptographic design of the core Signal protocol. As Signal protocol specifications only provide an informal discussion about security goals, design, and limitations [1], Section 4 surveys the various analyses and discoveries from the existing academic literature regarding the security of and possible extensions to the protocol, starting with [7]. While the core Signal protocol itself is theoretically proven to be secure, many implementations trade strong security protections for efficiency, however; see Sections 4 and 5 for more details for the security of Signal implementations in theory and in practice, respectively.

2 Preliminaries

2.1 Notation

For any value x, let $\mathbf{x}$ represent its byte-encoding with corresponding bit length $|x|$. Let $a \leftarrow A$ denote a uniformly random sampling of some value a from an arbitrary set A. Let p denote a sufficiently large prime order for a corresponding finite field $\mathbb{Z}_p$. For any $a \in \mathbb{Z}_p$, let a^{-1} denote the multiplicative inverse of a when $a \neq 0$, and 0 otherwise.

Let an elliptic curve E be described by a set of points $P = (x, y)$ containing coordinates $x, y \in \mathbb{Z}_p$. The elliptic curve is equipped with an addition operator over points, with additive

inverse $-P$ and identity point I. Furthermore, the elliptic curve is equipped with a scalar field $\mathbb{Z}_q$ with prime order q and base point B such that $q < p$ and $qB = I$. k-times addition of P is equivalent to scalar multiplication by $k \in \mathbb{Z}_q$, i.e., $P + P + \ldots + P = kP$ as expected.

Hereafter, assume unless otherwise specified that E is a twisted Edwards curve. See [8, 9] for more details about how twisted Edwards points P can be compressed then encoded into an equivalent b-bit point defined by a single $(b-1)$-bit coordinate y and a sign bit $s \in \{0, 1\}$. Twisted Edwards curves are *birationally equivalent* to a corresponding Montgomery curve with points $P' = (u, v)$ and associated function $\mathsf{u\_to\_y}$ mapping P' to P [8, 10].

Let H be a secure cryptographic hash function such as SHA-256 [11]. Unless specified otherwise, let public-private keypairs be denoted by e.g. (k, k^*), where k is the public key.

2.2 Cryptographic Assumptions

For the discrete logarithm problem to hold in an elliptic curve E, it should have sufficiently large embedding degree k with $k > (q-1)/100$ [12, 13]. The Signal protocol can relax these requirements somewhat, however, as it uses stronger Diffie-Hellman based assumptions:

Assumption (CDH; Computational Diffie-Hellman)**.** *Suppose that an elliptic curve E is defined over finite field $\mathbb{Z}_p$. Given the Montgomery points $P, \alpha P, \beta P$, where $\alpha, \beta \leftarrow \mathbb{Z}_q$ are independently random scalars, it is computationally difficult to compute $\alpha\beta P$.*

Assumption (DDH; Decisional Diffie-Hellman)**.** *Suppose that an elliptic curve E is defined over finite field $\mathbb{Z}_p$. Given the Montgomery points $P, \alpha P, \beta P, P'$, where $\alpha, \beta \leftarrow \mathbb{Z}_q$ are random scalars and $P' \coloneqq \alpha\beta P$, P' is computationally indistinguishable from γP where $\gamma \leftarrow \mathbb{Z}_q$.*

Assumption (GDH; Gap Diffie-Hellman)**.** *Even provided black-box access to a DDH oracle, it is computationally difficult to compute $\alpha\beta P$ when given $(\alpha P, \beta P)$.*

The strongest cryptographic assumption that the Signal protocol relies on is GDH assumption where, essentially, the DDH problem is easy but the CDH problem is hard. The proof of security relies on random oracle model (ROM), wherein all KDFs return uniformly-random outputs [14].

The Signal protocol relies on many more informal security assumptions regarding key compromise, however; see Sections 3.1 and 4 for a more detailed analysis of such assumptions.

2.3 Cryptographic Primitives

2.3.1 Elliptic curve

The Signal protocol allows the choice of either Curve25519 or Curve448 to instantiate the elliptic curve E; however, the Signal protocol must all use the same associated Diffie-Hellman function (X25519 or X448, respectively) and parameters throughout for every key generated by the protocol [15]. The twisted Edwards curve equation for Curve25519 is defined as $-x^2 + y^2 = 1 + dx^2y^2$, with curve constant d and birational map $\mathsf{u\_to\_y}(u) = (u-1) \cdot (u+1)^{-1} \bmod p$. Similarly, the twisted Edwards curve equation for Curve448 is defined as

$x^2 + y^2 = 1 + dx^2y^2$, with $\mathsf{u\_to\_y}(u) = (1 + u) \cdot (1 - u)^{-1} \bmod p$ [8, 10]. See RFC 7748 for more details regarding the secure instantiation of both of these curves [12].

2.3.2 Hash-to-point

Using a secure cryptographic hashing algorithm H such as SHA-256, it is possible to hash an integer r into a Montgomery point coordinate u which can, in turn, be converted into a twisted Edwards point P with $y \coloneqq \mathsf{u\_to\_y}(u)$ and the appropriate s bit. Finally, multiplying P by a cofactor $c \in \mathbb{Z}_q$ ensures that it lies within the correct order-q subgroup [8, 16].

2.3.3 Elliptic-curve digital signatures

As a part of its protocol, Signal defines custom digital signature schemes which are compatible with EdDSA ([17, 9]) but are non-deterministic and instead rely on X25519 and X448 [8].

At a high level, the XEdDSA digital signature scheme consists of the following algorithms:

$\mathsf{XEdDSA.Sign}(k, \mathbf{M}, \mathbf{Z}) \to (\mathbf{R} \,\|\, \mathbf{s})$ Given a Montgomery private key k, the byte sequence of message M, and 64 bytes of secure random data $\mathbf{Z}$, outputs a $2b$-bit signature where R is a twisted Edwards point and $s \in \mathbb{Z}_q$.

$\mathsf{XEdDSA.Ver}(\mathbf{u}, \mathbf{M}, (\mathbf{R} \,\|\, \mathbf{s})) \to \mathsf{T/F}$ Given the encoded Montgomery public key $\mathbf{u}$, the byte sequence of message M, and the signature, outputs whether or not the signature verifies.

The VXEdDSA signature scheme is defined similarly, except also producing a b-bit byte encoding of verifiable random function (VRF) output v:

$\mathsf{VXEdDSA.Sign}(k, \mathbf{M}, \mathbf{Z}) \to (\mathbf{V} \,\|\, \mathbf{h} \,\|\, \mathbf{s}), \mathbf{v}$ Given the same inputs as in $\mathsf{XEdDSA.Sign}$, outputs a $3b$-bit signature where V is a point and $h, s \in \mathbb{Z}_q$ encodes the VRF output v.

$\mathsf{VXEdDSA.Ver}(\mathbf{u}, \mathbf{M}, (\mathbf{V} \,\|\, \mathbf{h} \,\|\, \mathbf{s})) \to \mathbf{v/F}$ Given the same inputs as in $\mathsf{XEdDSA.Ver}$ (but with a different signature encoding), outputs $\mathbf{v}$ if the signature verifies, and F otherwise.

Unlike standard DSA, these variants—which hash the secret and unique random value Z along with the private key and message into an internal signed value r— prevents collisions on r and the derived signature $(R \,\|\, s)$ with high probability, assuming that a cryptographic hash is employed and Z is unique for each signature. Therefore, these non-deterministic variants of DSA allow for the private key k and corresponding public key u to be used in the long-term alongside the unique key Z. See [8] for more details regarding these signature schemes, including the elliptic-curve conversions and hashing to a point on the curve.

2.3.4 HMAC-based key derivation functions (HKDF) and KDF chains

As described in RFC 5869 [18], given a secure cryptographic hashing algorithm H it is possible to create a secure keyed HMAC function ([19]) which instead takes a salt value and key material as input and outputs a pseudorandom key [18].

Using HKDF as a primitive, the Signal protocol defines a *KDF chain* as a cryptographic function which, given an initial KDF key and input data, outputs pseudo-random data which

is split into the next KDF chain key and a new output key. Each KDF evaluation to generate a new output key is considered a "ratchet step". See Section 3.3 for more details.

2.3.5 Elliptic-curve Diffie-Hellman key agreement

As described in RFC 7748 [12], it is possible to use elliptic-curve Diffie-Hellman (ECDH) functions X25519 and X448 with the Diffie-Hellman protocol to agree on a shared secret key in the elliptic-curve setting. X25519 and X448 must have the properties specified in Section 2.2 for the Diffie-Hellman key agreement to remain secure.

2.3.6 Authenticated Encryption with Associated Data (AEAD)

An AEAD scheme is similar to standard authenticated-encryption (AE) schemes, except that AE schemes provide stronger privacy by both encrypting and authenticating all data; AEAD schemes contain associated data (AD) which is only authenticated along with the ciphertext [20]. Signal employs the Encrypt-then-MAC and ciphertext translation/header methods of instantiating the AEAD primitive [21], both of which are proven secure in [20].

3 The Signal Protocol

We now introduce the Signal protocol for sending end-to-end encrypted messages between two parties. Due to the complexity of analyzing the many extensions and implementations of the Signal protocol, in this section we only consider analysis of the core cryptographic protocol [1].

Without loss of generality, the Signal protocol considers Alice to be the party A initiating communication, and Bob the (perhaps offline) registered party B receiving her initial message. The initiation process is facilitated by a set of Signal servers S which are conceptually modeled as a single server for simplicity.

3.1 Security Goals and Threat Model

The Signal protocol is designed to remain secure within a fully adversarially-controlled network, where device compromise in the presence of passive adversaries is possible [22]. The intantiation was also carefully chosen to be amenable to constant-time implementations and resilient as a whole against side-channel analysis [8, 17]. Furthermore, to protect the privacy of its users, the X3DH protocol used by Signal offers cryptographic deniability (i.e. repudiation) of messages by default. However, if desired, the protocol can support non-repudiation as well [15]. Its informal security goals are as follows:

Confidentiality. Confidentiality guarantees that authorized individuals can only view messages sent between the desired parties. Further, non-intended recipients cannot view the message; this is primarily accomplished using encryption schemes.

Origin Integrity. Origin Integrity is a security goal which guarantees that when a user, Alice, receives a message from another user, Bob, Alice knows this message came from Bob; this can be accomplished by the use of cryptographic signatures over the party's keys.

Authenticity. Message Authenticity guarantees that a message between two parties has not been altered in any way by an unauthorized party; this can be accomplished by using message authentication codes (MACs) and authenticated encryption.

Forward Secrecy. Forward secrecy is a feature of key agreement protocols indicating that an adversary who compromises the message keys of a victim cannot decrypt messages from before a certain time.

Future Secrecy. (aka break-in recovery or post-compromise secrecy) Future secrecy is a much less common feature of key agreement protocols indicating that an adversary who compromises the message keys of a victim cannot decrypt messages after a certain time from the last key compromise.

(Optional) Deniability. (aka repudiation). Message repudiation relies on the encryption and authentication of message transcripts and the fact that Signal servers do not store conversation metadata. As a result, any participant can deny that they have sent a particular message.

3.2 Extended Triple Diffie-Hellman (X3DH) Key Agreement

The Extended Triple Diffie-Hellman (X3DH) key agreement protocol [15] is an extension of elliptic-curve Diffie-Hellman (ECDH) key agreement which supports mutual authentication, forward secrecy, and cryptographic deniability over a channel alongside Alice and Bob's usual agreement on a shared secret key for bidirectional communication.

Prior to engaging in the protocol, Alice holds keypairs with corresponding public keys (IK_A, EK_A), where IK is a long-term identity public key and EK_A is an ephemeral public key unique to each run of the X3DH protocol. Bob holds keypairs with corresponding public keys (IK_B, SPK_B, OPK_B), where SPK_B is a periodically-refreshed signed prekey and $OPK_B = \left\{OPK_B^{(j)}\right\}$ is a set of one-time public prekeys unique to each run of the X3DH protocol. Bob must first publish his public keys, along with prekey signature $\sigma_{SPK} \coloneqq \mathsf{Sign}(IK_B{}^*, \mathbf{SPK_B})$, to server S prior to executing the protocol.

Alice initiates by querying server S for Bob's *prekey bundle* $(IK_B, SPK_B, \sigma_{SPK}, OPK_B^{(j)})$. The server optionally provides one of Bob's one-time prekeys $OPK_B^{(j)}$ and deletes it; if no remaining one-time prekeys exist, then she receives $OPK_B^{(j)} \coloneqq \mathsf{nil}$.

After receiving Bob's prekey bundle, Alice uses the elliptic curve's Diffie-Hellman function as in RFC 7748 [12] and HKDF as in RFC 5869 [18] to calculate the session key SK as in Figure 1. $DH1$ and $DH2$ facilitate mutual authentication via long-term identity keys IK.

$DH1 = \mathsf{DH}(IK_A, SPK_B)$;
$DH2 = \mathsf{DH}(EK_A, IK_B)$;
$DH3 = \mathsf{DH}(EK_A, SPK_B)$;
if $OPK_B^{(j)} \neq \mathsf{nil}$ **then**
  $DH4 = \mathsf{DH}(EK_A, OPK_B^{(j)})$;
end
$SK = \mathsf{KDF}(DH1 \,\|\, DH2 \,\|\, DH3 \,\|\, DH4?)$;

Figure 1: The X3DH protocol.

$DH3$ and (optionally) $DH4$ facilitate forward secrecy via the Bob periodically refreshing the medium-term signed prekey SPK_B and the one-time key $OPK_B^{(j)}$, respectively.

Alice then deletes ${EK_A}^*$ and the DH outputs, calculates associated data $AD = \mathbf{IK_A} \,\|\, \mathbf{IK_B} \,\|\, \ldots$ identifying the parties, then sends Bob the initial message $(IK_A, EK_A, i_{SPK}, j, c)$ where c is an AEAD encryption of the first post-X3DH message with respect to AD and keyed by SK.

Once Bob is online, he parses Alice's initial message and looks up the corresponding private keys for the public keys that Alice identified. Bob then repeats the same ECDH and AD calculations as Alice, using the resultant SK to attempt to decrypt the initial message. If the decryption succeeds, Bob deletes ${OPK_B^{(j)}}^*$ for forward secrecy and the X3DH protocol is complete; alternatively, failed decryption indicates that mutual authentication has failed, in which case Bob should abort and delete SK.

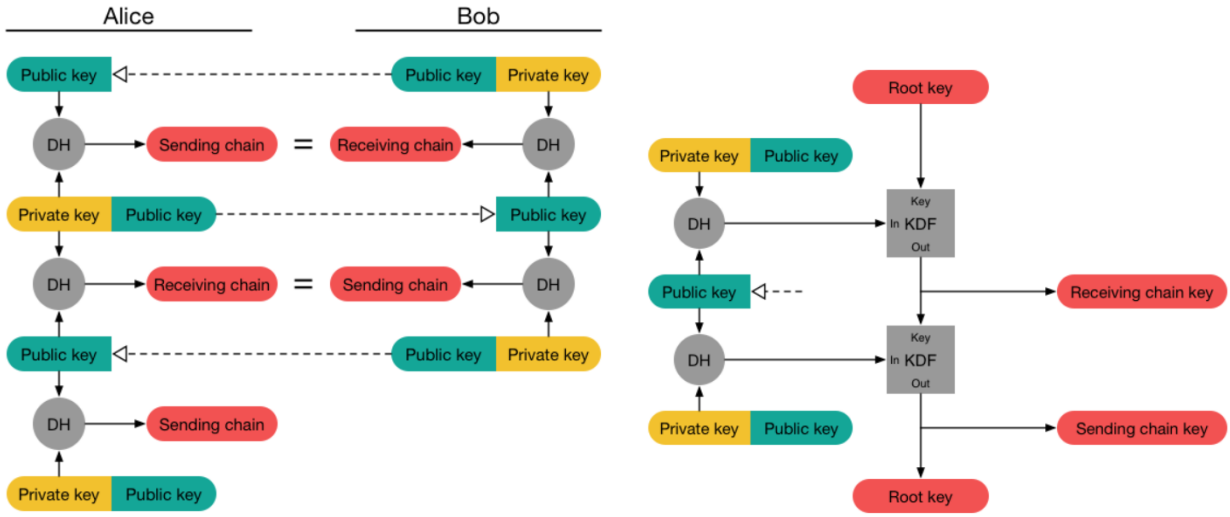
When computing AD, Alice and Bob must somehow compare their identity keys IK_A, IK_B and perhaps other identifying information such as usernames or certificates over an authenticated channel to cryptographically ensure mutual authentication; the Signal protocol leaves PKI out-of-scope. Furthermore, Alice and Bob must use the shared secret key SK in a controlled manner post-X3DH to preserve forward secrecy as well as mitigate the consequences of protocol replays and key reuse. See [15] and Section 3.3 for more details.

3.3 Double Ratchet Algorithm

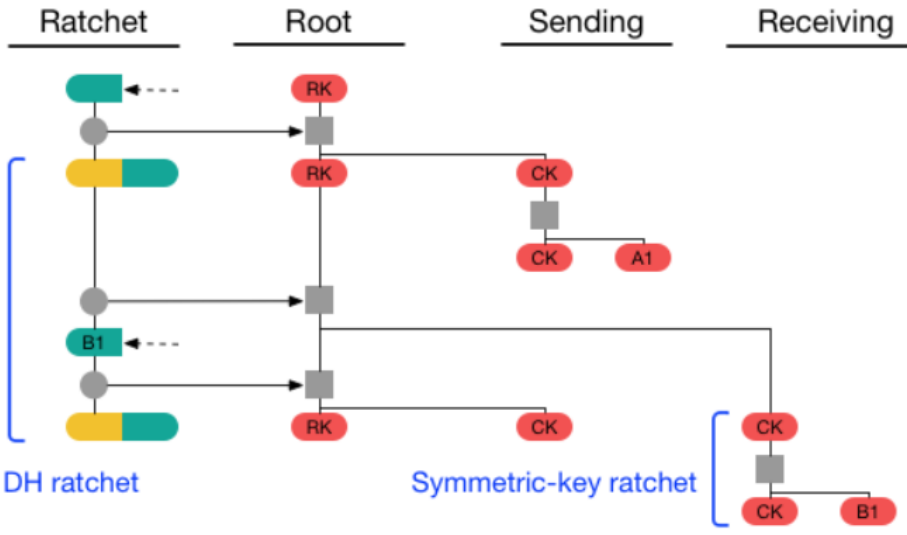
The Double Ratchet algorithm [21] uses a shared secret key SK and associated data AD to establish a session for exchanging encrypted messages while preserving forward secrecy and minimizing session key reuse. At a high level, the parties use ECDH and multiple KDF chains to reliably and asynchronously derive new symmetric keys for each message.

Initially, both Alice and Bob both initialize their *root chain* with the shared secret input $RK_0 \coloneqq SK$ and with Bob's initial keypair as $(SPK_B, {SPK_B}^*)$ in the post-X3DH instantiation. The *Diffie-Hellman ratchet* proceeds as follows, assuming Bob's view of the protocol by symmetry. Alice and Bob alternate in generating a new *ratchet keypair*. When Bob receives a new public key from Alice in the (plaintext) message header, he uses his *old* private key and her sent public key as ECDH inputs to generate a unique input to the root chain. Using this DH output, he advances the root chain to generate a new root key and initial chain

key CK_R, instantiating a new receiving chain[1]. Bob immediately proceeds by using his *new* private key and her sent public key as ECDH inputs, advancing the root chain again to generate the next root key RK' and initial sending chain key CK_S. He will publish his new public key in the header of the next message sent to Alice, which allows her to instantiate her own receiving-then-sending chain keys when she executes the next DH ratchet step. This allows the state of Alice's initial sending chain key and Bob's initial receiving chain key to be identical, and vice versa. After each of Alice and Bob have performed a single DH ratchet step, the high-level interaction with Bob's chain key computations are as follows [15]:



Now, consider Alice's perspective of the protocol by symmetry. Every time Alice receives / sends a new message with the old ratchet public key still in the header, she then performs a *symmetric ratchet step* in the receiving / sending chain, generating a new chain key CK' and message key B_i / A_j).



To mitigate lost or out-of-order messages when Alice sends the X3DH initial message and

[1]Alice excludes this first part of the DH ratchet step when she sends her initial message.

ensure that the parties' chains remain synchronized, however, Alice may need to repeatedly send that same message until she receives Bob's first response. Thereafter, through careful state management, this allows Bob's sending chains to directly correspond with Alice's receiving chains at any point in the message transcript, and vice versa. Furthermore, having message-specific session keys and regularly deleted-then-refreshed ratchet keypairs supports both forward and future security, ensuring that even after KDF key compromise, keys sufficiently far into both the past and future will still appear random. Thus, an adversary who compromises a key from the KDF chain is unable to decrypt past or future messages after the party refreshes that corresponding key. See [21] for more details regarding proper state management and reference implementation, and Section 4.3 for more details regarding the security limitations of this algorithm.

3.4 Sesame Protocol

The Sesame algorithm [22] allows for asynchronous and multi-device session management for message encryption algorithms such as the X3DH-based Double Ratchet algorithm instantiation described in Section 3.3. While not a cryptographic protocol itself, the manner in which the Signal protocol manages consistent multi-device and multi-party session states in the midst of asynchrony and device compromise nevertheless warrants discussion.

Sesame can be instantiated such that the identity keypairs (IK, IK^*), can be bound either per-user or per-device; the Signal implementation defaults to the latter. Regardless, each device stores a set of UserRecords, indexed by a correspondent's UserID. Each UserRecord may contain associated DeviceRecords, similarly indexed by unique DeviceID. The Signal server S manages an (unreliable) "mailbox" for each registered device, from which the device can fetch messages that are pending delivery. Since each session is initiated with particular identity public keys, messages can only be encrypted and decrypted from within the matching session context. Therefore, the Sesame protocol must manage how to switch between active and inactive sessions in a manner that preserves the consistency of the message transcript both between user devices and between communicating users, even after ad-hoc device registration, deletion, or desynchronization. At a high level, the Sesame algorithm operates as follows:

Updating Devices.

- If all sessions in a DeviceRecord are deleted or all DeviceRecords in a UserRecord are deleted, the record that contained them is also deleted.
- A session inserted into a DeviceRecord automatically becomes an active session.
- Activating a session requires moving the previously-active session to the head of DeviceRecord's inactive sessions list; this list can delete old sessions if it grows too large.
- Records corresponding to a deleted user or device can be marked as stale, allowing decryption of delayed messages; while Signal recommends deleting stale records after exceeding mailbox's MAXLATENCY, stale records cannot be expected to honor this.
- A device can conditionally update its $(\mathsf{UserID}, \mathsf{DeviceID}, IK)$ tuple by adding and/or replacing a new record if it does not exist or the received $IK' \neq IK$ for the associated user or device. A device does not contain its own DeviceRecord, however.
- Prior to encrypting to $(\mathsf{UserID}, \mathsf{DeviceID}, IK)$, the relevant records are deleted and up-

dated. A new and active initiating session is added if no session is currently active.

In the interests of presenting the Sesame algorithm in a more organized manner than in the Signal specifications, we have decided to reformulate the algorithm itself as pseudocode:

Sending Messages. Send encrypted message to all devices, including sender; see Figure 2.

Input: Plaintext M and recipient UserIDs (including sender)
State: Updates relevant records to current, or no changes on error ($\perp$)
Output: N/A

foreach uid $\in$ UserIDs **do**
  if *non-stale* ur $\coloneqq$ UserRecords[uid] **then**
    foreach *non-stale* (did : dr) $\in$ DeviceRecords ($\in$ ur) *with active session* s **do**
      Encrypt message C[did] $\coloneqq$ AEAD_Enc($M; AD$) with respect to s;
    end
  end
  Collect all encrypted messages C, keyed by associated DeviceIDs;
  Send (uid, C, DeviceIDs) to server S;
  /* If active uid and all current DeviceIDs, the server S adds to relevant mailboxes.
     Otherwise, S rejects with $\perp$, including any necessary updates to be made. */
  if S *accepts* **then** repeat, iterating to next uid;
  else
    if *Exceeded max # re-attempts for* uid **then** handle $\perp$;
    if uid *does not exist* **then**
      Update uid as indicated;
      Mark UserRecords[uid] as stale;
      Repeat, iterating to next uid;
    end
    else
      foreach *outdated* did **do** mark DeviceRecords[did] as stale;
      foreach *new* (did, IK) **do**
        Prepare records for (uid, did, IK) as necessary;
        Repeat with current uid;
      end
    end
  end
end

Figure 2: An algorithm for sending encrypted messages to all devices of correspondent users.

Receiving Messages. Receive encrypted message from sender via server S; see Figure 3.

Input (Mailbox): Encrypted message c and sender's $\mathsf{uid}, \mathsf{did}$ from server S
State: Updates relevant records and device session, or no changes on error ($\perp$)
Output: Decrypted plaintext M or error ($\perp$)

if *c is initial msg* $\wedge$ *(*$\mathsf{dr} \coloneqq \mathsf{DeviceRecords}[\mathsf{did}] = \emptyset \vee$ *no matching session* $s \in \mathsf{dr}$*)* **then**
    Extract new IK from header of message c;
    Prepare records for $(\mathsf{uid}, \mathsf{did}, IK)$ as necessary;
    Create a new session $s \in \mathsf{dr}$ using initial message c;
end
Decrypt message $M \coloneqq \mathsf{AEAD\_Dec}(c; AD)$ with respect to s;
Mark s as active in dr;
return M

Figure 3: An algorithm for fetching received encrypted messages for a user's device.

The manner by which Signal server(s) S deliver messages via the mailbox to specific devices is left out-of-scope. If any errors occur in parsing input, updating records, or sending or receiving messages, the device discards all state changes and the encrypted message it was processing; state updates are atomic. Furthermore, the user must bound the loop for processing updated $\mathsf{UserIDs}$ to prevent non-termination, and all lists should be appropriately bounded by a max number of entries and/or expiration. See [22] for more details regarding assumptions, design decisions, optional features, and other implementation considerations.

4 Survey of Related Work

4.1 Extensions

`zkgroup` is a recent extension to Signal which allows for private group chats in which user permissions are also managed in a completely end-to-end manner. While there exist both game-based and simulation-based security proof sketches for this extension, there is little precedent in the academic literature for what security properties a private group messaging protocol should capture [23, 24]. Furthermore, as Signal servers are often exposed to a significant amount of metadata about the messages being sent between parties, the Signal Foundation is implementing a "sealed sender" extension which minimizes metadata retention further by also hiding the communicating parties [25, 26].

There are many extensions which attempt to improve upon the security of the Signal protocol however; see the following section for more details.

4.2 Formal Security Analysis

The core Signal protocol (i.e. X3DH + Double Ratchet) has been proven theoretically secure under the GDH problem and ROM using a game-theoretical model which attempts to capture the informal design goals of Signal [7, 14]. Others in the academic literature have leveraged various automation tools as an alternative means of analyzing the security of Signal implementations, beyond simply the protocol itself [27]. Dion Van Dam [28] also

wrote a thesis detailing his process for performing both manual and automated analysis of the Signal implementation using state machines.

A different instantiation of the Signal protocol using generalized Double Ratchet and avoiding elliptic-curve and Diffie-Hellman based primitives reveals that the core Signal protocol can extend into the post-quantum setting. Fortunately, this generalization of the protocol does not even rely on the quantum variant of the random oracle model to remain secure [29].

As stated in the specifications, anyone who wishes to offer non-repudiation instead of deniability can modify the Signal protocol by requiring a third party to verify cryptographic proofs during the protocol [15]. On the other hand, if deniability is a desirable security property for a particular application of Signal, the X3DH's deniability not only formally holds, but also extends to *any* message within a Signal communication session.

4.3 Limitations

As the Signal specifications only provide very informal discussions regarding the security goals for the protocol, it is difficult to formally model and analyze security of the protocol in a way that captures its intended usage [14, 24]. Formal analyses are most often limited to the core protocol itself, and do not capture the many extensions that are used by the app in practice [14]. Furthermore, while the Signal Foundation chose to instantiate the protocol in a manner that is resilient to constant-time analysis, there exists few analyses in the academic literature verifying that this is the case [8, 17].

While Signal allows for *message* repudiation, there is no clear mechanism for preserving *session* repudiation built into the protocol. It is likely limited in the sense that it cannot offer deniability regarding session establishment [30]. It is even conjectured that communication protocols can only choose 2 of 1) online non-repudiation, 2) weak forward secrecy or 3) non-interactivity [31], implying that extending Signal must use an online third-party if non-repudiation were to be included as a feature; the strength of this conjecture, however, remains unclear.

The majority of the focus regarding Signal's security goals in the academic literature analyze its resiliency against key compromise. Identity keys are shared between both X3DH key agreement and signing prekeys, leading to potential risk regarding key reuse (although practical attack is known) [14]. One primary limitation is that the Signal protocol leaves verification and authentication of identity keys and medium-term signed prekeys to be out-of-band [14]. Furthermore, if clients do not keep a cache of ephemeral keys along with identity keys used by sender, key reuse during the X3DH protocol run is still possible when the optional one-time prekey is not used [27]. Furthermore, Bob's both long-term identity key and signed pre-key are compromised (e.g. the device itself is compromised), message authenticity is violated as an attacker is able to forge messages on behalf of Alice [27]. Out-of-order messages also leads clients to store old message keys until they arrive, and cannot be deleted. Elongating the lifetime of session-relevant keys in this manner inhibits forward security[21, 14]. These limitations as well as certain countermeasures have since been acknowledged in the Signal specifications [15].

The Signal protocol specification's choice of parameters, in alignment with RFC 7748 [12], do not closely match the models from which it has been proven secure. The likely moti-

vation for this decision is to improve efficiency of the protocol for mobile devices, and thus remain usable to general users [12, 14]. This choice of parameters and other implementation decisions may result in many Signal implementations having notably weaker future secrecy guarantees against active adversaries; in the event that a user's full state and keys are compromised and cloned for impersonation, for example, clone detection is difficult and varies by implementation [32].

5 Novel Analyses of Signal Implementation

In this section, we provide our own comparisons of the core Signal protocol with other similar E2EE messaging protocols, guided by the protocol limitations discussed in Section 4.3.

5.1 Comparison with WhatsApp

The E2EE protocol implementation used by WhatsApp is essentially a fork of the Signal protocol [3] as described in Section 3. However, there are notable differences to the implementation of the protocol used by Signal Messenger.

As a consequence of Signal's Sesame algorithm and how it handles the sending and receiving of messages to devices, a user who registers a new device under the same user ID will be unable to see prior messages sent by other devices, only syncing up message keys following the device's registration. By contrast, WhatsApp deviates from Signal's Sesame algorithm by offering more user reliability, instead regenerating all prior session keys after it detects a newly-registered device then re-encrypting all old messages using these new keys. This poses a significant security risk however; if the adversary falsely registers a compromised device under a victim user ID, they can compromise all prior keys and catastrophically violate forward secrecy. One possible attack vector which exploits this vulnerability could involve compromising the out-of-band backup keys[2] to obtain the full state of the victim device's sessions [33]. See Figure 4 for a visual demonstration of how this attack can occur.

Signal Messenger avoids this by only allowing new devices under the same user to sync on future messages, and so cloning a compromised device does not inherently compromise forward secrecy as WhatsApp does.

An adversary who obtains state from backups can violate future secrecy in this manner as well [33]. Another attack vector impacting future secrecy that applies to both WhatsApp and Signal Messenger implementations, however, involves de-synchronization issues in the underlying Double Ratchet algorithm. If the protocol cannot re-sync session states between multiple devices quickly enough, the messaging apps– and, in turn, the victims– may not notice that an adversary is using a compromised key to send and receive messages on behalf of the victim, acting in essence as a cloned device. Furthermore, they can even leverage this to impersonate as the other party in the communication session [32]. While the Sesame algorithm (or other session management protocols) are able to partially mitigate these issues through the regular deletion of keys, this exploit has been demonstrated in practice on multiple messaging applications which have implemented the Signal protocol [32].

[2]Some implementations of Signal store session backups in plaintext, including key material...

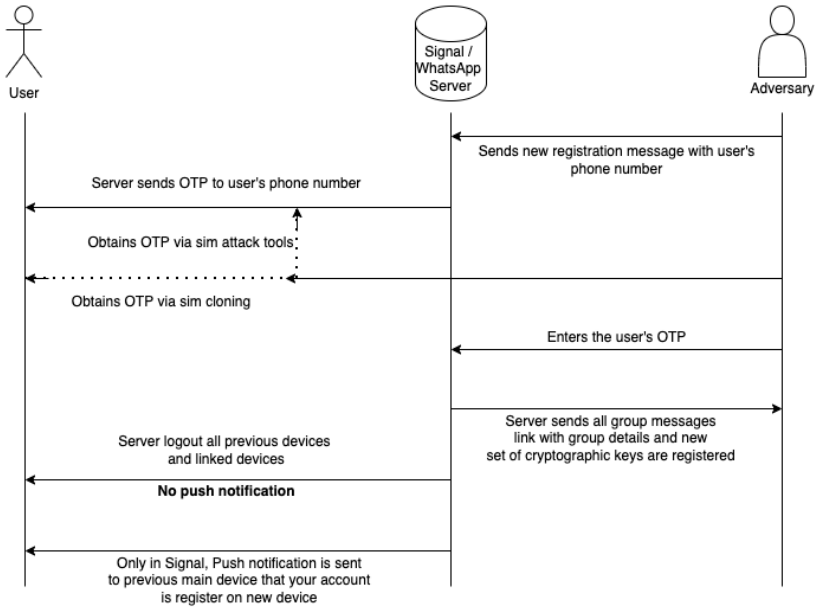


Figure 4: WhatsApp's lack of forward secrecy in the event of malicious device registration.

5.2 Comparison with MTProto

MTProto is a set of cryptographic protocols created by Telegram which are designed for implementing fast, scalable, and secure message exchange without relying on the security of an underlying transport protocol. Telegram uses these protocols to create shared keys between clients and servers and session keys between two clients for end-to-end encryption in secret chats, allow rekeying of secret chats, encrypt each message [34]. MTProto provides two different encryption protocols for cloud chats and secret chats. As of December 2017, Telegram began to phase out the MTProto 1.0 in favor of MTProto 2.0, bringing important changes such as using SHA-256 instead of SHA-1, and using 12..1024 padding bytes instead of 0..15 bytes [35].

As our interest is in the comparison of end-to-end encryption, in this section we will refer to MTProto as MTProto's secret chats, if not explicitly specified otherwise.

Cryptographic Primitives Both Signal and MTProto use 256-bit AES encryption[3] with SHA-256. Signal uses one of two elliptic curves to implement X3DH: curve X25519 or curve X448, while MTProto instead uses a 2048-bit RSA key for DH.

Forward Secrecy Signal renews the keys used for message encryption after Alice receives a message from Bob. In contrast, 'official Telegram clients' using MTProto will initiate re-keying once a key has been used to decrypt and encrypt more than 100 messages, or has been in use for more than one week provided the key has been used to encrypt at least one message [36]. In Signal, compromising the current key will only compromise a set of

[3] `libsignal` uses CBC encryption mode, while MTProto's encryption mode is unspecified [26]...

messages until the next message from the other party arrives, perhaps even none. In Telegram (MTProto), however, up to 100 messages can be recovered using a compromised key.

Asynchronous Communication To cope with the situations when users are not online at the same time, two different approaches have been developed: Signal Protocol uses a Key Distribution Server solution where servers are intermediaries who only store and relay information to let the communication be feasible and secure between the parties providing End-to-End Encryption. The encrypted messages are deleted once the device fetches it from the server. MTProto, however, relies on cloud functionality for its servers to function, where servers also compute encryptions and decryptions, storing and forwarding data to the interested users. The two different methods bring to light one of the main differences between these applications, i.e. the fact that in Telegram it is possible to access all conversations from different devices with the same account and expose the data via cloud storage. Since Signal stores all session data locally for each unique client, however, these potential features (or, from a privacy perspective, potential vulnerabilities) are not possible in Signal Messenger.

6 Conclusion

We compared and found implementation differences in the protocol specifications and between industry messaging applications like WhatsApp and Signal Messenger. We found a simple attack in which an adversary can steal user's identification just by getting OTP to register as user and then violate the privacy of the user by seeing all the group chats and members associated with that group chat. We also performed a comparison between Signal protocol and Telegram's MTProto protocol to show how both differ in terms of cryptography, implementation and security goals for the users.

While formally evaluating Signal against its informal design goals is outside the scope of this report, we believe both through our own analysis and through consensus of the academic literature that the implementation of the Signal protocol in Signal Messenger is mostly secure. However, its resilience against key compromise, while impressive in its own right, is not as foolproof as the specifications seem to imply. In either case, any implementation of the Signal protocol must carefully follow the provided specifications, and must take great care if the protocol must be modified in the interests of efficiency or usability.

Future work involves further analysis of extensions of the protocol, as well as more practical demonstrations of attacks in concrete implementations.

References

[1] Signal Messenger LLC. *Signal: Specifications*. 2016. URL: https://signal.org/docs/.

[2] Signal Messenger LLC. *Signal: Home*. 2022. URL: https://www.signal.org/#signal.

[3] Inc. Meta Platforms. *WhatsApp Security*. 2021. URL: https://www.whatsapp.com/security.

[4] Andy Greenberg. *You Can All Finally Encrypt Facebook Messenger, So Do It*. Oct. 2016. URL: https://www.wired.com/2016/10/facebook-completely-encrypted-messenger-update-now/.

[5] Microsoft Corporation. *What are Skype Private Conversations?* 2022. URL: https://support.skype.com/en/faq/FA34824/what-are-skype-private-conversations.

[6] Google. *Messages End-to-End Encryption Overview*. Tech. rep. Google, Feb. 2022, p. 12.

[7] Katriel Cohn-Gordon et al. "A Formal Security Analysis of the Signal Messaging Protocol". In: *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27.

[8] Trevor Perrin. *The XEdDSA and VXEdDSA Signature Schemes*. 2016. URL: https://signal.org/docs/specifications/xeddsa/.

[9] Daniel J. Bernstein et al. *EdDSA for more curves*. Tech. rep. 2015. URL: https://ia.cr/2015/677.

[10] Daniel J. Bernstein et al. "Twisted Edwards Curves". In: *Progress in Cryptology – AFRICACRYPT 2008*. Ed. by Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 389–405.

[11] NIST. *FIPS 180-4. Secure Hash Standard (SHS)*. Gaithersburg, MD, United States, 2012. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

[12] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. RFC 7748. Jan. 2016. DOI: 10.17487/RFC7748. URL: https://www.rfc-editor.org/info/rfc7748.

[13] Danial J. Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. Oct. 2013. URL: https://safecurves.cr.yp.to/twist.html.

[14] Katriel Cohn-Gordon et al. "A Formal Security Analysis of the Signal Messaging Protocol". In: *Journal of Cryptology* 33 (4 Oct. 2020).

[15] Moxie Marlinspike. *The X3DH Key Agreement Protocol*. Ed. by Trevor Perrin. 2016. URL: https://signal.org/docs/specifications/x3dh/.

[16] Daniel J. Bernstein et al. "Elligator: Elliptic-Curve Points Indistinguishable from Uniform Random Strings". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 967–980. DOI: 10.1145/2508859.2516734. URL: https://doi.org/10.1145/2508859.2516734.

[17] Daniel J. Bernstein et al. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* 2 (2 2012), pp. 77–89. DOI: 10.1007/s13389-012-0027-1.

[18] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://www.rfc-editor.org/info/rfc5869.

[19] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. DOI: 10.17487/RFC2104. URL: https://www.rfc-editor.org/info/rfc2104.

[20] Phillip Rogaway. "Authenticated-encryption with associated-data". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security – CCS 2002*. 2002, pp. 1–28.

[21] Moxie Marlinspike. *The Double Ratchet Algorithm*. Ed. by Trevor Perrin. 2016. URL: https://signal.org/docs/specifications/doubleratchet/.

[22] Moxie Marlinspike. *The Sesame Algorithm*. Ed. by Trevor Perrin. 2017. URL: https://signal.org/docs/specifications/sesame/.

[23] Katriel Cohn-Gordon et al. "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees". In: CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1802–1819. DOI: 10.1145/3243734.3243747. URL: https://doi.org/10.1145/3243734.3243747.

[24] Melissa Chase, Trevor Perrin, and Greg Zaverucha. "The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1445–1459. URL: https://doi.org/10.1145/3372297.3417887.

[25] jlund. *Technology preview: Sealed sender for Signal*. Oct. 2018. URL: https://signal.org/blog/sealed-sender/.

[26] Signal Messenger LLC. *Signal library*. https://github.com/signalapp/libsignal. 2022.

[27] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 2017, pp. 435–450. DOI: 10.1109/EuroSP.2017.38.

[28] Dion Van Dam. "Analysing the Signal Protocol: A manual and automated analysis of the Signal Protocol". PhD thesis. Nijmegen, Gelderland, Netherlands: Radboud University and Deloitte Netherlands, 2019.

[29] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. "The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol". In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Yuval Ishai and Vincent Rijmen. Cham: Springer International Publishing, 2019, pp. 129–158.

[30] Nihal Vatandas et al. "On the Cryptographic Deniability of the Signal Protocol". In: *Applied Cryptography and Network Security*. Ed. by Mauro Conti et al. Cham: Springer International Publishing, 2020, pp. 188–209.

[31] Nik Unger and Ian Goldberg. "Deniable Key Exchanges for Secure Messaging". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1211–1223. DOI: 10.1145/2810103.2813616. URL: https://doi.org/10.1145/2810103.2813616.

[32] Cas Cremers et al. "Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice". In: *Proceedings of the 2020 ACM SIGSAC Conference on Com-*

puter and Communications Security. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1481–1495. URL: https://doi.org/10.1145/3372297.3423354.

[33] Katriel Cohn-Gordon and Cas Cremers. *Mind the Gap: Where Provable Security and Real-World Messaging Don't Quite Meet*. Cryptology ePrint Archive, Report 2017/982. https://ia.cr/2017/982. 2017.

[34] Marino Miculan and Nicola Vitacolonna. "Automated Symbolic Verification of Telegram's MTProto 2.0". In: *CoRR* abs/2012.03141 (2020). arXiv: 2012.03141. URL: https://arxiv.org/abs/2012.03141.

[35] Telegram Messenger LLC. *MTProto Mobile Protocol*. 2020. URL: https://core.telegram.org/%20mtproto.

[36] Telegram Messenger LLC. *Perfect Forward Secrecy*. 2020. URL: https://core.telegram.org/api/end-to-end/pfs.