

# Prekey Pogo: Investigating Security and Privacy Issues in WhatsApp’s Handshake Mechanism

Gabriel K. Gegenhuber<sup>1,2</sup>, Philipp É. Frenzel<sup>3</sup>, Maximilian Günther<sup>1</sup>, and Aljosha Judmayer<sup>1</sup>

<sup>1</sup>University of Vienna, Faculty of Computer Science

<sup>2</sup>UniVie Doctoral School Computer Science

<sup>3</sup>SBA Research

## Abstract

WhatsApp, the world’s largest messaging application, uses a version of the Signal protocol to provide end-to-end encryption (E2EE) with strong security guarantees, including Perfect Forward Secrecy (PFS). To ensure PFS right from the start of a new conversation—even when the recipient is offline—a stash of ephemeral (one-time) prekeys must be stored on a server. While the critical role of these one-time prekeys in achieving PFS has been outlined in the Signal specification, we are the first to demonstrate a targeted depletion attack against them on individual WhatsApp user devices. Our findings not only reveal an attack that can degrade PFS for certain messages, but also expose inherent privacy risks and serious availability implications arising from the refilling and distribution procedure essential for this security mechanism.

## 1 Introduction

WhatsApp is the world’s largest messaging application, with more than 3 billion users worldwide [31]. Under the hood, WhatsApp uses its own version of the Signal protocol for end-to-end encryption (E2EE) of messages [2].

The Signal protocol suite consists of several different protocols [19–22, 27] which together form one of the best end-to-end encrypted communication options available to end users today. Parts of the protocol suite have also been formally analyzed and proven secure in their respective security models [3, 6, 9, 10, 12]. Nonetheless, it remains crucial to continuously analyze protocols in their entirety—including their real-world composition and implementations—to uncover and test new attack strategies, identify real-world limitations, and enhance them accordingly. For our research, we depleted the *ephemeral prekeys* (also *one-time prekeys*) of our test accounts to analyze attacks on *perfect forward secrecy* (PFS) and to highlight novel privacy and availability implications arising from the current replenishing and distribution mechanisms for such prekey bundles.

The importance of one-time prekeys for the PFS of initial messages has already been noted in the specification of

Signal’s X3DH protocol [22]:

*“This reduction in initial forward secrecy could also happen if one party maliciously drains another party’s one-time prekeys, so the server should attempt to prevent this, e.g. with rate limits on fetching prekey bundles.”*

To the best of our knowledge, we are not only the first to test this concrete attack against forward secrecy, but also the first to analyze its feasibility and the general implications of this feature regarding the privacy of users. Hereby, we not only show that WhatsApp currently does not employ any rate limiting on fetching prekey bundles of participants, but also highlight that the lack of a detailed specification on how to handle and replenish ephemeral one-time prekeys, allows for device fingerprinting and gives away the online status of the targeted device. Moreover, extensively querying prekey bundles for a targeted account may cause errors, potentially preventing the retrieval of *any* prekey bundle for that account (even without one-time prekeys). As a result, no one would be able to establish new chat sessions with the victim, leading to an availability issue. While PFS is undoubtedly affected as well, we consider the real world confidentiality impact of this attack to be modest. This is due to the careful design and a clever defense-in-depth strategy of the Signal protocol suite. After being able to circumvent the noise protocol [26], an attacker would still need to get his hands on the long- and medium-term keys of a victim to exploit the lack of forward secrecy and decrypt the previously recorded messages. Even without one-time prekeys, the “self-healing” properties of the double ratchet [27] restore forward secrecy after the first round trip. Nevertheless, the attack on PFS shows that the strong claim from the WhatsApp whitepaper, would at least require a footnote that this currently might not hold for all messages:

*“Due to the ephemeral nature of the cryptographic keys, even in a situation where the current encryption keys from a user’s device are physically compromised, they cannot be used to decrypt previously transmitted messages.” [2]*

## 1.1 Threat Model

We consider two different attack models: A *PFS attack model* where the attacker is assumed to have far-reaching capabilities, as well as a *privacy and availability attack model* where the sole requirement for the attacker is having a WhatsApp account and the phone number of the target.

### 1.1.1 PFS Attack Model

The goal of the attacker in this case is as follows:

- (G1) **PFS Degradation and Exploitation:** Force Bob’s communication partners to send initial messages without forward secrecy and exploit this by recording the respective messages for possible decryption later on after Bob’s long- and medium- term secret keys have been compromised.

**Attacker Capabilities.** In this attack, we assume a passive attacker that has access to WhatsApp’s data center internal network communication, s.t., they are able to gain access to end-to-end encrypted messages of WhatsApp users. In other words, the attacker is able to strip the first layer of transport encryption – usually provided by TLS or the Noise protocol framework between the client and the server [26] – on top of the E2EE communication used in the Signal protocol suite. This is not an unrealistic scenario if a nation state actor, or compromised WhatsApp inter-server communication (e.g., by an internal employee), is considered. Moreover, minimizing the trust in the operator of the servers (i.e., WhatsApp in this case), is an explicit design goal of the Signal protocol family [8]. It is therefore reasonable to assume that an attacker could, under certain circumstances, gain access to end-to-end encrypted (E2EE) messages.

Moreover, we require the attacker to know the phone number of the target user, as well as a WhatsApp account.

**Prekey Depletion.** Under these assumptions, we describe the prekey depletion attack on a user Bob, in which the PFS of initial messages sent to him is violated by constantly depleting the ephemeral (one-time) prekeys, usually automatically deposited by Bob on the WhatsApp servers. We assume Bob to be a security-cautious user with a very strict deletion policy regarding messages (i.e., disappearing messages set to the lowest value; currently 24h). Therefore, a compromise of his long- and medium-term security keys usually would not lead to a compromise of message content, if PFS guarantees hold, as the plaintext of messages would no longer be available on his device.

In our attack, the attacker (Eve) with passive access to the end-to-end encrypted messages, tries to constantly deplete the ephemeral prekeys of a user Bob, s.t. a new session with Bob initiated by another user Alice will have no PFS for the initial messages sent from Alice to Bob. Note that it is not uncommon, even for long-term communication partners, to

create new sessions with each other. This is mainly due to the increasing popularity of WhatsApp Web, which usually results in more short-lived browser sessions. An active already established, smartphone app session between both communication partners will not be directly affected by this attack though.

Note that this paper focuses exclusively on point-to-point communication and does not consider group messaging scenarios.

### 1.1.2 Privacy and Availability Attack Model

In this case the attacker Eve is not interested in uncovering the content of Bobs conversations, but in gathering information about’s Bob behavior and his devices and in denying that any account can communicate with Bob via WhatsApp at all. Therefore, the goals in this case are as follows:

Although the objectives of the two attack models differ, both exploit the same underlying prekey mechanisms and functionalities. As a result, the attacks are closely related and interconnected, underscoring the inherent privacy and security challenges associated with this mechanism.

- (G2) **Device Status Tracking:** Monitoring the online/offline status and activity phases (active use vs. standby) of Bob’s device(s).
- (G3) **Fingerprinting:** Gather information about Bob’s operating system (Android, iOS, macOS, Windows), their device’s age and the number of (new) interactions within a given time span.
- (G4) **Denial of Service:** Deny any other account to establish a new communication session via WhatsApp with Bob.

**Attacker Capabilities.** For this attack model, the only requirements for the attacker are knowing the telephone number of the target and having a WhatsApp account.

**Prekey Side-channel and Refills.** To achieve their goals, the attacker inspects subtle differences in the way the victim pushes fresh prekeys to the server.

## 2 Background

This section should provide a high level overview of the necessary protocol aspects of the Signal protocol. For more details we refer to Appendix B.

In this paper we target to the current WhatsApp adaption of the Signal protocol(s) described in their Whitepaper [2], but since the official WhatsApp client software is not open source the exact implementation of the protocols is not easily obtainable. The origins of the Signal protocol, date back

Keys	Description
$ipk^A$	Long-term identity key pair of Alice
$ipk^B$	Long-term identity key pair of Bob
$prepk^A$	Medium-term prekey pair of Alice, aka. <i>signed prekey</i>
$prepk^B$	Medium-term prekey pair of Bob, aka. <i>signed prekey</i>
$eprepk_n^A$	Short-term prekey pair number $n$ of Alice, aka. <i>ephemeral prekey</i> or <i>one-time prekey</i>
$eprepk_n^B$	Short-term prekey pair number $n$ of Bob, aka. <i>ephemeral prekey</i> or <i>one-time prekey</i>
$\langle ipk^A, prepk^A, \text{Sig}(ik^A, prepk^A), [eprepk_n^A] \rangle$	A prekey bundle deposited by Alice on the server
$\langle ipk^B, prepk^B, \text{Sig}(ik^B, prepk^B), [eprepk_n^B] \rangle$	A prekey bundle deposited by Bob on the server

Table 1: Main cryptographic keys of the signal protocol relevant for our attacks. Public keys in asymmetric schemes always end in  $pk$ . The naming convention of the keying material is according to Cohn-Gordon et al. [9]. The ephemeral prekeys (one-time prekeys), which are considered optional in the prekey bundle, are depicted in **red**. This is the key type drained in our attack. The secret keys of Bob which an attacker has to compromise to benefit from the violation of forward secrecy, i.e., if no ephemeral prekeys can be used, are depicted in **orange**.

to the messaging App *TextSecure*, started in 2010, which introduced a *double ratchet* construction<sup>1</sup> where communicating parties derive new keys for sending and receiving messages. The Signal protocol actually consists of an entire family of protocols [19–22, 27] which been studied in a variety of works [3, 6, 9–12, 32].

In this paper we focus on the desired (*perfect*) *forward secrecy* (PFS) guarantees of the handshake protocol and the practical implications regarding privacy to ensure it. NIST defines perfect forward secrecy, or just *forward secrecy* for short, as follows:

”Forward Secrecy (FS): Assurance obtained by one party in a key-agreement transaction that the keying material derived during that transaction is secure against the future compromise of the static private key-agreement keys (if any) of the participants.” [25]

The Signal protocol uses three different types of Diffie-Hellman public keys to ensure forward secrecy right from the start: Long-term identity keys, medium-term (signed) prekeys and short-term (one-time) ephemeral prekeys (see Table 1 for an overview). In our scenario Alice is the *initiator* and wants to establish a secure connection with Bob (the *responder*). The Signal protocol, as also implemented by WhatsApp, uses prekey bundles deposited by every user at a central server to allow any *initiator* to negotiate a shared secret (via Diffie-Hellman Key exchange) even if the *responder* is currently not online/available using the X3DH protocol [22]<sup>2</sup>.

The initial handshake works as depicted in Figure 1 and starting with Formula 1. For more details we refer the reader to the Appendix B. First, the prekey bundle of the responder (in our case Bob) is fetched from the server by the initiator (in

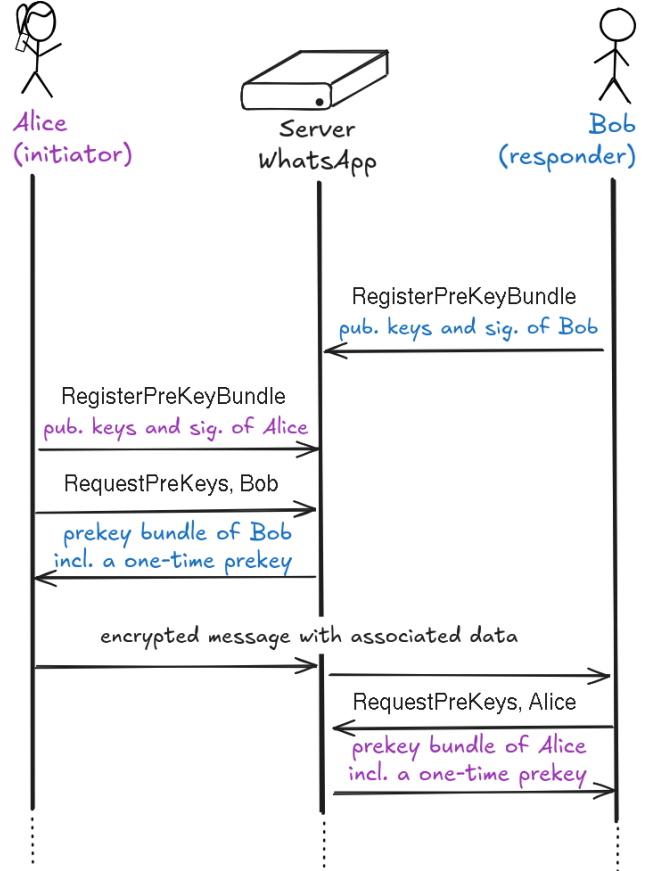


Figure 1: High-level overview of the intended prekey bundle data deposit and retrieval process, illustrating the ideal case in which everything functions as expected (i.e., prekey bundle of Bob contains also a one-time prekey).

<sup>1</sup> Initially referred to as *Axolotl Ratchet* to emphasize the self-healing properties of the protocol.

<sup>2</sup> Since late 2023 Signal replaced X3DH by PQXDH [19]. On a high level, PQXDH is comparable to X3DH with the difference that an additional key from a post quantum key encapsulation mechanism (KEM) is used.

our case Alice). The information from the prekey bundle is verified by Alice through checking the signature on the *signed prekey* using the (long-term) identity public key of Bob. As within other works [9], it is assumed that Alice has already verified out-of-band that the long-term identity public key indeed belongs to Bob.

Then the public keys from Bob’s prekey bundle are used to compute shared keys for the ratcheting and message encryption and authentication. Here now we have to distinguish between the case where an ephemeral (one-time) prekey  $eprek^B$  of Bob is available or not. If no ephemeral prekey is available, the DH invocation  $dh_4$  in Formula 5 is omitted. Since this is the key type drained in our attack it is colored in red.

In any case, before initiating the session and sending the first message to Bob, Alice generates ephemeral keys including a ephemeral key pair  $(ek^A, epk^A)$ . Those are also used in the initial handshake and to initialize the DH ratchet construction, also referred to as the *asymmetric ratchet*.

$$dh_1 \leftarrow DH(ik^A, prepk^B) \quad (1)$$

$$dh_2 \leftarrow DH(ek^A, ipk^B) \quad (2)$$

$$dh_3 \leftarrow DH(ek^A, prepk^B) \quad (3)$$

$$[dh_4 \leftarrow DH(ek^A, eprek^B)] \quad (4)$$

$$rk_0 \leftarrow KDF_r(dh_1 || dh_2 || dh_3 || [DH4]) \quad (5)$$

⋮

At the end, a new message key  $mk$  is derived using a key derivation function (KDF). This message key is then used to encrypt and authenticate (AE) the first chat *message* from Alice to Bob, as well as to authenticate some associated data  $AD$  consisting of the ephemeral public keys generated previously by Alice. This is shown in Formula 6 and 7.

$$AD \leftarrow \langle \text{ephemeral public keys of Alice,} \quad (6)$$

$$\text{id of } prepk^B, [eprek^B] \rangle$$

$$AE_{mk}, AD \leftarrow E(mk, message, AD) \quad (7)$$

Once Bob receives this initial message, he can compute the same shared keys using his identity key  $ik^B$  and his prekey  $prek^B$ , as well as the public keys of Alice consisting of her identity key  $ipk^A$  (Formular 8) and her ephemeral handshake public keys (Formular 9 and 10). Since the later are transmitted in the associated data  $AD$ , they are authenticated, but not encrypted. Tho benefit from the PFS violation (due to missing one-time prekys) and successfully decrypt recorded E2EE messages, the orange keys have to be compromised by the attacker later on.

$$dh_1 = DH(ipk^A, prek^B) \quad (8)$$

$$dh_2 = DH(epk^A, ik^B) \quad (9)$$

$$dh_3 = DH(epk^A, prek^B) \quad (10)$$

$$rk_0 \leftarrow KDF_r(dh_1 || dh_2 || dh_3) \quad (11)$$

⋮

The received message is then decrypted using the previously computed shared message key  $mk$ :

$$message, AD \leftarrow D(mk, AE_{mk}, AD) \quad (12)$$

If no ephemeral prekeys have been fetched by Alice, this initial message sent by Alice has no forward secrecy if observed by an attacker. Therefore, an attacker who is able to compromise Bobs medium-term and long-term secret keys  $prek^B$  and  $ik^B$  later on, can recompute the same message key  $mk$ , which highlights that there is no forward secrecy for this message. If Alice sends multiple messages before receiving any response from Bob, all these messages are affected as well, as the keys for these messages come from the symmetric ratchet. Forward secrecy is restored through the asymmetric ratchet, when Bob responds to a message. As soon as these ephemeral ratchet keys are deleted, forward secrecy is regained for this as well as subsequent messages. Note, that even if forward secrecy is regained in a chat session, the initial messages sent from Alice to Bob have been encrypted using the symmetric ratchet only. Therefore, they have no forward secrecy for the entire lifetime of the signed prekey  $prek^B$  of Bob. According to the specifications, the signed prekey should be periodically rotated [2, 22, 23], where suggested intervals reach from once a week to once a month<sup>3</sup>. WhatsApp refreshes signed prekeys usually once every month. To the best of our knowledge, the open source implementations *whatsmeow*, *baileys* and *cobalt* never replace the initially uploaded signed prekey, which of course would increase the impact of a loss in forward secrecy.

### 3 Testing Environment

To effectively test WhatsApp’s session and encryption procedures, we set up a test and experimentation environment which we describe in this section. To capture and understand low-level protocol messages, we base our work on existing community projects (i.e., unofficial WhatsApp clients that are based on reverse-engineering of the official implementation).

We’ve used the community projects that are shown in Table 2 to dynamically send and inspect requests from our own

<sup>3</sup>In practice Signal rotates the signed prekey every two days <https://github.com/signalapp/Signal-Android/blob/481dc162d80292a046b4229cceb2ac2f2a73f36/app/src/main/java/org/thoughtcrime/securesms/jobs/PreKeysSyncJob.kt#L57-L66>



Project	GitHub Stars	Lines of Code	Project Scope
Baileys	4,856	134,052	Emulating WhatsApp Web (companion) devices
whatsmeow	2,549	67,088	Emulating WhatsApp Web (companion) devices
Cobalt	708	41,115	Emulating main (Android/iOS) and WhatsApp Web (companion) devices
CobaltAnalyzer	37	331	Capturing decrypted traffic of legitimate WhatsApp Web browser sessions

Table 2: Relevant WhatsApp community projects and their offered features that we used throughout our analysis.

WhatsApp devices (Android, iOS, Web, Desktop). Furthermore, we wrote a custom client that allows querying the existing WhatsApp devices and their cryptographic keys (*ipk*, *prepk*, *eprepk*) for an arbitrary telephone number.

### 3.1 Relevant Endpoints and Message Structs

To uniquely identify and address users within the messaging service, WhatsApp uses so-called JIDs (Jabber ID), following a specific addressing scheme: `<phoneNumber>@<serverName>`.

Every device that is registered for a specific phone number, gets their own *device ID*. The device ID is an auto-incrementing index for each user, that is reset whenever the user sets up WhatsApp on their phone. Device 0 always represents the main device (i.e., the smartphone), while non-zero device IDs are used for companion devices (i.e., web- or desktop clients). To address specific devices within a JID, the device ID is encoded between the phone number and the server name: `<phoneNumber>:<deviceId>@<serverName>`. For example, `123456789:1@s.whatsapp.net` represents the first companion device that is registered for the US-based phone number +123456789.

Using our custom client, we can fetch the available device IDs for an arbitrary phone number:

```
pogo@prekey: $ ./query-devices -t 123456789
Querying registered devices for target number.

Found 3 existing devices: [0, 1, 3]
```

In this case, the target has one main device (index 0) and two companion devices (index 1 and 3). Since no device with index 2 is available in this list, we can deduce that there has been a linked device (e.g., a desktop computer or WhatsApp web session) with index 2, which has been logged out (unlinked). At some later point a new device has been linked, which due to the auto-incrementing nature of the device IDs, now has index 3.

WhatsApp’s encryption scheme requires the message sender to individually encrypt and send messages for every device of the recipient. Thus, the sender can query a users’ current device list from the server via so-called *usync* infoqueries. WhatsApp also allows sending messages to new contacts (or unknown phone numbers), thus this endpoint

is not only limited to known contacts, but can also be queried for external numbers. Using our testing client, we can retrieve (and consistently monitor) the registered devices and their corresponding device IDs for arbitrary phone numbers as already demonstrated in [15].

Besides knowing the recipient’s device list, the sender also needs to retrieve each device’s DH keys, to individually encrypt the message for every target device. Again, the corresponding *inforquery* can be issued to retrieve a *prekey bundle* for an arbitrary phone number. Listing 1 shows an example for the information that is returned by this query. In summary, the endpoint returns,

- the three DH public keys (*ipk*, *prepk*, *eprepk*).
- their corresponding *key IDs* (registration ID, signed prekey ID and one-time prekey ID).
- the signature of the signed prekey *prepk*.
- an epoch timestamp indicating when the device last updated the relevant object (i.e., pushed a new *prepk* or *eprepk* to the server).

The *key ID* of the signed prekeys and the *key ID* of the one-time prekeys are also incremented with every new key (of the respective type) uploaded to the server, but these IDs do not always start with zero (for more information we refer to Section 4.3).

While the device’s long-term (static) identity key and the medium-term signed prekey typically remain unchanged across subsequent queries, the included one-time prekey changes with each query. As the name suggests, this prekey is intended for single use and is therefore discarded from the server after being disclosed to a third party. In practice, the endpoint is not called very often from a single account, as a device’s prekey bundle only needs to be retrieved from the server when initiating a new session. For active sessions, the key material is continuously renewed and embedded within ongoing direct messages between the communicating parties. However, as we show, a malicious actor can deliberately request multiple prekey bundles from the server, effectively draining a device’s one-time prekey reserve that is saved at the server.

<sup>3</sup>Baileys: [github.com/Whiskeysockets/Baileys](https://github.com/Whiskeysockets/Baileys)  
whatsmeow: [github.com/tulir/whatsmeow](https://github.com/tulir/whatsmeow)  
Cobalt: [github.com/Auties00/Cobalt](https://github.com/Auties00/Cobalt)  
CobaltAnalyzer: [github.com/Auties00/CobaltAnalyzer](https://github.com/Auties00/CobaltAnalyzer)

Listing 1: WhatsApp prekey bundle containing identity key, signed prekey and a single one-time prekey, queriably for arbitrary phone numbers. The values for the byte arrays are shortened due to the limited space.

```
{
  "jid": "123456789:1@s.whatsapp.net",
  "t": "1740182155" // epoch timestamp
  "registration": "000005DB",
  "type": "05", // key type (djb)
  "identity": "76..77", // 32 bytes pubkey
  "skey": {
    "id": "000001", // signed prekey
    "value": "44...6a", // 32 bytes pubkey
    "signature": "0d..02" // 64 bytes
  },
  "key": {
    "id": "0001a", // one-time prekey
    "value": "0e..0b" // 32 bytes pubkey
  }
}
```

## 3.2 Testing Methodology

To assess whether an attacker can deplete the saved prekeys of a specific target device and to compare official implementations and the impact across different device categories, we systematically retrieve and analyze the returned key values in various settings and scenarios.

### 3.2.1 Server-side Prekey Output and Rate Limits

In our custom client, we’ve implemented a function that queries the prekeys for a specific target device repeatedly. By targeting our own devices, we want to investigate whether there are server-side protections or rate-limiting mechanisms, stopping an attacker from doing so. Furthermore, we want to test how fast an attacker is able to consume prekeys and whether consistent prekey depletion is potentially feasible. Finally, we check whether the server properly removes the prekeys, or whether certain keys are accidentally returned more than once.

### 3.2.2 Client-side Prekey Input and Push Behavior

According to the protocol design, client devices push new prekeys to the server whenever necessary. We want to investigate how many prekeys are typically saved on the server and under which circumstances they’re refilled.

**Prekey Reserve Batches.** Our analysis covers different device types (Android, iOS, Windows, macOS, Web) and different prekey states (initial reserve vs. prekey refills). To measure the amount of prekeys that were pushed in different client states, we consume (and count) all available prekeys for our target device using our custom client. To remove additional







Device	Standby		Screen On	
	WiFi	4G	WiFi	4G
 iPhone SE	85%	94%	90%	93%
 iPhone 8	90%	88%	89%	88%
 iPhone 11	80%	96%	74%	80%
 Poco X3	76%	55%	18%	17%
 Galaxy A54	10%	9%	18%	4%
 Redmi 10	15%	72%	13%	19%

Table 3: Success rate among different devices, i.e., how likely it is that a prekey bundle fetched for a new session will *not* contain a one-time prekey even if the target is currently online and powered on and an attacker is actively depleting its one-time prekeys.

noise during the measurement and to prevent the target device from immediately refilling the prekey buffer, we put it into flight mode.

**Prekey Refill Trigger.** After measuring typical prekey batch sizes, we repeat the above experiment without putting the target device into flight mode. A prekey refill also updates the epoch timestamp that is sent alongside the prekey bundle infoquery response (cf. Listing 1). Thereby, we can also observe when the last prekey refill mechanism was triggered for the target device.

**Fingerprinting Device Types and Activity States.** Besides identifying general conditions that trigger a prekey refill, we want to investigate whether different device types or activity states (e.g., standby) influence how fast the prekey reserve is refilled.

### 3.2.3 Exploration and Exploitation

After understanding the design decisions of the official WhatsApp clients and potential rate-limits on the server, we outline various exploitation scenarios. To test and verify them in practice, we use our custom client to target our own testing devices. A detailed list of the used devices can be found in Section 5 in the Appendix.

## 4 Results and Exploitation

Our tests showed that depleting a target’s prekeys is possible and that WhatsApp currently employs little to no countermeasures against it. We furthermore show our detailed results and outline the abuse potential and specific security- and privacy implications.

### 4.1 Perfect Forward Secrecy Degradation

For this section, we consider the *PFS attack model* described in Section 1.1.1, so the goal  $\text{G1}$  of the attack is it to de-

grade PFS for new sessions initiated with Bob by other users (such as Alice). Therefore, Eve wants to deplete all one-time prekeys of Bob.

For our first attack we consider the simplest case, where the targeted device is currently offline and thus cannot refill depleted one-time prekeys at the moment. This attack is depicted in Figure 2. Our tests showed, since WhatsApp does not enforce any rate limiting, constantly querying prekey bundles for any user is possible, thereby eventually depleting all available one-time prekeys. Although one-time prekeys are crucial for ensuring PFS, messages can also be transmitted when only identity and signed prekey are available to generate a shared secret (i.e., when no one-time prekeys are available on the server). In our tests, this PFS degradation is *not* indicated to the users UI (e.g. by a security notification in the WhatsApp client initiating the session). Therefore, the PFS degradation attack can be executed without the affected users noticing in their UI. Note that even if Bob’s device comes online after Alice has initiated a new communication session without a one-time prekey, messages sent by Alice will not achieve PFS until Bob’s device responds for the first time within that session.

Now let’s consider the case where the targeted device of Bob is constantly online. In this case, it receives a notification from the server as soon as there are less than 11 one-time prekeys left. Depending on the device type and its current power saving state, it reacts upon this notification and uploads 812 new one-time prekeys to the server. For measurements on how fast depletion of one-time prekeys is possible in case the targeted device is online, as well as the detailed behavior and availability implications, see Section 4.4 and 4.5. Table 3 provides an overview of the measured success probabilities of the one-time prekey depletion attack in case the targeted device is online. In other words, it indicates the likelihood that another party (e.g., Alice) will initiate a new communication session with Bob without using a one-time prekey while our depletion attack is in progress.

## 4.2 Device Online Status Leak

For this and the remaining section we consider the *privacy and availability attack model* described in Section 1.1.2. In this section we focus on the goal  $\text{G2}$  *device status tracking*.

A device can only push fresh prekeys to the server, if it is turned on and connected to the Internet. When a device’s prekeys are drained by an attacker and the reserve on the server drops to less than 11 prekeys, the targeted device will receive a notification from the server to refill its one-time prekeys. Consequently, depending on whether or not the one-time prekeys are refilled timely, this could also leaks the current online state of this particular device: If the one-time prekeys are not refilled timely, the targeted device is probably offline. Furthermore, any refill will update the corresponding epoch timestamp of the prekey bundle (cf. Listing 1), which

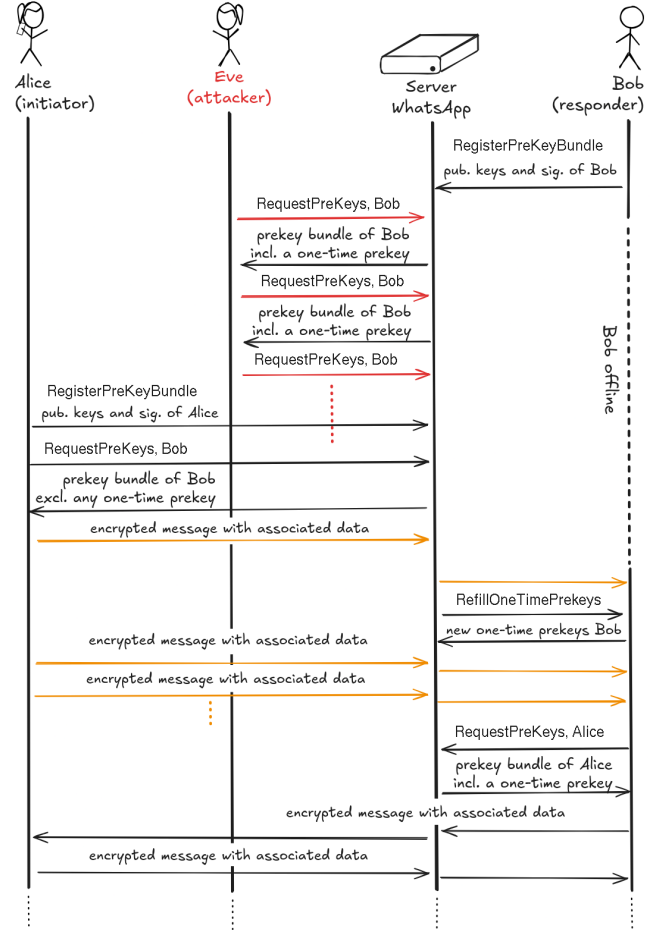


Figure 2: High-level overview of the attack. Eve is flooding the server with requests for prekey bundles of Bob. This prevents any other user, such as Alice, from obtaining a one-time prekey of Bob. Therefore, in this case all messages from Alice to Bob (orange) do not have forward secrecy, as long as the associated secret key of the signed prekey belonging to Bob is not deleted. New messages in this session after a response of Bob are not affected since new ephemeral keys are used and therefore forward secrecy is regained as soon as these new ephemeral keys are deleted.

Client Implementation	Initialization Values for <i>key IDs</i>			Prekey Batch Size		Refill Trigger
	Registration	Signed PK	One-Time PK	Initial	Refill	
Android	$R$	0	$R$	812	812	10
iPhone	$R$	$R$	1	812	812	10
WhatsApp Web <sup>a</sup>	$R$ & 0x3FFF	1	1	200	812	10
Desktop App macOS	$R$	$R$	1	200	812	10
Desktop App Windows	$R$	1	1	50	812	10

$R$  Random number. <sup>a</sup> Verified on Firefox, Chrome, Safari.

Table 4: Different initialization values, prekey batch sizes, and incrementing ID patterns used across various implementations enable device fingerprinting (e.g., OS, device age). Beyond being a privacy risk, this could be exploited by an attacker during the reconnaissance phase to tailor further attacks.

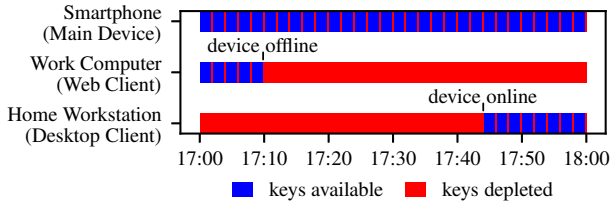


Figure 3: Each device’s online status can be independently, consistently and stealthily monitored, possibly leaking the victim’s location and daily routines.

thereby can be seen as a lower bound for the last online time of the respective device. Since this attack can be executed independently for every device of the victim, it can be used for stealthy and consistent tracking of connection states throughout the day. This is especially critical for companion devices that are usually not always online, such as desktop computers. For example, Eve could use this to track the victim’s daily routines (and thus, corresponding locations) when switching between devices, as shown in Table 3. To match devices to a specific context or location (e.g., work vs. home), the attacker could monitor the devices over a period of multiple weeks.

Main devices are usually always online and are thus less prone to this attack. Nevertheless, omission of prekey refills for extended periods could still leak information about their current activity. For example, the attack could be used to determine whether a person is currently on a flight, in a shielded building, or other locations without any Internet connection. Delayed refills could hint to blind spots in cellular reception, e.g., when driving through a tunnel. Finally, some people use the phone’s flight mode to mute all notifications during the night, disclosing a person’s sleep schedule. The next Section addresses the question, how one-time prekeys are actually refilled by different devices and what information can be gathered through this feature.

### 4.3 Device Fingerprinting

Table 4 shows characteristic *key ID* initialization values and particular prekey batch sizes used when uploading fresh keys to the server. Different client implementations use different initialization values for assigning initial key IDs. While this also holds true regarding the initial prekey batch size, all implementations use a fixed refill batch of 812 elements. Furthermore, we did not see any difference regarding the refill trigger, as all clients push a new prekey batch (812 elements) to the server as soon as it has only 10 remaining prekeys left. In detail, this is triggered by a server-side notification that is received by the client. Pushing new one-time prekeys to the server always invalidates any previous one-time prekeys. After the initial assignment, signed prekeys and one-time prekeys are issued by incrementing their IDs.

**OS Disclosure.** Finding out a target’s operating system might be a valuable information for an attacker during the reconnaissance phase to efficiently prepare for further attacks. Due to different strategies for the used initialization values and the fact that we can naturally differentiate main and companion devices by their *device IDs* (0 for main devices,  $> 0$  for companion devices), we can distinguish the used client implementation and runtime environment for an arbitrary device with high confidence. For users utilizing WhatsApp web, all operating systems and browsers showed the same behavior (because all browsers execute the same javascript code).

Fingerprinting a target primary device by characteristic *signed prekey IDs* (e.g., Android vs. iOS) is possible with high confidence and just requires the attacker to query the prekey bundle once for the victims phone number and device ID 0. If the the signed prekey ID is high ( $>> 0$ ), then it has probably be chosen at random and thus it must be an iPhone.

Distinguishing companion device types by their initial prekey batch sizes requires repeated querying, but is still feasible. Due to homogeneous refill batch sizes across all device types, the initial prekey batch size can be deduced at any later point in time, by consuming the entire (812 element-sized) prekey batch from the server and looking at the returned min-



imum and maximum prekey IDs.

To facilitate this kind of OS fingerprinting, our client collects all fetched prekey bundles and summarizes the stats of the returned values:

```
pogo@prekey: $ ./deplete-keys -t 123456789
Fetching all available prekeys of target device...

All prekeys depleted, consumed 812 bundles in 45s.
[Prekey Stats] Cnt: 812, MinID: 1674, MaxID: 2486
```

In the above example, we know that the victim uses a Windows desktop application, where a batch of 50 prekeys were uploaded in the initial batch (corresponding calculation:  $1,674 - 812 - 812 = 50$ ). If the initial batch size would be 200, we could still distinguish between macOS and WhatsApp Web by the value of the signed prekey ID as described before.

**Device Age and Activity Score.** The registration ID and identity key are static for the lifetime of the device installation. In contrast, signed- and one-time prekeys (and their IDs) change over time. When the device updates the signed prekey (usually done approx. once a month in WhatsApp) it also increments the corresponding signed prekey ID. Thus, for all devices except iOS and macOS, the current signed prekey ID roughly corresponds to the device's age in months.

Similarly, one-time prekey IDs are incremented by all devices. When asked by a client, the server always randomly selects one of the available one-time prekeys, making it more cumbersome to monitor for the attacker. However, when consuming all available prekeys, the attacker can still deduce how many prekeys have been used since the last refill, by calculating the difference between 812 and the number of returned prekeys. For all devices that initialize the one-time prekey with 1 (all except Android), the attacker can also derive the total amount of used one-time prekeys since the initial setup of the device. Due to the natural depletion of the one-time prekeys that is caused by new people (or devices) contacting the victim, this can be used to estimate an activity- or chattiness score of the target.

## 4.4 Observing Characteristic Refill Behavior

While measuring the approximate success rates for our PFS downgrade attack (Table 3 in Section 4.1), we noticed that the refill behavior can vary widely, depending on the victim's phone and its current state (e.g., current access technology and screen on/off state).

For example, across all captured experiments, iPhones were more vulnerable to prekey depletion –thus, took longer to react on a drained prekey bundle– than Android devices. Among the various Android models, the Samsung Galaxy A54 consistently showed the fastest refill times.

In addition to identifying the victim's operating system through specific key ID values, as presented in Section 4.3,

continuous monitoring of refill behavior could help determine the victim's operating system or device model.

Interestingly, the Xiaomi Poco X3 also showed significantly faster reactions when the screen was active compared to when it was in standby mode. This aligns with previous work [15], exposing activity fingerprinting by measuring message RTT times via delivery receipts.

Characteristic examples for the monitored refill behavior of different phones can be found in Figure 4 in the Appendix.

## 4.5 Rapid Retrieval and Denial of Service

For the previously presented attacks, our client sends synchronous queries to the server. Thus, before sending another query requesting the victim's prekey bundle, we always waited for the response of the previous request, leading to a depletion rate of up to 20 prekeys per second. In practice, the time for every request is limited by the connection round trip time (RTT) between client and server. Furthermore, the process seems to be significantly influenced by the current server load, with the depletion of all 812 prekeys taking anywhere from 40 seconds to 2 minutes within our different experiments.

One-time prekeys are supposed to be returned just once, which requires synchronization across concurrent requests. To test for the maximum retrieval rate for a single session, we increased the request rate, by sending queries in an asynchronous manner. Using parallel requests, we were able to consistently deplete 812 prekeys within just 10 seconds. After a certain retrieval rate (roughly more than 50 requests per second), the server occasionally returns a 503 *Service Unavailable* instead of the actual prekey bundle.

We observed that the 503 server error is not merely a rate limit affecting the current client session. Instead, generating a high volume of requests in one session also causes unsuccessful queries for other unrelated clients requesting prekeys for the same device for all querying devices.

By further increasing the request rate to  $> 2,000$  requests per second, we can entirely clog the prekey retrieval for the corresponding victim device. We showed the feasibility of this attack by clogging prekey retrieval with one session and concurrently trying to retrieve a valid prekey bundle by an unrelated client.

**Denial of Service.** In Section 4.1, the attacker only tries to eliminate the one-time prekey layer from the key exchange, thus, starting a new conversation is still possible. In contrast, the failure to retrieve the entire prekey bundle will generally hinder any new conversation attempts with the victim.

We verified this in practice by trying to contact our victim from different phones while simultaneously clogging prekey bundle retrieval using our custom client implementation. In all cases, the phones were not able to effectively start a new session and send the corresponding message, but kept stuck at the sending symbol ☹. Also, trying to audio/video call the

target via WhatsApp resulted in the call being immediately dropped<sup>4</sup>. To demonstrate the attack in practice, we prove a short demonstration video<sup>5</sup>.

We tried to infer the retransmission strategy empirically. Thereby, we did not see any timing-based back-off strategies for message retransmission (i.e., even when we stopped our DoS attack, the corresponding clients were not automatically trying to resend the message). Re-entering the chat or minimizing/activating the application to/from standby does not automatically trigger a retry-procedure. However, when entirely closing and reopening the WhatsApp application, the client makes another attempt to request the prekey bundle and eventually transmits the message to the target. Our testing attempts to execute this attack over extended time periods (e.g., multiple hours) were not blocked by WhatsApp, thus performing this attack consistently seems currently feasible. Thereby, an attacker could completely prevent any new conversation attempt to the target and thus force a downgrade to use less secure messaging solutions (e.g., SMS, Telegram).

## 4.6 Battery Drainage

Consistently generating new prekeys and pushing them to the server results in additional battery drain and likely prevents the phone from entering deep sleep states. We measured this extra drain under conditions of rapid (i.e., asynchronous) prekey depletion, which forces the device to frequently regenerate and upload fresh prekeys. For this case, we use our Samsung Galaxy A54 5G as target, since it refilled prekey almost immediately, even when being put into standby mode (cf. Section 4.4), presumably increasing abuse potential and battery drain. Our measurements showed an additional battery drain of approximately 2% per hour (measured during standby in LTE). During this time, prekeys were depleted roughly every 15 seconds, and the process of uploading new prekeys resulted in about 8 MB of additional data usage per hour. While this attack could definitely be annoying for the victim (i.e., forcing them to recharge their phone throughout the day), we consider this only a minor availability issue. Nevertheless, similar to the previously discussed vulnerabilities, any WhatsApp user can be targeted covertly, with minimal evidence left behind on the victim’s device.

## 4.7 Peripheral Observations

Besides the presented exploits, we made additional observations that could be relevant for WhatsApps security or privacy.

**Desynchronization of Prekey Depletion State.** During rapid prekey depletion, we observed instances where the

prekey state between the server and client became desynchronized. As a result, the client was not aware that the prekeys had been drained and thus did not push fresh prekeys to the server. In practice, this increases the success rate for our PFS depletion attack.

Additionally, when inspecting the decrypted traffic of legitimate WhatsApp web clients (using *CobaltAnalyzer*), we observed that prekey refills of the client were occasionally rejected with a 503 *Service Unavailable* error. While the client was eventually able to upload a fresh prekey bundle, this of course also enlarges the available time window for a PFS degradation attack.

**Repeated Prekey Distribution.** In the course of depleting prekeys from our test devices to evaluate the effectiveness of the PFS downgrade attack, we collected and analyzed the returned prekey bundles to verify whether one-time prekeys were correctly discarded after use.

While the server generally behaved as expected – successfully synchronizing concurrent queries from two independent sessions targeting the same device– we did observe rare instances in which one-time prekeys were handed out more than once. In total, we documented four such occurrences of prekey reuse, potentially indicating isolated failures in the server’s prekey distribution.

**Omitted Prekey IDs (Android).** While adhering to the uniform batch-size of 812 elements, we noticed that for every additional prekey batch that is uploaded to the server, 2 prekey IDs are omitted by the Android client. This behavior suggests the presence of a potential off-by-two error in the Android client’s prekey generation implementation. While simply skipping prekey IDs is not a security issue by itself, it could again be abused to determine the victim’s operating systems as presented in Section 4.3.

**Blocked Contacts.** Because the attacker interacts only with the server and never sends direct packets to the victim, the victim has no means of identifying the source of the attack. Nevertheless, we evaluated whether blocking an account in WhatsApp has any impact by testing a scenario in which the victim had already blocked the attacker prior to the attack. The results show that blocking has no effect, thus the attack remains fully feasible.

## 5 Related Work

**Security and Privacy at Instant Messaging.** Schrittwieser et al. were the first to investigate security and privacy issues in mobile instant messaging and VoIP applications, uncovering vulnerabilities such as account hijacking and number spoofing [24, 29]. Beyond Over-the-Top (OTT) applications, similar security vulnerabilities have been identified in VoIP-based messaging solutions like VoLTE, VoWiFi and RCS [16, 17, 30, 33]. In many cases, these vulnerabilities re-

<sup>4</sup>According to a technical report of Meta, the call initiation should have been unaffected, as the required prekey bundle is sent directly via webRTC [23], as the communication partner needs to be online anyway to establish a call.

<sup>5</sup><https://drive.proton.me/urls/H2P7VCW9R4#6ZGjnwFjf4TT>

mained undiscovered for years due to proprietary clients and the lack of tooling for security experiments. We adopt a similar security testing approach to find vulnerabilities within WhatsApp, leveraging open-source tools to emulate a real client sending protocol queries to the server. In contrast, Hagen et al. [18] demonstrated that large-scale account enumeration is possible in major messaging applications (e.g., WhatsApp, Signal) simply by automating interactions with the regular user interface. Beyond that, efficient account enumeration can also be achieved by using open-source clients to directly extract data from internal user APIs [14].

**Fingerprinting and Side Channels.** Previous work has shown that convenience features, such as read receipts in WhatsApp and other instant messaging apps, are frequently misused for stalking, even by non-technical users [13]. Beyond read receipts, delivery receipts expose message round-trip times (RTTs), which can be exploited to infer a user’s coarse geolocation [28]. Recent work [4] has further revealed that WhatsApp’s multi-device feature inadvertently leaks users’ device lists and the specific device used to send a message, due to the design approach used for end-to-end encryption (E2EE). Moreover, Gegenhuber et al. have demonstrated that read receipts can be leveraged for malformed and thus invisible messages in multi-device settings, enabling independent tracking and fingerprinting of all a user’s devices, as well as their online status and operating system [15]. As a potential mitigation, WhatsApp now allows users to block messages from unknown accounts<sup>6</sup>. However, while our prekey depletion exploit can be used for similar fingerprinting techniques—such as tracking a user’s device online status—we do not send any direct messages to the victim’s phone. Instead, our approach interacts solely with WhatsApp’s central prekey server.

**Signal Protocol.** The Signal Protocol and its variants has been analyzed in a series of works [6, 7, 9–12, 32]. Cohn-Gorden et. al [9] provides a nice overview of the protocol as well as a formal security analysis of the triple Diffie-Hellman (X3DH) key agreement and the Double Ratchet (DR). The DR was initially analyzed in [3]. A new variant of the key agreement, called PQXDH, which includes PQ-KEM as been described and formally analyzed in [19]. Post-Compromise Security (PCS) was initially defined and analyzed in [10]. Attacks on PCS in a multi device setting have been described in [11, 32]. The entire conversation layer, potentially consisting of multiple sessions/devices of a user, has been analyzed in [12] also with a focus on PCS and cloned devices.

## 6 Discussion

### 6.1 Ethical Considerations

For our measurements and during experimentation we only targeted WhatsApp accounts under our direct control. Additionally, we tried to adhere to WhatsApp’s protocol through the use of community-proven open source projects (some of them being used in widely deployed production systems<sup>7</sup>). In our depletion experiments, we issued a substantially higher volume of prekey queries compared to typical client implementations. However, this increased traffic is unlikely to pose a significant risk to WhatsApp’s infrastructure, which is built to serve more than three billion users. Moreover, we limited our offensive depletion to at most two concurrent client sessions. Lastly, none of our testing accounts were blocked throughout the study, hinting that we did not cause any significant harm and most likely were not even noticed by the platform operator. Lastly, to accurately assess the feasibility of the proposed attack, it was essential to conduct experiments against the actual WhatsApp infrastructure. Given the minimal risk of adverse effects on other users or the service itself, as argued above, we considered this a reasonable approach. Finally, all our findings have been responsibly disclosed to Meta.

### 6.2 Limitations

Although WhatsApp is the most popular instant messenger using the Signal protocol, many other messaging applications rely on the same protocol suite. While our analysis specifically focuses on WhatsApp, some of our findings may generalize to other Signal-based messengers. Due to WhatsApp’s closed-source nature, we were unable to directly inspect the source code of the official clients or the server backend. However, given its immense popularity, it is crucial to scrutinize its overall security. We hope this work serves as a first step toward shedding light on WhatsApp’s real-world implementation and the design decisions underlying its deployment of the Signal protocol.

### 6.3 Countermeasures

Many of the exploits and side channels presented in this work are inherent to Signal’s session handshake protocol, which relies on the availability of fresh one-time prekeys. As a result, completely eliminating these issues is challenging; for example, the device’s online state will inevitably be exposed when new prekeys are uploaded. Nevertheless, we propose several mitigations that would substantially reduce the practical exploitability of the identified vulnerabilities.

**Rate Limiting.** A single account should not be able to constantly query prekey bundles for the same device in rapid

<sup>6</sup><https://faq.whatsapp.com/3379690015658337/>

<sup>7</sup><https://github.com/element-hq/mautrix-whatsapp>

succession. Given the fast refill rate of most Android devices, even a modest artificial slow down (i.e., rate limiting), would reduce the likelihood of a successful one-time prekey depletion attack against these devices significantly.

**Reduce Signed Prekey Renewal Interval.** The lifetime of a signed prekey in WhatsApp is higher ( $\approx$  month), than the lifetime of a signed prekey in Signal (two days). Reducing the lifetime of signed prekeys would also reduce the impact regarding PFS through missing one-time prekeys.

**Visual Indication of Missing PFS in the UI.** Currently neither sender, nor receiver are notified if a new session is established without a one-time prekey, therefore there is no obvious way to detect such an attack as a user. To not flood all users with complicated warnings and to prevent misunderstandings, the settings could offer a verbose option for security-cautious or high-profile users, which would show such security related UI notifications when messages lack PFS.

**Signed Prekey Update on Demand.** If a prekey bundle without a one-time prekey is used to initiate a new session, the responder device could update his signed prekey together with the next batch of one-time prekeys it pushes to the server. This would minimize the damage a lack of PFS could cause, in case the responder device is online. Due to asynchronous communication, there may still exist prekey bundles in circulation that contain outdated signed prekeys, but this should not be a large problem since there is no valid use case for keeping them around and not immediately initiate a new session. To prevent an attacker from turning this countermeasure into a DoS attack, there should be a minimum validity period (in the order of minutes) for signed prekeys. Otherwise an attacker could trigger signed prekey updates all the time by initiating new sessions without one-time prekeys, which would prevent everybody else from establishing a new session as signed prekeys are constantly outdated.

**Redesign Key IDs.** Due to their initial values and the fact that they are incremented by one, key IDs leak information and make device fingerprinting possible. The question is, if key IDs could not be enlarged and entirely be replaced with hashes of the respective public keys they refer to, which would completely mitigate this information disclosure vulnerability.

## 7 Conclusion

In this work we have demonstrated that WhatsApp does not enforce any rate limiting regarding the querying of prekey bundles, thereby violating the Signal X3DH specification. This enables an attacker to deplete all one-time prekeys of a targeted device, subsequently degrading the perfect forward secrecy (PFS) of new sessions initiated with the victim. Although, PFS is undoubtedly effected by such an attack, the successful exploitation of this degraded forward secrecy would still require a compromise of the involved long- and medium-

term secret keys, as well as passive eavesdropping capabilities to record the respective encrypted messages.

In contrast to this rather strong attacker model, we also describe attacks on privacy and availability, with the sole requirement of having a WhatsApp account. Hereby, we were able to show that the refilling of one-time prekeys necessarily leaks the current online status of the respective device, as well as in certain cases: the device age, operating system and the approximate total number of new sessions initiated with the targeted device. Moreover, we were able to highlight a DoS issue by rapidly querying prekey bundles of a device such that the retrieval of any prekey bundle (even without one-time prekeys) was no longer possible. As a consequence, for the duration of the attack no new session can be established with the victim. All attacks described in this paper can be executed covertly, and targeted at any of WhatsApp’s more than 3 billion users.

To mitigate the discovered issues, we suggest a range of countermeasures. Most notably the notification of users regarding the degraded PFS in the UI, as well as the introduction of rate limits regarding the repeated fetching of prekey bundles for the same device from a single account.

## Artifact Evaluation

The artifact accompanying this paper is publicly available at <https://github.com/sbaresearch/prekey-pogo/tree/woot25ae>. It includes a modified WhatsApp client based on *whatsmeow*, which enables the retrieval of a user’s WhatsApp devices and their associated prekey material. As part of the WOOT 2025 artifact evaluation process, we applied for and were awarded both the *Artifacts Available* and *Artifacts Functional* badges.

## Acknowledgments

This material is based upon work partially supported by (1) the Christian-Doppler-Laboratory for Security and Quality Improvement in the Production System Lifecycle; The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the Nation Foundation for Research, Technology and Development and University of Vienna, Faculty of Computer Science, Security & Privacy Group is gratefully acknowledged; (2) the FFG Bridge project 46322124 SecKey; (3) SBA Research (SBA-K1 NGC), a COMET Center within the COMET – Competence Centers for Excellent Technologies Programme, funded by BMIMI, BMWET, and the federal state of Vienna. The COMET Programme is managed by FFG.

We further would like to thank Markus Maier for his outstanding support in providing and maintaining the IT infrastructure, which was essential for the success of this work. Moreover, we would also like to thank our anonymous reviewers for their valuable feedback and suggestions.



## References

- [1] CRYSTALS cryptographic suite for algebraic lattices. Retrieved Aug 26th, 2024 from <https://pq-crystals.org/index.shtml>.
- [2] WhatsApp encryption overview: Technical white paper. Retrieved Aug 26th, 2024 from <https://faq.whatsapp.com/82012443585354>.
- [3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer.
- [4] Tal A. Be’ery. WhatsApp with privacy? privacy issues with IM e2ee in the multi-device setting. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 11–16. USENIX Association.
- [5] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [6] Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. Post-quantum asynchronous deniable key exchange and the signal handshake. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part II*, volume 13178 of *Lecture Notes in Computer Science*, pages 3–34. Springer.
- [7] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1445–1459. ACM.
- [8] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1445–1459. ACM, 2020.
- [9] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. 33(4):1914–1983.
- [10] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 164–178. IEEE Computer Society.
- [11] Cas Cremers, Jaiden Fairuze, Benjamin Kiesl, and Aurora Naska. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1481–1495. ACM, 2020.
- [12] Cas Cremers, Charlie Jacomme, and Aurora Naska. Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1235–1252. USENIX Association.
- [13] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. “A Stalker’s Paradise” How Intimate Partner Abusers Exploit Technology. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–13, 2018.
- [14] Gabriel K. Gegenhuber, Philipp É. Frenzel, Maximilian Günther, Johanna Ullrich, and Aljosha Judmayer. Hey there! You are using WhatsApp: Enumerating Three Billion Accounts for Security and Privacy, 2025.
- [15] Gabriel K. Gegenhuber, Maximilian Günther, Markus Maier, Aljosha Judmayer, Florian Holzbauer, Philipp É. Frenzel, and Johanna Ullrich. Careless Whisper: Exploiting Silent Delivery Receipts to Monitor Users on Mobile Instant Messengers, 2024.
- [16] Gabriel K. Gegenhuber, Florian Holzbauer, Philipp É. Frenzel, Edgar Weippl, and Adrian Dabrowski. Diffie-Hellman Picture Show: Key Exchange Stories from Commercial VoWiFi Deployments. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 451–468, Philadelphia, PA, August 2024. USENIX Association.

- [17] Gabriel K. Gegenhuber, Wilfried Mayer, Edgar Weippl, and Adrian Dabrowski. MobileAtlas: Geographically Decoupled Measurements in Cellular Networks for Security and Privacy Research. In *Usenix Security Symposium 2023*, 2023.
- [18] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are US: Large-scale abuse of contact discovery in mobile messenger. In *28th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 21 - February 25, 2021*. The Internet Society.
- [19] Ehren Kret and Rolfe Schmidt. The PQXDH key agreement protocol. Retrieved Aug 26th, 2024 from <https://signal.org/docs/specifications/pqxdh/pqxdh.pdf>.
- [20] Moxie Marlinspike. Private group messaging. Retrieved Aug 26th, 2024 from <https://signal.org/blog/private-groups/>.
- [21] Moxie Marlinspike and Trevor Perrin. The sesame algorithm: Session management for asynchronous message encryption. Retrieved Aug 26th, 2024 from <https://signal.org/docs/specifications/sesame/sesame.pdf>.
- [22] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. Retrieved Aug 26th, 2024 from <https://signal.org/docs/specifications/x3dh/x3dh.pdf>.
- [23] Meta. Messenger end-to-end encryption overview, December 2023. Accessed: 2025-03-11.
- [24] Robin Mueller, Sebastian Schrittwieser, Peter Fruehwirt, Peter Kieseberg, and Edgar Weippl. What's new with WhatsApp & Co.? Revisiting the Security of Smartphone Messaging Applications. iiWAS '14, page 142–151, New York, NY, USA, 2014. Association for Computing Machinery.
- [25] National Institute of Standards and Technology. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography (Revision 3). Technical Report NIST SP 800-56A Rev. 3, National Institute of Standards and Technology, April 2018.
- [26] Trevor Perrin. The Noise Protocol Framework. Online, 2018. Version 34, Retrieved March 5, 2025.
- [27] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. Retrieved Aug 26th, 2024 from <https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf>.
- [28] Theodor Schnitzler, Katharina Kohls, Evangelos Bitsikas, and Christina Pöpper. Hope of delivery: Extracting user locations from mobile instant messengers. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society.
- [29] Sebastian Schrittwieser, Peter Fruehwirt, Peter Kieseberg, Manuel Leithner, Martin Mulazzani, Markus Huber, and Edgar Weipp. Guess who is texting you? Evaluating the security of smartphone messaging applications. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5 - February 8, 2012*. The Internet Society.
- [30] Guan-Hua Tu, Chi-Yu Li, Chunyi Peng, Yuanjie Li, and Songwu Lu. New security threats caused by IMS-based SMS service in 4G LTE networks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [31] WhatsApp. About WhatsApp, 2024. Retrieved Aug 26th, 2024 from <https://www.whatsapp.com/about/>.
- [32] Jan Wichelmann, Sebastian Berndt, Claudius Pott, and Thomas Eisenbarth. Help, My Signal has Bad Device! - Breaking the Signal Messenger's Post-Compromise Security Through a Malicious Device. In Leyla Bilge, Lorenzo Cavallaro, Giancarlo Pellegrino, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 18th International Conference, DIMVA 2021, Virtual Event, July 14-16, 2021, Proceedings*, volume 12756 of *Lecture Notes in Computer Science*, pages 88–105. Springer.
- [33] Yaru Yang, Yiming Zhang, Tao Wan, Chuhan Wang, Haixin Duan, Jianjun Chen, and Yishen Li. Uncovering Security Vulnerabilities in Real-world Implementation and Deployment of 5G Messaging Services. In *Proceedings of the 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2024.

## Appendix

### A Tested Phone Models and OS Versions

Device	Modem Chipset	OS	WhatsApp
iPhone SE 2020	Intel	iOS 18.3.1	2.25.4.77
iPhone 8	Intel	iOS 16.7.10	2.25.5.74
iPhone 11	Qualcomm	iOS 18.3.1	2.25.5.74
Xiaomi POCO X3 NFC	Qualcomm	Android 12 (MIUI 14.0.5)	2.25.2.78
Samsung Galaxy A54 5G	Exynos	Android 14	2.25.2.78
Xiaomi Redmi 10 5G	MediaTek	Android 14	2.25.2.78

Table 5: Overview of the devices including software versions that were used throughout our tests. For our WhatsApp Web tests (Chrome, Firefox, Safari) we’ve used the most recent browser and WhatsApp web versions available (testing date 2025-02-21).

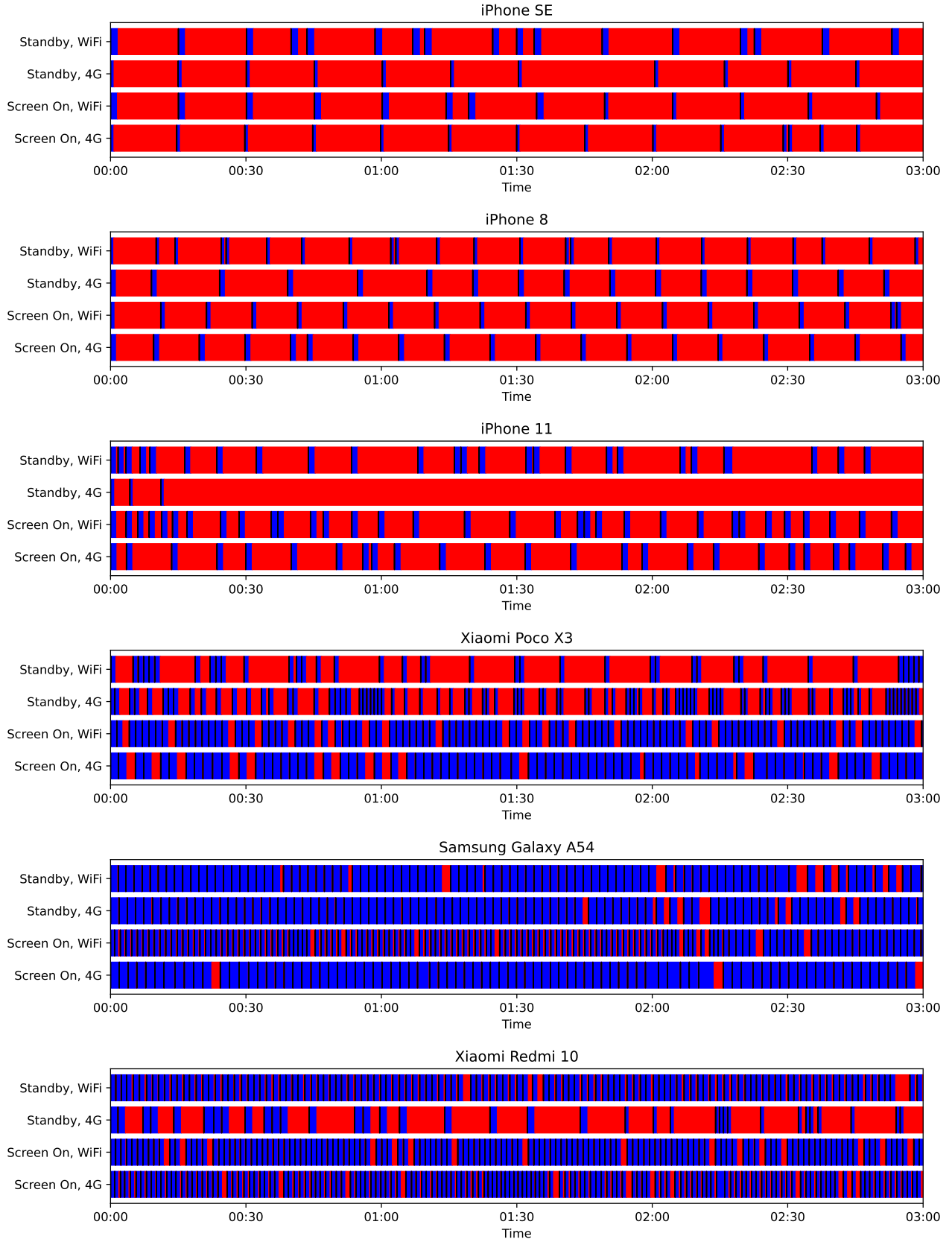


Figure 4: Characteristic refill behavior of different smartphone models in various device and connection states. Time where no one-time prekeys are available is shown in red.



## B Signal Protocol in More Detail

The Signal protocol actually consists of an entire family of protocols [19–22, 27] which been studied in a variety of works [3, 6, 9–12, 32].

Internally, the signal protocol uses two key derivation functions (KDF), one to derive ratchet keys, which we denote by  $KDF_r(\cdot)$  and one to derive the message keys, denoted by  $KDF_m(\cdot)$ . The KDFs are implemented with HKDF and HMAC-SHA256. The thereby generated derived keys are used for authenticated encryption with associated data (AEAD), using AES256 in CBC mode for encryption and HMAC-SHA256 for authentication. We denote the symmetric encryption function by  $E(\text{key}, \text{plaintext}, \text{associated data})$  and the decryption function by  $D(\text{key}, \text{ciphertext}, \text{associated data})$ . Moreover, the signal protocol relies on Diffie-Hellman key exchange to compute shared keys and achieve its design goals. We denote the key exchange algorithm by  $DH(\cdot)$ , which is implemented over Curve25519 [5] in practice

The Signal protocol uses three different types of Diffie-Hellman public keys to ensure forward secrecy right from the start: Long-term identity keys, medium-term (signed) prekeys and short-term (one-time) ephemeral prekeys (see Table 6 for an overview). In our scenario Alice is the *initiator* and wants to establish a secure connection with Bob (the *responder*). The Signal protocol, as also implemented by WhatsApp, uses prekey bundles deposited by every user at a central server to allow any *initiator* to negotiate a shared secret (via Diffie-Hellman Key exchange) even if the *responder* is currently not online/available using the X3DH protocol [22] <sup>8</sup>.

The initial handshake works as depicted in figure 5 in the appendix and here starting with formula 13. First the prekey bundle of the responder (in our case Bob) is fetched from the server by the initiator (in our case Alice). The information from the prekey bundle is verified by Alice through checking the signature on the *signed prekey* using the (long-term) identity public key of Bob. As within other works [9], it is assumed that Alice has already verified out-of-band that the long-term identity public key indeed belongs to Bob.

Then the public keys from Bob's prekey bundle are used to compute shared keys for the ratcheting and message encryption and authentication. Here now we have to distinguish between the case where an ephemeral (one-time) prekey  $eprepk^B$  of Bob is available or not. If no ephemeral prekey is available, the DH invocation  $dh_4$  in formula 17 is omitted.

In any case, before initiating the session and sending the first message to Bob, Alice generates two ephemeral key pairs: The *ephemeral handshake key pair* denoted  $(epk^A, ek^A)$  and the *ephemeral ratchet key pair* denoted  $(rchk^A, rchk^A)$ .

Those are used for the initial handshake and to initialize the DH ratchet construction, also referred to as the *asymmetric ratchet*.

$$dh_1 \leftarrow DH(ik^A, prepk^B) \quad (13)$$

$$dh_2 \leftarrow DH(ek^A, ipk^B) \quad (14)$$

$$dh_3 \leftarrow DH(ek^A, prepk^B) \quad (15)$$

$$[dh_4 \leftarrow DH(ek^A, eprepk^B)] \quad (16)$$

$$rk_0 \leftarrow KDF_r(dh_1 \parallel dh_2 \parallel dh_3 \parallel [DH4]) \quad (17)$$

$$DH_{ratchet} \leftarrow DH(rchk_0^A, prepk^B) \quad (18)$$

$$rk_1, ck_{0,0}^{i \rightarrow r} \leftarrow KDF_r(rk_0, DH_{ratchet}) \quad (19)$$

$$ck_{0,1}^{i \rightarrow r}, mk_{0,0}^{i \rightarrow r} \leftarrow KDF_m(ck_{0,0}^{i \rightarrow r}) \quad (20)$$

The derived message key  $mk_{0,0}^{i \rightarrow r}$  is then used to encrypt and authenticate (AE) the first chat *message* from Alice to Bob, as well as to authenticate some associated data *AD* consisting of the ephemeral public keys ( $epk^A$  and  $rchkp_0^A$ ) generated previously by Alice.

$$AD \leftarrow \langle rchkp_0^A, epk^A, \text{id of } prepk^B, [eprepk^B] \rangle \quad (21)$$

$$AE_{mk_{0,0}^{i \rightarrow r}}, AD \leftarrow E(mk_{0,0}^{i \rightarrow r}, \text{message}, AD) \quad (22)$$

Once Bob receives this initial message, he can compute the same shared keys using his identity key  $ik^B$  and his prekey  $prepk^B$ , as well as the public keys of Alice consisting of her identity key  $ipk^A$ , her ephemeral handshake key  $epk^A$  and her ephemeral ratchet key  $rchkp^A$ . Since the later two are transmitted in the associated data *AD*, they are authenticated, but not encrypted.

$$dh_1 = DH(ipk^A, prepk^B) \quad (23)$$

$$dh_2 = DH(epk^A, ik^B) \quad (24)$$

$$dh_3 = DH(epk^A, prepk^B) \quad (25)$$

$$rk_0 \leftarrow KDF_r(dh_1 \parallel dh_2 \parallel dh_3) \quad (26)$$

$$DH_{ratchet} \leftarrow DH(rchkp_0^A, prepk^B) \quad (27)$$

$$rk_1, ck_{0,0}^{i \rightarrow r} \leftarrow KDF_r(rk_0, DH_{ratchet}) \quad (28)$$

$$ck_{0,1}^{i \rightarrow r}, mk_{0,0}^{i \rightarrow r} \leftarrow KDF_m(ck_{0,0}^{i \rightarrow r}) \quad (29)$$

The received message is then decrypted using the previously computed shared message key  $mk_{0,0}^{i \rightarrow r}$ :

$$\text{message}, AD \leftarrow D(mk_{0,0}^{i \rightarrow r}, AE_{mk_{0,0}^{i \rightarrow r}}, AD) \quad (30)$$

If no ephemeral prekeys have been fetched by Alice, this initial message sent by Alice has no forward secrecy if observed by an attacker. Therefore, an attacker who is able to

<sup>8</sup>Since late 2023 Signal replaced X3DH by PQXDH [19] and started a process to use PQXDH for new sessions if supported by both peers. On a high level, PQ3DH is comparable to X3DH with the difference that an additional key from a CRYSTALS-Kyber [1] key encapsulation mechanism (KEM) is used in the KDF.

Keys		Description
$ipk^A$	$ik^A$	Long-term identity key pair of Alice
$ipk^B$	$ik^B$	Long-term identity key pair of Bob
$prepk^A$	$prek^A$	Medium-term prekey pair of Alice, aka. <i>signed prekey</i>
$prepk^B$	$prek^B$	Medium-term prekey pair of Bob, aka. <i>signed prekey</i>
$eprepk_n^A$	$eprek_n^A$	Short-term prekey pair number $n$ of Alice, aka. <i>ephemeral prekey</i> or <i>one-time prekey</i>
$eprepk_n^B$	$eprek_n^B$	Short-term prekey pair number $n$ of Bob, aka. <i>ephemeral prekey</i> or <i>one-time prekey</i>
$\langle ipk^A, prepk^A, \text{Sig}(ik^A, prepk^A), [\textcolor{red}{eprek_n^A}] \rangle$		A prekey bundle deposited by Alice on the server
$\langle ipk^B, prepk^B, \text{Sig}(ik^B, prepk^B), [\textcolor{red}{eprek_n^B}] \rangle$		A prekey bundle deposited by Bob on the server
$epk^A$	$ek^A$	Ephemeral handshake key pair of Alice
$rchkp_0^A$	$rchk_0^A$	Ephemeral ratchet key pair of Alice
$rk_x$		Symmetric (shared) root key of ratchet number $x$
$ck_{x,y}^{i \rightarrow r}$		Symmetric (shared) chaining key number $y$ , in the $x^{th}$ initiator to responder ratchet
$mk_{x,y}^{i \rightarrow r}$		Symmetric (shared) message key number $y$ , in the $x^{th}$ initiator to responder ratchet

Table 6: Main cryptographic keys of the signal protocol relevant for our attacks. Public keys in asymmetric schemes always end in  $pk$ . The naming convention of the keying material is according to Cohn-Gordon et al. [9]. The ephemeral prekeys, which are considered optional in the prekey bundle, are depicted in red. The secret keys of Bob which an attacker has to compromise to benefit from the violation of forward secrecy, i.e., if no ephemeral prekeys can be used, are depicted in orange.

compromise Bobs medium-term and long-term secret keys  $prek^B$  and  $ik^B$  later on, can recompute the same message key  $mk_{0,0}^{i \rightarrow r}$ , which highlights that there is no forward secrecy for this message. If Alice sends multiple messages before receiving any response from Bob, all these messages are affected as well, as the keys for these messages come from the symmetric ratchet. This is illustrated by the following example starting with formula 31, which depicts encrypting a second message from Alice to Bob. Here,  $y$  is 0 at the beginning and later set to  $y = 1$  for the second message and so forth:

$$ck_{0,2}^{i \rightarrow r}, mk_{0,1}^{i \rightarrow r} \leftarrow KDF_m(ck_{0,1}^{i \rightarrow r}) \quad (31)$$

$$y \leftarrow y + 1 \quad (32)$$

$$AD \leftarrow (rchkp_0^A, ipk^A, ipk^B, y) \quad (33)$$

$$AE_{mk_{0,1}^{i \rightarrow r}}, AD \leftarrow AE(mk_{0,1}^{i \rightarrow r}, message, AD) \quad (34)$$

Forward secrecy is restored through the asymmetric ratchet, when Bob responds to a message. If Bob responds to one of Alice messages, he also computes a new ephemeral ratchet key pair  $(rchkp_1^B, rchk_1^B)$ , s.t.  $x = 1$ , and thereby advances the asymmetric ratchet as follows:

$$DH_{ratchet} \leftarrow DH(rchkp_{x-1}^A, rchk_x^B) \quad (35)$$

$$tmp, ck_{x,0}^{r \rightarrow i} \leftarrow KDF_r(rk_x, DH_{ratchet}) \quad (36)$$

$$ck_{x,1}^{r \rightarrow i}, mk_{x,0}^{r \rightarrow i} \leftarrow KDF_m(ck_{x,0}^{r \rightarrow i}) \quad (37)$$

$$AD \leftarrow (rchkp_x^B) \quad (38)$$

$$AE_{mk_{x,y}^{r \rightarrow i}} \leftarrow E(mk_{x,y}^{r \rightarrow i}, message, AD) \quad (39)$$

As soon as these ephemeral ratchet keys are deleted, forward secrecy is regained for this as well as subsequent messages. Note, that even if forward secrecy is regained in a chat session, the initial messages sent from Alice to Bob have been encrypted using the symmetric ratchet only. Therefore, they remain vulnerable for the entire lifetime of the signed prekey  $prek^B$  of Bob. According to the specifications, the signed prekey should be periodically rotated [2, 22, 23], where suggested intervals reach from once a week to once a month <sup>9</sup>.

<sup>9</sup>In practice Signal rotates the signed prekey every two days <https://github.com/signalapp/Signal-Android/blob/481dc162d80292a046b4229ccea2ac2f2a73f36/app/src/main/java/org/telegram/secrests/jobs/PreKeysSyncJob.kt#L57-L66>

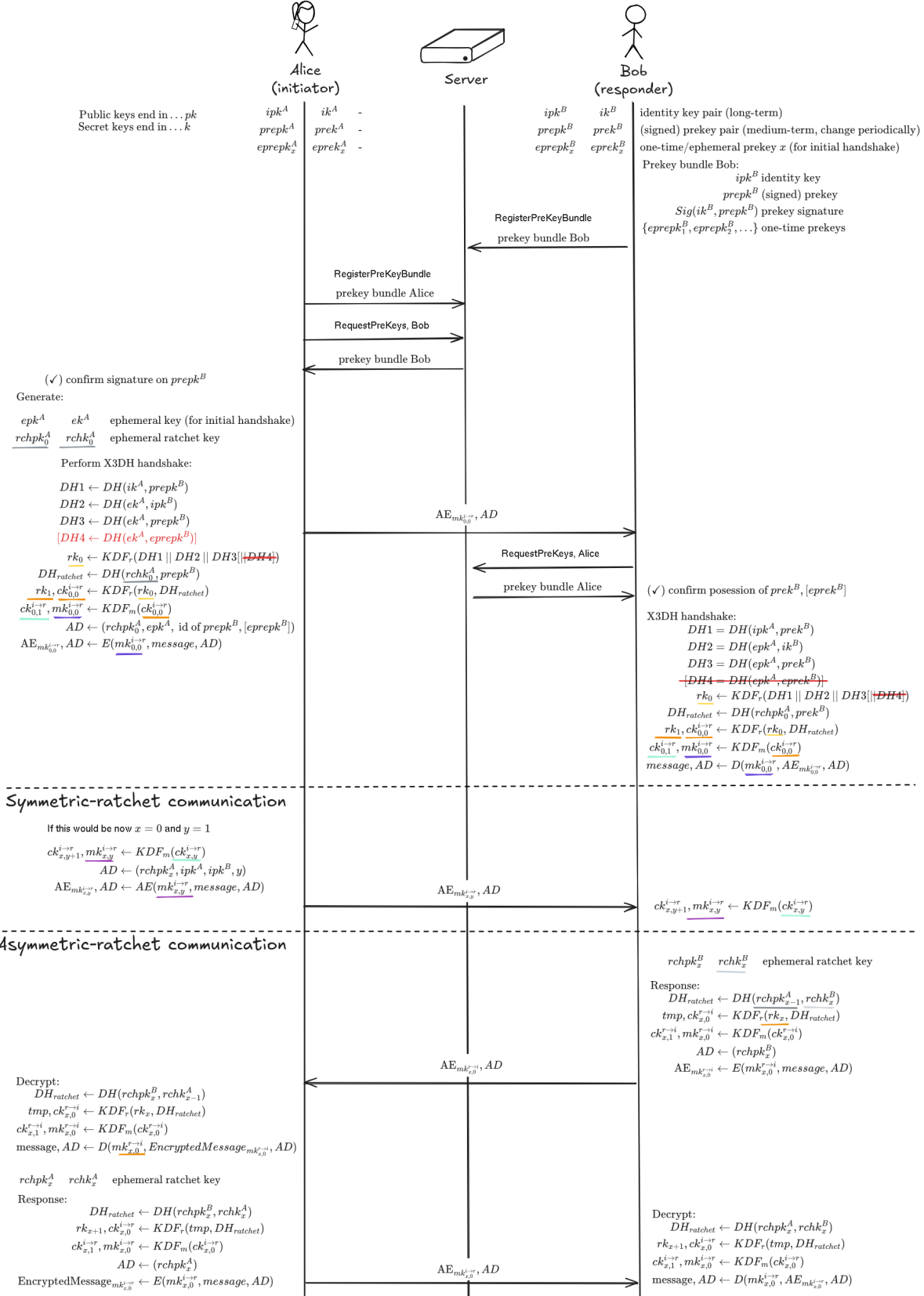


Figure 5: Signal protocol layout, when no *ephemeral (one-time) prekeys* are available on the server. Identical keys are highlighted with the same color.