2016 IEEE European Symposium on Security and Privacy

How Secure is TextSecure?

Tilman Frosch*†, Christian Mainka†, Christoph Bader†, Florian Bergsma†, Jörg Schwenk†, Thorsten Holz†
*G DATA Advanced Analytics GmbH
{firstname.lastname}@gdata.de
†Horst Görtz Institute for IT-Security
Ruhr University Bochum
{firstname.lastname}@rub.de

***Abstract*—**

***Instant Messaging* has gained popularity by users for both private and business communication as low-cost short message replacement on mobile devices. However, before releases about mass surveillance performed by intelligence services such as NSA and GCHQ and Facebook's acquisition of WHATSAPP, most mobile messaging apps did not protect confidentiality or integrity of the messages.**

A messaging app that claims to provide secure instant messaging and has attracted a lot of attention is TEXTSECURE. Besides numerous direct installations, its protocol is part of Android's most popular aftermarket firmware CYANOGENMOD. TEXTSECURE's successor *Signal* continues to use the underlying protocol for text messaging. In this paper, we present the first complete description of TEXTSECURE's complex cryptographic protocol, provide a security analysis of its three main components (key exchange, key derivation and authenticated encryption), and discuss the main security claims of TEXTSECURE. Furthermore, we formally prove that—if key registration is assumed to be secure—TEXTSECURE's push messaging can indeed achieve most of the claimed security goals.

1. Introduction

Since more than a decade, *Instant Messaging* (IM) is an alternative to classical e-mail communication, for both private and business communication. IM has different features; most importantly, messages are delivered in real-time, but only if both parties are online. However, in contrast to security mechanisms available for e-mail such as PGP [1] and S/MIME [2], instant messages were sent unprotected: In the early days, many popular IM solutions like MSN MESSENGER and YAHOO MESSENGER did not provide any security mechanisms at all. Today, many clients implement only client-to-server encryption via TLS, although security mechanisms like *Off the Record* (OTR) communication [3] or SCIMP [4] providing end-to-end confidentiality and integrity are available.

With the advent of smartphones, low-cost short-message alternatives gained popularity. However, in the context of mobile applications, the assumption of classical instant messaging, for instance, that both parties are online at the time the conversation takes place, is no longer necessarily valid. Instead, the mobile context requires solutions that allow for asynchronous communication, where a party may be offline for a prolonged time. In this setting, existing solutions, such as OTR, are only applicable in a limited fashion.

Secure Messaging and TextSecure. In the light of the recent revelations of mass surveillance actions performed by intelligence services such as NSA and GCHQ, several secure text messaging (TM) solutions that claim not to be prone to surveillance and to offer a certain level of security have appeared on the market [5].

One of the most popular apps for *secure* TM is TEXTSECURE[1], an app developed by *Open WhisperSystems* that claims to support end-to-end security of text messages. While previously focusing on encrypted short message service (SMS) communication, *Open WhisperSystems* introduced data channel-based push messaging in February 2014. Thus, the app offers both an iMessage- and WhatsApp-like communication mode, providing SMS+data channel or data channel-only communications [6]. Following Facebook's acquisition of WHATSAPP, TEXTSECURE gained in popularity among the group of privacy-conscious users and has currently more than 500,000 installations via *Google Play*. Its encrypted messaging protocol has also been integrated into the OS-level SMS-provider of CyanogenMod [7], a popular open-source aftermarket Android firmware that has been installed on about 10 million Android devices [8]. According to media reports [9], TextSecure's protocol has additionally been implemented in WhatsApp's Android client. While we did not verify this claim, in consequence the protocol's security would affect several hundred million users. Despite this popularity, the messaging protocol behind TEXTSECURE has not been rigorously reviewed so far. While the developers behind TEXTSECURE have a long history of research in computer security, a security assessment is needed to carefully review the approach.

Contribution. In summary, we make the following contributions:

1. The name of the App has been changed to SIGNAL in November 2015 to be consistent with the iOS App.

978-1-5090-1752-2/16 $31.00 © 2016 IEEE
DOI 10.1109/EuroSP.2016.41

- We are the first to completely and precisely document and analyze TEXTSECURE's secure push messaging protocol. Our description was confirmed by the developers of TEXTSECURE.
- We show that the main protocol of TEXTSECURE consists of three building blocks: A *cached One-Round Key Exchange (cORKE)* protocol, a secure *key derivation function*, and *authenticated encryption*. We give formal security definitions and security proofs for these blocks.
- We found subtle, but avoidable flaws in the protocol that allows for an Unknown Key-Share attack. We have documented the issues and show how they can be mitigated. They have been communicated to the developers of TEXTSECURE. We show that our proposed method of mitigation actually solves the issues.
- We discuss how and to which extent deniability, perfect forward secrecy (PFS) and future security (FS) are realized. While TEXTSECURE meets PFS and FS, deniability is only achieved partially in practice.

2. High-level Overview of TextSecure and related protocols

TEXTSECURE was previously compared [10] to the Off-the-Record Protocol (OTR) and the Silent Circle Instant Messaging Protocol (SCIMP) [11]. In the following, we discuss common elements and differences.

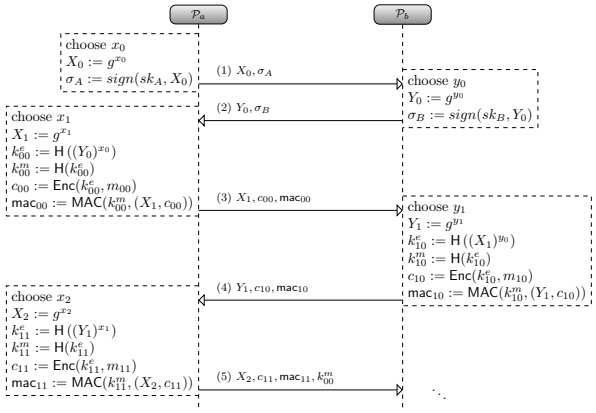

Figure 1: The Off-the-Record protocol.

Off-the-Record Protocol. OTR was proposed by Borisov et al. [3] as a method to secure conventional instant messaging. OTR's focus differed significantly from previous message prKGotection mechanisms like OpenPGP and S/MIME by introducing two novel properties: *Perfect Forward Secrecy* and *Deniability*.

Figure 1 shows how the OTR protocol version 1 works: After an initial signed Diffie-Hellman (DH) key exchange, novel DH shares are exchanged with every message, and the resulting DH key is constantly changed. OTR uses malleable encryption [12] in combination with MACs (instead of digital signatures). The OTR protocol reveals the MAC keys one round later to the public. This is essential for the *deniability* property of the protocol: anyone can change the value of the plaintext message, as inverting bits of the ciphertext will result in an inversion of the same bits at the same positions in the plaintext. Thus, the received messages are authentic at the time of reception only (given a party verifies the first signature and the following MACs). Since the MAC keys are derived as hash values of the encryption keys, revealing MAC keys does not compromise the security of the former, and the exchanged messages remain confidential. Private DH shares x_i and y_j are deleted as soon as the key k_{ij} has been computed. This guarantees *perfect forward secrecy* since without these private shares the encryption keys cannot be recomputed from the public shares X_i, Y_j later.

Di Raimondo *et al.* [13] showed that OTR v1 is vulnerable to an unknown key-share (UKS) attack (also called *identity misbinding attack*) [14]. We will discuss this kind of attack in Section 4.2. OTR version 2 did address this issue by introducing a four message handshake that follows the SIGMA protocol paradigm [15], effectively mitigating the UKS attack. Moreover, the protocol achieves deniability: public keys and signatures are exchanged within a confidential channel, leaving no trace of participation for an eavesdropper. However, these strong capabilities come at the cost of a four-message handshake.

OTR and Mobile Messaging. Instant Messaging connections are typically short-lived and online, whereas text messaging conversation may last for prolonged spans of time, and parties may be offline temporarily. Additionally, text messaging may be asynchronous, such that a sender sends several messages before receiving an answer.

The first adaption needed to derive a secure text messaging protocol from OTR is to make OTR work in offline scenarios. The basic idea here is due to ElGamal [16]. Thus OTR can be adapted to an offline scenario by storing many ephemeral DH shares of each party on a server.

The second adaption concerns key bookkeeping: In OTR, an ephemeral DH share must be protected by a MAC computed with a previous key, and must be acknowledged by the recipient B before being used by the sender A. This secure chaining of keys through MACs needs a lot of bookkeeping, as well as the acknowledgment. Here, TEXTSECURE adapts to the scenario by replacing MAC chaining by a secret value derived from long-lived (g^a, g^b) and ephemeral (g^{x_a}, g^{x_b}) DH shares, which is fed into any key generation step.

Silent Circle Instant Messaging protocol (SCIMP). In SCIMP [11] the idea of using different generations of symmetric keys is essential: After sending a message, the sender updates his own key by replacing it with its own hash value: $k_{new} := hash(k_{old})$. In doing so, perfect forward secrecy *between* two DH key exchanges can be achieved: Even if the sender is forced to reveal all his keys, an adversary can no longer decrypt the message sent, since this key has been deleted.

TextSecure. Informally speaking, TEXTSECURE builds on a (cached) *one-round key exchange (ORKE)* protocol [17] executed between parties A and B to compute the long-lived secret, a *key derivation function (KDF)* which takes as input the long-lived secret and a fresh DH secret, and an authenticated encryption scheme.

For the first message, a *cached* ephemeral key of the receiver (called *prekey*)[2] is fetched from the TEXTSECURE server, together with its long-lived public key. Later, new ephemeral public DH shares are included in each (first) response to a message. The process of changing ephemeral DH shares is called *ratcheting* in OTR and TEXTSECURE terminology. If a party sends several messages before receiving an answer, it updates the symmetric key used for each message in a one-way fashion by applying a key derivation function on an intermediate value (called *chaining key*), from which all cryptographic keys are computed. The structure of the TEXTSECURE communication is depicted in Figure 2. A complete representation of the protocol can be found online[3].

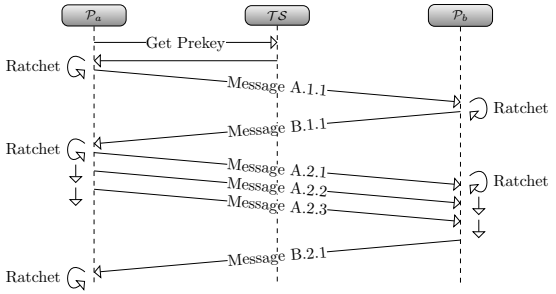

Figure 2: High-Level view on the *Axolotl-ratchet* protocol in TEXTSECURE.

3. TextSecure Protocol

We analyzed the source code of the Android app to recover the individual building blocks of the protocol. TEXTSECURE builds upon a set of cryptographic primitives. For ECDH operations, Curve25519 [18] is used as it is implemented in Google's Android native cryptographic library. Symmetric encryption in TEXTSECURE relies on AES [19] in both, counter mode without padding and cipher block chaining mode with PKCS7 [20] padding. HMAC-SHA256 [21], [22] is used for message integrity. Security considerations of the cryptographic primitives are not within the scope of this paper.

For push messaging via data channel, TEXTSECURE relies on a central server[4] ($\mathcal{TS}$) to relay messages to the intended recipient. Parties communicate with $\mathcal{TS}$ via a REST-API using HTTPS. $\mathcal{TS}$'s certificate is self-signed; the certificate of the signing CA is hard-coded in the TEXTSECURE app. Actual message delivery is performed via Google Cloud Messaging.

From TEXTSECURE's description in Google's Play store, the authors' blog and github we can identify some security goals of TEXTSECURE[5][6]. These are *end-to-end security, deniability, forward secrecy*, and *future secrecy*. We formalize and prove the first of these goals in Section 5, and discuss the remaining goals in Section 6.

TEXTSECURE's protocol consists of several phases as shown in Figure 3. We distinguish (1) *registration*, (2) *key comparison*, (3) *sending/receiving a first message*, (4) *sending a follow-up message*, and (5) *sending a reply*.

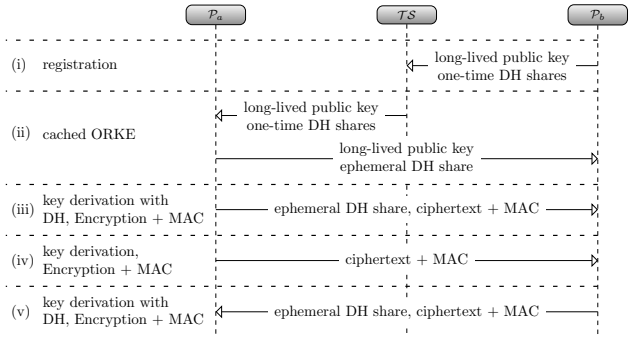

Figure 3: High-Level view on the different phases in TEXTSECURE.

Notation

In the protocol description, we use the following notation: Each protocol participant $\mathcal{P}_a$ is associated with a set of parameters, which all share the same index: e.g. $\mathsf{phone}\#_a$ is the phone number associated with $\mathcal{P}_a$. We denote the private, long-lived key of this party with a, and its public counterpart with g^a. *Prekeys* (recall that prekeys actually are ephemerals keys stored on the server) belonging to $\mathcal{P}_a$ are named $g^{x_{a,z}}$ with $z \in \{0, 99\}$, their private counterparts are $x_{a,z}$. The t-th ephemeral public Diffie-Hellman (DH) share chosen by $\mathcal{P}_a$ is denoted as $g^{\bar{x}_{a,t}}$, with its private counterpart $\bar{x}_{a,t}$. (Please note the overline symbol.)

Phase 1: TEXTSECURE Registration

The registration process is depicted in detail in Figure 5. To register with the TEXTSECURE server $\mathcal{TS}$, a party $\mathcal{P}_a$ requests a verification token by transmitting its phone number ($\mathsf{phone}\#_a$) and its preferred form of transport to $\mathcal{TS}$ (Step 1), which $\mathcal{TS}$ confirms with a HTTP status 204 (Step 2). Depending on the transport $\mathcal{P}_a$ chose, $\mathcal{TS}$ then dispatches either a short message or a voice call containing a random token (Step 3) to the number transmitted in Step

2. We refer to prekeys as *cached* ephemeral keys to distinguish them from ephemeral keys which are chosen at the time of protocol execution, whereas prekeys are chosen in advance and then stored on the server.

3. http://bit.ly/1IPIJ3Y

4. textsecure-service.whispersystems.org

5. https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms

6. https://github.com/trevp/axolotl/wiki

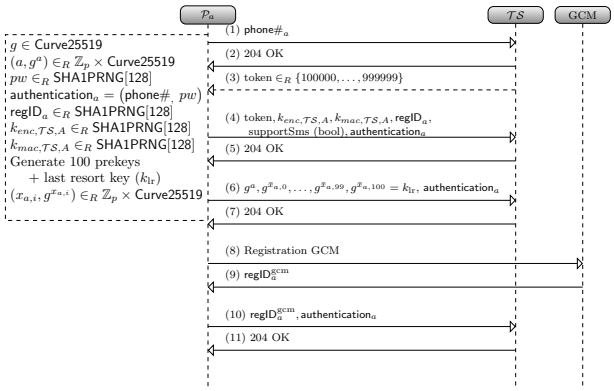

Figure 5: TEXTSECURE registration.

1. $\mathcal{P}_a$ performs the actual registration in Step 4, where it shows ownership of phone$\#_a$ by including the token, registers its credentials with the server via HTTP basic authentication [23], and sets its *signaling keys* $k_{mac,\mathcal{TS},A}$ and $k_{enc,\mathcal{TS},A}$[7]. In this step, the client also states whether it wishes to communicate only via data channel push message or also accepts short messages. The server accepts if the token corresponds to the one supplied in Step 3 and the phone number has not been registered yet.

In Step 6, $\mathcal{P}_a$ supplies its 100 *prekeys* (or one-time keys) and k_{lr} to $\mathcal{TS}$. Prekeys are not transmitted individually, but within a JSON structure consisting of a *keyID* z, a *prekey* $g^{x_{a,i}}$, and the long-term key g^a. The *last resort key* k_{lr} (the meaning of which will be clear at the end of this section) is transmitted in the same way and identified by *keyID* 0xFFFFFF. The server accepts, if the message is well-formed and HTTP basic authentication is successful. $\mathcal{P}_a$ then registers with Google Cloud Messaging (GCM) in Step 8 and receives its regID$_a^{\mathrm{gcm}}$ (Step 9), which it transmits to $\mathcal{TS}$ in Step 10 after authenticating again.

Taking a look ahead, a prekey for $\mathcal{P}_a$ is needed whenever a new session is to start in which $\mathcal{P}_a$ plays the role of the responder. Thus, the number of prekeys on the server decreases. If there are only a few prekeys left on the server, $\mathcal{P}_a$ may store new prekeys on the server. In the case that no more prekeys are available, the last resort key is used. This key, however, will not be erased from the server.

7. These keys are later used by $\mathcal{TS}$ to encrypt messages that are transmitted via GCM so that GCM cannot see any metadata, e.g. regID$_a$

Phase 2: Key Comparison

In an attempt to establish that a given public key indeed belongs to a certain party, TEXTSECURE offers the possibility to display the fingerprint of a user's long-term public key. Two parties can then compare fingerprints using an out-of-band channel, for example, a phone call or an in-person meeting. If two parties meet in person, TEXTSECURE also offers to conveniently render the fingerprint of one's own long-term public key as a QR code, using a third-party application on Android, which the other party can then scan using the same application on its mobile device. TEXTSECURE then compares the fingerprint it just received to the party's fingerprint it received as part of a conversation.

Phase 3.1: Sending an Initial Message

Before a first message can be sent from $\mathcal{P}_a$ to $\mathcal{P}_b$, three main steps have to be completed (see left side of

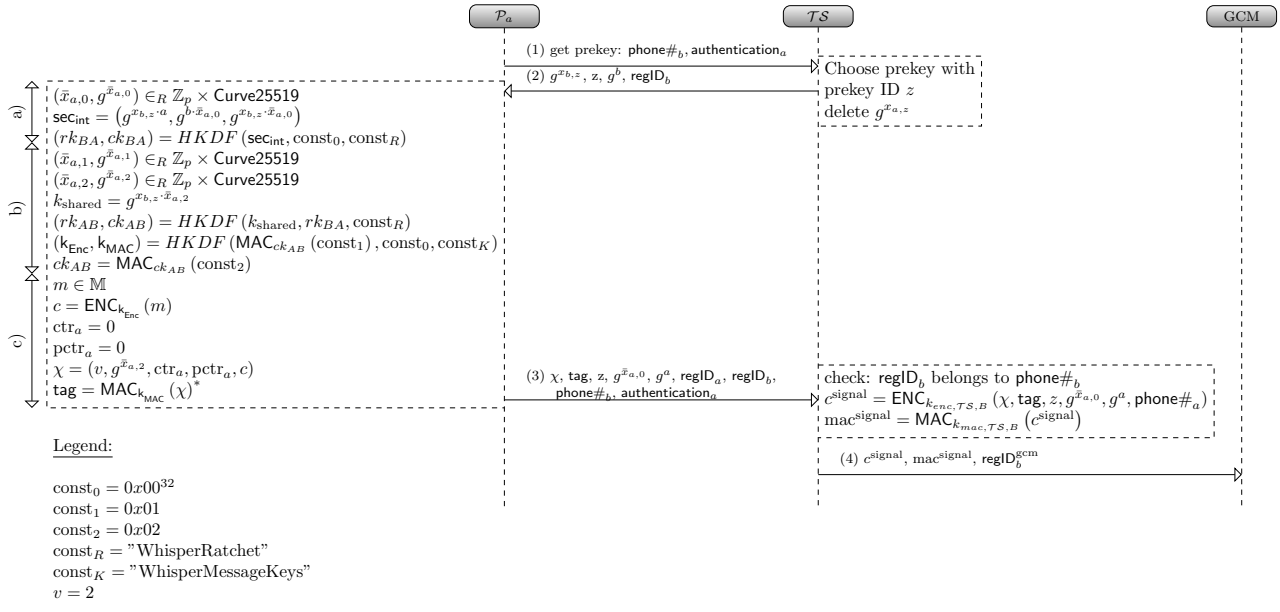

Figure 4: Sending an initial TEXTSECURE message.

Figure 4): (a) a (cached) ORKE-type key exchange protocol (cf. Section 5.1) to establish a shared, long-lived symmetric secret rk_{BA} (also called *root key*), (b) a key derivation and update protocol (the so-called *Axolotl-ratchet* [24]), which updates the root key, and generates a chaining key from which the encryption and MAC keys are derived, and (c) an authenticated encryption scheme. The process is depicted in detail in Figure 4.

(a) Establishment of shared secret. In the first step, $\mathcal{P}_a$ requests a *prekey* for $\mathcal{P}_b$ and receives a JSON structure consisting of prekey-ID z, a prekey $g^{x_{b,z}}$, and $\mathcal{P}_b$'s long-term key g^b. $\mathcal{P}_a$ also receives regID_b from $\mathcal{TS}$ and then chooses a new ephemeral key $\bar{x}_{a,0}$ to calculate an intermediate secret $\mathsf{sec_{int}}$ as the concatenation of three DH operations, combining $\mathcal{P}_b$'s prekey, $\mathcal{P}_a$'s long-term key, $\mathcal{P}_b$'s long-term key, and $\mathcal{P}_a$'s freshly chosen ephemeral key, $sec_{int} = \left(g^{x_{b,z} \cdot a}, g^{b \cdot \bar{x}_{a,0}}, g^{x_{b,z} \cdot \bar{x}_{a,0}}\right)$. From this value, the first root key is derived via the HKDF key derivation function.

(b) Key Management (*Axolotl-ratchet*). After $\mathcal{P}_a$ has computed the shared secret, a fresh secret k_{shared} is derived from the received prekey and a freshly generated private Diffie-Hellman share $\bar{x}_{a,2}$: $k_{shared} := (g^{x_{b,z}})^{\bar{x}_{a,2}}$.[8] From k_{shared} and rk_{BA}, a new root key rk_{AB} and a chaining key ck_{AB} are derived, again via HKDF.

The chaining key ck_{AB} is finally used to derive (again via HKDF) the encryption and MAC keys (K_{Enc}, K_{MAC}) to encrypt and protect the integrity of the first message.

(c) Authenticated Encryption. A message $m \in \mathbb{M}$ is encrypted using AES in counter mode without padding. That is, $c = \mathsf{ENC}_{\mathsf{k_{Enc}}}(m)$. $\mathcal{P}_a$ then forms message (3.) and thus calculates $\mathsf{tag} = \mathsf{MAC}_{\mathsf{k_{MAC}}}(\chi)$, where $\chi = (v, g^{\bar{x}_{a,2}}, \mathrm{ctr}_a, \mathrm{pctr}_a, c)$. v represents the protocol version and is set to $0x02$. For ordering messages within a conversation ctr and pctr are used. Both are initially set to 0. ctr is incremented with every message a party sends, while pctr is set to the value ctr carried in the message a party is replying to. Message (3.) is sent to $\mathcal{TS}$.

Forwarding the message to GCM. Upon receiving message (3.), $\mathcal{TS}$ checks if regID_b corresponds to $\mathsf{phone\#}_b$. It then encrypts the parts of message (3.) intended for $\mathcal{P}_b$ with $\mathcal{P}_b$'s signaling key ($k_{enc,\mathcal{TS},B}$), using AES in CBC mode with PKCS5 padding. $\mathcal{TS}$ additionally calculates a MAC over the result, which we denote as $\mathrm{mac}^{\mathrm{signal}}$. $\mathcal{TS}$ sends both, encrypted message data c^{signal} and $\mathrm{mac}^{\mathrm{signal}}$, to the GCM server, together with $\mathsf{regID}_b^{\mathrm{gcm}}$ as the recipient. The result of this additional encryption layer is that Google's Cloud Messaging servers will only be able to see the recipient but not the sender of the message.

Phase 3.2: Receiving a message

The receiving process is depicted in Figure 6. $\mathcal{P}_b$ receives the message in Step (5.). First, $\mathcal{P}_b$ verifies $\mathrm{mac}^{\mathrm{signal}}$ and, if successful, decrypts c^{signal}. It looks up its long-lived private key and the one-time private key that corresponds to prekey-ID z and calculates $\mathsf{sec_{int}}$ and the first root key rk_{BA}. From now on, all computations are identical to those done by the sender, until the message keys (k_{Enc}, k_{MAC}) are derived. $\mathcal{P}_b$ now verifies the MAC and, if successful, decrypts the message.

Phase 4: Sending a Follow-up Message

If $\mathcal{P}_a$ wants to send a message before $\mathcal{P}_b$ replies, $\mathcal{P}_a$ first derives a new chaining key $ck_{AB} := \mathsf{MAC}_{ck_{AB}{}^{old}}(\mathsf{const}_2)$, and derives a new pair $(\mathsf{k_{Enc}}, \mathsf{k_{MAC}}) = HKDF(\mathsf{MAC}_{ck_{AB}}(\mathsf{const}_1), \mathsf{const}_0, \mathsf{const}_K)$.

Phase 5: Sending a Reply

If $\mathcal{P}_b$ wants to reply to a message within an existing session with $\mathcal{P}_a$, it first chooses a new ephemeral keypair $(\bar{x}_{b,0}, g^{\bar{x}_{b,0}})$ and calculates a new k_{shared} as the output of

8. Please note that the DH share $(\bar{x}_{a,1}, g^{\bar{x}_{a,1}})$ is only generated because of code reuse, but never used.

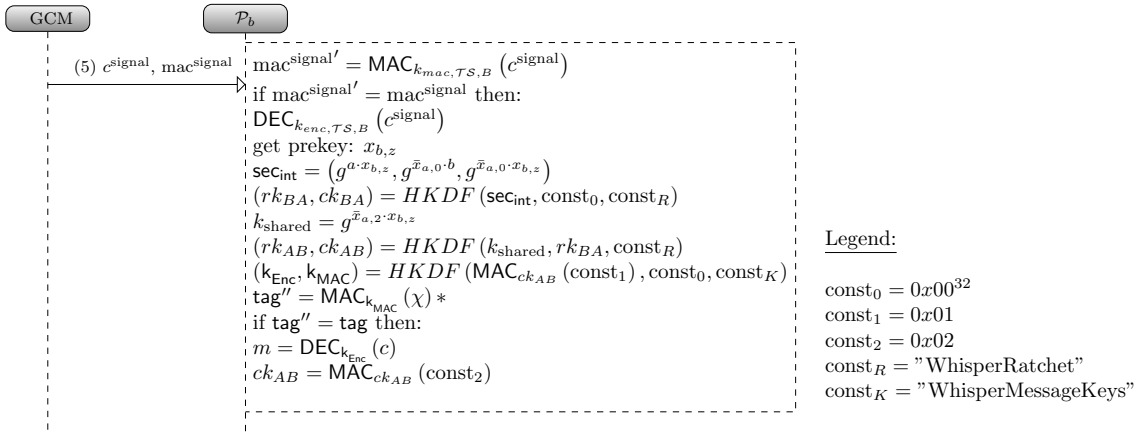

Figure 6: Receiving an initial TEXTSECURE message.

a DH operation that takes $\mathcal{P}_a$'s latest ephemeral public key $g^{\bar{x}_{a,2}}$ and its own freshly chosen ephemeral private key $\bar{x}_{b,0}$ as input: $k_{\text{shared}} = g^{\bar{x}_{a,2} \cdot \bar{x}_{b,0}}$. $\mathcal{P}_b$ then updates the root key and derives a new chaining key by seeding HKDF with the new k_{shared} and rk_{AB}: $(rk_{BA}, ck_{BA}) = HKDF\,(k_{\text{shared}}, rk_{AB}, const_R)$. From the chaining key, a new encryption and MAC key pair is derived, the message is protected with these keys, and the encrypted message is sent together with $g^{\bar{x}_{b,=}}$ back to $\mathcal{P}_a$.

4. Issues with TextSecure

During our analysis, we found several issues that we discuss in the following.

4.1. Password-based Key Registration

In TEXTSECURE, the password pw is both required to register prekeys with the server and to send messages on behalf of a party. The password is stored in the application-specific storage on an Android device. When we first analyzed TEXTSECURE, the easiest way to recover the password of another party was to trigger TEXTSECURE's encrypted export function, which exported an encrypted representation of the user's messages and an XML file containing the *unencrypted* pw. The export function placed the data on the device's SD card, thus easily accessible. This is a very limited level of protection for a token that is rather attractive for an attacker. Having access to pw does not only facilitate one way of conducting the Unknown Key Share attack we describe below, but allows for much more severe scenarios, where an attacker can register arbitrary key material on behalf of a party, as well as, send arbitrary messages. The encrypted export function has been silently removed in more recent versions of TEXTSECURE.

In more recent versions of the TEXTSECURE protocol, digital signatures have been added for the ephemeral public shares. In particular, a conversion of the long-lived key $(a, g^a) \in \mathbb{Z}_p \times \mathsf{Curve25519}$ is used to sign prekeys before sending them to the TEXTSECURE server $\mathcal{TS}$. The signature scheme used here is *Ed25519*, a variant of the Schnorr-Signature scheme which is known to be a non-interactive proof of knowledge of the secret key a. However, prekeys are still not bound to an identity, as it is the case with PGP keys. Also, as the proof is non-interactive, it can be forwarded.

4.2. Unknown Key-Share Attack

An *Unknown Key-Share Attack* (UKS) is an attack vector first described by Diffie *et al.* [14]. Informally speaking, if such an attack is mounted against $\mathcal{P}_a$, then $\mathcal{P}_a$ believes to share a key with $\mathcal{P}_b$, whereas in fact $\mathcal{P}_a$ shares a key with $\mathcal{P}_e \neq \mathcal{P}_b$.

For a better understanding how this can be related to TEXTSECURE, suppose the following example (*jealous spouse attack*): Bart ($\mathcal{P}_b$) wants to trick his friend Milhouse ($\mathcal{P}_a$). Bart knows that Milhouse will invite him to his birthday party using TEXTSECURE (e.g., because Lisa already told him). He starts the UKS attack by replacing his own public key with Nelsons ($\mathcal{P}_e$) public key and lets Milhouse verify the fingerprint of his new public key. This can be justified, for instance, by claiming to have a new device and having simply re-registered, as that requires less effort than restoring an encrypted backup of the existing key material. Now, as explained in more detail below, if Milhouse invites Bart to his birthday party, then Bart may just forward this message to Nelson who will believe that this message was actually sent from Milhouse. Thus, Milhouse ($\mathcal{P}_a$) believes that he invited Bart ($\mathcal{P}_b$) to his birthday party, where in fact, he invited Nelson ($\mathcal{P}_e$).

In detail, the attacker (Bart, $\mathcal{P}_b$) has to perform the steps shown in Figure 7 for this attack (only the important protocol parameters and steps are mentioned):

(1-2) $\mathcal{P}_b$ requests $g^{x_{e,z_0}}, \ldots, g^{x_{e,z_i}}$ from $\mathcal{TS}$ using $\mathsf{phone}\#_e$.

(3-4) $\mathcal{P}_b$ commits $g^{x_{e,z_0}}, \ldots, g^{x_{e,z_i}}$ to $\mathcal{TS}$ as his own prekeys plus g^e as its own long-term public key.

(5) $\mathcal{P}_b$ lets $\mathcal{P}_a$ verify the fingerprint of its new public key g^e. Note that this step uses QR-codes, therefore it is offline.

(6-7) Once $\mathcal{P}_a$ wants to send the message to $\mathcal{P}_b$, $\mathcal{P}_a$ requests a prekey for $\mathcal{P}_b$ by using $\mathsf{phone}\#_b$. $\mathcal{TS}$ returns $g^{x_{b,z_j}} = g^{x_{e,z_j}}$ and the long-term key $g^b = g^e$.

(8-9) $\mathcal{P}_a$ computes the $\mathsf{sec}_{\mathsf{int}}$ using $g^{x_{b,z_j}}$ and g^b from which $\left(\mathsf{k}_{\mathsf{Enc},AB}, \mathsf{k}_{\mathsf{MAC},AB}\right)$ are going to be derived. For computing those keys, he uses in fact $\mathcal{P}_e$'s prekey and identity key although he believes to use $\mathcal{P}_b$'s ones. He then encrypts message $m \in \mathbb{M}$, computes the respective MAC tag, and sends it to $\mathcal{P}_b$ (GCM is omitted for simplicity).

(10-11) $\mathcal{P}_b$ is neither able to verify the tag nor to decrypt the message c. He sends the ciphertext and message tag to $\mathcal{P}_e$.

(12) $\mathcal{P}_e$ processes the incoming message as usual. He computes the same $\mathsf{sec}_{\mathsf{int}}$ as $\mathcal{P}_a$, because $g^{x_{b,z_j}} = g^{x_{e,z_j}}$ and $g^b = g^e$. The $\mathsf{sec}_{\mathsf{int}}$ is then used to compute $\left(\mathsf{k}_{\mathsf{Enc},AE} = \mathsf{k}_{\mathsf{Enc},AB}, \mathsf{k}_{\mathsf{MAC},AE} = \mathsf{k}_{\mathsf{MAC},AB}\right)$ so that $\mathcal{P}_e$ is able to read and verify the message.

In Step 10, $\mathcal{P}_b$ has to forward the message to $\mathcal{P}_e$, such that it appears to be sent by $\mathcal{P}_a$. Therefore, he needs to include $\mathsf{authentication}_a$ for $\mathcal{TS}$ and $\mathsf{phone}\#_a$ in Step 11, so that $\mathcal{P}_e$ will receive $\mathsf{phone}\#_a$ with the forwarded message. This can be achieved in several ways:

- $\mathcal{TS}$ is corrupted. In this case, it is a trivial task to get or circumvent $\mathsf{authentication}_a$.
- If $\mathcal{TS}$ is benign, an attacker might be able to eavesdrop $\mathsf{authentication}_a$. Although TLS is used for all connections between clients and server, future or existing issues with TLS implementations [25]–[29] can not be ruled out and would allow for a compromise of $\mathsf{authentication}_a$. Another way to obtain

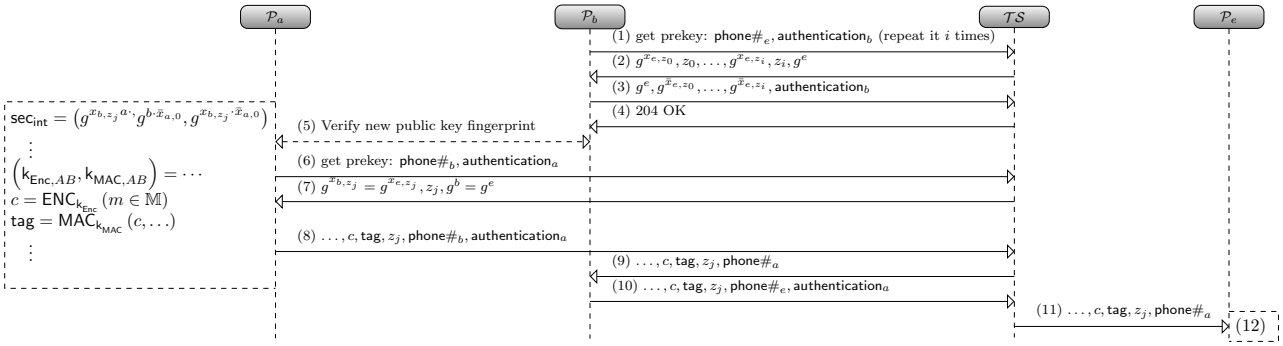

Figure 7: UKS attack on TEXTSECURE: $\mathcal{P}_a$ believes to share a key with $\mathcal{P}_b$ but shares one with $\mathcal{P}_e$.

authentication$_a$ could be a governmental agency (legally) enforcing access to the TLS keys.

- In contrast to a party's other key material, the password is stored unencrypted and is not protected by TEXTSECURE's master password. Thus, the easiest possibility to realize this attack might be for an attacker to recover the password for authentication$_a$ from TEXTSECURE's preferences[9].

Remark 1. *The signing of prekeys, as it is implemented in current versions of* TEXTSECURE*, does not prevent the attack. As the keys are still not cryptographically bound to the parties' identities, an attacker can still download keys of an arbitrary party and pass them off as their own. Thus the UKS attack described in this section was not mitigated.*

4.3. Mitigation of Unknown Key-Share Attack

Let us consider the message that is sent in Step 8 of Figure 7:

$$\chi, \mathsf{tag}, g^{\bar{x}_{a,0}}, g^a, \mathsf{regID}_a, \mathsf{regID}_e, \mathsf{phone}\#_e, \mathsf{authentication}_a,$$

where $\chi = (v, g^{\bar{x}_{a,2}}, \mathrm{ctr}_a, \mathrm{pctr}_a, c)$ and $\mathsf{tag} = \mathsf{MAC}_{\mathsf{k}_{\mathsf{MAC}}}(\chi)$. Intuitively, if both $\mathcal{P}_a$'s and $\mathcal{P}_e$'s identity were protected by the tag, then the attacks above do not longer work. As identities we propose to use the respective parties' phone numbers, as they represent a unique identifier within the system. χ would thus be formed as

$$(v, g^{\bar{x}_{a,2}}, \mathrm{ctr}_a, \mathrm{pctr}_a, \mathsf{phone}\#_a, \mathsf{phone}\#_e, c).$$

If $\mathsf{k}_{\mathsf{MAC}}$ is secret (i.e., only shared among $\mathcal{P}_a$ and $\mathcal{P}_e$) and if MAC is secure, the inclusion of both identities in the tag provides a proof of $\mathcal{P}_a$ towards $\mathcal{P}_e$ that $\mathcal{P}_a$ is aware of $\mathcal{P}_e$ as its peer, i.e., that the message is indeed intended for $\mathcal{P}_e$. Moreover, $\mathcal{P}_e$ is convinced that $\mathcal{P}_a$ actually sent the message. Thus, $\mathcal{P}_b$ will not be able to mount the above attacks.

4.4. Mitigation of Authentication Issue

While the *Unknown Key-Share Attack* is mitigated if the message in Step 8 is modified as we propose in Section 4.3, the underlying problem is not resolved. It results from a party's erroneous assumption that a communication partner's long-term identity key is authentic, if they have compared key fingerprints, and these fingerprints matched their assumptions. However, this is not necessarily the case. Given the attack scenario described in Section 4.2, a malicious party would always be able to present a third party's long-term public key as their own, as only fingerprints are compared – a party is not required to show their knowledge of the corresponding secret key.

To resolve this issue and to enforce that only a party in possession of the valid long-term identity key for this respective party may register new key shares with the server, we propose to use an *interactive* zero-knowledge proof of knowledge. Instead of presenting pw before uploading new prekeys, a party could present a proof that it knows the private key that corresponds to the identity key already registered with the server and used for signing previous prekeys. Thus, the authentication required for registering keys would now depend on a strongly protected secret: a party's identity key.

The Schnorr identification protocol [30] works ideally in this case. Let $(sk_a, pk_a) = (a, g^a) \in \mathbb{Z}_p \times Ed25519$ be the key pair of Party P_a. Then the Schnorr protocol can be used to prove possession of a to P_b (the verifier) as follows:

1) P_a chooses $r \xleftarrow{\$} \mathbb{Z}_p$ and sends $G = g^r$ (the commitment) to P_b.
2) P_b samples $e \xleftarrow{\$} \mathbb{Z}_p$ and sends e (the challenge) to P_a.
3) P_a computes $s = a \cdot e + r \mod p$ and sends s (the response) to P_b. P_b accepts if $g^s = pk_a^e \cdot G$.

The protocol could easily be carried out between a TEXTSECURE client and the server $\mathcal{TS}$ to proof possession of the corresponding secret key, each time a party attempts to upload keys.

It is well known that the Schnorr protocol is an *honest verifier* zero knowledge proof of knowledge [31]. That is, it holds that:

1) Computing a proof requires knowledge of the secret key, and
2) An honest verifier, i.e., a verifier that follows the protocol honestly, learns nothing about a.

In the sequel we assume public keys to be authentic.

9. File: *shared-prefs/org.thoughtcrime.securesms_preferences.xml*

Remark 2. *If we do not want to trust the server $\mathcal{TS}$, the Schnorr protocol can also be carried out when to parties meet face to face via exchange of QR-codes.*

This would mitigate the UKS attack, because in addition to being authenticated with his real identity at the TEXTSECURE server, the attacker would have to prove knowledge of the private long-lived key of the victim, which he does not know.

5. Security Proofs

In the following, we provide security proofs for the building blocks of TEXTSECURE. Namely, we give a security proof for the cached ORKE protocol and prove the key derivation function and authenticated encryption to be secure.

We note that we do only prove security of the building blocks of TEXTSECURE here, not of their composition. We see this as a first step to provide a thorough security analysis of TEXTSECURE and encourage further research in this direction. This approach was succesful, for example when proving the security of TLS where Jager et al. [32] built on provable security results of the building blocks (e.g., [33]).

5.1. The computation of the long-lived symmetric secret is a secure ORKE protocol

One of the main design criteria as described by Marlinspike in his blog entry on "*Axolotl-ratchet*" [10] is the replacement of the three-message ratchet ("key refreshment") used by OTR by a two-message ratchet. This, however, disables the MAC-based "chain of trust" for successive ephemeral DH shares. As a solution, TEXTSECURE first establishes a shared, long-lived, symmetric secret rk_{BA} between sender A and receiver B. This secret is then used, together with a fresh DH secret, in the TEXTSECURE key derivation function.

In this section we describe the establishment of this secret as a *cached one-round key establishment protocol* (cORKE, cf. Figure 8), where the first message has been cached on the TEXTSECURE server. In the TEXTSECURE cORKE protocol, each party B generates one *long-lived* key pair (b, g^b) and several *one-time* DH key pairs (x_b, g^{x_b}). The public shares of these key pairs are then registered. Before party A can send the first message to B, he has to request the long-lived and one one-time public DH share from the TEXTSECURE server. Party A then chooses a fresh ephemeral DH key pair (x_a, g^{x_a}), and sends the public share g^{x_a} and his long-lived public share g^a together with the encrypted message and additional data to B. A and B then both derive an intermediate secret sec_{int} by computing 3 of the 4 possible combinations of their public shares: $sec_{int} := (DH(g^a, g^{x_b}), DH(g^b, g^{x_a}), DH(g^{x_a}, g^{x_b}))$. The long-lived secret rk_{BA} shared between A and B is then derived by applying the key derivation function $HKDF$ to this intermediate secret and two constant values: $(rk_{BA}, ck_{BA}) = HKDF(sec_{int}, \text{const}_0, \text{const}_R)$.

Security models for ORKE protocols mostly follow Canetti and Krawczyk [34], and the many different variants of these models are often summarized under the term "extended Canetti-Krawczyk" (eCK) model. All these models differ slightly in the definition of a "protocol session", and in the adversarial capabilities.

Computational model. We model each instance of TEXTSECURE installed on a mobile device as a party P_i, $i \in \{1, ..., n\}$. Each party keeps track of the long-lived and medium-lived variables (i.e., for party P_A the long-lived private key a, the corresponding public key g^a, the identity A, and all the registered one-time key pairs).

For each communication started with another party, P_i forks off a (medium-lived) process π_i^s, $s \in \{1, ..., \ell\}$, to maintain the different session key variables. For cORKE (cf. Figure 8) and π_A^s, these variables include the identity B of the communication partner P_B as requested in message (2), the long-lived key g^b and the one-time key g^{x_b} as received in message (3), the ephemeral secret key x_a, the value sec_{int}, and the pair (rk_{BA}, ck_{BA}).

Each child process may request a registered private key from the party by presenting a reference to the corresponding public key.

Remark: This computational model seems overidealized, since the different sessions are most likely implemented as threads, not as separate processes. However, in order to allow Reveal queries targeting only session-specific variables, some kind of memory separation should be in place, to avoid fatal effects like reading the complete OpenSSL memory with a HeartBleed attack. So this computational model should be seen as without memory separation, the proof may not hold. Or to put it the other way round: Without memory separation, we must remove Reveal queries from the adversarial model to be able to prove security.

The TEXTSECURE server is modelled as an additional party, since this server does not compute any session-specific values.

Adversarial model. Adversarial capabilities are formalized as queries, where the fact that an active adversary controls the whole communication network is modeled as a Send query, together with various variants of queries which reveal secret data from all but the Test session. The multiplicity of different secret values established during the TEXTSECURE protocol (cf. Section 6.2) makes it difficult to correctly define queries revealing any long-, medium- or short-lived values to the adversary. However, this is not the goal of this section: we only want to prove security of the cORKE building block. Therefore we formalize adversarial capabilities as the following queries:

- Send(π_i^s, m). With this query, the adversary $\mathcal{A}$ can send message m to process oracle π_i^s. The oracle will process this input according to the protocol specifications, and return any output it produces to $\mathcal{A}$. The Send query may also be used to send messages to the TEXTSECURE server. This query models that an active attacker may control the com-

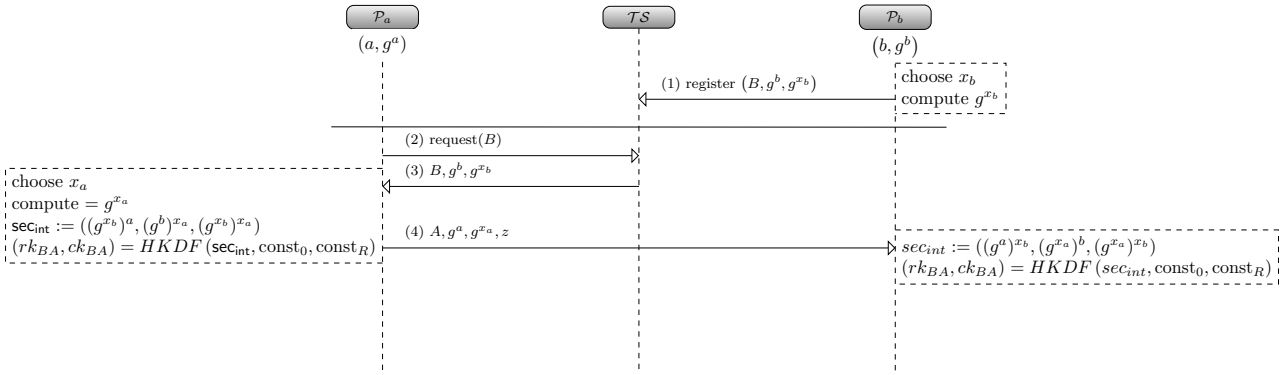

Figure 8: The cached ORKE protocol in TEXTSECURE to compute a shared secret $secret$ between parties A and B. Please note that z is a pointer to g^{x_b}.

plete communication between different process oracles and the TEXTSECURE server, and may delete, modify, delay or inject arbitrary messages.

- Reveal(π_i^s). This query will reveal the values (rk_{BA}, ck_{BA}) if they have already been computed by the process oracle. If this computation is not finished yet, the query will return $\perp$.
- Corrupt(P_i). This query will return the long-lived secrets of party P_i to $\mathcal{A}$, which consist of the long-lived private key and the password. After this $\mathcal{A}$ is able to completely impersonate P_i, including registering one-time keys at the TEXTSECURE server.
- Test(π_i^s). This query can only be asked once, and is answered by the oracle in the following way: The oracle flips a fair coin b, and returns k_b, where $k_0 = (rk_{BA}, ck_{BA})$ and $k_1 = (r_1, r_2)$ for two random values r_1, r_2 chosen from the same distribution.

Please note that the TEXTSECURE server may be addressed with Send queries, but not with (a) Reveal or (b) Corrupt queries, because (a) he does not compute any session secrets, and (b) corruption of the TEXTSECURE server would also reveal long-lived passwords, and thus the registration phase would become insecure by definition.

Security model. To define security, first we have to exclude trivial attacks which result from a combination of the above queries, but which do not correspond to any real attacks. The simplest example is asking Reveal and Test to the same oracle, which does not make sense. More subtly, we must not allow this combination of queries against pairs of oracles which should compute the same key, i.e., we must define what a *protocol session* is. For protocol sessions, recently Cremers [35] and Bergsma *et al.* [36] independently came up with a satisfactory definition, which we use.

Definition 1 (Origin session)**.** *Consider two parties A and B with processes π_A^s and π_B^t, and let $M^{in}(\pi)$ and $M^{out}(\pi)$ be the sequences of messages received and sent by process π, resp. We say π_A^s has* origin session *π_B^t, if $M^{in}(\pi_A^s) = M^{out}(\pi_B^t)$, and denote this by $\pi_A^s \leftarrow \pi_B^t$.*

In Figure 8 we assume that the one-time key retrieved by P_a in message (2) is unique, and that the TEXTSECURE server does not issue the same key twice. (We do not consider the last ressort key here.) The adversary $\mathcal{A}$ may forward message (4) to different process oracles at party P_b, which consequently would all compute the same key. We consider this (trivial) attack by restricting the security game with the help of the concept of origin sessions.

Definition 2 (Freshness)**.** *We say that an oracle π_i^s is* fresh *if neither party P_i nor the intended partner P_j of this oracle is corrupted, and*

- *if the oracle is an initiator oracle (i.e., if it retrieves a one-time key from the* TEXTSECURE *server), then no* Reveal *query has been asked against any oracle π_j^t with $\pi_j^t \leftarrow \pi_i^s$, or*
- *if the oracle is a responder oracle, the no* Reveal *query has been asked against the unique initiator oracle π_j^t with $\pi_i^s \leftarrow \pi_j^t$*

Definition 3 (cORKE Security Game)**.** *Let $\mathcal{E}$ be a polynomial time adversary, and let $\mathcal{C}$ be a challenger who simulates the* TEXTSECURE *communication network. After observing and manipulating several instances of the cached ORKE protocol from Figure 8 by using the* Send*,* Reveal *and* Corrupt *queries, $\mathcal{A}$ may choose one fresh oracle which just has successfully completed this protocol, and send a TEST query to this oracle. In our security game, this oracle will now set $s_0 := (rk_{BA}, ck_{BA})$ and $s_1 \xleftarrow{\$} \{0,1\}^{length(HKDF_{out})}$. Then the party throws a fair coin and uses the result b to return s_b to the adversary. The adversary may now perform some additional computations involving* Send*,* Reveal *and* Corrupt*, and eventually issue a bit b'. He wins the game if $b' = b$.*

Since an attacker may always win this game with probability $\frac{1}{2}$ by just guessing b, we cannot use this winning probability directly in our security definition. A protocol should only be considered broken if an attacker can do *significantly* better that just guessing.

Definition 4 (cORKE security)**.** *Let $Pr_{\mathcal{E}}(b' = b)$ be the probability that attacker $\mathcal{E}$ wins the game described in*

Definition 3. We say that an attacker breaks *the cORKE protocol if* $Adv_{\mathcal{E}} := |Pr_{\mathcal{E}}(b' = b) - \frac{1}{2}|$ *is non-negligible. The cORKE protocol is* secure *if no such attacker exists.*

Theorem 1. *If the Gap-DH assumption holds*[10] *in* Curve25519 *and we model* HKDF *as a non-programmable random oracle, then cORKE is a secure cached ORKE protocol and we have*

$$Adv_{\mathcal{A}} \leq \frac{(n\ell)^2}{2^q} + (n\ell)\epsilon_{gDH}.$$

Proof. The proof is organized as a sequence of games. Let $\mathcal{A}$ be an arbitrary adversary, and let $\mathcal{C}$ be the cORKE challenger. Let q be the order of Curve25519.

Game 0: This is the original cORKE security game.

Game 1: In this game we exclude collisions in the DH shares for the cORKE subprotocol. (For the full TEXT-SECURE protocol we would need a different, but still negligible bound, because of the ephemeral shares used for the ratcheting.) Since there are n parties, and each party may fork off at most ℓ process oracles, at most $n\ell$ DH shares (both one-time and ephemeral) are used in the different cORKE protocols. The challenger will now abort the game if any of these shares have the same value. We get $Pr^0_{\mathcal{A}}(b' = b) \leq Pr^1_{\mathcal{A}} + \frac{(n\ell)^2}{2^q}$, and therefore also

$$Adv^0_{\mathcal{A}} \leq Adv^1_{\mathcal{A}} + \frac{(n\ell)^2}{2^q}.$$

Game 2: In this game we guess which of the $n\ell$ oracles will be asked the Test query. Note that according to our security definition, neither the party of this oracle, nor its intended partner may be corrupted. If our guess is wrong, the simulation is aborted and the bit b' is chosen randomly. This gives us

$$Adv^1_{\mathcal{A}} = Adv^2_{\mathcal{A}} \cdot (n\ell).$$

Game 3: We now want to embed our CDH challenge (g, g^c, g^d), where g is the generator of Curve25519 used in our computations, into the simulation. To do this, we have to distinguish two cases:

1) The test oracle is a initiator oracle π^i_A. In this case we know that x_a is chosen by the test oracle itself, so that the adversary cannot influence this value. Additionally, we assume that the public key g^b of the intended partner has been authenticated offline. On the other hand, $\mathcal{A}$ may choose x_b by herself. Thus we set $(g^b, g^{x_a}) = (g^c, g^d)$.
2) The test oracle is a responder oracle π^t_B. In this case, $\mathcal{A}$ may choose x_a, and thus we have to embed the CDH challenge as $(g^a, g^{x_b}) = (g^c, g^d)$.

Since HKDF is modeled as a random oracle, we may replace HKDF by a programmable random oracle RO simulated by the challenger $\mathcal{C}$. The adversary may now follow two strategies: either he tries to compute b' by querying RO, or he doesn't. If he doesn't, then his advantage in this game is 0, since s_0 and s_1 are both random.

10. For instance, the CDH assumption holds but we have an oracle to simulate the DDH assumption [37, p. 278f].

If he does query RO, then he has to provide the full input $(g^{x_b \cdot a}, g^{b \cdot x_a}, g^{x_a \cdot x_b}, \text{const}_0, \text{const}_R)$. In case 1, the second value is the solution to the CDH challenge, and in case 2 it is the first value. Since $\mathcal{C}$ simulates RO, he sees these values. With the help of the DDH oracle $\mathcal{C}$ can check if these values are the solution to the CDH problem.

For all other simulation steps, $\mathcal{C}$ only has to check if the input provided by either $\mathcal{A}$ or by one of the simulated oracles has already been asked to RO, and either return a freshly chosen random value, or the value stored in the database. To be able to do so, we need the DDH oracle from the gap-DH assumption. For example, if we use g^b to embed one value from the CDH assumption, and the adversary sends message (4) where x_a is chosen by the adversary, then $\mathcal{C}$ cannot compute $(g^{x_a})^b$; however, with the help of the DDH oracle, $\mathcal{C}$ can check if $CDH(g^{x_a}, g^b)$ equals any value queried before, and thus adapt the answer.

Thus we have

$$Adv^2_{\mathcal{A}} = Adv^3_{\mathcal{A}} + \epsilon_{gDH}.$$

□

5.2. Key derivation

Definition 5 (Key derivation function [38])**.** *A key derivation function* KDF *is a function* KDF *that takes as input 1) a value* SKM *sampled from a source of keying material, 2) a length value l, 3) a salt value* XTS *, and 4) a context variable* CTXInfo *(the last two inputs being optional) and outputs a string of l bits.*

Let us elaborate on this definition. First of all, one may wonder, why we cannot simply use a secure pseudo-random function for key derivation. The reason is that PRFs require a key of a certain length, e.g., a key of 512 bits in HMAC SHA-256 and with sufficient high min-entropy, say, at least 160 random bits. However, it may occur that though the key SKM that is input to KDF provides enough entropy, it is not of the correct size to be used with the considered PRF. This may occur, if e.g., SKM is the result of Diffie-Hellman key exchange in a group of prime order. Here, it is usually the case that the Diffie-Hellman protocol is carried out in a group of order roughly 160 bits. Since by the Diffie-Hellman assumption the secret (g^{ab}) is indistinguishable from a random value in that particular group, it provides us with 160 uniform bits. However, to thwart attacks that deploy the structure of the field, the group is actually a subgroup of a group of order roughly 2048 bit. Thus, though the secret provides us with 160 uniform bits, the element that represents the secret is actually of size 2048 bits.

Thus, the goal of a key derivation is actually twofold: On the one hand, the random bits from SKM need to be extracted (for reasons described above) and on the other hand they need to be expanded, that is an output of the desired length l is to be produced (we often omit l where the output length is clear from the context). The value XTS (extractor salt) in Definition 5 may contain additional *public* information on the extraction step (see below). It may,

Algorithm 1 $\mathsf{HKDF}(\mathsf{key}, \mathsf{salt}, \mathsf{contextInfo})$

$k_{pr} \leftarrow \mathsf{HMAC}(\mathsf{salt}, \mathsf{key})$
$k_0 \leftarrow \mathsf{HMAC}(k_{pr}, \mathsf{contextInfo}||0x00)$
$k_1 \leftarrow \mathsf{HMAC}(k_{pr}, k_0||\mathsf{contextInfo}||0x01)$
return (k_0, k_1)

however, also be empty. The context-variable is used to use SKM for different purposes.

Let us in the following assume that there is an algorithm $(\mathsf{SKM}, \alpha) \xleftarrow{\$} \Sigma(1^k)$ that inputs the security parameter and outputs a key SKM and additional information α that is available to the adversary about SKM. We will also refer to Σ as the distribution of (SKM, α) and we will denote by Σ_{SKM} and Σ_α the corresponding marginal distributions on SKM and α. Moreover, let us denote by μ_{SKM} the Min-Entropy of Σ_{SKM}.

In our scenario, Σ will be the cORKE protocol from above, where SKM is the key and α is the transcript that is available to the attacker. In case cORKE is secure it holds that $2^{-\Sigma_{\mathsf{SKM}}} \leq \mathsf{negl}$.

Next, we define the security of a KDF with respect to Σ. To this end, consider the following security game that is played between a challenger and an adversary $\mathcal{A}$ and that is parametrized by Σ, and q the number of queries the adversary may issue. We slightly deviate from the definition in [38] in that we do not allow $\mathcal{A}$ side information on SKM. This suffices in our case since the key exchange phase is secure (see above). On the other hand, we allow $\mathcal{A}$ to choose XTS adaptively for each evaluation of KDF.

1) First, $\mathcal{C}$ runs $(\mathsf{SKM}, \alpha) \xleftarrow{\$} \Sigma(1^k)$ and passes α to $\mathcal{A}$.
2) $\mathcal{A}$ may now adaptively query $\mathcal{C}$ on arbitrary values $(\mathsf{XTS}_i, c_i, l_i)$, for $i = 1, \ldots, q' \leq q$. When $\mathcal{A}$ issues query (c_i, l_i), $\mathcal{C}$ returns $\mathsf{KDF}(\mathsf{SKM}, l_i, \mathsf{XTS}, c_i)$.
3) At some point $\mathcal{A}$ chooses values XTS^*, c^*, l^* such that $\mathsf{XTS}^* \notin \{\mathsf{XTS}_1, \ldots, \mathsf{XTS}_{q'}\}$. $\mathcal{C}$ flips a bit $b \xleftarrow{\$} \{0,1\}$. If $b = 1$, then $\mathcal{C}$ returns $\mathsf{KDF}(\mathsf{SKM}, l^*, \mathsf{XTS}, c^*)$, otherwise, it provides $\mathcal{A}$ with a uniformly random bit string of length l.
4) $\mathcal{A}$ may continue to adaptively query $\mathcal{C}$ on $(\mathsf{XTS}_i, c_i, l_i), i = q' + 1, \ldots, q$ such that $\mathsf{XTS}_i \neq \mathsf{XTS}^*$.

Definition 6 (Security of KDF)**.** *We say that a KDF* KDF *is secure with respect to* Σ *if it is computationally hard to determine the value of* b *significantly better than by guessing in the above game.*

Key Derivation in TextSecure. The core of the key derivation in TEXTSECURE is the HMAC based key derivation function HKDF due to Krawczyk [38] that is depicted in Algorithm 1. Krawczyk showed that under reasonable assumptions, HKDF is a secure key derivation function.

Let us denote by rk_{BA} the secret established in the cORKE phase and by $g^{x_{b,z}}$ the one-time key of B. After sampling $(x_{a,2}, g^{x_{a,2}}) \xleftarrow{\$} \mathbb{Z}_p \times \mathsf{Curve25519}$ and computing $k_{\text{shared}} = g^{x_{b,z} \cdot x_{a,2}}$, key derivation in TEXTSECURE proceeds as depicted in Algorithm 2.

Algorithm 2 $\mathsf{KDF}_{\mathsf{TS}}(rk_{BA}, k_{\text{shared}}, \mathsf{aux})$

1: $(\text{const}_R, \text{const}_2, \text{const}_1, \text{const}_0, \text{const}_K) \leftarrow \mathsf{aux}$
2: $(\mathsf{k}^r_{AB}, \mathsf{k}^c_{AB}) \leftarrow \mathsf{HKDF}(k_{\text{shared}}, rk_{BA}, \text{const}_R|| \text{ const}_2)$
3: ${k_{\text{shared}}}^{new} \leftarrow \mathsf{HMAC}(\mathsf{k}^c_{AB}, \text{const}_1)$.
4: $(\mathsf{key}_{\mathsf{Enc}}, \mathsf{key}_{\mathsf{MAC}}) \leftarrow \mathsf{HKDF}({k_{\text{shared}}}^{new}, \text{const0}, \text{const}_K)$.

Here, variable aux contains the strings const_R, const_2, const_1, const_0 and const_K and is parsed as these in the first step. The length variable l is set to $l = 2$ for $\mathsf{KDF}_{\mathsf{TS}}$.

We note that though the source of keying material SKM is (close to) uniformly distributed over Curve25519, it contains only ephemeral and non-authenticated values. rk_{BA}, the only value that is uniformly and authenticated, acts as *salt*, when HKDF is first called (and not as source of keying material). Thus, the key to $\mathsf{KDF}_{\mathsf{TS}}$ is rk_{BA} the secret from the cORKE protocol and not k_{shared} which is the key when HKDF is called first. Therefore we can not apply [38, Theorem 4] on the KDF-security of HKDF directly. Rather we show the following:

Theorem 2. $\mathsf{KDF}_{\mathsf{TS}}$ *is a secure* KDF *with respect to cORKE if* HKDF *is modelled as a non-programmable random oracle and* HMAC *is a secure* PRF.

Proof. Again, the proof follows the sequence of games approach. Let χ_i denote the event that the adversary breaks security of $\mathsf{KDF}_{\mathsf{TS}}$ in game i.

GAME 0. This is the real security game as per Definition 5. Note that here, we have $\mathsf{SKM} = rk_{AB}, \mathsf{XTS} = k_{\text{shared}}$ and $\mathsf{CTXInfo} = \mathsf{aux}$. In particular, CTXInfo and l are fixed.

GAME 1. In Game 1, we proceed similar to Game 0, except for the following. Instead of sampling $(\mathsf{SKM}, \alpha) \xleftarrow{\$} \Sigma(1^k)$ (i.e., running $\Sigma(1^k)$) we sample SKM and α *independently* from the marginal distributions Σ_{SKM} and Σ_α. Stated otherwise, we run $(\mathsf{SKM}', \alpha) \xleftarrow{\$} \Sigma(1^k)$ and afterwards replace SKM′ with a uniformly random value SKM. By the security of the cORKE protocol (Theorem 1) we have

$$\mathsf{Pr}[\chi_0] - \mathsf{Pr}[\chi_1] \leq \mathsf{negl}$$

GAME 2. Game 2 proceeds similar to Game 1, except that $\mathcal{A}$ loses whenever $\mathcal{A}$ issues a query $(\cdot, \mathsf{SKM}, \cdot)$ to the HKDF-oracle. Since SKM is sampled indpendent of α from a source with Min-Entropy μ_{SKM} such that $2^{-\mu_{\mathsf{SKM}}} \leq \mathsf{negl}$, we have

$$\mathsf{Pr}[\chi_1] - \mathsf{Pr}[\chi_2] \leq \mathsf{negl}$$

GAME 3. Game 3 proceeds similar to Game 2, except for the following. When $\mathcal{A}$ issues XTS^* to the challenger, it samples $(k^r_{AB}, k^c_{,}AB)$ uniformly at random, instead of computing $(k^r_{AB}, k^c_{,}AB) \leftarrow \mathsf{HKDF}(k_{shared}, rk_{BA}, \text{const}_R||\text{const}_2)$. Due to Game 2, $\mathcal{A}$ will never issue such query to the HKDF-oracle (recall that $\mathsf{SKM} = rk_{BA}$) and thus, the change in Game 3 is purely conceptual. It follows:

$$\mathsf{Pr}[\chi_2] = \mathsf{Pr}[\chi_3]$$

GAME 4. In Game 4, we proceed similar to Game 3, for the following. When $\mathcal{A}$ issues XTS^*, the challenger does not compute $k^{new}_{shared} \leftarrow \mathsf{HMAC}(k^c_{AB}, \mathrm{const}_1)$ but samples k^{new}_{shared} uniformly at random. By the PRF-properties of HMAC we have

$$\Pr[3] - \Pr[4] \leq \mathsf{negl}$$

GAME 5. Game 5 proceeds similar to Game 4, except that $\mathcal{A}$ loses whenever $\mathcal{A}$ issues a query $(k^{new}_{shared}, \cdot, \cdot)$ to the HKDF-oracle. By Game 4 it follows

$$\Pr[\chi_4] - \Pr[\chi_5] \leq \mathsf{negl}$$

Claim 1. $\Pr[\chi_5] \leq \mathsf{negl}$.

Due to Game 5, $\mathcal{A}$ never issues a query $(k^{new}_{shared}, \cdot, \cdot)$ to the HKDF-oracle. By Game 4, we have that (k^{new}_{shared}) is the key that is input to this last evaluation of HKDF when $\mathcal{A}$ issues XTS^*. It follows that the output of HKDF is uniformly random from the adversaries point of view. The claim follows. □

5.3. Encryption of Messages is Authenticated Encryption

The goals of authenticated encryption are indistinguishability of plaintext even in the presence of a *decryption* oracle (IND-CCA-security) and integrity of ciphertexts (INT-CTXT-security) [39] (which also guarantees plaintext-integrity).

Algorithm 3 $\mathsf{TS.Enc}(k, hd, m)$

$c \leftarrow \mathsf{Enc}_{\mathsf{k}_{\mathsf{Enc}}}(m)$
$\chi \leftarrow (v, g^{\overline{x}_{a,2}}, \mathsf{ctr}_a, \mathsf{pctr}_a, c)$
$\mathsf{tag} \leftarrow \mathsf{MAC}_{\mathsf{k}_{\mathsf{MAC}}}(\chi)$
return $\mathbf{C} \leftarrow (\chi, \mathsf{tag})$

Algorithm 4 $\mathsf{TS.Dec}(k, hd, \mathbf{C})$

parse $\mathbf{C}$ **as** $\mathbf{C} = (\chi', \mathsf{tag}')$
parse χ' **as** $\chi' = (v', (g^{\overline{x}_{a,2}})', \mathsf{ctr}'_a, \mathsf{pctr}'_a, c')$
$\mathsf{tag}'' \leftarrow \mathsf{MAC}_{\mathsf{k}_{\mathsf{MAC}}}(\chi')$
if $\mathsf{tag}'' \neq \mathsf{tag}'$ **return** $(\perp, \perp)$
$m \leftarrow \mathsf{Dec}_{\mathsf{k}_{\mathsf{Enc}}}(c)$
if $m = \perp$ **return** $(\perp, \perp)$
return m

IND-CCA-security guarantees security of encrypted messages, even if the adversary is provided with encryption and decryption oracles. The practical motivation stems from the discovery of padding oracle attacks (e.g., [40]). INT-CTXT-security guarantees that an adversary can not create well-formed ciphertexts, even when having access to an encryption oracle.

The ENCRYPT-then-MAC (EtM) construction produces a ciphertext by encrypting the plaintext with a symmetric encryption scheme (SE) and protecting the output with a message authentication code (MAC). Bellare and Namprempre have shown [39] that the EtM construction achieves these goals under the assumptions that 1) MAC is *strongly* unforgeable and 2) SE is IND-CPA-secure. As depicted in Algorithms 3 and 4, TEXTSECURE also follows this EtM-approach.

Thus, we obtain the following:

Theorem 3. $(\mathsf{TS.Enc}, \mathsf{TS.Dec})$ *provides* IND-CCA*-security and* INT-CTXT*-security if AES in counter mode is* IND-CPA*-secure and* HMAC *is a strongly unforgeable message authentication code.*

It thus suffices to show that the underlying primitives satisfy the respective security notions. We recall these properties here:

- IND-CPA-security, i.e., security of encrypted messages, even if the adversary is provided with an encryption oracle, holds for AES in counter-mode if AES itself is a *pseudo-random permutation*.
- A message authentication code is *strongly unforgeable* if it is computationally hard for an adversary that has access to an oracle that computes tags for messages of the adversary's choice, to output a pair (m^*, tag^*) such that tag^* is a valid tag for m that has not been previously output by the oracle. In particular, we require that it is hard for the adversary to compute a tag tag' for a message m, even if it is given pairs (m, tag) for the same message such that $\mathsf{tag} \neq \mathsf{tag}'$. For any message m and key k there is only one tag such that $\mathsf{tag} = \mathsf{HMAC}(k, m)$ and thus there is a unique tag that will be accepted by the verification-algorithm of HMAC for any message. It is well known [41] that any unique MAC is strongly unforgeable if and only if it is a secure MAC (i.e., if an existential forgery is computationally hard). Thus, by the results of [42], we may assume that this property is satisfied for HMAC.

6. Claimed Additional Security Features

We now briefly discuss how the TEXTSECURE protocol achieves *forward secrecy* and also a security notion called *future secrecy*. Furthermore, we show that TEXTSECURE does *not* achieve deniability in the sense of the definition given for the OTR protocol.

6.1. Future Secrecy

The authors of TEXTSECURE formulated the security goal of *future secrecy* [43]. Future secrecy guarantees that if, in a conversation between A and B, at some point in time $\mathsf{k}_{\mathsf{Enc}}, \mathsf{k}_{\mathsf{MAC}}$ or ephemeral DH exponents were leaked, the protocol will recover and from a certain point in time on future messages will be secured again.

Please note that future secrecy does not hold if the whole key material of one of the parties is leaked: for example, if

rk_{BA} (rk_{AB}, resp.) and one private exponent is leaked, the attacker may act as a man-in-the-middle and read and alter all future messages. So one precondition for future secrecy is that long-lived keys remain secret.

If the long-lived keys remain secret, future secrecy will be enforced with each new ratchet: Since rk_{AB} (rk_{BA}, resp.) is one input to the key derivation function, no attacker will be able to control the new keys from this point on.

However, the question whether TEXTSECURE does achieve future secrecy depends on the amount of information that may leak to an attacker. We discuss this issue in more depth below for the classic notion of perfect forward secrecy.

6.2. Perfect Forward Secrecy

Perfect forward secrecy (PFS) [44]–[46] is a desirable goal in cryptographic systems since its guarantees that if one of the communicating parties is forced to reveal its private and secret keys, nevertheless all messages remain secure (i.e. they cannot be decrypted with the revealed keys) that were sent *prior* to this event. All major key exchange protocols support PFS (e.g. TLS-DHE, SSH and IPSec IKE), and a formalization has been proposed by Jager et al. [32]. PFS does not prevent messages sent after this event from being read, since an attacker may now impersonate one of the communicating parties by using the revealed keys.

When claiming PFS for TEXTSECURE, we first have to define which keys could be revealed. Since the TEXTSECURE server does not store any private or secret end-to-end communication keys, those keys can only be revealed from a TEXTSECURE app. Here the following types of keys are stored:

- Long-lived keys. The TEXTSECURE app used by party $\mathcal{P}_a$ has to store the private DH key a for the lifetime of the app, or until the user changes this value. Additionally, it has to store the long-lived shared secrets sec_{AB} for any communication partner $\mathcal{P}_b$.
- Medium-lived prekeys. Here party $\mathcal{P}_a$ must store the private prekeys $x_{a,z}$ for all public prekeys stored at the TEXTSECURE server. These keys should be erased once they have been used in a communication, and a chaining key has been derived. The only exception from this deletion rule is the last resort key, which can only be erased if new prekeys have been generated and uploaded to the server.
- Short-lived keys. For each new ratchet in the communication flow, $\mathcal{P}_a$ must generate and store the ephemeral private DH key x_a until it is used a second time to derive master and chaining keys.

The revealing of long-lived keys alone does not compromise the security of any message in TEXTSECURE, since they only influence one parameter of the key derivation, namely the shared secret sec_{AB}.

If however in addition some medium or short-lived keys are revealed (which are stored on the same device), the situation becomes more complex: If a key $x_{a,z}$ has already been used for sending messages to A, but A has not received these messages at the time of the key reveal, then somehow "previous" messages can be decrypted. The same holds for keys x_a who must be kept stored until an answer from the other party B has been received.

So strictly speaking, TEXTSECURE again only fulfills a weaker variant of PFS, but of course the large majority of "previous" messages is protected by PFS.

6.3. Deniability

Deniability, which was one of the primary design goals in the OTR protocol, can be described as follows: *Both* sender and receiver of a message should be able to deny that they have sent or generated a message, by showing that *any* other party may also have sent and generated a particular message. OTR achieves this property by using a malleable encryption scheme and periodically publishing the MAC key after the receiver has verified a message's authenticity. Thus, *anyone* could have modified the messages plaintext and produced a valid MAC.

TEXTSECURE encrypts and authenticates all messages with symmetric primitives. Assuming the key-exchange provides authenticated keys, either of the two communication parties knows that the received messages come from the communication partner. However, since both parties compute the same keys for encryption and MAC, neither of them can cryptographically attribute authorship. This is the nature of symmetric cryptography.

Looking at the protocol, we see that parties are able to simulate the whole key exchange themselves, since direct interaction of a second party is unnecessary thanks to prekeys cached by the server. The same can be done during any real conversation: at time t, a party $\mathcal{P}_a$, communicating with $\mathcal{P}_b$ begins to derive new ephemeral DH shares for $\mathcal{P}_b$ to simulate an ongoing conversation. As long as there is no proof that $\mathcal{P}_b$ was able to re-enter the session at a time $t + n$, $\mathcal{P}_b$ can deny any involvement. To conclude the discussion, the design of the TEXTSECURE protocol achieves deniability on a protocol level.

However, in practice the following happens: when a party sends a message, it needs to do so via the TEXTSECURE server. The message itself is encrypted, but in order to guarantee correct delivery, the identities of the sender (regID_a) and recipient (regID_b) have to be transmitted to the server. The sending request is authenticated with the sender's phone number and password.

The server is not able to read the content of any message and handles only the delivery. Therefore no one can prove—even if the server collaborates—which content was sent. However, one major goal regarding deniability is the ability to plausibly disclaim any involvement in a conversation or to deny having a conversation to some party at all. Looking at the delivery procedure of TEXTSECURE , this goal is not achieved if the server logs all delivery requests.

In conclusion, TEXTSECURE only achieves deniability theoretically. Content deniability is provided due to our

security proof but we can not prove that no delivery request will be recorded at the TEXTSECURE server.

7. Related Work

At the 2004 Workshop on Privacy in the Electronic Society (WPES), Borisov et. al. [3] presented a protocol for "Off the Record" (OTR) communication. The OTR protocol was designed to provide authenticated and confidential instant messaging communication with strong perfect forward secrecy and *deniability*: no party can cryptographically prove the authorship of a message. The deniability property of OTR has been discussed by Kopf and Brehm [47]. The work of Di Raimondo et. al. [13], who analyzed the security of OTR, is in its nature closely related to our paper. The authors point out several issues with OTR's authentication mechanism and also describe a UKS attack on OTR, as well as, a replay attack along with fixes. We note, however, that the authentication mechanisms of OTR and TEXTSECURE have little in common: Though it aims to provide deniability, OTR explicitly uses signatures for authentication while TEXTSECURE does not. The UKS attack on OTR described by Raimondo et al. [13] directly targets the key exchange mechanism of the protocol, whereas the attacks presented in this paper are rather subtle and exploit the protocol structure and key derivation of TEXTSECURE .

Besides OTR, which has been widely adopted, there exist protocols for secure instant messaging like IMKE [48], which aim at being verifiable in a BAN-like logic, but has never found a wider adoption. SILC [49] also has received a certain adoption and some discussion, but is rarely used today, as is FiSH [50], a once popular plugin for IRC clients that used Blowfish with pre-shared keys to encrypt messages.

Thomas [51] analyzed the browser-based instant messaging client Cryptocat and found that, due to an implementation error, Cryptocat used private keys of insufficient length when establishing a group chat session. Green [52] recently discussed Cryptocat's group chat approach from a protocol perspective and points out several issues.

Further protocols exist that aim at securing instant messaging communication but have, to the best of our knowledge, not received public scrutiny. Among these are Silent Circle's SCIMP, THREEMA[11] and SURESPOT[12]. Additionally, the security of GCM in Android has been analyzed at CCS'14 [53]. The work revealed serious security issues allowing to intercept user messages, for example, Skype messages, or even command Android service apps, like (un-)installing any apps stealthily. However, our work concentrates on the cryptographic protocol analysis and treats TEXTSECURE 's GCM usage transparently.

8. Conclusion and Future Work

Mobile messaging apps with security guarantees are becoming more and more popular. Prominent mobile applications for *secure* IM are THREEMA, SURESPOT, and TEXTSECURE. In this paper, we have provided a detailed security analysis of TEXTSECURE. First, we precisely described the protocol and then performed a security analysis of the individual steps of the protocol. This led to the discovery of several peculiarities, most notably the protocol's susceptibility for a UKS attack. We proposed a mitigation and showed that, if our mitigation is applied, TEXTSECURE actually provides authenticated encryption. To the best of our knowledge, this is the first formal verification of the security guarantees offered by the app.

While TEXTSECURE's implementation is open source, little is known about the competing messaging applications. While THREEMA makes use of the open source library NaCl [54] for cryptographic operations, its protocol is kept confidential.

SURESPOT is an open source project with its own cryptographic protocol; an analysis of its protocol has (as far as we know) not been done yet. A comparison of TEXTSECURE and SURESPOT will be an interesting project for future work.

On October 18th 2014, Open Whispersystems and several news outlets reported that WHATSAPP, a mobile messaging application with more than 500M installations via *Google Play*, now uses TEXTSECURE's protocol to provide end-to-end encryption for its users [55], multiplying the protocol's user base. Besides its Android app, WHATSAPP does also offer an app for Apple's iOS platform, and a web-based client. TEXTSECURE's protocol can only be used with WHATSAPP, if the two communicating parties are both using Android and one of the latest version of WHATSAPP. In the context of the protocol's role-out to WhatsApp, this results in WHATSAPP's server deciding whether two users can communicate encrypted [56]. However, there is no indication to users, whether they are communicating end-to-end encrypted or not. While it is not unexpected that WHATSAPP does not offer a way for users to authenticate each other's key – naturally, WHATSAPP suffers from the same shortcomings, as TEXTSECURE– WHATSAPP does not even offer a way for users to compare key fingerprints. Altogether the existing differences warrant the conclusion that one should not deduce WHATSAPP's security properties from TEXTSECURE's. Instead, a thorough analysis of WHATSAPP can be subject of future research.

Acknowledgments

The authors thank Dominik Preikschat for the initial documentation of TextSecure's source code during his bachelor thesis, David L. Gil for highlighting the application of HMAC in the Random Oracle Model, Brian Smith for pointing out several publications, and Tibor Jager, as well as, our anonymous reviewers and Trevor Perrin for their comments and insightful discussion.

The research was supported by the German Ministry of Research and Education (BMBF) as part of the VERTRAG research project.

11. https://threema.ch/
12. https://surespot.me/

References

[1] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880 (Proposed Standard).

[2] B. Ramsdell and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification," RFC 5751 (Proposed Standard).

[3] N. Borisov, I. Goldberg, and E. A. Brewer, "Off-the-record communication, or, why not to use pgp," in *WPES*, 2004, pp. 77–84.

[4] V. Moscaritolo, G. Belvin, and P. Zimmermann, "Silent circle instant messaging protocol protocol specification," 2012.

[5] Electronic Frontier Foundation, "Secure messaging scorecard," 2014. [Online]. Available: https://www.eff.org/secure-messaging-scorecard

[6] Open WhisperSystems, "The new TextSecure: Privacy beyond SMS," Feb. 2014. [Online]. Available: https://whispersystems.org/blog/the-new-textsecure/

[7] ——, "TextSecure, now with 10 million more users," Dec. 2013. [Online]. Available: https://whispersystems.org/blog/cyanogen-integration/

[8] M. Crider, "CyanogenMod is now installed on over 10 million android devices," Dec. 2013, [online] Available: http://www.androidpolice.com/2013/12/22/ cyanogenmod-is-now-installed-on-over-10-million-android-devices/.

[9] T. Fox-Brewster, "WhatsApp adds end-to-end encryption using TextSecure," November 2014, http://www.theguardian.com/technology/2014/nov/19/whatsapp-messaging-encryption-android-ios.

[10] M. Marlinspike, "Advanced cryptographic ratcheting," https://whispersystems.org/blog/advanced-ratcheting/, November 2013.

[11] P. Z. Vinnie Moscaritolo, Gary Belvin, "Silent circle instant messaging protocol," https://silentcircle.com/sites/default/themes/silentcircle/assets/downloads/SCIMP_paper.pdf, December 2012.

[12] D. Dolev, C. Dwork, and M. Naor, "Nonmalleable cryptography," *SIAM Journal on Computing*, vol. 30, no. 2, pp. 391–437, 2000.

[13] M. D. Raimondo, R. Gennaro, and H. Krawczyk, "Secure off-the-record messaging," in *WPES*, 2005, pp. 81–89.

[14] W. Diffie, P. C. van Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges," *Des. Codes Cryptography*, 1992.

[15] H. Krawczyk, "SIGMA: The "SIGn-and-MAc" approach to authenticated Diffie-Hellman and its use in the IKE protocols," in *CRYPTO 2003*, 2003.

[16] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *CRYPTO'84*, 1984.

[17] C. Boyd, Y. Cliff, J. G. Nieto, and K. G. Paterson, "Efficient one-round key exchange in the standard model," in *ACISP 08*, 2008.

[18] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *PKC*, 2006. [Online]. Available: http://cr.yp.to/ecdh/curve25519-20060209.pdf

[19] "Advanced encryption standard (aes)," National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.

[20] R. Housley, "Cryptographic Message Syntax (CMS)," RFC 5652 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5652.txt

[21] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104 (Informational).

[22] D. E. 3rd and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)," RFC 4634 (Informational).

[23] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication," RFC 7235 (Proposed Standard).

[24] T. Perrin, "Axolotl ratchet," Jun. 2014. [Online]. Available: https://github.com/trevp/axolotl/wiki

[25] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting ssl/tls implementations: New bleichenbacher side channels and attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[26] A. Karjalainen and N. Mehta, "The heartbleed bug," 2014. [Online]. Available: http://heartbleed123.com/

[27] C. Meyer and J. Schwenk, "Sok: Lessons learned from ssl/tls attacks," in *WISA*, 2013, pp. 189–209.

[28] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," in *IEEE Symposium on Security and Privacy*, 2013.

[29] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt, "On the security of rc4 in tls," in *USENIX Security*, 2013.

[30] C.-P. Schnorr, "Efficient identification and signatures for smart cards," in *CRYPTO'89*, 1989.

[31] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," in *CRYPTO'94*, 1994.

[32] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *CRYPTO 2012*, 2012.

[33] K. G. Paterson, T. Ristenpart, and T. Shrimpton, "Tag size does matter: Attacks and proofs for the TLS record protocol," in *ASIACRYPT 2011*, ser. LNCS, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, Dec. 2011, pp. 372–389.

[34] R. Canetti and H. Krawczyk, "Analysis of key-exchange protocols and their use for building secure channels," Cryptology ePrint Archive, Report 2001/040, 2001, http://eprint.iacr.org/2001/040.

[35] C. J. F. Cremers and M. Feltz, "Beyond eCK: Perfect forward secrecy under actor compromise and ephemeral-key reveal," in *ESORICS 2012*, 2012.

[36] F. Bergsma, T. Jager, and J. Schwenk, "One-round key exchange with strong security: An efficient and generic construction in the standard model," in *PKC 2015*, 2015.

[37] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC Press, 2014.

[38] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *CRYPTO 2010*, 2010.

[39] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," *Journal of Cryptology*, vol. 21, no. 4, pp. 469–491, Oct. 2008.

[40] S. Vaudenay, "Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ..." in *EUROCRYPT 2002*, 2002.

[41] M. Bellare, O. Goldreich, and A. Mityagin, "The power of verification queries in message authentication and authenticated encryption," Cryptology ePrint Archive, Report 2004/309, 2004, http://eprint.iacr.org/.

[42] M. Bellare, "New proofs for NMAC and HMAC: Security without collision-resistance," in *CRYPTO 2006*, 2006.

[43] Open WhisperSystems, "The new TextSecure: Privacy beyond SMS," Feb. 2013. [Online]. Available: https://whispersystems.org/blog/advanced-ratcheting/

[44] C. G. Günther, "An identity-based key-exchange protocol," in *EUROCRYPT'89*, 1989.

[45] M. Bellare, D. Pointcheval, and P. Rogaway, "Authenticated key exchange secure against dictionary attacks," in *EUROCRYPT 2000*, 2000.

[46] H. Krawczyk, "HMQV: A high-performance secure Diffie-Hellman protocol," in *CRYPTO 2005*, 2005.

[47] G. Kopf and B. Brehm, "Phrack magazine: Secure function evaluation vs. deniability in OTR and similar protocols," Apr. 2012. [Online]. Available: http://phrack.org/issues/68/14.html

[48] M. Mannan and P. C. van Oorschot, “A protocol for secure public instant messaging,” in *Financial Cryptography and Data Security*, 2006.

[49] P. Riikonen, “Secure internet live conferencing protocol specification DRAFT,” 2007. [Online]. Available: http://tools.ietf.org/id/draft-riikonen-silc-spec-09.txt

[50] Unkown, “FiSH – secure communications with internet relay chat,” 2007. [Online]. Available: http://ultrx.net/doc/fish/

[51] S. Thomas, “DecryptoCat,” 2013. [Online]. Available: http://tobtu.com/decryptocat-old.php

[52] M. Green, “Noodling about IM protocols,” Jul. 2014. [Online]. Available: http://blog.cryptographyengineering.com/2014/07/noodling-about-im-protocols.html

[53] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, “Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services,” 2014.

[54] D. J. Bernstein, T. Lange, and P. Schwabe, “The security impact of a new cryptographic library,” in *LATINCRYPT 2012*, 2012.

[55] M. Marlinspike, 2014. [Online]. Available: https://openwhispersystems.org/

[56] ——, “Second thoughts on whatsapp encryption,” 2014. [Online]. Available: https://www.mail-archive.com/messaging@moderncrypto.org/msg01138.html