DECIM: Detecting Endpoint Compromise In Messaging

Full Version*

Jiangshan Yu[†‡ §] Mark Ryan [§] Cas Cremers [¶]

Abstract

We present DECIM, an approach to solve the challenge of detecting endpoint compromise in messaging. DECIM manages and refreshes encryption/decryption keys in an automatic and transparent way: it makes it necessary for uses of the key to be inserted in an append-only log, which the device owner can interrogate in order to detect misuse.

We propose a multi-device messaging protocol that exploits our concept to allow users to detect unauthorised usage of their device keys. It is co-designed with a formal model, and we verify its core security property using the Tamarin prover. We present a proof-of-concept implementation providing the main features required for deployment. We find that DECIM messaging is efficient even for millions of users.

The methods we introduce are not intended to replace existing methods used to keep keys safe (such as hardware devices, careful procedures, or key refreshment techniques). Rather, our methods provide a useful and effective additional layer of security.

1 Introduction

Spurred by government surveillance [1–3] and users' desire for strong security [4], a new trend of using end-to-end secure communication has spread. Large companies and security communities have started to deploy

*An extended abstract of this work appeared at IEEE TIFS, 2017.

[†]Corresponding author, J.Yu.Research@gmail.com

[‡]SnT, University of Luxembourg.

[§]School of Computer Science, University of Birmingham, UK.

[¶]Department of Computer Science, University of Oxford, UK.

and provide message services with end-to-end encryption, which include Apple *iMessage*, Facebook *WhatsApp*, Google *End-to-End* email encryption, and *Telegram Messenger*, to millions of users.

One challenge in providing end-to-end encrypted messaging concerns how to authenticate public keys. Even though methods based on the CA-model (e.g. S/MIME) and the web-of-trust (e.g. OpenPGP) have been available for decades, they have failed to be widely deployed because of the security and usability concerns [5]. Recently, CIRT [6] and CONIKS [7] have been proposed to solve the key authentication problem for messaging, by making all issued key bindings transparent to end-users. Both CIRT and CONIKS support multiple devices, and detect misbehaviours or compromise of the key certification authority. However, while these services provide a good level of protection on users' communication, they still rely on the assumption that the end-device cannot be compromised. Yet, this assumption is rather hard to justify in practice: new software vulnerabilities [8–10] are discovered every day, and malware is common on mobile devices such as phones and tablets [11] as well as on traditional platforms like desktop PCs.

Signal [12] (formerly *TextSecure*) moves a step towards handling device compromise. It rotates keys through a ratcheting process (a.k.a. Axolotl protocol), which generates three types of keys, namely root key, chain key, and message key. The root key is a relatively long-term key generated from users' public keys and updated through the ratchet process. The chain key and message key are ephemeral keys derived from the associated root key. Each chain key is a session key, and the associated message keys are used to encrypt/decrypt messages exchanged in that session. (We refer the reader to [13] for more detail.)

An attacker who learns the chain keys and message keys will not be able to learn messages that have been exchanged in other sessions. However, if the root key has also been compromised, then the attacker is able to perform a man-in-the-middle (MITM) attack to intercept future messages. Additionally, the ratcheting process can lock the attacker out from the point that the attacker discontinued being the MITM. The ratcheting process has been built into several systems including WhatsApp, one of the most popular messaging platforms.

Whilst Signal is an important contribution to message security, it leaves open the question of how to defend against an attacker (e.g., a platform operator or an internet service provider) who is in a unique position to act as a persistent MITM, and has previously compromised a victim's device.

This paper explores a different part of the complex design space inhabited by CIRT/CONIKS and Signal. We develop DECIM, a method to Detect

Endpoint Compromise in Messaging applications.

Contribution Our first contribution is to develop an attacker model in which platforms may be *periodically compromised*. That means that they can be compromised by an attacker at any time, but we assume that the victim periodically takes steps to remove malware and eliminate vulnerabilities. Unfortunately, the compromise could have revealed long-term keys. We thus propose security goals that aim to detect the subsequent usage of such keys by the attacker.

Second, we propose an approach for detecting endpoint compromise in messaging (DECIM), to transparently manage ephemeral encryption/decryption keys. It enables users to detect subsequent usage of compromised long-term keys by the attacker even against a persistent MITM attacker, while avoiding the use of expensive and inconvenient manual process for re-authenticating and distributing keys through the underlying PKIs (e.g. applying for a new certificate from a CA), unless attacks are actually detected.

We develop two DECIM protocols. The first is a basic protocol that makes strong assumptions about the participants being simultaneously online, and serves mostly to explain the concepts. The second protocol is a more fully developed messaging application, supporting multiple devices per user and allowing the receiver to be offline at the time the sender sends a message.

We provide a proof-of-concept implementation of the detailed messaging application, and conduct a performance evaluation on the system. It shows that the protocol is efficient and scalable: even in an extreme case, i.e. the messaging system has been operating for 100 years with 10^9 users (each with 3 devices), clients only need to download 2.2 KB extra data for the compromise detection. The memory usage on the server side for enrolling 10^5 new devices of distinct users is only 410 MB, and it takes roughly 5.7 milliseconds on average for each request.

Our third contribution is the security analysis which shows that the protocols satisfy precise properties expressing software damage containment. Informally, if an attacker controlled device has been recovered from a compromised state to a secure state, then our system can automatically detect a (persistent) MITM attacker. Therefore the victim will be prompted to manually revoke the key and generate a new one. We use the Tamarin prover to prove several core properties of our protocol.

We proceed in the following way. In Section 2, we present the background and related work. We detail our attacker model in Section 3 and present the

main idea of our DECIM protocols in Section 4. The implementation of our messaging protocol is presented in Section 5 in full detail. We analyse the security of our proposal in Section 6, present the performance evaluations in Section 7, and conclude in Section 8.

2 Related work

Axolotl protocol As mentioned previously, the Axolotl protocol implemented in Signal [12] uses a ratchet process to handle device compromise against a non-persistent MITM attacker. Similar security guarantees are also provided by other messaging protocols; see [14] for a detailed survey.

FlipIt FlipIt is an abstract game-theoretic framework for modelling security scenarios similar to the attacker model of our paper. In the FlipIt game [15], the attacker player moves by compromising a system, and the defender player moves by recovering it into a secure state. The FlipIt paper explores strategies for defender and attacker, based on an abstract notion of costs associated with moves.

Drifting keys Drifting keys [16] is an approach for detecting device impersonation when an attacker has obtained a copy of pre-shared secret keys stored in constrained devices (such as sensors). Roughly speaking, each key is updated by the sender by appending a random bit. If two inconsistent keys of the same device are detected by the receiver, then it learns that the pre-shared keys at the sender side (i.e. the constrained devices) has been compromised and used by an attacker to impersonate the device.

Certificate transparency *Certificate transparency* (CT) [18] is a technique proposed by Google aiming to detect mis-issued public key certificates. CT achieves this by recording all issued certificates in an append-only Merkle tree log. CT has been extended to handle revocation [6], and much work on building transparent systems has been proposed based on the concept of CT. Examples include ARPKI [19] and PoliCert [20] for transparent PKI, and CIRT [6] and CONIKS [7] for transparency in messaging systems.

3 Threat model and design goals

Assumptions We assume a role called sender, that sends messages, and another one called receiver, that receives messages. Users can perform one or both of those roles. Each user has one or more devices, and can pick any

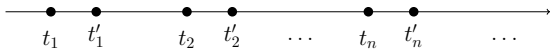

Figure 1: A device is compromised at time t_1, and then restored into a secure state at time t'_1. This cycle is repeated. Thus, the device is in a compromised state during the intervals $\{(t_j, t'_j) \mid j \in \{1, 2, 3, \ldots\}\}$.

of his/her devices to send a message, and can receive messages on any of them. We use **S**ally and **R**obert to refer to an arbitrary sender and receiver, respectively.

Threat model The attacker has control over the network and the messaging server. This means he can eavesdrop, modify, insert and suppress any messages, and as many of them he wants. In this way, he can act as a persistent MITM. However, we also assume that the parties can occasionally communicate short messages, possibly through an independent, low-bandwidth and unreliable channel. The attacker has only partial control of this additional channel — he can intercept, modify and suppress messages, but not all of them all of the time (occasionally, a message will get through)[1]. In other words, we assume that the attacker can block all communications in one channel, but cannot block all communications in all possible diverse channels.

In addition, the attacker may compromise any user's devices at any time. After compromising a device, the attacker fully controls it, and can retrieve and store all the data (including secret keys) that are stored on it.

Periodically and routinely, users detect and remove malware on their devices, upgrade the operating system, and install software patches that remove known vulnerabilities. This can put the device back into a trustworthy state. The users do not regenerate long-term keys or change passwords unless evidence of a compromised device has been found.

Thus, we assume that devices are *periodically trustworthy*. An attacker compromises the device by exploiting a vulnerability, and sometime later the device owner restores it into a secure state. This cycle repeats, as illustrated in Figure 1.

The problem Once a device is compromised, then the victim's secrets stored in the device are exposed to the attacker. Performing security updates

[1]The idea of this secondary channel is to enable the users to detect a misbehaving log server that shows different versions of the log to different users. It has been used in other transparent log based systems, such as in CT, CIRT and CONIKS. We indicate how this works at the end of the section 4.3 on page 6.

and removing malware is insufficient to prevent the attacker masquerading as the victim.

Security goals To solve this problem, our system detects key usages by the attacker. We state our security goal here, and explain how to achieve the goal in the following sections. In the security statements below, we assume a parameter ζ, which is a time period set by the user. A shorter ζ brings greater security. However, devices are automatically unregistered from the system if they are not used for periods longer than ζ, and have to be re-registered. Thus, a very short ζ reduces usability. Typically, ζ would be about two days. We discuss ζ and other system parameters later.

In the next section, we develop two protocols: the basic DECIM protocol and the full DECIM messaging application. These offer slightly different guarantees.

- **Basic DECIM protocol.**
  Suppose receiver Robert's device is compromised during the periods $\{(t_j, t'_j) \mid j \in N\}$. Suppose a message is sent by sender Sally at time t from a device in a trustworthy state, and the plaintext is obtained by the attacker. Robert can detect this attack provided his device
  - was well within a trustworthy state when the message was sent; that is, $t'_j + \zeta \leq t \leq t_{j+1} - \zeta$ for some j.
- **Messaging application (many users each with many devices).**
  Suppose Robert's devices are periodically compromised, as before: D_i is compromised during the intervals
  $\{(t_{i,j}, t'_{i,j}) \mid j \in N\}$. Suppose a message is sent by Sally at time t from a device in a trustworthy state, and the plaintext is obtained by the attacker. Robert can detect this attack provided, for each of his devices D_i,
  - D_i was well within a trustworthy state when the message was sent; that is, $t'_{i,j} + \zeta \leq t \leq t_{i,j+1} - \zeta$ for some j, or
  - D_i was in a compromised state, but had not been used by Robert since $t - \zeta$.

  The last condition reflects the fact that one can tell that a device has been compromised if the device was not being used at the time its key was used. Later, in Section 4.2, we show the user interface that allows a user to check this.

As part of our solution, we introduce an auxiliary role called the log maintainer. In practice, there can be one or more agents acting as log maintainers. We do not require that any of these log maintainers are trusted and assume that the attacker controls them.

4 Overview of DECIM

We present an overview of two protocols for detecting endpoint compromise. In the first, the participants are a single sender and a single receiver, assisted by a log maintainer. This situation is too simple to be useful, but serves to illustrate the core concepts. The second protocol is more involved; there are multiple senders and receivers, and each of them has multiple devices. This reflects a more realistic situation, and the multiple devices assist in the detection of attacks.

4.1 The basic DECIM protocol

Our solution involves three roles: senders, receivers, and a log maintainer. We assume all of these can be compromised. We assume a log maintainer is capable of receiving data and storing it in an append-only log.

During the bootstrapping phase, the receiver Robert obtains or generates a long-term signing and verification key pair (sk_R, vk_R), and the sender Sally obtains an authentic copy of vk_R. The log maintainer has a signing key sk_L, and Robert and Sally have an authentic copy of the corresponding verification key vk_L. How these keys are securely distributed is not the subject of this paper; we assume it can be done through PKIs such as S/MIME [24], PGP [25–27], CIRT [6], or CONIKS [7].

The log maintainer signs and publishes *digests* of the log. We use 'digest' to denote a short data item that uniquely summarises the log (in practice, it is the root tree hash of a Merkle tree). The maintainer is able to create cryptographic proofs that given data is present or absent from the log. Data is never deleted from the log represented by a given digest.

The log maintainer can also create proofs that a given digest represents an append-only extension of the log represented by a previous digest.

Sally and Robert track the digests issued by the log, all the time checking the proofs issued by the log that later digests represent extensions of earlier ones. Sally and Robert also periodically directly exchange the digests they know about, and request and check proofs of extension of those digests with respect to those they already have. Our assumption that the attacker cannot suppress all messages ensures that they are being presented with the same version of the log.

The transmission part of the basic DECIM protocol then runs as follows (see Figure 2).

- To prepare for receiving a message, Robert's device creates an ephemeral encryption and decryption key pair (ek, dk), and certifies it with his

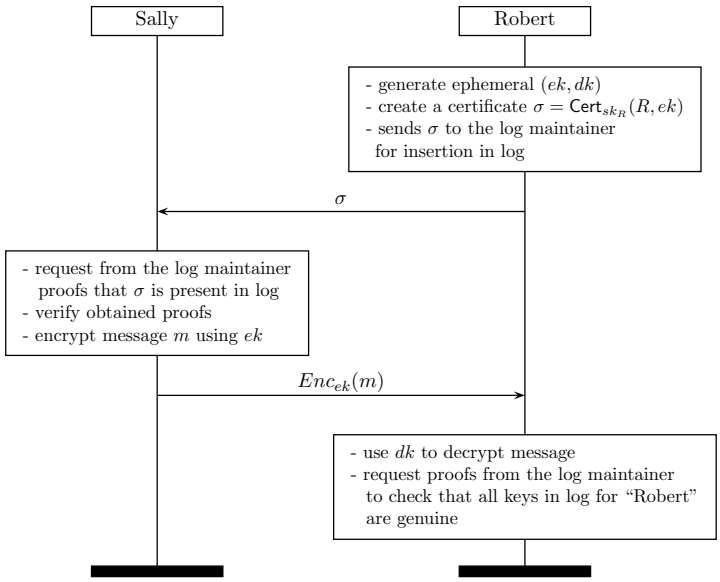

Figure 2: The basic DECIM protocol. Robert has a pair (sk_R, vk_R) of long term keys for signature signing and verification. He generates an ephemeral key pair (ek, dk) for encryption, creates the certificate $\sigma = \mathsf{Cert}_{sk_R}(R, ek)$ on ek, and sends the certificate to the log maintainer for insertion into the public log. Meanwhile, Robert also sends the certificate to Sally. After receiving σ, Sally requests from the log maintainer proofs that the certificate is present in the log. If the proof is valid, Sally sends a message m to Robert encrypted with ek. Robert requests proofs from the log maintainer to enable him to verify whether the log contains signatures that he did not generate.

long-term signing key sk_R. He publishes the certificate $\mathsf{Cert}_{sk_R}(R, ek)$ in the log. Publishing the certificate in the log assures Sally that it is a valid encryption key belonging to Robert.

- To send a message, Sally's device retrieves $\mathsf{Cert}_{sk_R}(R, ek)$ from the log along with a proof of its currency in the log. She encrypts the message with ek and sends it to Robert. Sally will not use a key whose certificate is not in the log.
- Robert's device receives the encrypted message and decrypts it.

Additionally, Robert's device periodically checks (where the period is determined by the parameter ζ) that all the keys ek' for which a certificate $\mathsf{Cert}_{sk_R}(R, ek')$ exists in the log were put there by him. If he finds entries in the log not corresponding to his actions, then he knows that his long term

credentials have been disclosed and abused by an attacker.

The basic protocol assumes that Robert is online at the time that Sally wants to send a message. In the messaging application protocol below, we generalise this to work when Robert is offline.

Intuitively, our protocol design detects compromise attacks because an attacker in possession of Robert's long term key would have to leave evidence of its usage of the key in the log. We give examples of how this detection works in Section 4.3. We perform a formal analysis of our designs in Section 6.

Properties of the log The security of the method requires that an attacker cannot remove information from the log. To achieve this, the log is typically stipulated to be append-only. It is also a requirement that users of the log (including Robert) can verify that no information has been deleted from the log. For this purpose, the log can be organised as a Merkle tree [28] in which data is inserted by extending the tree to the right. Such a log was designed and introduced in *certificate transparency* [18]. The log maintainer can provide efficient proofs that (A) some particular data is present in the log, and (B) the log is being maintained in an append-only manner. Proof A is referred to as *proof of presence* (PoP) and proof B is referred to as *proof of extension* (PoE).

Certificate transparency has been extended to provide proofs that all data associated to some attribute (e.g. keys associated to a user identity) is absent from the log, and proofs that some data associated to some attribute is the latest valid data in the log. The former is referred to as *proof of absence*, and the latter as *proof of currency* [6, 7].

4.2 DECIM Messaging application

The DECIM messaging application generalises the basic DECIM protocol, allowing the users to have multiple devices. Sally can choose any of her devices to send a message, and Robert is able to receive the message on all of his devices. Although this makes the protocol a bit more complicated, it also allows us to obtain a stronger security guarantee, because even if one of Robert's devices is in an untrustworthy state we are able to leverage security from the other ones.

As before, we assume a log, with the same capabilities mentioned above. We also assume that Robert and the log maintainer have long-term signing and verification key pairs (sk_R, vk_R) (sk_L, vk_L) respectively, and all parties have authentic copies of the verification keys they need.

The parameters δ, ϵ and ζ The protocol is parameterised by three values:

- δ is the period between the times at which device registration requests are processed. It is set by the log maintainer. We expect it to be typically one hour.
- ϵ is the period between the times at which key update requests are processed. We refer to such periods as "epochs". It is also set by the log maintainer, and is typically one day.
- ζ is the maximum lifetime of a key. It is set by the user. Different users can choose different values of ζ, subject to the constraint $\epsilon \leq \zeta$. We expect it to be about two or three days.

The messaging protocol has three main sub-protocols: enrolling, message transmission, and key updates. We describe these in turn.

Enrolling a device To enroll a device D_ℓ, Robert needs to install sk_R onto it. We assume that sk_R is derived from a passphrase that Robert types into D_ℓ. Next, D_ℓ needs to create a key pair and publish its certificate in the log. More precisely:

- D_ℓ generates a new ephemeral encryption key pair (ek_ℓ, dk_ℓ) and sends the certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$ to the log maintainer. Here, t_ℓ is the key creation time. The key will be used from the current time until the next epoch beginning, for the purpose of encrypting messages for Robert's device.
- After time δ, the log maintainer has inserted the certificate into the log and sends to D_ℓ the list of device certificates $\mathsf{Cert}_{sk_R}(D_i, ek_i, t_i)$ for Robert present in the log, together with a proof that the list is complete, and current in the log.
- D_ℓ verifies the proof of currency for $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$. It displays the table (D_i, t_i) (for each i) to Robert, so he can check that the devices mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is evidence of an attack (§ 4.4). Figure 3 presents an example of the envisaged GUI to show how the information is likely to be presented to Robert.

The device is now ready to be used. When Sally encrypts a message, her device will obtain all the public parts of the current ephemeral keys for Robert from the log, and encrypt the messages with each of them.

Remark. The method of displaying on a user's device the user's activities on other devices is well-known (for example, in Gmail, a user can click "last account activity" to see a table of the sessions open by other devices). A crucial difference in our protocol is that the displaying device can fully verify the veracity of the account activity provided by the untrusted log maintainer.

Sending and receiving a message

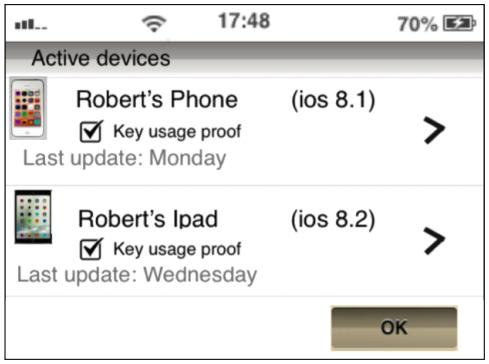

Figure 3: An example of envisaged GUI that presents the table (D_i, t_i) for $i = \{1, 2\}$ to Robert. The ticked box against the "key usage proof" indicates that the proofs about the usage statement (e.g., last update time) have been cryptographically verified.

- To send a message, Sally retrieves $\mathsf{Cert}_{sk_R}(D_i, ek_i, t_i)$ (for each available i) from the log along with proofs of currency. Her device encrypts a copy of the message by using each received ek_i according to the specific end-to-end secure messaging protocol that they both use[2]. It sends the encrypted message and together with the encrypted k to each of Robert's devices.
- Robert picks up any of his devices, receives the encrypted message, and decrypts it.

Updating the keys Whenever Robert invokes the messaging app on a device D_ℓ, the device checks to see if it is the first time it has run the app during that ϵ-long epoch. If so, it generates a new device key which will become the key for the following epoch. More precisely, on the first invocation during an epoch:

- D_ℓ requests and verifies proof of currency for all of the current epoch's device certificates $\mathsf{Cert}_{sk_R}(D_i, ek_i, t_i)$ for each available i. It verifies that ek_ℓ is indeed the one it created and sent the previous epoch; if this verification fails, it is evidence of an attack (§ 4.4). D_ℓ displays the table (D_i, t_i) (each i) to Robert, so he can check that the devices

[2]The design of DECIM is agnostic about the specific end-to-end secure messaging protocol used; e.g. it could be PGP, Axolotl, or something else. For simplicity and concreteness, in the detailed presentation of DECIM in section 5, and also in the Tamarin proofs of section 6, we encrypt messages by using the hybrid mechanism deployed in PGP and iMessage.

mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is again evidence of an attack.

- D_ℓ next creates a new ephemeral encryption key pair (ek'_ℓ, dk'_ℓ) and sends the certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek'_\ell, t_\ell)$ to the log maintainer. Here, t_ℓ is the key creation time.
- By the next epoch, the log maintainer has inserted into the log all the device keys thus received. If a device does not send a new key during an epoch, the old key is retained in subsequent epochs until a period ζ has elapsed. At that time, keys of devices that did not send new keys are revoked.
- When a new key becomes valid, D_ℓ securely removes the old key in the device.

In other words, devices change their key every epoch, and if they don't do so (because the application is not invoked during a particular epoch) then their key is reused for a certain period, and then revoked. In this last case, the device can't be used until it re-registers.

4.3 Detecting attacks: examples

To provide intuition on how our protocol allows users to detect attacks, we explain some potential attack detection scenarios. We will present our formal security analysis in Section 6.

Attacks from a third party

Suppose one of Robert's devices, say his phone, is infected with malware, allowing an adversary to misuse all the keys stored on the device. Suppose the adversary is the messaging service provider acting as a persistent MITM. The adversary may decrypt messages encrypted with ephemeral keys in that epoch, and may create new signed ephemeral keys by using the phone's long term key and inserting them into the log to perform MITM attacks in future epochs.

Robert routinely performs malware scanning and software patching, which may or may not help him regain the control of his phone depending on the robustness of the malware. It is obvious that one can do nothing for the epoch in which the adversary has all the ephemeral secrets for decryption. We focus on the more interesting case, namely, the security of messages exchanged in future epochs.

If Robert regains control of his phone, and the attacker continues to use the phone's long-term key to create ephemeral keys, the phone can detect this activity via the log, and report it to the user.

If the adversary remains in full control of the phone, then Robert might still be able to detect the device compromise by monitoring the long-term key usage – he notices unexpected usage of phone using the GUI of Figure 3. The figure shows the GUI displayed on another device of Robert's. It informs him that (so far in the current epoch) the keys corresponding to his phone and his iPad have been active. If Robert has not used his phone in the epoch, then he learns that it has been compromised. The GUI also confirms that the proofs about the usage statement have been cryptographically verified.

Attacks on or by the log maintainer Suppose the log maintainer is malicious or compromised. It may provide fake proofs, or provide no proofs at all. This is readily detected by client software. It may maintain the log incorrectly, either by not correctly recording signed ephemeral keys or by incorrectly recording fake ephemeral keys. These attacks are detected when the key owner requests a complete proof of presence.

A more interesting attack arises if the log maintainer shows different versions of the log to different users. A receiver may see a version in which his ephemeral keys are correctly recorded, while the sender sees a version in which attacker-owned keys are present instead. This would allow the attacker to play man-in-the-middle attacks, preventing the sender and receiver ever exchanging information about the log digests they have. In DECIM, users can detect such attacks by gossiping with their contacts, for example, through an out-of-band channel as used in Signal [12], or through a gossip protocol [21–23] as recommended by Google CT [18] and CONIKS [7]. Such a procedure will ensure that the log maintainer is not misbehaving. We refer readers to the referenced work for more detail.

4.4 Responding to attacks

If Robert detects unexpected activity on a device, or some verification fails, this is evidence of an attack. Robert's response should be to fix the software on his devices. He should generate a new long-term key, in order to prevent attacks occurring (and being detected) due to the disclosure of his current long-term key. The corresponding public key can be distributed using the method used in the bootstrapping phase. Furthermore, he can inform Sally that some of her recent messages to him may have been compromised.

Robert can also detect failure when he verifies the actions of the log maintainer. His response is to change to a different provider.

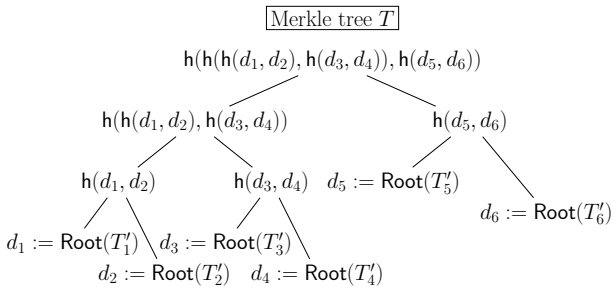

Figure 4: An example of the log containing six updates $\{d_1, d_2, \ldots, d_6\}$. The log is an append-only Merkle tree T whose leaves are ordered chronologically.

5 Detailed messaging protocol

In this section we present our proposal's details in several parts. We first present the log structure in Section 5.1. We then turn to describe the protocol in more detail in Section 5.2. The procedures that ensure that we detect malicious log maintainers are described in Section 5.3. we consider privacy concerns in Section 5.4.

5.1 Log structure

The public log is organised as a tree of trees: the top-level tree is append-only, and its leaves are lexicographically ordered trees.

The top-level tree of the log is implemented by a append-only Merkle tree [28]. The digest of a log is the root hash value and the size of this tree. A Merkle tree is a tree in which every node is labelled with the hash of the labels of its children nodes. Suppose a node has two children labelled with hash values h_1, h_2. Then the label of this node is $\mathsf{h}(h_1, h_2)$. Merkle trees allow efficient proofs that they contain certain data. To prove that a certain data item d is part of a Merkle tree requires an amount of data proportional to the log of the number of nodes of the tree. (This contrasts with hash lists, where the amount is proportional to the number of nodes.) If a Merkle tree is append-only, i.e. the only supported operation is to append some data to the tree, then it supports efficient proof that a version of the tree is extended from a previous version. If items in a Merkle tree are ordered lexicographically, then the Merkle tree supports efficient proof that some data is absent from the tree. The sizes of all the above proofs are proportional to the log of the number of nodes of the tree. More examples can be found in [6, 18]. Table 1 shows methods that a Merkle tree supports.

Table 1: The methods supported by the Merkle tree.

Method	Input	Output
Size	T	The size of the Merkle tree T
Root	T	The root value of the Merkle tree T
Last	T	The data stored in the rightmost side leaf of Merkle tree T
PoP	(T, d)	*Proof of Presence*: The proof that d is in T
PoC	(T, d)	*Proof of Currency*: The proof that d is the last leaf in T
PoA	(T, a)	*Proof of Absence*: The proof that any data d having attribute a is absent from the Merkle tree T. This proof can only work if items in T are ordered lexicographically according to the attribute.
PoE	(T, dg')	*Proof of Extension*: The proof that the Merkle tree T is an extension of another Merkle tree whose digest is dg'. This proof can only work if T is append-only.

The append-only Merkle tree T (as shown in Figure 4) records the entire update history. Items in T are stored only in leaves and ordered chronologically, and each leaf is labelled by the root hash value of another Merkle tree T' (presented in Figure 5). Items in T' are also stored only in the leaves, but ordered according to user identity. Each leaf of T' is labelled by users' identity and a list of ephemeral certificates for different devices of the same user.

We give some examples based on Figure 4 and 5 to show how the proof can be done with our log. We will explain how to verify that the log is maintained correctly — i.e. the log maintainer only appends data in T, and items in every T' are ordered lexicographically — in §5.3.

Example of *proof of presence* To prove that data d'_2 for Bob is in T'_6 (see Figure 5), the log maintainer only needs to give the data needed to compute the label of parent node from d'_2 to the root of the tree.

$$\mathsf{PoP}(T'_6, d'_2) = [w, d'_1, h_{(3,4)}, h_{(5,7)}]$$

where $w = l \cdot l \cdot r$ is the path to d'_2, and l (resp. r) indicates the path to the left (resp. right) child. So, given d'_2, $\mathsf{Root}(T'_6)$, and the proof $\mathsf{PoP}(T'_6, d'_2)$, one can verify the proof by reconstructing the root value $h_T = \mathsf{h}(\mathsf{h}(\mathsf{h}(d'_1, d'_2), h_{(3,4)}), \mathsf{h}((5,7)))$. If $h_T = \mathsf{Root}(T'_6)$, then the proof is valid.

Example of *proof of currency* The proof of currency is the same as the proof of presence, but there is an extra constraint for the verifier to check, namely that the path to the root of the lexicographic tree (e.g., the path from the root to d_6 in Figure 4) is of the form $r \cdot r \ldots \cdot r$, i.e., the leaf should be the rightmost leaf of the tree.

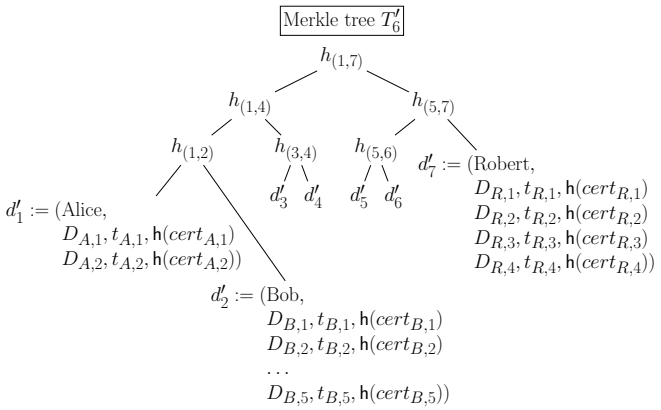

Figure 5: An example of the data structure T' recording data in each update. Items in T' are ordered lexicographically. For all $a, b \in [1, 7]$, $h_{(a,b)}$ is the root hash value of a Merkle tree containing data from d'_a to d'_b. For example, $h_{(1,2)} = \mathsf{h}(d'_1, d'_2)$, and $h_{(1,7)} = \mathsf{h}(h_{(1,4)}, h_{(5,7)})$. Each leaf of T' is labelled by $(\mathsf{h}(ID),\ (D_j, t_j, \mathsf{h}(cert_j))_{j=1}^{n})$, such that $cert_j$ is a certificate on (D_j, ek_j, t_j) issued by ID, where D_j is the identity of the j^{th} device of ID, ek_j is an (ephemeral) public encryption key, and t_j is the issuing time.

Example of *proof of extension* To prove that the current version of the log represented by T is an extension of a previous version (T_{old}) containing four updates (i.e. $\mathsf{Root}(T_{old}) = \mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4))$ and $\mathsf{Size}(T_{old}) = 4$), the log maintainer gives $\mathsf{h}(d_5, d_6)$ as the proof. Given the two digests and this proof, the verifier can verify that T is extended from T_{old} by reconstructing $\mathsf{Root}(T)$. A well defined algorithm for generating the proof in different cases is presented in §5.1.2 of [18].

Example of *Proof of absence* To prove that no certificates for user identity 'Bill' is included in T'_6, the log maintainer needs to prove that any node whose label containing Bill is absent from T'_6, by performing the following steps.

- Locate node A such that the user identity contained in its label is lexicographically the largest one smaller than Bill. In our example, the label of node A is d'_1 which contains user identity 'Alice'.
- Locate node B such that the user identity contained in its label is lexicographically the smallest one greater than Bill. In our example, the label of node B is d'_2 which contains user identity 'Bob'.
- Prove that d'_1 and d'_2 are present in T'_6, and they are siblings (so no

node is placed in between of them). The former is proved by using proof of presence, and the latter one can be verified by checking the path to d'_1 and d'_2.

5.2 Messaging protocol details

We recall the defined system parameters in Table 2.

Table 2: System parameters.

Parameter	Explanation
δ	the period between the times at which device registration requests are processed. It is set by the log maintainer. (Typically one hour.)
ϵ	the regularity of processing key update requests on the server side. It is set by the log maintainer. (Typically one day.)
ζ	the lifetime of the device ephemeral keys, defined by each individual user.

5.2.1 Enrolling a device (Figure 6)

We assume that all Robert's devices have shared his long-term signing key sk_R. To enrol a device D_ℓ, it generates a new ephemeral certificate, and publishes it in the log. In more detail, as presented in Figure 6:

- D_ℓ generates a new ephemeral key pair (dk_ℓ, ek_ℓ) for decryption and encryption, respectively. Then, D_ℓ issues a certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$ on $(D_\ell, ek_\ell, t_\ell)$ by using sk_R, where t_ℓ is the key creation time; and sends the signed registration request $m_1 = (req_1, R, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$ to the log, where req_1 is the request identity, R is the identity of Robert, and $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$ is the digest of the log that Robert possibly has previously stored (it is likely to happen if Robert is re-enrolling his device D_ℓ).
- After the log maintainer receives the request, it verifies the signature and the certificate, and that t_ℓ is in the time interval of the current update epoch δ. If they are all valid, it stores the request, and issues a signed confirmation $\mathsf{sign}\{\mathsf{Root}(log), \mathsf{Size}(log), \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)\}_{sk_L}$, where log is organised as T, as explained in §5.1. If dg_{old} is provided, the log maintainer also generates a proof P of extension that the current log is extended from the log represented by dg_{old}, and sends the proof together with signed confirmation as the message m_2 to Robert.

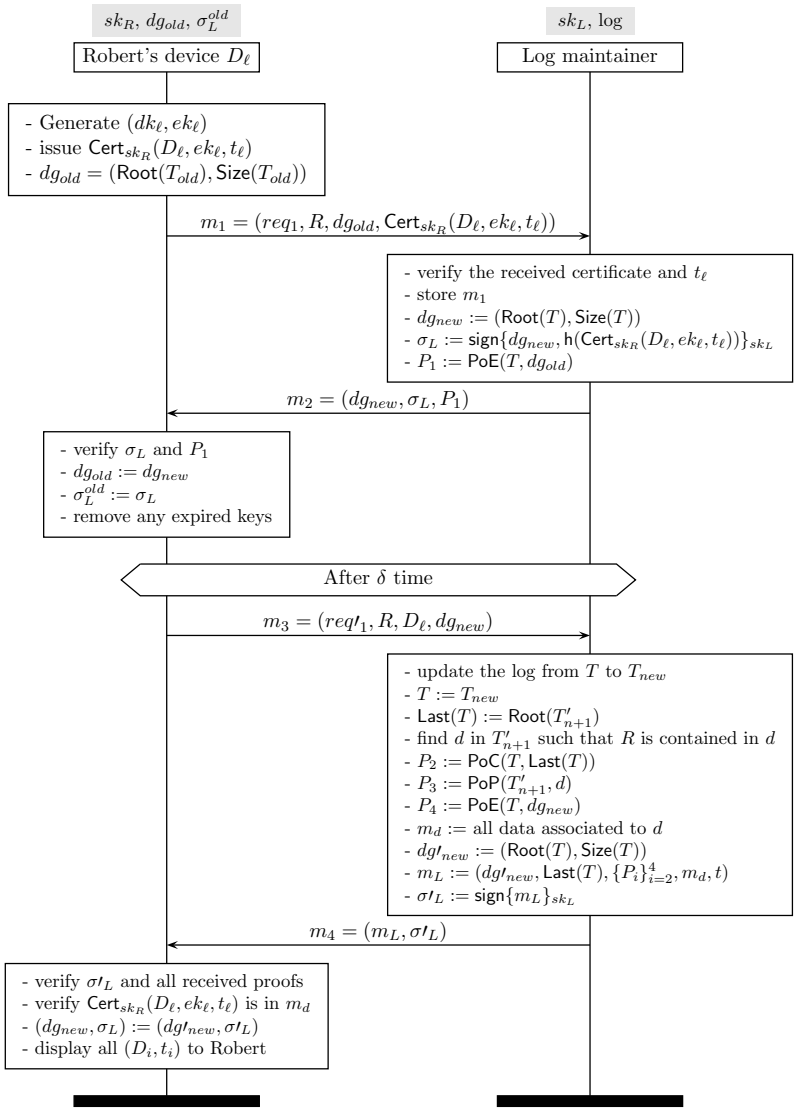

Figure 6: The protocol for (re-)enrolling a device. In the protocol, if Robert is re-enrolling his device, then dg_{old} and σ_L^{old} are the previously stored digest and signature received from the log maintainer, respectively.

- D_ℓ verifies the received signature and proof, stores the new digest dg_{new} with signature σ_L, and sends the request m_3 containing a request identity req'_1, Robert and the device's identity (R, D_ℓ), and current

observed digest to the log maintainer after δ time.

- After each period of length δ, the log maintainer updates the log according to the list of device enrollment requests received from its customers. The list of requests should be in the form

$$(R_i, (\mathsf{Cert}_{sk_{R_i}}(D_{i,j}, ek_{i,j}, t_{i,j}))_{j=1}^{P})_{i=1}^{M}$$

  where R_i is the client identity, P is the number of devices that a client has requested to enroll this update, and M is the total number of clients who have sent the enrollment request for this update.
  To update the log, the log maintainer retrieves the current T'_n such that $\mathsf{Root}(T'_n) = \mathsf{Last}(T)$, and creates T'_{n+1} by adding each request to the appropriate node of T'_n, where n is the size of the current log. It then extends T with a new rightmost node T'_{n+1}.

  In addition, the log maintainer proves that the list of certificates (including the ones in the enrollment request) for each participant R_i is complete, and current in the log. If R_i has previously observed a digest dg_{old} of the log, then log maintainer also generates a proof of extension that the current log is extended from the log represented by dg_{old}. To do so, the log maintainer locates the node labelled with d for R_i in T'_{n+1}, and generates:
  - $\mathsf{PoP}(T'_{n+1}, d)$ that d is present in T'_{n+1};
  - $\mathsf{PoC}(T, T'_{n+1})$ that the root hash value of T'_{n+1} is the label of the rightmost leaf in T; and
  - $\mathsf{PoE}(T, dg_{old})$ that the current log is extended from the log represented by dg_{old}.

  So R_i can verify that d — which contains a full list of certificates for his devices (including the newly enrolled ones) — is present in the latest update of the log.
- D_ℓ verifies the received proofs and signatures. Additionally, it displays the table (D_i, t_i) (for all $i \in [1, P]$) to Robert, so he can check that the devices mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is evidence of an attack that an attacker who has used his long-term key without authorisation and has inserted a certificate for him.

The device is now ready to be used. A similar process is used to unregister a device with the log maintainer.

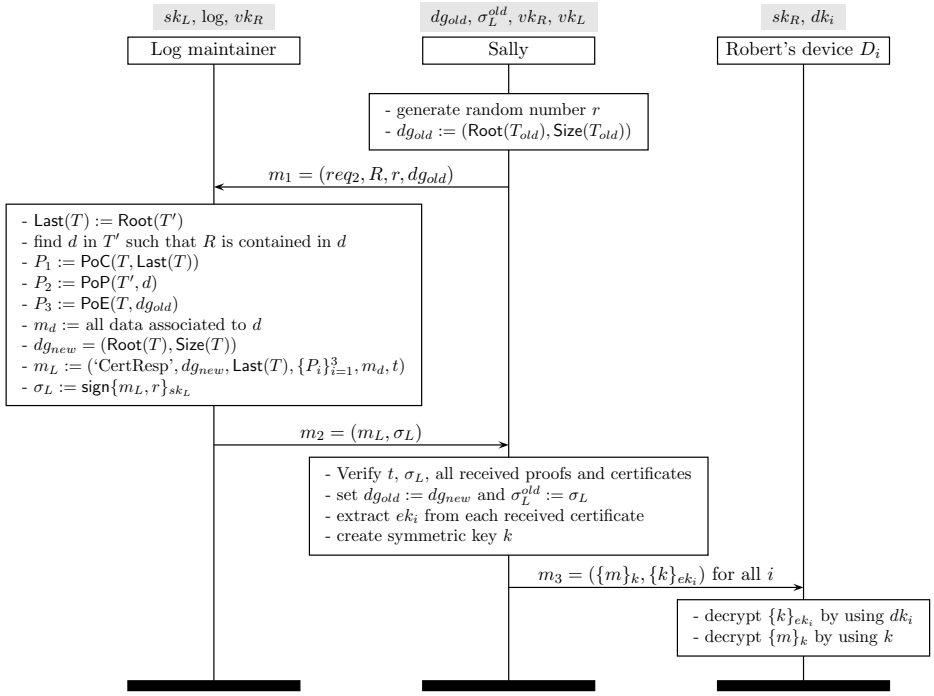

Figure 7: The protocol for sending and receiving a message. In which, σ_L^{old} is the signature received from the log maintainer in the last session. If any of the stated verification checks fails, the agent aborts the protocol.

5.2.2 Sending and receiving a message (Figure 7)

To send a message to Robert, Sally's device retrieves all the current device certificates for Robert from the log, and encrypts the messages with each of them. More precisely (as presented in Figure 7), to send a message:

- Sally sends request $m_1 = (req_2, R, r, dg_{old})$ to the log, where req_2[3] is the request identity, R is the identity of Robert, r is a random number, and where $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$ is the digest of the log that Sally received in the last session.
- After receiving the request, the log maintainer locates the leaf whose label d contains R in the latest update T' (that is represented by the rightmost leaf of T), and generates the proof P_1 that $\mathsf{Root}(T')$

[3]This request corresponds to the 'CertReq' in our Tamarin code.

is current in T, proof P_2 that d is in T', and proof P_3 that the current log is an extension of the log that Sally has previously observed. It then sends m_2 to Sally. In particular, m_2 is the signed message $(\text{'CertResp'}, dg_{new}, \mathsf{Last}(T), P_1, P_2, P_3, r, m_d, t)$, where 'CertResp' is a tag, $dg_{new} = (\mathsf{Root}(T), \mathsf{Size}(T))$, $m_d = (R,\ (D_j, t_j, ek_j, \mathsf{Cert}_j)_{j=1}^{P})$ is the data associated to d, and t is the time to identify the current epoch.

- After receiving the message from the log maintainer, Sally verifies if t corresponds to the current epoch, and verifies the received signature, proofs, and certificates. If all verifications succeed, she replaces dg_{old} and σ_L^{old} by dg_{new} and σ_L, respectively, where σ_L is the signature from the log maintainer.

  Her device encrypts a copy of the message with a fresh symmetric key k, and encrypts k with each received ek_i. It sends the encrypted message and together with the encrypted k to each of Robert's devices.
- Robert picks up any of his devices, receives the encrypted message, and decrypts it.

Note that in the protocol, if there is no certificate for Robert in the latest update, then a proof of absence that the identity of Robert is not in the latest update is provided to the user.

5.2.3 Updating the keys (Figure 8)

Devices change their key every epoch w.r.t. ϵ, and if they don't do so (because the application is not invoked on a particular day), then their key will be reused for a certain period (e.g. a few more ϵ), and then will not be included in the log for the next further update epoch. In this last case, the device can't be used for receiving and reading messages until Robert uses the device again — it will re-register the device automatically. So, after Robert can use this device again in δ time (e.g. one hour). Note that if Robert has un-registered the device, then the device will not *automatically* re-register itself; and Robert has to re-register it *manually* in this case.

More precisely, whenever Robert invokes the messaging app on a device D_ℓ, the device checks to see if it is the first time it has run the app during that epoch w.r.t. ϵ. If so,

- D_ℓ creates a new ephemeral key pair (dk_ℓ, ek_ℓ), issues a certificate $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$, which will become the valid key in next epoch, where t_ℓ is the key creation time. Then, he sends the signed request $m_1 = (req_3, R, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$ to the log maintainer, where

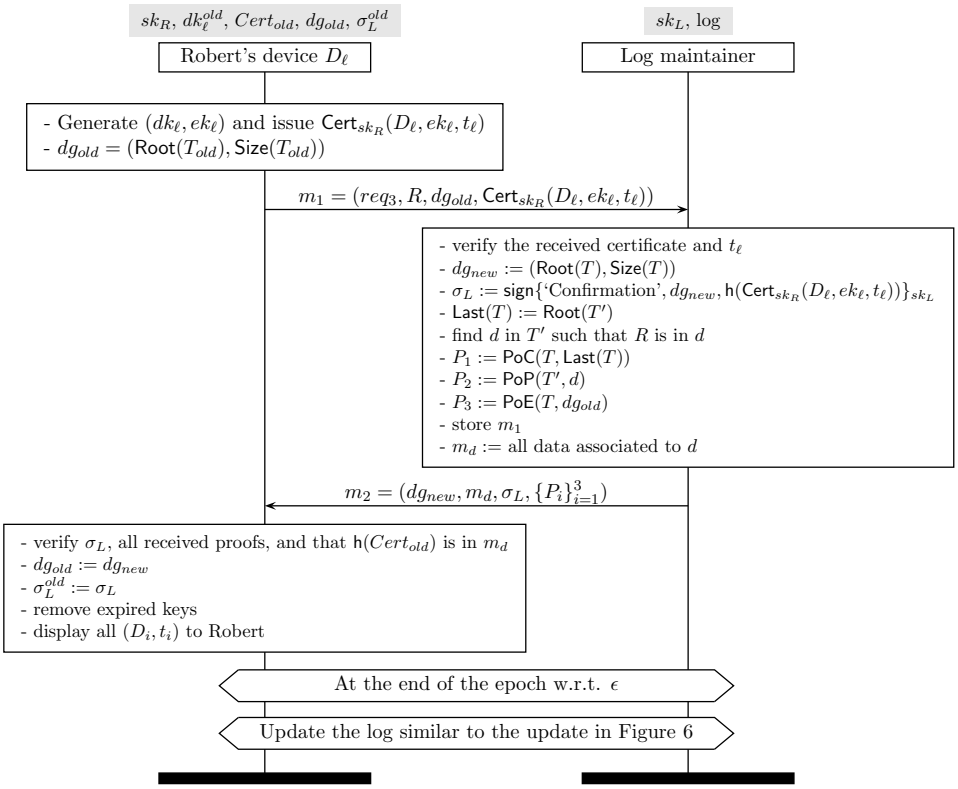

Figure 8: The protocol for updating keys. In the protocol, dk_ℓ^{old} is the current valid ephemeral secret key, $Cert_{old}$ is the corresponding certificate, dg_{old} and σ_L^{old} are the digest and signature received from the log maintainer in the last session, respectively.

req_3[4] is the identity of update request, $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$ is the digest of the log that he observed in the last session.

- After receiving the request, the log maintainer verifies the signature, time t_ℓ, and the received certificate. If they are all valid, then it generates a commitment $\sigma_L = \mathsf{sign}\{\text{'Confirmation'}, dg_{new}, \mathsf{h}(\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))\}_{sk_L}$ that it will put the received new certificate in the log by the end of this epoch. The log maintainer locates the node d for Robert in the latest update of the log, and generates the proof P_1 that the root hash value of T' is the label of the rightmost leaf in T, proof P_2 that d is

[4]This request corresponds to the 'UpdateReq' in our Tamarin code.

present in T', and the proof P_3 that T is an extension of the log that Robert has observed in the last session. Note that P_1 and P_2 together form the proof that d is the latest update for Robert in the log. The log maintainer sends the generated signature and proofs to D_ℓ.

- Upon receiving the response, D_ℓ verifies all signatures and proofs. Additionally, it verifies that the hashed certificate (contained in d) for D_ℓ in the latest update is indeed corresponding to the one it created and sent in the previous epoch. This verification ensures that no unauthorised request has been generated and recorded in the current log. (We will explain in the §5.3 that why we don't need to require D_ℓ to verify all history certificates for D_ℓ in the log are indeed generated by D_ℓ.) If all verifications succeed, D_ℓ removes any expired keys stored in D_ℓ, replaces the stored digest of the log with the new one, and displays the table (D_i, t_i) (for each possible i) to Robert, so he can check that the devices mentioned are indeed recently used. If Robert sees a device mentioned that he has not recently used, it is evidence of an attack.
- At the turn of the epoch, the log maintainer inserts all received update request into the log. Suppose in the current epoch, the log maintainer which maintains the log (represented by T of size n) has the tree T'_n containing

$$\begin{aligned}(&\mathit{Alice},\ D_{A,1}, t_{A,1}, \mathsf{h}(\mathit{cert}_{A,1})\\ &\qquad D_{A,2}, t_{A,2}, \mathsf{h}(\mathit{cert}_{A,2})),\\ (&\mathit{Bob},\ D_{B,1}, t_{B,1}, \mathsf{h}(\mathit{cert}_{B,1})\\ &\qquad D_{B,2}, t_{B,2}, \mathsf{h}(\mathit{cert}_{B,2})\\ &\qquad \cdots\\ &\qquad D_{B,5}, t_{B,5}, \mathsf{h}(\mathit{cert}_{B,5})),\\ &\cdots\ \cdots\end{aligned}$$

and receives

$$(R_i, (\mathsf{Cert}_{sk_{R_i}}(D_{i,j}, ek_{i,j}, t_{i,j}))_{j=1}^{P'})_{i=1}^{M'}$$

for some identity R_i and certificates for its devices $D_{i,j}$, where P' is the number of a user's devices that have sent a key update request, and M' is the total number of clients who have sent the key update request in this epoch.

To update the log, the log maintainer performs the following steps:

Step 1) creates a new tree T'_{n+1} by copying and pasting the entire T'_n;

Step 2) replaces the old certificates with the corresponding new ones in T'_{n+1};

Step 3) checks if any un-replaced certificate is older than ζ; if there is any, the log maintainer removes them from T'_{n+1};

Step 4) extends T with a new rightmost node $\mathsf{Root}(T'_{n+1})$.
Similar to the idea explained in §5.2.1, the log maintainer can provide the proof that the list of certificates (including the ones in the key update request) for R_i is complete, and current in the log; and the proof that the current log is an extension of the log that R_i has previously observed.
If a device has not updated ephemeral keys and has been excluded from the latest update by the log maintainer, then the device will automatically re-register itself when the owner has used the device again, so the device will be included in the log and be ready to receive and decrypt messages in δ time.

5.3 Crowd-sourced verification

Since we want to guarantee some security even when the log maintainer is not trusted, we need to monitor the log maintainer's behaviour to see if the log is maintained correctly. This can be easily verified by allowing any interested party to download and check the entire log at any time. Parties can set themselves up as monitors to perform such checks as a public service. Alternatively or in addition, to avoid having to rely on such monitors, we can use crowd-sourced verification by breaking the verification work into independent little pieces, and distribute each piece to different devices.

First, we need to verify that the log update history recorded in T is maintained in an append-only manner. This is achieved by verifying the proof of extension performed in the protocols for enrolling a device, updating the keys, and sending/receiving a message. Hence, there is no need for any additional verification.

Second, we need to verify that in each update T'_i, items are ordered lexicographically according to the user identity. It can be verified by asking each device to pick a random leaf in an update T'_i, and verify that the user identity recorded in its left (or right) neighbour leaf is lexicographically smaller (resp. greater) than the user identity in the picked leaf.

Third, in our protocol a device only checks its latest certificate in the log, instead of verifying all certificates recorded in the log. So, it cannot guarantee that no attacker-generated certificates have been previously included in the log. To detect such behaviour, we need to verify that the time of the key generation for the same device in different updates of the log is only going forward. To achieve this, each device picks a random leaf for a user in an update T_i, and verifies that either the record in an update is the same as the one in the previous update, or it is different and the time in the

node for the same device of the user in the left (or right) neighbour update T_{i-1} (or T_{i+1}) is no greater (or no smaller) than the time in the picked leaf, respectively. Additionally, if the two times are equal, then the hash values of the certificates should also be equal. A missing associated record in a new update is evidence of misbehaviour. If no leaf for the user is included in the neighbour update, then a proof of absence that a node containing the user identity is not included in the update is provided.

Remark. Note that these checks ensure that the log is maintained correctly, and the most recently published device key of all user devices are recorded in the latest log update (i.e. the rightmost leaf of the top level tree, see §5.1). Any unexpected record is evidence of misbehaviour of the log maintainer. Thus, to detect the un-authorised usage of the long-term keys, users only need to check their device records against the records in the latest log update, as stated in the protocol for *enrolling a device* and *for updating the keys*.

5.4 Privacy considerations

The public log may cause some privacy concerns. For example, depending on deployment specifics, one may want to hide the user identities contained in a log against potential spammers, the total number of communications of a user, or the time distribution of a user's communications, etc. We provide an informal discussion here, and leave a detailed formal study on the privacy of transparent log based systems as a future work.

To hide the user identity, the log maintainer can issue a signature on a user identity, then use a hash value of the signed user identity in the labels of leaves in each log update, rather than containing the user identity directly in the labels (see Figure 5). The signature scheme used should be deterministic and unforgeable, as suggested in [7]. Hence, users that have the recipient's address can request the signed user identity from the log maintainer, and verify the log; but an attacker who has downloaded the entire log cannot recover the identity of users, based on the unforgeability of the chosen signature scheme. In this case, the nodes in each update tree T'_i will be ordered lexicographically according to the hash value of the signed user identity. In addition, users can also make the log to be only available to a fixed set of contacts. To hide the real number of communications associated to a given client of the log, the client can generate some noise — for example, the client can make 'spoof queries' to the log maintainer through an anonymous channel (e.g. Tor network).

6 Security Analysis

We provide all input files required to understand and reproduce our security analysis at [29]. In particular, these include the complete DECIM models. The proof assumes that all users see the same log (a gossip protocol can be used to detect attacks in which different views of the log are presented to different users). We also assume that each user has only one device. The detection of some attacks when a user has multiple devices would need the user's involvement. A formal study on the user behavior and security analysis with multiple devices is an interesting future work.

Security properties Our messaging protocol achieves both classical security properties as well as novel ones. In a classical sense, Sally obtains the guarantee that if Robert's devices are not compromised, then the attacker will not learn the messages she sends.

The more interesting properties are achieved in the cases where Robert's devices get compromised. In this case, we cannot avoid that messages sent by Sally in the same epoch are also compromised. However, we prove that if any of Sally's messages from different epochs are compromised, then Robert will be able to detect this.

Formal analysis We analyse the main security properties of the protocol using the Tamarin prover [30]. The Tamarin prover is a symbolic analysis tool that can prove properties of security protocols for an unbounded number of instances and supports reasoning about protocols with mutable global state, which makes it suitable for our log-based protocol. Protocols are specified using multiset rewriting rules, and properties are expressed in a guarded fragment of first order logic that allows quantification over timepoints.

Tamarin is capable of automatic verification in many cases, and it also supports interactive verification by manual traversal of the proof tree. If the tool terminates without finding a proof, it returns a counter-example. Counter-examples are given as so-called dependency graphs, which are partially ordered sets of rule instances that represent a set of executions that violate the property. Counter-examples can be used to refine the model, and give feedback to the implementer and designer.

Modeling aspects We used several abstractions during modeling. We model the Merkle hash trees as lists, similar to the abstraction used in [19].

We model the protocol roles S (sender), R (receiver) and L (log maintainer) by a set of rewrite rules. Each rewrite rule typically models receiving a message, taking an appropriate action, and sending a response message. Our modeling approach is similar to most existing Tamarin models. Our

modeling of the roles directly corresponds to the protocol descriptions in the previous sections. TAMARIN provides built-in support for a Dolev-Yao style network attacker, i.e., one who is in full control of the network. We also specify rules that enable the attacker to compromise devices and learn their long and short-term secrets.

The final DECIM model consists of 450 lines for the base model, and six main property specifications, examples of which we will give below.

Proof goals We state several proof goals for our DECIM model, exactly as specified in TAMARIN's syntax. Since TAMARIN's property specification language is a fragment of first-order logic, it contains logical connectives (`|`, `&`, `==>`, `not`, ...) and quantifiers (`All`, `Ex`). In Tamarin, proof goals are marked as `lemma`. The `#`-prefix is used to denote timepoints, and "`E @ #i`" expresses that the event E occurs at timepoint i.

The first goal is a check for executability that ensures that our model allows for the successful transmission of a message. It is encoded in the following way.

```
lemma protocol_correctness:
 exists-trace
  " /* It is possible that */
   Ex d R skR dkR m #i.
     /* R received an encrypted message m on device d */
     MsgReceived(d, R, skR, dkR, m) @ #i
     /* without the adversary compromising any device. */
   & not (Ex d2 A ltk dkR #j.
             Compromise_Device(d2, A, ltk, dkR) @ #j)"
```

The property holds if the TAMARIN model exhibits a behaviour in which one of R's devices received a message without the attacker compromising any device. This property mainly serves as a sanity check on the model. If it did not hold, it would mean our model does not model the normal (honest) message flow, which could indicate a flaw in the model. Tamarin automatically proves this property in a few seconds and generates the expected trace in the form of a graphical representation of the rule instantiations and the message flow.

We additionally proved several other sanity-checking properties to minimize the risk of modeling errors.

The second example goal is the core secrecy property with respect to a classical attacker, and expresses that unless the attacker compromises one of Robert's keys, he cannot learn any messages sent by Sally. Note that `K(m)` is a special event that denotes that the attacker knows m at this time.

```
lemma message_secrecy:
```

```
"All R skR ekR m #i.
  /* If S sent a message m to R */
    ( MsgSent(R, skR, ekR, m) @ #i &
  /* without the adversary compromising any of Robert's
     devices */
    not (Ex #j d sk dkR.
                Compromise_Device(d, R, sk, dkR) @ #j)
    ) ==>
  /* then the adversary cannot know m */
    (not ( Ex #j. K(m) @ #j) ) "
```

Tamarin also proves this property automatically.

The above result implies that if Robert receives a message that was sent by Sally, and the attacker did not compromise his device during the current epoch, then the attacker will not learn the message.

The final property encodes the unique security guarantees provided by our protocol. If the attacker compromises Robert's device in an epoch, he can use the private ephemeral key to decrypt Sally's messages in that epoch. We prove that if he uses the compromised long-term key of Robert to learn messages sent by Sally in other epochs, then he will be detected once Robert checks the log.

```
lemma detect_usage_S_sends:
 "All d skR dkR m #i1 #i2 #i3 detectionresult R k.
    /* If S sent to R an encrypted message m,
       where pk(dkR)=ekR */
      ( MsgSent(R, skR, pk(dkR), m) @ #i1 &
    /* and the adversary knows m */
      K(m) @ #i2 &
    /* and the ephemeral key used by the sender was
       not compromised, i.e., the compromise occurred
      in a different epoch */
      not (Ex #j sk .
               Compromise_Device(d, R, sk, dkR) @ #j ) &
    /* and Robert afterwards checks the log */
      CheckedLog(d, R, detectionresult, k ) @ #i3
      &  #i1 < #i3
    ) ==>
    /* then we detect a compromise */
      ( (detectionresult = 'bad') ) "
```

The property states that if Sally sends a message when Robert's device is not controlled by an attacker in the current epoch (but might have been compromised previously), and the attacker learns the message, then Robert detects the fact that his key was previously compromised when he next verifies the log.

The above properties are all proven automatically by the Tamarin prover on a laptop within a few minutes. Overall, the modeling effort was in

Table 3: The size of messages in different protocols. In which, size_P is the size of proofs in the corresponding message, and size_M is the maximum size of a message.

	Enrolling a device	Fetching keys from log	Updating the keys	Crowd-sourced verification
Size of request message	1.6 KB	78 B	1.5 KB	-
Size of response message	2.5 KB	6.9 KB	2.5 KB	5.9 KB
Size of proof in (response) message	2.2 KB	2.2 KB	2.2 KB	5.3 KB
Total message size	4.1 KB	7 KB	4 KB	5.9 KB

the order of weeks, with several iterations to debug both the abstract model and the property specifications. The verification process helped us not only to prove, but also to refine the precise security properties of our protocol.

7 Realization in practice

7.1 Estimating communication cost

To check if deployment might be feasible, we estimate the expected cost of our protocol design. As an example, we consider the following scenario. We assume that there are 10^9 users, each user has 3 devices, the log has been operating for 100 years, the log update period δ for registration request is 1 hour, and the log update epoch ϵ for certificate update is 1 day.

In this scenario, the size of T will be $100{\cdot}365{+}100{\cdot}365{\cdot}24 = 912500 < 2^{20}$, and the size of each T' is 10^9 which is less than 2^{30}. In addition, we assume that the size of a hash value is 256 bits (e.g. SHA256), the size of a signature is 64 Bytes (e.g. ECDSA), and the size of a certificate is 1.5 KB.

In addition, we assume that the size of a user (or device) identity is 12 Bytes, and time is in the 64-bit format, a random number is 28 bytes (recommended by TLS 1.2 [31]), each request identifier is 4 bits, and the size of a digest of a log is 300 bits.

The size of a proof of presence that some data is in T and is in T' will be at most 640 bytes and 960 bytes, respectively; the size of the proof that a version of the log is extended from a previous version is at most 640 bytes. We present the size of messages in the protocol in our example scenario in Table 3.

From Table 3 we can see that up to 5 KB data are needed to be transferred for both enrolling a device and updating keys. The protocol for fetching keys from the log is the most expensive one, as the sender has to download all certificates for different devices of the same users. In our example,

the sender needs to download 3 certificates, the size of which is already 4.5 KB.

The results of our analysis indicate that the space cost of our system is acceptable.

7.2 Proof-of-concept log server prototype

To demonstrate the deployment of DECIM in a real-world setting, we built a proof-of-concept prototype of the log server. We implemented a full log server implementation with interfaces, and client-side code for (a) adding users/devices, (b) rotating keys at the end of each epoch, and (c) sending messages. This involves all the operations to manipulate the log (consisting of a tree of trees), produce various proofs, and produce and verify the appropriate signatures. Anticipating a deployment on platforms such as Google's App Engine, we implemented our code in Python. We use basic caching mechanisms for previously computed results.

On a quad-core 4 GHz Intel Core i7 with 32 GB of memory, we obtain the following times. The times are measured locally and therefore do not include network latency. Performing 100000 (1e05) enrollment requests from distinct users takes 1526 seconds, i.e., 15 milliseconds per request on average. When 100000 (1e05) users enroll 3 devices each, enrollment takes 1708 seconds, i.e., 5.7 milliseconds on average. The delay experienced by the user is therefore dominated by the network latency of transmitting 4.1 KB (Table 3), which is certainly less than a second.

When the tree contains 10000 (1e04) entries, the server produces 100000 (1e05) responses to message queries in 14.1 seconds, i.e., 0.14 milliseconds per message query. Updating a tree by simultaneously adding 10000 (1e04) entries takes about 1 second, which is mostly spent in creating the leaf data structures. Once again, the user's experience is mostly affected by the network latency, which is small because the data transferred is a few KB.

The memory usage when 100000 (1e05) users enroll one device is 410 MB (computed using "heapy" for the full process, not just reachable objects). If they enroll three devices each, memory usage increases to 900 MB.

Thus, even though our proof-of-concept implementation is not yet optimized for efficiency or storage, its performance already indicates our scheme is feasible.

8 Conclusion

End-to-end encryption has become popular in the years since the Snowden revelations, motivating attackers wishing to intercept messages to instead turn their attention to client end-points. To address this, we have presented a novel messaging protocol that offers security guarantees even when an attacker can access all the secret keys in a user's devices. In particular, (a) the protocol limits the impact of a compromise, since the attacker can only learn messages sent in the same epoch without being detected, and (b) if the attacker uses compromised long-term keys to impersonate users, then the protocol allows the participants to detect this, and therefore to take remedial action. Our protocol supports multiple devices per user, and the multiplicity of devices helps detect attacks by intuitive indicators to users about which (device) keys have recently been active.

The methods we introduce are not intended to replace existing methods used to keep keys safe. Existing technologies such as Axolotl ratcheting, TPMs, smart-cards, and ARM TrustZone are all useful for securing keys. However, none of these technologies are completely secure. For example, even if hardware security is used, malware may be able to trigger usages of the key without having the ability to copy the key. Our methods can also detect such cases. Thus, DECIM adds an additional layer of security that allows users to detect when other layers fail.

Acknowledgment

This work was partially supported by the Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128; and by the HP Research Chair in Cyber Security at Birmingham. We also specially thank Katriel Cohn-Gordon for his assistance on the proof-of-concept implementation and performance evaluation.

References

[1] B. Gellman and L. Poitras, "U.S., British intelligence mining data from nine U.S. internet companies in broad secret program," The Washington post, June 2013. [Online]. Available: http://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-

broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html

[2] S. Braun, A. Flaherty, J. Gillum, and M. Apuzzo, "Secret to prism program: Even bigger data seizure," June 2013. [Online]. Available: http://bigstory.ap.org/article/secret-prism-success-even-bigger-data-seizure

[3] E. MacAskill, N. Davies, N. Hopkins, J. Borger, and J. Ball, "GCHQ intercepted foreign politicians' communications at G20 summits." The Guardian, June 2013. [Online]. Available: http://www.guardian.co.uk/uk/2013/jun/16/gchq-intercepted-communications-g20-summits

[4] M. Madden, "Public perceptions of privacy and security in the post-snowden era," Pew Research Internet Project, Nov. 2014.

[5] A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: A usability evaluation of PGP 5.0," in *USENIX Security*, 1999.

[6] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail," in *NDSS*, 2014.

[7] M. S. Melara, A. Blankstein, J. Bonneau, M. J. Freedman, and E. W. Felten, "CONIKS: A privacy-preserving consistent key service for secure end-to-end communication," *IACR Cryptology ePrint Archive*, 2014.

[8] "Common vulnerabilities and exposures," https://cve.mitre.org/cve/index.html, Retrieved Feb. 2015.

[9] A. Greenberg, "Hack brief: Update ios now to fix a serious imessage crypto flaw," March 2016. [Online]. Available: http://www.wired.com/2016/03/hack-brief-update-ios-fix-serious-imessage-crypto-flaw/

[10] L. Chang, "Apple just fixed an imessage bug that researchers called easily exploitable," Yahoo Tech, April 2016. [Online]. Available: https://www.yahoo.com/tech/apple-just-fixed-imessage-bug-014700260.html

[11] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.

[12] "Signal," https://whispersystems.org/, Retrieved April. 2016.

[13] M. Marlinspike, "Advanced cryptographic ratcheting," Whisper System Blog, 2013 Nov. [Online]. Available: https://whispersystems.org/blog/advanced-ratcheting/

[14] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith, "Sok: Secure messaging," in *2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 232–249.

[15] M. van Dijk, A. Juels, A. Oprea, and R. L. Rivest, "Flipit: The game of "stealthy takeover"," *J. Cryptology*, vol. 26, no. 4, pp. 655–713, 2013.

[16] K. D. Bowers, A. Juels, R. L. Rivest, and E. Shen, "Drifting keys: Impersonation detection for constrained devices," in *Proceedings of the IEEE INFOCOM, Turin, Italy, April 14-19, 2013*, 2013, pp. 1025–1033.

[17] J. Håstad, J. Jonsson, A. Juels, and M. Yung, "Funkspiel schemes: an alternative to conventional tamper resistance," in *ACM CCS*, 2000, pp. 125–133.

[18] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962 (Experimental), Internet Engineering Task Force, 2013.

[19] D. A. Basin, C. Cremers, T. H. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: attack resilient public-key infrastructure," in *ACM CCS*, 2014.

[20] P. Szalachowski, S. Matsumoto, and A. Perrig, "PoliCert: Secure and flexible TLS certificate management," in *ACM CCS*, 2014.

[21] L. Nordberg, "Transparency gossip," INTERNET-DRAFT, Internet Engineering Task Force, 2014.

[22] ——, "Transparency gossip HTTPS transport," INTERNET-DRAFT, Internet Engineering Task Force, 2014.

[23] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and M. Eran, "Efficient Gossip Protocols for Verifying the Consistency of Certificate Logs," in *Proceedings of the IEEE Conference on Communications and Networks Security (CNS)*. IEEE, 2015.

[24] B. Ramsdell and S. Turner, "Secure/multipurpose internet mail extensions (S/MIME) version 3.2 message specification," RFC 5751, Internet Engineering Task Force, Jan. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5751.txt

[25] "Internet Mail Consortium, S/MIME and OpenPGP," http://www.imc.org/smime-pgpmime.html, Retrieved Feb. 2015.

[26] R.Zimmermann, "The official PGP users guide," 1995, MITPress, Cambridge, MA, USA.

[27] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880, Internet Engineering Task Force, Nov. 2007, updated by RFC 5581. [Online]. Available: http://www.ietf.org/rfc/rfc4880.txt

[28] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO*, 1987, pp. 369–378.

[29] J. Yu, M. Ryan, and C. Cremers, "Tamarin models for the DECIM protocol," 2016, http://www.jiangshanyu.com/doc/paper/DECIM-proof.zip.

[30] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *CAV 2013, Saint Petersburg, Russia, July 13-19, 2013.*, 2013, pp. 696–701.

[31] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," RFC 5246, Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt