How to detect unauthorised usage of a key

Jiangshan Yu
School of Computer Science
University of Birmingham, UK
jiangshan.yu@me.com

Mark Ryan
School of Computer Science
University of Birmingham, UK
m.d.ryan@cs.bham.ac.uk

Cas Cremers
Dept. of Computer Science
University of Oxford, UK
cas.cremers@cs.ox.ac.uk

Abstract

Encryption is useful only if the decryption key has not been exposed to adversaries; in particular, it requires that the device performing the crypto operations is free of malware. We explore ways in which some security guarantees can be achieved even if an attacker has succeeded in obtaining access to all the keys in a device, e.g. by exploiting software vulnerabilities.

We develop a new protocol concept that allows the device owner to detect if another party is using the device's long-term key. We achieve this by making it necessary for uses of the key to be inserted in an append-only log, which the device owner can interrogate. We propose a multi-device messaging protocol that exploits our concept to allow users to detect unauthorised usage of their device keys. We prove the main properties of our protocol using the Tamarin prover.

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

1. INTRODUCTION

Encryption is the main mechanism used to protect the confidentiality of messages sent between computers. It relies on the assumption that the computer end-points can securely store and use cryptographic keys. If this assumption does not hold, then encryption does not guarantee confidentiality. Yet, this assumption is rather hard to justify in practice. New software vulnerabilities [5] are discovered every day, and malware is pervasive on mobile devices such as phones and tablets [8] as well as on traditional platforms like desktop PCs.

The security architecture of mobile devices running Android and iOS is an improvement over the PC security architecture, thanks to better security sandboxing of apps. This sandboxing aims to contain the effects of malware, preventing it accessing secrets of other apps and of the operating system. Unfortunately, the sandboxing model has been shown to be insecure (e.g., [6]), allowing privilege escalation attacks to take place. Android malware is very prevalent, with 1200 samples being collected in a single year [20]. Although new efforts at containing malware are much needed, it does not seem likely that completely secure platforms will be built soon.

We explore ways in which some security guarantees can be obtained, even if the end-point devices have software vulnerabilities that allow an attacker to obtain keys and/or control the device.

If a device is compromised by exploiting software vulnerabilities, and is then made secure again, the attacker remains in possession of secrets (such as keys) he obtained during the compromise. Since victims do not know when compromises take place, they are not motivated to revoke their keys. In practice, it is impractical to ask users to revoke their keys and distribute new ones after every security update.

We develop messaging protocols that allow users to detect if their long-term keys have been compromised and are being used by an attacker. It is clear that if a recipient's device becomes temporarily compromised and leaks all of its secrets, it is impossible to ensure the secrecy of messages sent during the compromised period. Informally speaking, we achieve the following unique security guarantee: if an attacker abuses compromised secrets to learn the contents of messages sent during trustworthy periods, for example by using the recipient's long-term key to impersonate him, then the recipient will detect this whenever he returns to a trustworthy state.

Our proposal not only detects situations in which the adversary has copied a key and uses it, but also situations in which he has access to a key but is not able to copy it (for example, if it is protected in a TPM).

We make use of two ideas in order to obtain our security guarantees:

- We assume that devices are *periodically trustworthy.* That is, a device may become vulnerable or infected at any time, but at some later time it will be again made secure. In other words, we assume that users periodically successfully perform malware scans, operating system upgrades, and software updates, bringing their devices back into a trustworthy state.
- If a user has multiple devices participating, e.g., in a messaging application, then devices that are in a trustworthy state can help detect attacks brought about by devices that are untrustworthy.

We minimise the burden placed on users: in particular, reflecting the fact that it is perceived as a burden to do so, we do not require that users routinely change passwords and regenerate long-term keys. This means that an attacker that has compromised a device and obtained its secrets continues to possess the secrets even after the device has been restored to a trustworthy state.

Contribution Our first contribution develops the idea of *malware damage containment*. This recognises that no architecture is immune from software vulnerabilities and consequent malware, and therefore it is useful to find new ways of limiting the impact that they have. We complement traditional software mitigation techniques (such as sandboxing and privilege limitation) by enabling a victim to detect that secret keys have been compromised. To make this precise, we develop an attacker model in which platforms are *periodically compromised*. That means that they can be compromised by the attacker at any time, but we assume that the user periodically takes steps to remove malware and eliminate vulnerabilities.

Second, we develop two protocols for Key Usage Detection (KUD), in which a receiver with a long-term key can detect if another party has obtained and is using his key. 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.

Our third contribution is the security analysis which shows that the protocols satisfy precise properties expressing software damage containment. Suppose that an attacker has gained possession of a message receiver Robert's long-term private key. We prove that if the attacker uses this key, then Robert will be able to detect that; and therefore will be prompted to revoke the key and generate a new one. We use the TAMARIN prover to prove several key properties of our protocol.

Key usage detection could be applied to solve other problems. One example is to apply it to identity-based signatures (IBS) to mitigate the key escrow problem, as it allows a signer to detect that an unauthorised signature (e.g. one made by the identity provider) has been issued.

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 our usage detection protocols in Section 4. Our protocols depend on a log, whose implementation we detail in Section 5. We analyse the security of our proposal in Section 6 and conclude in Section 7. The Appendix describes our messaging protocol in full detail.

2. BACKGROUND AND RELATED WORK

Apple iMessage Apple iMessage is an encrypted messaging service for iOS devices and Mac computers, with per-device public keys. Although Apple asserts that it cannot decrypt the data [2], this is not verifiable because the software source code and all details beyond what is given in [2] are proprietary and secret. Moreover, even with the architecture described in place, Apple could inspect the contents of messages simply by creating additional public keys corresponding to fake devices. Our paper builds on the iMessage architecture by adding verifiable mechanisms to guarantee that the security measures work.

Google account activity A Google account user can inspect her Google account activity to detect unauthorised usages, based on information including date and time, IP address, device identity, and device software versions. This is useful to defend against third-party attackers, but obviously it does not defend against abuse by Google itself. Our protocol builds on the account activity idea; we use it to present usages of keys to the user. Additionally, our protocol includes verifiable mechanisms to guarantee that the usage activity reported is complete, including possible usages that may be mounted by the service provider.

FlipIt FLIPIT is an abstract game-theoretic framework for modelling security scenarios similar to the attacker model of our paper. In the FLIPIT game [19], 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.

Certificate transparency (CT) *Certificate transparency* (CT) [10] is a technique proposed by Google that aims to efficiently detect fake public key certificates issued by corrupt certificate authorities, by making certificate issuance transparent. Certificates are stored using an append-only Merkle tree log. This enables the log maintainer to provide two types of verifiable cryptographic proofs: (a) a proof that the log contains a given certificate, and (b) a proof that a snapshot of the log is an extension of another snapshot (*i.e.*, only appends have taken place between the two snapshot). The time and size for proof generation and verification are logarithmic in the number of certificates recorded in the log. Domain owners can obtain the proof that their certificates are recorded in the log, and provide the proof together with the certificate to their clients, so the clients can get a guarantee that the received certificate is recorded in the log.

Extended certificate transparency (ECT) *Extended certificate transparency* (ECT) [17] is a proposal for managing certificates for end-to-end encrypted email. It proposes an idea to address the revocation problem left open by CT. It provides transparent key revocation, and reduces reliance on trusted parties by designing the monitoring role so that it can be distributed among user browsers.

Gossip protocol A potential problem in CT or ECT arises if an attacker shows different versions of the log to different clients. This is sometimes called the "bubble" problem; two clients in different bubbles could see different keys for the same subject. A gossip protocol is a mechanism that allows clients of a log to directly exchange digests of the log, in order to ensure that they have the same view of the log. If Alice holding digest dg_A receives a digest dg_B from Bob, she can challenge the log maintainer to prove that dg_A and dg_B are related by extension. Gossip protocols for log transparency are currently being specified [14, 15].

3. ATTACKER MODEL

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 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.

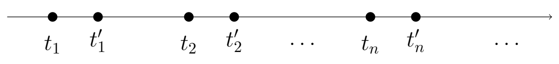


Figure 1: A device is compromised at time t_1, and then restored into a secure state at time t'_1. This cycle is repeated. Thus, the device is in a compromised state during the intervals $\{(t_j, t'_j) \mid j \in \{1, 2, 3, \ldots\}\}$.

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 puts the device back into a trustworthy state. The users do not regenerate long-term keys or change passwords.

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 be exposed to the attacker. Performing security updates 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.

- **Basic KUD 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, $t'_j + \zeta \le t \le t_{j+1} - \zeta$ 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, $t'_{i,j} + \zeta \le t \le t_{i,j+1} - \zeta$ 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.

4. OVERVIEW OF KUD

We present an overview of two protocols for Key Usage Detection (KUD). 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 KUD protocol

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. We assume a log maintainer, which is capable of receiving data and storing it in an append-only log. The log may sign its outputs using its 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 [16], PGP [9, 18, 4], ECT [17], or CONIKS [12].

The log maintainer signs and publishes *digests* of the log. We use 'digest' to denote the unique presentation of the log. 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, and the new added data is considered to be the update of old ones. The log maintainer can also create proofs that a given digest represents an append-only extension of the log represented by a previous digest.

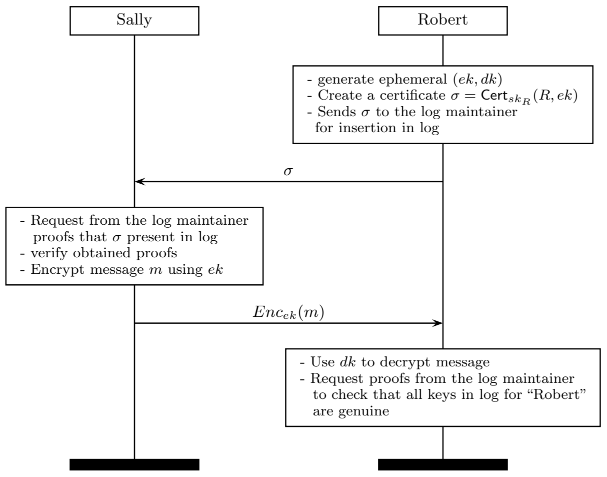
Sally and Robert track the digests issued by the log, all the time checking the proofs issued by the log that later digests represent extensions of earlier ones. Sally and Robert also periodically exchange the digests they have received from the log, and request and check proofs of extension of those digests with respect to those they already have ("gossip protocol").

The KUD 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 long-term signing key sk_R. He publishes the certificate $\mathsf{Cert}_{sk_R}(R, ek)$ in the log.
- To send a message, Sally's device retrieves $\mathsf{Cert}_{sk_R}(R, ek)$ from the log along with a proof of its currency. She encrypts the message with ek and sends it to Robert.
- Robert's device receives the encrypted message and decrypts it.

Additionally, Robert's device periodically checks 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.



Sally
Robert
- generate ephemeral (ek, dk)
- Create a certificate $\sigma = \mathsf{Cert}_{sk_R}(R, ek)$
- Sends σ to the log maintainer for insertion in log
σ
- Request from the log maintainer proofs that σ present in log
- verify obtained proofs
- Encrypt message m using ek
$Enc_{ek}(m)$
- Use dk to decrypt message
- Request proofs from the log maintainer to check that all keys in log for "Robert" are genuine

Figure 2: The basic KUD 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.

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 [13] in which data is inserted by extending the tree to the right. Such a log was designed and introduced in *certificate transparency* [10]. 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* [17, 12].

It is also a requirement that the log maintainer cannot maintain different versions of the log which it shows to different sets of users. Gossip protocols are a known technique to ensure that.

4.2 Messaging application

The messaging application generalises the basic protocol, allowing the users to have multiple devices. In terms of functionality, it is inspired by Apple iMessage (see § 2), but it adds key usage detection and verifiability of the security. 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; and we assume that senders and receivers gossip and check digest extensions. 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.

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 device key 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 $\mathsf{Cert}_{sk_R}(R, ek_\ell, t_\ell)$ 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 $\mathsf{Cert}_{sk_R}(R, ek_i, t_i)$ 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 $\mathsf{Cert}_{sk_R}(R, ek_\ell, t_\ell)$. 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 current device 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

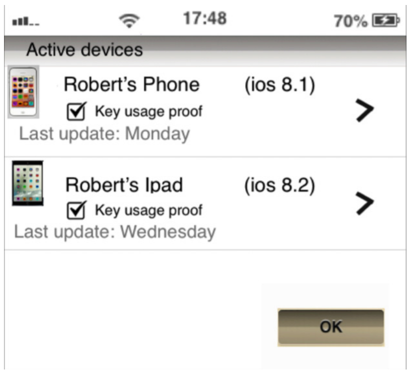


Figure 3: An example of envisaged GUI that presents the table (D_i, t_i) for $i = \{1, 2\}$ to Robert. Section 4.3 describes how Robert uses this information. The figure gives an impression of the kind of user interface we envisage. Usability is important and difficult to get right, and we need to work with HCI experts to design the interface fully.

the log maintainer.

Sending and receiving a message

- To send a message, Sally retrieves $\mathsf{Cert}_{sk_R}(R, ek_i, t_i)$ (for each available i) from the log along with proofs of currency. 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.

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 $\mathsf{Cert}_{sk_R}(R, ek_i, t_i)$ 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 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 $\mathsf{Cert}_{sk_R}(R, ek'_\ell, t_\ell)$ 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 mis-use all the keys stored on the device. The adversary may decrypt messages encrypted with ephmeral keys, and may create new signed ephemeral keys by using the phone's long term key and inserting them into the log. While the phone remains under the control of the attacker, the decryption activity is not detected. However, the long-term key usage is detected if the user 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 verified.

Suppose Robert regains control of his phone, through routine malware scanning and software patching. If the adversary 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.

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. Such attacks will be detected by the gossip protocol: the sender and receiver can detect that the versions of the log they see are inconsistent.

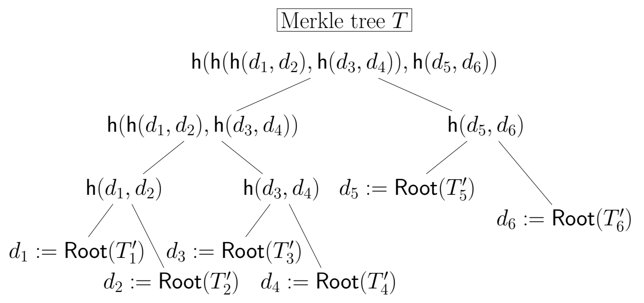
4.4 Responding to attacks

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 boostrapping 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.

5. LOG IMPLEMENTATION AND PROOFS

5.1 Log structure



Merkle tree T
$\mathsf{h}(\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), \mathsf{h}(d_5, d_6))$
$\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4))$
$\mathsf{h}(d_5, d_6)$
$\mathsf{h}(d_1, d_2)$
$\mathsf{h}(d_3, d_4)$
$d_5 := \mathsf{Root}(T'_5)$
$d_6 := \mathsf{Root}(T'_6)$
$d_1 := \mathsf{Root}(T'_1)$
$d_3 := \mathsf{Root}(T'_3)$
$d_2 := \mathsf{Root}(T'_2)$
$d_4 := \mathsf{Root}(T'_4)$

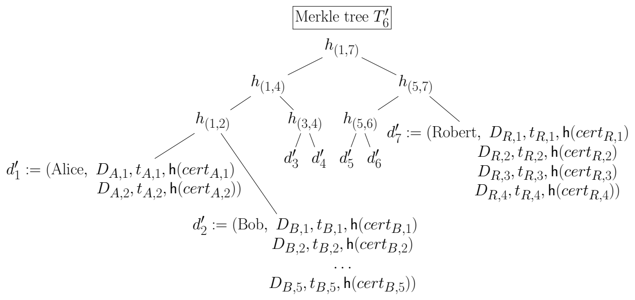
Figure 4: An example of the log containing six updates $\{d_1, d_2, \ldots, d_6\}$. The log is maintained as an append-only Merkle tree T whose leaves are ordered chronologically.

The public log is organised as an append-only Merkle tree [13], and the digest of a log is the root hash value and the size of the 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 [10, 17]. Table 1 shows methods that a Merkle tree supports.

In more detail, our log is organised as a tree of trees. The log is an append-only Merkle tree T (as shown in Figure 4), which 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

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.



Merkle tree T'_6
$h_{(1,7)}$
$h_{(1,4)}$
$h_{(5,7)}$
$h_{(1,2)}$
$h_{(3,4)}$
$h_{(5,6)}$
$d'_7 :=$ (Robert, $D_{R,1}, t_{R,1}, \mathsf{h}(cert_{R,1})$
$D_{R,2}, t_{R,2}, \mathsf{h}(cert_{R,2})$
$D_{R,3}, t_{R,3}, \mathsf{h}(cert_{R,3})$
$D_{R,4}, t_{R,4}, \mathsf{h}(cert_{R,4})$)
d'_3 d'_4 d'_5 d'_6
$d'_1 :=$ (Alice, $D_{A,1}, t_{A,1}, \mathsf{h}(cert_{A,1})$
$D_{A,2}, t_{A,2}, \mathsf{h}(cert_{A,2})$)
$d'_2 :=$ (Bob, $D_{B,1}, t_{B,1}, \mathsf{h}(cert_{B,1})$
$D_{B,2}, t_{B,2}, \mathsf{h}(cert_{B,2})$
$\ldots$
$D_{B,5}, t_{B,5}, \mathsf{h}(cert_{B,5})$)

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.

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 base 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, 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 proof of presence, but there is an extra constraint for the verifier to check, namely that the path to the node is of the form $r \cdot r \ldots \cdot r$.

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 $\mathsf{h}(d_3, d_4)$ 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 of how to generate the proof in different cases is presented in §5.1.2 of [10].

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 Updating the log

We detail how a log maintainer updates the log, and generate proofs for its clients.

5.2.1 Log update for "enrolling a device"

After each period of length δ, the log maintainer has received a list of device enrollment requests

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

where R_i is the client identity, P is the number of devices that a client has requested to enrolling 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, 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 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'_i for R_i in T'_{n+1}, and generates:

- $\mathsf{PoP}(T_{n+1}, d')$ that d' is in present of T';
- $\mathsf{PoC}(T, T'_{n+1})$ 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'_i — which contains a full list of certificates for his devices (including the newly enrolled ones) — are present in the latest update of the log.

5.2.2 Log update for "updating the keys"

Suppose in the current epoch, the log maintainer which maintains the log (represented by T of size n) has the tree T'_n containing

$$\begin{aligned}(\mathit{Alice},\ & D_{A,1}, t_{A,1}, \mathsf{h}(cert_{A,1}) \\ & D_{A,2}, t_{A,2}, \mathsf{h}(cert_{A,2})), \\ (\mathit{Bob},\ & D_{B,1}, t_{B,1}, \mathsf{h}(cert_{B,1}) \\ & D_{B,2}, t_{B,2}, \mathsf{h}(cert_{B,2}) \\ & \dots \\ & D_{B,5}, t_{B,5}, \mathsf{h}(cert_{B,5})), \\ \dots\ & \dots\end{aligned}$$

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 has sent a key update request, and M' is the total number of clients who has sent the key update request in this epoch.

At the turn of the epoch, the log maintainer updates the log by performing 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.

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 if we introduce a trusted party to monitor the entire log. Alternately, to avoid having a trusted party, 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 each of above protocols, namely 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, rather than verifies 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 only going forward. To verify this, each device picks a random leaf for a user in an update T_i, and verifies that 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 corresponding certificates should also be equal. Moreover, 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.

Last, to ensure that the log maintainer did not show different logs to different users, a gossip protocol is needed to exchange the digest of the log that users observed.

Table 2: 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.

	Message	size_P	size_M
Enrolling a device			
	request	-	1.6 KB
	response	2.2 KB	2.5 KB
Total			4.1 KB
Fetching keys from log			
	request	-	78 B
	response	2.2 KB	9.9 KB
Total			10 KB
Updating the keys			
	request	-	1.5 KB
	response	2.2 KB	2.5 KB
Total			4 KB
crowd-sourced verification			
Total		5.3 KB	5.9 KB

5.4 Communication cost

To check if deployment might be feasible, we provide a rough estimation on 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 5 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 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 [7]), each request identifier is 4 bits, and the size of a digest of a log is 300 bits.

The size of 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 2.

From Table 2 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 5 certificates, the size of which is already 7.5 KB. In the crowd-sourced verification, we have excluded the gossip protocol in our calculation, as it depends on how the gossip protocol is designed.

The results of our high-level analysis indicate that the cost of our system is acceptable.

5.5 Privacy considerations

The public log may cause some privacy concerns. For example, one may want to hide the user identities contained in a log, the total number of communications of a user, or the time distribution of a user's communications, etc.

In the above examples, to hide the user identity, the log maintainer can 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 [12]. 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).

There are many other possible solutions (e.g. server side access control). We do not detail them here as they are not the main focus of this paper.

6. SECURITY ANALYSIS

We provide all input files required to understand and reproduce our security analysis at [1]. In particular, these include the complete KUD models.

6.1 Security properties

Our 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.

6.2 Formal analysis

We analyse the main security properties of the KUD protocol using the TAMARIN prover [11]. 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 [3].

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 the one used in most 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 additionally specify rules that enable the attacker to compromise devices and learn their long and short-term secrets.

The final KUD 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 KUD 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 a key, 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 device */
        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 next two properties encode 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. The first main property we prove is 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, but the attacker learns the message without compromising that epoch's ephemeral key, (which is based on impersonating Robert), then the next verification of the log will contain evidence of the misbehaviour.

The final property extends the previous for the messages that Robert actually receives from Sally, and shows that this also leads to detection of the key's abuse.

```
lemma detect_usage_R_receives:
 "All d skR dkR dkR2 m #i1 #i2 #i3 #i4 detectionresult R k.
      /* If S sent to R an encrypted message m,
         where pk(dkR)=ekR */
      ( MsgSent(R, skR, pk(dkR), m) @ #i1 &
        // /* and R receives it */
        MsgReceived(d, R, skR, dkR2, m) @ #i2 &
        /* and the adversary knows m */
        K(m) @ #i3 &
        /* and the ephemeral key used by the sender was
           not compromised, i.e., the compromise was in
           a different epoch then when m was sent.
         */
        not (Ex #j sk .
                Compromise_Device(d, R, sk, dkR) @ #j ) &
        /* and Robert afterwards checks the log */
        CheckedLog(d, R, detectionresult, k ) @ #i4 &
        #i2 < #i4
      )
      ==>
      ( /* then we can detect a compromise */
        (detectionresult = 'bad')
      )
 "
```

The last two properties are proven automatically by TAMARIN on a laptop within a few minutes.

Overall, the modeling effort was in the order of weeks, with several iterations to debug both the abstract model and the property specifications.

Our initial model and property specification could not be automatically verified by TAMARIN and we used the tool's interactive mode to determine the cause of non-termination. Ultimately, this enabled us to use TAMARIN's lightweight heuristics-influencing mechanism, which boils down to adding two lines of code per property, to guide the prover to find the proofs automatically and efficiently. This took several iterations and also revealed errors in earlier specifications, which made it clear that the complexity of the model required us to specify and prove several sanity checks.

Overall, the process helped us not only to prove, but also to refine the precise security properties of our protocol.

7. CONCLUSION

We have presented a novel messaging protocol that offers security guarantees even when an attacker can access 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 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, KUD adds an additional layer of security that allows users to detect when other layers fail.

8. REFERENCES

[1] Anonymous. Tamarin models for the KUD protocol, 2015. `https://www.dropbox.com/sh/8cuvuo4x90wea0q/AABe3m8VTUEn4we2JQOt_Gxra?dl=0`.

[2] Apple Inc. iOS Security. `https://www.apple.com/ca/fr/ipad/business/docs/iOS_Security_Feb14.pdf`, February 2014.

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

[4] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), Nov. 2007. Updated by RFC 5581.

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

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Information Security*, pages 346–360. Springer, 2011.

[7] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[8] 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*, pages 3–14. ACM, 2011.

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

[10] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), 2013.

[11] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 696–701, 2013.

[12] 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.

[13] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[14] L. Nordberg. Transparency gossip. INTERNET-DRAFT, 2014.

[15] L. Nordberg. Transparency gossip HTTPS transport. INTERNET-DRAFT, 2014.

[16] B. Ramsdell and S. Turner. Secure/multipurpose internet mail extensions (S/MIME) version 3.2 message specification. RFC 5751 (Proposed Standard), Jan. 2010.

[17] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. in network and distributed system security. In *NDSS*, 2014.

[18] R.Zimmermann. The official PGP userâĂŹs guide, 1995. MITPress, Cambridge, MA, USA.

[19] M. van Dijk, A. Juels, A. Oprea, and R. L. Rivest. Flipit: The game of "stealthy takeover". *J. Cryptology*, 26(4):655–713, 2013.

[20] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109. IEEE, 2012.

A. Detailed protocol

This section provides the detailed protocols with figures. It would help readers to see the exact exchanged messages. In addition, it would help readers to understand our TAMARIN code provided at [1].

A.1 Enrolling a device

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:

- D_ℓ generates a new ephemeral key pair (dk_ℓ, ek_ℓ) for decryption and encryption, respectively. Then, D_ℓ issues a self-signed 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 reg_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 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.
- D_ℓ verifies signatures and proofs, and sends the request m_3 containing a request req_1', its identity 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 as described in §5.2.1.
- 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 will be used to un-register a device with the log maintainer.

A.2 Sending and receiving a message

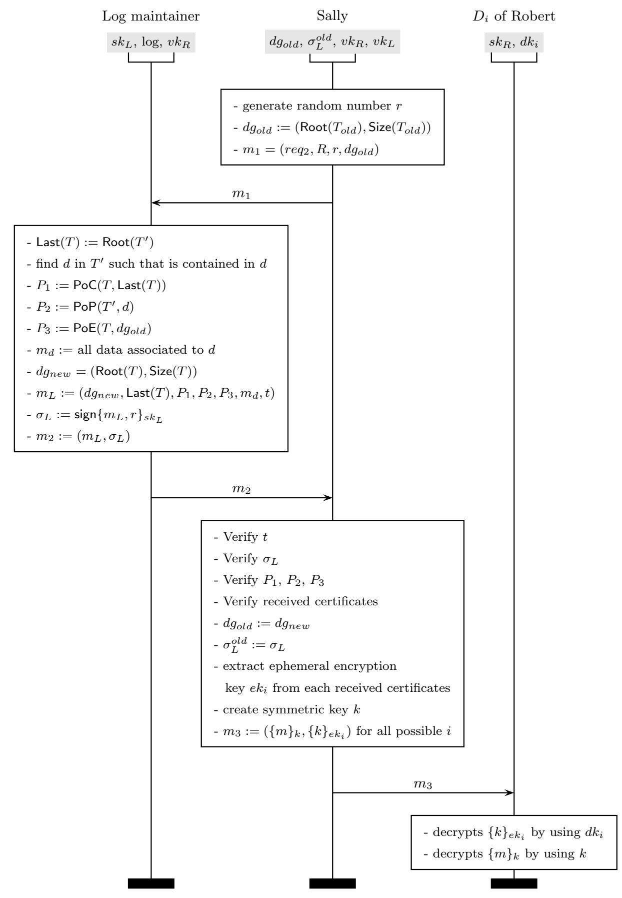
To send a message to Robert, Sally's device will retrieve all the current device certificates for Robert from the log, and encrypt the messages with each of them. More precisely (as presented in Figure 6), to send a message:

- Sally sends request $m_1 = (req_2, R, r, dg_{old})$ to the log, where req_2[1] 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 receives 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 $\mathsf{Root}(T')$ 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 the signed message ('CertResp', $dg_{new}, \mathsf{Last}(T), P_1, P_2, P_3, r, m_d, t$) to Sally, 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 receives the message from the log maintainer, Sally verifies if t is corresponding 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. Similar as what PGP does, 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.

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

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.

Remark 1. *The signed t is used to prevent attacks that replay a selected version of the log from the (compromised) log maintainer. Let x be the version of the log that Sally has previously observed, and z be the latest update. The replay attack is that the log maintainer picks and replays a version y of the log, such that $x < y < z$, and Robert's ephemeral key that is valid in the version y is compromised by the attacker. In this case, even with gossip protocol, all verification will succeed (if without having t_{new}), because version y is a genuine version of the log (though it is not the latest one).*



Log maintainer
Sally
D_i of Robert
sk_L, log, vk_R
dg_{old}, σ_L^{old}, vk_R, vk_L
sk_R, dk_i
- generate random number r
- $dg_{old} := (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$
- $m_1 = (req_2, R, r, dg_{old})$
m_1
- $\mathsf{Last}(T) := \mathsf{Root}(T')$
- find d in T' such that is contained in d
- $P_1 := \mathsf{PoC}(T, \mathsf{Last}(T))$
- $P_2 := \mathsf{PoP}(T', d)$
- $P_3 := \mathsf{PoE}(T, dg_{old})$
- $m_d :=$ all data associated to d
- $dg_{new} = (\mathsf{Root}(T), \mathsf{Size}(T))$
- $m_L := (dg_{new}, \mathsf{Last}(T), P_1, P_2, P_3, m_d, t)$
- $\sigma_L := \mathsf{sign}\{m_L, r\}_{sk_L}$
- $m_2 := (m_L, \sigma_L)$
m_2
- Verify t
- Verify σ_L
- Verify P_1, P_2, P_3
- Verify received certificates
- $dg_{old} := dg_{new}$
- $\sigma_L^{old} := \sigma_L$
- extract ephemeral encryption key ek_i from each received certificates
- create symmetric key k
- $m_3 := (\{m\}_k, \{k\}_{ek_i})$ for all possible i
m_3
- decrypts $\{k\}_{ek_i}$ by using dk_i
- decrypts $\{m\}_k$ by using k

Figure 6: 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 fails, the agent aborts the protocol.

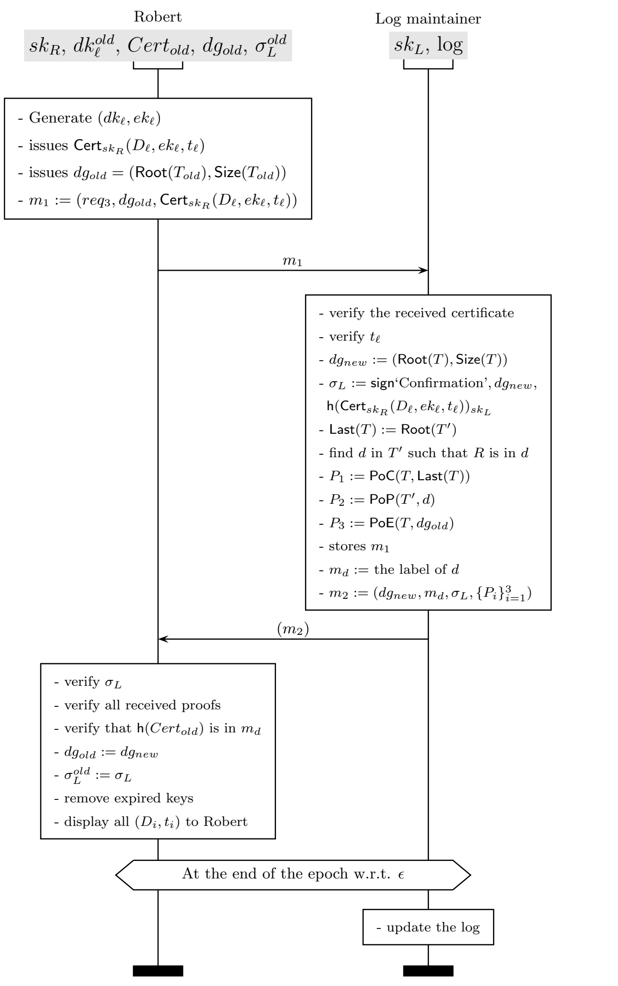
A.3 Updating the keys

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. 3 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 $m_1 = (req_3, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$ to the log maintainer, where req_3[2] 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 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 in present of 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, as described in §5.2.2. If a device has not update 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 (e.g. an hour).

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



Robert
sk_R, dk_ℓ^{old}, $Cert_{old}$, dg_{old}, σ_L^{old}
Log maintainer
sk_L, log
- Generate (dk_ℓ, ek_ℓ)
- issues $\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell)$
- issues $dg_{old} = (\mathsf{Root}(T_{old}), \mathsf{Size}(T_{old}))$
- $m_1 := (req_3, dg_{old}, \mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))$
m_1
- verify the received certificate
- verify t_ℓ
- $dg_{new} := (\mathsf{Root}(T), \mathsf{Size}(T))$
- $\sigma_L := \mathsf{sign}$'Confirmation', dg_{new}, $\mathsf{h}(\mathsf{Cert}_{sk_R}(D_\ell, ek_\ell, t_\ell))_{sk_L}$
- $\mathsf{Last}(T) := \mathsf{Root}(T')$
- find d in T' such that R is in d
- $P_1 := \mathsf{PoC}(T, \mathsf{Last}(T))$
- $P_2 := \mathsf{PoP}(T', d)$
- $P_3 := \mathsf{PoE}(T, dg_{old})$
- stores m_1
- $m_d :=$ the label of d
- $m_2 := (dg_{new}, m_d, \sigma_L, \{P_i\}_{i=1}^3)$
(m_2)
- verify σ_L
- verify all received proofs
- verify that $\mathsf{h}(Cert_{old})$ is in m_d
- $dg_{old} := dg_{new}$
- $\sigma_L^{old} := \sigma_L$
- remove expired keys
- display all (D_i, t_i) to Robert
At the end of the epoch w.r.t. ϵ
- update the log

Figure 7: 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.