Revoke and Let Live: A Secure Key Revocation API for Cryptographic Devices

Véronique Cortier
CNRS, Loria, UMR 7503
Vandœuvre-lès-Nancy,
F-54506, France
cortier@loria.fr

Graham Steel
INRIA
Paris, France
graham.steel@inria.fr

Cyrille Wiedling
CNRS, Loria, UMR 7503
Vandœuvre-lès-Nancy,
F-54506, France
cyrille.wiedling@loria.fr

ABSTRACT

While extensive research addresses the problem of establishing session keys through cryptographic protocols, relatively little work has appeared addressing the problem of revocation and update of long term keys. We present an API for symmetric key management on embedded devices that supports key establishment and revocation, and prove security properties of our design in the symbolic model of cryptography. Our API supports two modes of revocation: a passive mode where keys have an expiration time, and an active mode where revocation messages are sent to devices. For the first we show that once enough time has elapsed after the compromise of a key, the system returns to a secure state, i.e. the API is robust against attempts by the attacker to use a compromised key to compromise other keys or to keep the compromised key alive past its validity time. For the second we show that once revocation messages have been received the system immediately returns to a secure state. Notable features of our designs are that all secret values on the device are revocable, and the device returns to a functionally equivalent state after revocation is complete.

Categories and Subject Descriptors

K.6.m [**Miscellaneous**]: Security

Keywords

Revocation, API, formal methods

1. INTRODUCTION

Embedded systems deployed in hostile environments often employ some dedicated tamper-resistant secure hardware to handle cryptographic operations and keep keys secure. Examples include mobile phones (which contain SIM cards), smartphones (recent models include 'Secure Elements'), public transport ticketing systems (such as the Calypso system which employs smartcards and 'SAM' modules [10]), smart utility meters (that include a smartcard-like chip for cryptography), on-vehicle cryptographic devices to support vehicle-to-vehicle networking [16] *et cetera*. In such systems, it is often necessary to support the possibility of remotely revoking and updating the long-term keys on the device. However, while extensive research addresses the problem of establishing a new session key or determining what security properties can be guaranteed in the event of long-term key corruption, relatively little work has appeared addressing the problem of revocation and update of long-term keys.

Most existing solutions for key revocation follow one of two approaches: either key revocation actually relies on some 'longer term' key that cannot be itself revoked, or key revocation is simply achieved by disabling, resetting or isolating the compromised device. For many applications, both these approaches are unsatisfactory. We propose that a key revocation API should ideally satisfy the following properties:

1. *The device should remain functional* - specifically, whenever possible, the device should return to an equivalent functional state after revocation (but with fresh keys in place).

2. *Any key should be revocable* - side-channel attacks may compromise (perhaps with significant effort) any of the keys stored on a cryptographic device, and the more sensitive a key is, the more likely an attacker is to dedicate effort to breaking it. Hence it is not prudent to decide in advance which keys may or may not be compromised.

The first main contribution of this paper is the design of an API with update and revocation functionalities that satisfy these properties. We assume a very general scenario in which a population of user devices (which could be anything from employee smartcards to enterprise HSMs and can include clients and servers) wish to establish keys and communicate securely between themselves, and a small number of administrator devices are used only to setup new devices and to revoke compromised keys. We start from an existing API for symmetric key management [4] capable of supporting a variety of protocols and show how to extend it with commands for revocation and update of long-term keys. Our design allows the right to update long-term keys to be shared or delegated in a secure way, making it suitable for the modern embedded systems environment where many actors (manufacturer, infrastructure provider, service provider, application developer) may need different access rights. We also discuss some limitations of revocation when symmetric key

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

cryptography is the only operation available (as is in fact the case in many real-world embedded systems).

Our second main contribution is a formal proof of security of our API. We show that it ensures two main properties. First, when a key is lost, the system is able to repair itself after some time. Indeed, our first technique for revoking a key is time-based: each key expires after some delay. Second, if the need to revoke a key is urgent, an explicit revocation command allows us to immediately secure the device, by prohibiting keys of certain levels for a certain period of time. We show in an example how with careful labelling of key levels this can be used to maintain full functionality. Our API makes use of special management keys, which may of course be revoked as well. Our security proofs are carried out in a symbolic model, based on terms for representing messages. The detailed proofs are available in a research report [5].

Related work.

Most deployed key revocation mechanisms are quite simple. For example, the Trusted Platform Module [15] supports a `TPM_OwnerClear` command that resets the 'ownership' status of the device and erases most keys such as the root key for the storage hierarchy. It does not support, for example, the revocation of individual keys in the hierarchy. In multicast group key management schemes, hierarchies are also standard, and here revocation of a key corresponds either to permanent removal of an entity from the group, or removal until the entity resubscribes, in which case long term secrets are assumed to be still intact. Our work will not make this assumption. One lesson from this domain is that many proposals published without proofs of security have turned out to have attacks [14].

Richer key revocation schemes have received some previous attention in the academic literature, e.g. in the key management schemes of wireless sensor networks. Here nodes in the network are expected to be deployed in hostile environments where key compromise may occur. Eschenauer and Gligor propose a scheme under which every node i shares a long-term symmetric key K_{Ci} with a control server [7]. If a node n is compromised, the server sends a fresh signature key S_i to each nodes i sharing keys with n encrypted under K_{Ci}, and then signs a list of keys to be revoked and sends this list to each i. The nodes then delete the keys, isolating the compromised node. This scheme has two disadvantages for the general case: first it assumes that a central authority knows the key identifiers of all the keys in the network and between what parties they are shared, and second it has no way to recover a device which has lost its K_{Ci} to the attacker. An alternative scheme is KeyRev [17], which proposes the use of a secret sharing scheme to distribute session keys only to unrevoked nodes, thereby isolating compromised notes without explicitly revoking their long term keys. Again this has the disadvantage that the loss of a long term key means the unrecoverable loss of a device: acceptable in a wireless sensor network perhaps but not in the general case. In this paper we aim to be able to securely update all keys on the device, provided not too many keys have been compromised.

The Sevecom API [9] is a proposal for an on-board tamper-resistant device to handle cryptography in next-generation vehicles supporting VANETs (vehicle to vehicle ad-hoc networks). It includes two root public keys which are used to check authenticity of messages coming from a central server. The signed messages from the central server are used to trigger updates of working keys. The Sevecom API is interesting because it allows update of the long term root keys using a simple two step protocol. We will examine this example in more detail in the next section.

There are other proposals for key management APIs with security proofs in the literature, but none of these address the question of revocation [3, 6, 8].

2. REVOCATION API DESIGN ISSUES

We set the following requirements for our design: (1) the device should return to full functionality whenever possible, and (2) all secret values should be revocable. Resources within the tamper-resistant boundary of an embedded crypto device are usually tightly restricted. We therefore design our API to use a minimum of memory (instead of storing lists of blacklisted keys, we blacklist sets of keys through their attributes) and cryptographic functionality (just symmetric encryption using an authenticated encryption scheme). We do however require that the devices contain loosely synchronized real-time clocks. This assumption is standard in some domains, e.g. on-vehicle cryptographic units [9], Smartmeters, but is not suitable for every application - we will comment briefly in section 6 on how to adapt our design to devices without clocks.

Our starting point is a generalisation of the symmetric key management API of Cortier and Steel [4], which defines a simple hierarchy on keys under management whereby keys higher in the hierarchy are allowed to encrypt (wrap) keys lower down, for storage outside the device or as part of a key exchange protocol. We will describe the API fully in the next section. Here we note some design constraints that are imposed on us by our key revocation requirements, since these may be of independent interest to future designers of revocation APIs.

Firstly, since we can have no single upper bound to the key hierarchy in order to satisfy requirement (2), our design must incorporate a set of keys of the same level Max which can revoke each other. The Sevecom API mentioned in section 1 tackles this by having two root keys, where knowledge of the private halves of both is required in order to revoke and update any one of them. The update protocol is given in figure 1, where the terms to the left of the semi-colon represent protocol messages, and the terms to the right are predicates on the state of the device. Initially, the device is in a 'two key state', with root keys k_1 and k_2 available for use. Then a revoke message is given for key k_1, which consists of k_1 signed under its own private half $inv(k_1)$. The device receives this and deletes k_1, moving into a 'one key state' where no further revocation messages will be processed. Then an update message is given, using k_2 to sign the new root key k_3.

$\mathsf{RevokeRootKey1}:$

$$\{k_1\}_{inv(k_1)}; \mathsf{keys}(k_1, k_2) \quad \rightarrow \quad ; \mathsf{keys}(_, k_2)$$

$\mathsf{UpdateRootKey1}:$

$$\{k_3\}_{inv(k_2)}; \mathsf{keys}(_, k_2) \quad \rightarrow \quad ; \mathsf{keys}(k_2, k_3)$$

Figure 1: Sevecom Revocation Protocol (simplified)

At first it may seem that this protocol is achieves its security goal: unless the intruder manages to obtain access to both $inv(k_1)$ and $inv(k_2)$, he cannot replace a key with one of his own. However, analysis by Mödersheim and Modesti showed that there is another attack scenario [12]: suppose an attacker had corrupted only $inv(k_2)$, and then waited until the server sent a revocation message for k_1. He could then intervene and inject his own root key k_i using $inv(k_2)$ to sign it. In a footnote, Mödersheim and Modesti propose that both signature keys be used to sign both messages, and the current public keys be included in the message to prevent replays of old updates. They did not verify this solution however. In our scheme we generalise the idea to a scheme where some number N_{Max} of the K level Max keys are required in order to revoke and replace one. We similarly require the revocation message to demonstrate knowledge of the current key in order to avoid replays, but we do this by encrypting under the old key rather than adding it to the plaintext, to prevent possible key cycles. Since the old key may be corrupted, it is the innermost encryption. A revocation message, e.g. 'replace k_i with k_j', will therefore have the following form:

$$\{\mathsf{update}, k_j\}_{k_i\,k_1 \cdot^{\cdot^{\cdot}} k_n}$$

In our setting, we only have symmetric key cryptography available, so unlike the public keys in Sevecom, our root keys must remain confidential. This imposes an upper limit to the security we can achieve: if at any point in the future, it is possible for the attacker to obtain all the keys $k_1, \ldots, k_n$, then he will obtain also k_j, and by repeating this for all the subsequent key updates be able to obtain N_{Max} of the current level Max keys. To prevent this, for our security proofs, we demand that 'honest' updates to the level Max keys, i.e. updates generated by the server, remain unknown to the attacker. In practice this could mean that the level Max keys are only updated when the device is connected to a trusted host. If this is too cumbersome for the application then a quantitative risk assessment would have to be undertaken to set the N and K high enough to achieve the required degree of security. A full solution would require asymmetric cryptography, as we will discuss in section 6. Note that we will still prove that the device will resist attacks against fake update messages constructed by the intruder, even when he has corrupted individual level Max keys.

For keys of lower level than Max, we assume that the attacker can see all (encrypted) update messages. We need some way to ensure freshness of messages containing keys encrypted by other keys, otherwise revocation will be ineffective. To see this, consider the following example: Assume $\{..., k_3, ...\}_{k_5}$ and $\{..., k_3, ...\}_{k_4}$ have been sent out through normal key encryption commands, with k_i at level $i (< \mathsf{Max})$ in the hierarchy. Assume that at some point k_4 is lost to the attacker. Then he learns k_3 as well (by decrypting the second message). Our revocation API would allow us to remove k_4, and if we are smart we will also remove k_3 since it must be assumed to be lost as well. However, this will not suffice: the attacker could replay the key distribution message $\{..., k_3, ...\}_{k_5}$, and therefore re-inject the corrupted k_3 into the device. To avoid this one could blacklist all keys below a corrupted key, which is problematic (this blacklist will take up device memory and will have to be kept indefinitely), or revoke all the keys in the hierarchy all the way to the top (impractical), or assure some kind of freshness of key distribution messages. We choose the latter option, requiring that every key is given a validity time when it is issued. After the expiry date passes, the device will refuse to use the key. Without this property, we would need some other way to ensure freshness of all messages, such as challenge-response exchanges for every message.

Finally, note that though we design our API to use memory efficiently, we do not explicitly analyse its resistance to denial of service attacks.

3. DESCRIPTION OF THE API

We describe the design and commands for our key management and key revocation API.

3.1 Setting

We assume that keys are structured in a hierarchy, such that a key k_1 may encrypt a key k_2 only if k_1 is greater than k_2. More precisely, we assume a set of levels $\mathcal{L}$ equipped with a (partial) order $<$, a maximal element Max and minimal element 0. Management keys used for revocation or updates will be keys of maximum level Max, and knowledge of at least N_{Max} of them will be required for many operations. We also assume that a level can only be compared to finite number of levels. More precisely, we assume that for any $l \in \mathcal{L}$, the set $\{l' \mid l' < l\}$ is finite. This will ensure that we have no infinite sequence of the form $\{\ldots k_1 \ldots\}_{k_2}, \{\ldots k_2 \ldots\}_{k_3}, \ldots$

We assume that each tamper resistant device (TRD) a has:

- a clock, whose current time is given by $t_a \in \mathcal{T}$ where $\mathcal{T}$ is an infinite ordered set of time events. For example, $\mathcal{T}$ may be $\mathbb{R}^+$, the set of non negative real numbers. For simplicity, we will assume that all clocks are synchronized with a global clock, referred to as the time of the global system. Our security properties could be adapted to take account of clock drift, provided some limit on the drift is assured.

- a table Θ_a of keys. Each entry in the table is indexed by a *handle* h and the corresponding entry $\Theta_a(h)$ is (k, l, v, m) where k is the actual key stored on the TRD, l is its corresponding level, v its *validity date* of the key, and $m \in \mathcal{M}$ is a *miscellaneous field* that may describe some other attributes of the key (e.g. describing the purpose of the key).

- a blacklist $\mathfrak{B}_a$ of elements of the form (l, t) where l is a level and t is an expiration time. Intuitively, whenever (l, t) occurs in a blacklist, it means that the TRD will never accept a key of level l (or below), unless time t has now passed.

Intuitively, the design of our API will ensure that a key may only encrypt other keys lower in the hierarchy whose validity dates have not expired.

We also assume that the TRDs share a function $\delta : \mathcal{L} \to \mathcal{T}$, that associates lifetime to keys depending on their levels. We may sometimes abbreviate $\delta(l)$ by δ_l. Since we have assumed that a level can only be compared to finite number of levels, we can compute the maximum lifetime of a chain of levels, smaller than a given level. Formally, we consider

the function

$$\Delta : l \mapsto \max_{l_1 < \cdots < l_n < l, n \in \mathbb{N}} \sum_{i=1}^{n} \delta(l_i)$$

Intuitively, $\Delta(l)$ is the time where compromising a key k of level l may compromise keys of lower levels, even if the validity time of k has expired, due to chains of encryptions (see Section 5).

For initialisation, we assume that each device contains at least N_{Max} keys of level Max which are known to an administrator. From this, using the commands of the API, the administrator can bootstrap the system and, if necessary, update all the level Max keys.

3.2 Commands

We first give the standard commands which do not concern revocation or keys of level Max, but suffice for normal key management operations. The are mostly the same as in the original API, [4], where they were shown to be sufficient to implement a number of key establishment protocols while always keeping sensitive keys in the secure memory of the TRD, never exposing them in the clear on the host machine. We have generalised the ordering on the hierarchy and introduced checking of the validity time of keys and a blacklist $\mathfrak{B}_a$ of key levels together with expiration times. We will write $l \in \mathfrak{B}_a$ if there is $(l', t) \in \mathfrak{B}_a$ with $l \leq l'$ and t a valid time. We also assume a test $\mathsf{Distinct}(h_1, \ldots, h_n)$ which checks that the h_i are pairwise distinct.

Public generation

The generation command for public (level 0) data (e.g a fresh public nonce), which possibly takes an argument $m \in \mathcal{M}$, is defined as follows. It returns both the value and a handle to the value as stored on the device.

```
generatePublic(m)
    let h = Fresh(H_a) in
    H_a := H_a ∪ {h}
    let n = Fresh(N) in
    N := N ∪ {n}
    let v = t + δ(0) in
    Θ_a := Θ_a ∪ {h → (n, 0, v, m)}
    return h, n
```

where $\mathsf{Fresh}(E)$ (respectively with $E \in \{H_a, N, K\}$) returns a element of the set $\mathcal{E} \setminus E$ (respectively with $\mathcal{E} \in \{\mathcal{H}_a, \mathcal{N}, \mathcal{K}\}$).

Secret generation

The generation command for a secret key, with level of security l and, possibly, an argument $m \in \mathcal{M}$:

```
generateSecret(l, m)
    if 0 < l < Max
        let h = Fresh(H_a) in
        H_a := H_a ∪ {h}
        let k = Fresh(K) in
        K := K ∪ {k}
        let v = t_a + δ(l) in
        Θ_a := Θ_a ∪ {h → (k, l, v, m)}
        return h
```

Encryption

The encryption command takes as input a list of data that are meant to be encrypted (which may include handles pointing to values stored on the device) and a handle for the key k that will be used for encryption. We check that k has not expired and is not below a level which has been blacklisted. For each term to be encrypted, we check the level is lower than that of k, the expiration date is valid, and that the level is not blacklisted. Note that break aborts the entire command.

```
encrypt([X_1, ..., X_n], h)
    let (k, l, v, m) = Θ_a(h) in
    if (l = Max) ∨ (v ≤ t_a) ∨ (l ∈ B_a)
        break
    for i = 1..n
        if X_i = M_i, m_i /* message */
            Y_i := (M_i, 0, t_a + δ(0), m_i)
        if X_i = h_i /* handle */
            let (k_i, l_i, v_i, m_i) = Θ_a(h_i) in
            if (l_i < l) ∧ (v_i > t_a) ∧ (l ∉ B_a)
                Y_i := (k_i, l_i, v_i, m_i)
            else break
    return {Y_1, ..., Y_n}_k
```

Decryption

The decryption command takes as inputs a handle for the key that will be used for decryption and a cipher-text. We also assume that decrypt throws break on failure of authentication (i.e. we assume an authenticated encryption scheme). Note that the checks on the levels etc. performed during encryption are repeated. This is important for security in the presence of corrupted keys.

```
decrypt(C, h)
    let (k, l, v, m) = Θ_a(h) in
    if (l = Max) ∨ (v ≤ t_a) ∨ (l ∈ B_a)
        break
    let X_1, ..., X_n = dec(k, C) in
    for i = 1..n
        let (k_i, l_i, v_i, m_i) = X_i in
        if (l_i = 0) ∧ (t_a < v_i ≤ t_a + δ(l_i)) ∧ (l_i ∉ B_a)
            Y_i := k_i, m_i /* message */
        elsif (l_i < l) ∧ (t_a < v_i ≤ t_a + δ(l_i)) ∧ (l_i ∉ B_a)
            let h_i = Fresh(H_a) in
            H_a := H_a ∪ {h_i}
            Y_i := h_i /* handle */
            Θ_a := Θ_a ∪ {h_i → (k_i, l_i, v_i, m_i)}
        else break
    return Y_1, ..., Y_n
```

3.3 Management of Revocation keys

We now introduce the commands for managing keys of level Max, which will only need to be used if a compromise to one of these keys is suspected. These keys, called *revocation keys*, can be used either to revoke and update lower level keys (keys with a smaller level than Max), or to revoke and update the revocation keys. At least N_{Max} revocation keys must be given to revoke or update a revocation key.

Update Max

If we cannot trust a certain revocation key anymore, the administrator can update it using the updateMax function. It takes as inputs a cipher-text of the key to be updated and its update encrypted with n revocation keys $k_1, \ldots, k_n$ such that $n \geq N_{\mathsf{Max}}$ and takes also the corresponding handles where these keys are stored, $h_1, \ldots, h_n$. updateMax commands will be assumed to be sent on secure channels to avoid that, if an attacker breaks very old Max keys, he could immediately deduce the current active Max keys. Actually, we do not need all updateMax commands to be runned under secure channels but simply that this occurs sufficiently regularly.

updateMax$(C, h_1, \ldots, h_n)$
  for $i = 1..n$
    let $(k_i, \mathsf{Max}, v_i, m_i) = \Theta_a(h_i)$ in
  if $\exists\, j \in [\![1, n]\!]$ s.t. $v_j \leq t_a \vee \neg\mathsf{Distinct}(h_1, \ldots, h_n)$
    break
  let $(\mathsf{updateMax}, k', v'_k, m'_k) = \mathsf{dec}(k_1, \ldots, \mathsf{dec}(k_n, C))$ in
  if $(v'_k \leq t_a) \vee (v'_k > t_a + \delta_{\mathsf{Max}})$
    break
  $\Theta_a(h_1) := (k', \mathsf{Max}, v'_k, m'_k)$

Note that the updateMax command does not introduce key cycles and avoid replays.

3.4 Management of Working keys

We call any key of level $l : l < \mathsf{Max}$ a *working key*. We include separate commands for creating working keys on the device and for updating or revoking them, each of which require a N_{Max} level Max keys. These commands are in addition to the usual operational key management functions arising from the encryption and decryption commands given in section 3.2: they are intended to be used for bootstrapping the system or for removing and updating possibly compromised keys. In the descriptions below, $n \geq N_{\mathsf{Max}}$.

Create

The create function takes as inputs a cipher-text containing the "order" to create and the different data to create, the whole encrypted with n revocation keys $k_1, \ldots, k_n$ and takes also the corresponding handles where these keys are stored, $h_1, \ldots, h_n$.

create$(C, h_1, \ldots, h_n)$
  for $i = 1..n$
    let $(k_i, \mathsf{Max}, v'_i, m'_i) = \Theta_a(h_i)$ in
  if $\exists\, j \in [\![1, n]\!]$ s.t. $v'_j \leq t_a \vee \neg\mathsf{Distinct}(h_1, \ldots, h_n)$
    break
  let $C' = \mathsf{dec}(k_1, \ldots, \mathsf{dec}(k_n, C))$ in
  let $(\mathsf{create}, x_1, l_1, v_1, m_1, \ldots, x_p, l_p, v_p, m_p) = C'$ in
  if $\exists\, j$ s.t. $(l_j \geq \mathsf{Max}) \vee (v_j \leq t_a) \vee$
    $(v_j > t_a + \delta(l_j)) \vee (l_j \in \mathfrak{B}_a)$
    break
  for $j = 1..p$
    let $h'_i = \mathsf{Fresh}(H_a)$ in
    $H_a := H_a \cup \{h'_i\}$
    $\Theta(h'_i) := (x_i, l_i, v_i, m_i)$
  return $h'_1, \ldots, h'_p$

Update

The update function takes as inputs a cipher-text containing the "order" to update, the different values to change and their updates, the whole encrypted with n revocation keys $k_1, \ldots, k_n$ and takes also the corresponding handles where these keys are stored, $h_1, \ldots, h_n$.

update$(C, h_1, \ldots, h_n)$
  for $i = 1..n$
    let $(k_i, \mathsf{Max}, v_i, m_i) = \Theta_a(h_i)$ in
  if $\exists\, j \in [\![1, n]\!]$ s.t. $v_j \leq t_a \vee \neg\mathsf{Distinct}(h_1, \ldots, h_n)$
    break
  let $C' = \mathsf{dec}(k_1, \ldots, \mathsf{dec}(k_n, C))$ in
  let $(\mathsf{update}, x_1, x'_1, l'_1, v'_1, m'_1, \ldots, x_p, x'_p, \ldots, m'_p) = C'$ in
  for $j = 1..p$
    for $h \in H_a$ s.t. $\Theta(h) = (x_j, l, v, m)$
    if $(l < \mathsf{Max}) \wedge (l'_j = l) \wedge (t_a < v'_j \leq t_a + \delta(l_j))$
    $\wedge (l'_j \notin \mathfrak{B}_a)$
      $\Theta(h) := (x'_j, l'_j, v'_j, m'_j)$

Revoke

We first define the revoke command in its most general form. The administrator may wish to revoke keys, for example if he suspects them to be compromised. He may wish to revoke a precise set of keys but he may also wish to revoke them on the basis of their attributes, e.g. validity time. To be as flexible as possible, we consider a revoke command that is parametrized by a function F such that :

$$F : \mathcal{L} \times \mathcal{T} \times \mathcal{M} \rightarrow \{\bot, \top\}$$

which defines, according to some criteria (e.g. level, validity, etc) what is going to be kept or deleted.

The revoke function takes as input such a function F, encrypted with n revocation keys, and the corresponding handles $h_1, \ldots, h_n$.

revoke$(C, h_1, \ldots, h_n)$
  for $i = 1..n$
    let $(k_i, \mathsf{Max}, v_i, m_i) = \Theta_a(h_i)$ in
  if $\exists\, j \in [\![1, n]\!]$ s.t. $v_j \leq t_a \vee \neg\mathsf{Distinct}(h_1, \ldots, h_n)$
    break
  let $(\mathsf{revoke}, F) = \mathsf{dec}(k_1, \ldots, \mathsf{dec}(k_n, C))$ in
  for $h \in H_a$ s.t. $\Theta_a(h) = (x, i, v, m)$
    if $F(i, v, m) = \top$
      $\Theta_a := \Theta_a \setminus \{h \mapsto (x, i, v, m)\}$

Note that, in practice an implementation should offer some specialized revoke functions for the particular application. Indeed, sending an arbitrary Boolean function as a parameter would raise both implementation and security issues. However, by defining F in this general way we ensure that our security results hold for any choice of F.

Blacklist

The Blacklist function takes as input a cipher-text containing the levels to blacklist together with expiration times, encrypted with n revocation keys $k_1, \ldots, k_n$. The effect of the command is to add the levels to the blacklist and erase all the keys of the corresponding levels, including smaller ones. Blacklisting the level instead of all the keys saves memory. It also means the administrator does not need to retain the actual values of all the working keys.

blacklist$(C, h_1, \ldots, h_n)$
  for $i = 1..n$
    let $(k_i, \mathsf{Max}, v_i, m_i) = \Theta_a(h_i)$ in

```
if ∃ j ∈ ⟦1, n⟧ s.t. v_j ≤ t_a ∨ ¬Distinct(h_1, ..., h_n)
  break
let (blacklist, (l_1, t_1), ..., (l_p, t_p)) =
  dec(k_1, ..., dec(k_n, C)) in
for i = 1..p
/* add to blacklist */
  𝔅_a := 𝔅_a ∪ {(l_i, t_i)}
/* erase affected keys */
    for h ∈ H_a s.t. Θ_a(h) = (x', l', v', m')
      if l' ≤ l_i
        Θ_a := Θ_a \ {h ↦ (x', l', v', m')}
```

3.5 Example

Let us consider two TRDs a and b (see Figure 2) initialized with handles containing keys of level Max k and k' and sharing key k_1 of level l_1. In a first step, the user of device a generates, using the generateSecret command, a secret key k_2 with a level l_2 (such that $l_2 < l_1$) and a validity v_2, this new secret appears in TRD a stored under a new handle. To share the key with b, a encrypts k_2 under k_1 using encrypt. When the message is received by the user of b he uses the decrypt command which will store the new key k_2 under a new handle.

3.6 Threat Scenario

We consider a scenario where the attacker:

- controls the network (he may read and send messages on the network),
- controls the host machines on which the TRD are connected; therefore he can execute API commands on any TRD
- may break some arbitrary keys of his choice, typically by brute-forcing some keys or employing side-channel attacks.

In the remainder of the paper we show that, provided the attacker does not break too many keys of level Max (an attacker should not break more than $N_{\mathsf{Max}} - 1$ keys simultaneously stored on a TRD), then our system can self-repair given time or one can explicitly fix a TRD. More precisely, we show the two following results:

- If some key of level $l \neq \mathsf{Max}$ is lost then after some time (actually, at time $v + \delta_l$ where v is the validity date of the lost key) the keys of level l are secure again.
- Moreover, if some key of level $l \neq \mathsf{Max}$ is lost and a TRD receives a command blacklisting the level l, then all the keys on the TRD are secure again.

Note that our analysis will not deal with denial of service attacks.

4. FORMAL MODEL

We study the security offered by our model in a symbolic model, where messages are represented by terms, following a now standard approach used for many protocols (see e.g. [1, 2, 11]).

4.1 Syntax

We assume a finite set of names $\mathcal{A}$ and two infinite sets $\mathcal{N}$ and $\mathcal{K}$ respectively for nonces and keys. We also assume a finite set $\mathcal{H}$ representing handles and an infinite set $\mathcal{M}$ representing the miscellaneous fields, with $\epsilon \in \mathcal{M}$ representing the empty element. We recall that $\mathcal{L}$ denotes the set of levels. Messages are represented using a term algebra Terms defined by the following grammar:

$$T, T_1, T_2, \ldots := a \mid n \mid k \mid l \mid t \mid m \mid \{T\}_k \mid \langle T_1, T_2 \rangle$$

where $a \in \mathcal{A}$, $n \in \mathcal{N}$, $k \in \mathcal{K}$, $l \in \mathcal{L}$, $t \in \mathcal{T}$, $m \in \mathcal{M}$. The term $\{t\}_k$ represents the message t (symmetrically) encrypted by the key k while the term $\langle t_1, t_2 \rangle$ represents the concatenation (or more precisely, the pairing) of the two messages t_1 and t_2. For simplicity, we will often write $t_1, t_2, \ldots, t_n$ instead of $\langle t_1, \langle t_2, \ldots \langle t_{n-1}, t_n \rangle \ldots \rangle$. The notion of *subterm* is defined as usual: t' is a subterm of t if t' occurs at some position p in t, that is $t|_p = t'$. We denote by $\mathrm{St}(m)$ the set of subterms of m and by extension $\mathrm{St}(S)$ is the set of subterms of terms in S.

A *global state* of our system is described by a tuple $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ where $\mathcal{P} \subseteq \mathcal{A}$ is the set of TRDs in the system, $t \in \mathcal{T}$ represents the current time, $\mathfrak{M} \subseteq$ Terms represents the set of messages sent so far over the network, N and K are respectively sets of currently used nonces and keys[1] by all the APIs and $\mathcal{I}$ is a function:

$$\mathcal{I} : a \mapsto (\Theta_a, H_a, \mathfrak{B}_a, t_a, N_a, K_a)$$

that represents the current local state of the TRD $a \in \mathcal{P}$. More precisely, $H_a \subseteq \mathcal{H}$ represents the finite set of handles currently used in the API a and $\mathfrak{B}_a \subseteq \mathcal{L} \times \mathcal{T}$ represents the set of blacklisted levels for which the TRD does not accept keys anymore. $N_a \subseteq N$ and $K_a \subseteq K$ are respectively the nonces and keys that have been generated or stored on the API. Θ_a represents the key table of the TRD. Formally, it is a function

$$\Theta_a : H_a \to (\mathcal{K} \cup \mathcal{N}) \times \mathcal{L} \times \mathcal{T} \times \mathcal{M}$$

Indeed, as seen in Section 3.1, each handle $h \in H_a$ points to an entry (x, l, v, m) corresponding a nonce or a key x and its attributes: the level l, its validity v, and other miscellaneous information m.

As indicated in the definition of a global state, all keys come with a level.

Definition 1. *Let $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ be a global state and let k be a key in K. The level of k is defined as follows :*

$$\mathsf{Level}(k) = \{l \mid \exists\, a \in \mathcal{P}, h \in H_a \ s.t.\ \Theta_a(h) = (k, l, v, m)\}$$

It is *a priori* possible for a key k to have several levels (i.e. $\mathsf{Level}(k)$ is not a singleton) but we will show in our proofs that this never happens for uncompromised keys. We will therefore say that a key k is of level l if $l \in \mathsf{Level}(k)$.

[1] N and K are here as artifacts of the model. Since an API gets a random key (or nonce) when it creates it, there is very little chance that it generates a key (or nonce) which has already been used. To capture this, we model a global knowledge of all nonces and keys used by all APIs to be sure that freshly generated keys (or nonces) are new.

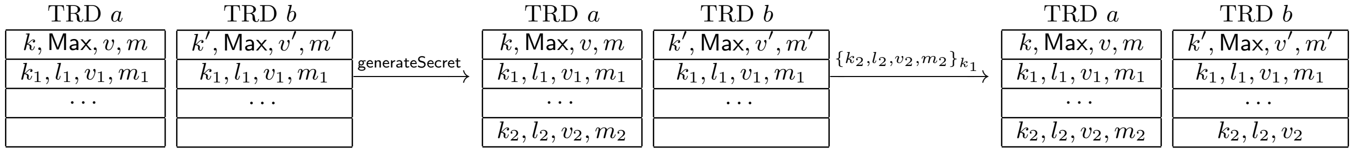

Figure 2: A simple execution. A key is created on device a and shared with device b.

$$\frac{t \in \mathfrak{M}}{\mathfrak{M} \vdash t} \qquad \frac{\mathfrak{M} \vdash x, \mathfrak{M} \vdash y}{\mathfrak{M} \vdash \langle x, y \rangle} \qquad \frac{\mathfrak{M} \vdash \langle x, y \rangle}{\mathfrak{M} \vdash x}$$

$$\frac{\mathfrak{M} \vdash \langle x, y \rangle}{\mathfrak{M} \vdash y} \qquad \frac{\mathfrak{M} \vdash x, \mathfrak{M} \vdash y}{\mathfrak{M} \vdash \{x\}_y} \qquad \frac{\mathfrak{M} \vdash \{x\}_t, \mathfrak{M} \vdash y}{\mathfrak{M} \vdash x}$$

Figure 3: Deduction Rules (Intruder)

4.2 Semantics

The behavior of our API is modeled by the transitions of our formal system. We define four kinds of transitions: transition of time, forgery by the attacker, silent transitions, and key management commands.

Transition of time.

We assume that the execution of API commands is instantaneous (this could be adapted to take into account worst case execution time). The effect of the time is modelled by a separate and explicit TIM transition:

$$\text{(TIM)} \qquad (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\text{(Time passes)}} (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t')$$

with $t' > t$.

Forgery by the attacker.

As is usually the case in formal models, the ability of the attacker to construct, or *deduce*, is modeled by a relation $\vdash$. We write $S \vdash m$ if m can be deduced from S. The deduction relation is formally defined in Figure 3 Intuitively, the attacker can deduce any term that can be obtained by pairing, encrypting, projecting, and decrypting whenever it has the encryption key.

EXAMPLE 1. *Let $S = \{\{k_1\}_{k_2}, \langle k_2, k_3 \rangle, \{k_4\}_{k_5}\}$. Then $S \vdash k_1$, $S \vdash k_2$, $S \vdash k_3$, but $S \not\vdash k_4$, $S \not\vdash k_5$. We also have $S \vdash \langle k_2, k_1 \rangle$, but $S \not\vdash \langle k_2, k_4 \rangle$.*

The attacker may send any deducible message over the network. This is reflected by the following transition.

$$\text{(DED)} \quad (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\text{(Deduction)}} (\mathcal{P}, \mathcal{I}, \mathfrak{M} \cup \{m\}, N, K, t)$$

provided $\mathfrak{M} \vdash m\}$.

Silent transitions.

Each API command executed on a TRD a with input m defines a transition $\Theta_a, H_a, \mathfrak{B}_a \to \Theta'_a, H'_a, \mathfrak{B}'_a$. Moreover, the TRD outputs some message m'. Given a global state $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ such that $m \in \mathfrak{M}$, the corresponding global transition is

$$(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \to (\mathcal{P}, \mathcal{I}', \mathfrak{M} \cup \{m'\}, N \cup N', K \cup K', t)$$

with N' and K' being respectively the sets of nonces and keys generated by a during the execution of the command. The function $\mathcal{I}'$ is defined by $\mathcal{I}' : a \mapsto (\Theta'_a, H'_a, \mathfrak{B}'_a, t_a)$ for the corresponding API a and $\mathcal{I}'(a') = \mathcal{I}(a')$ otherwise.

Key management commands.

As explained in Section 2, at least some revocation commands need to be kept hidden from the attacker. Otherwise, assuming that the attacker controls and memorizes all the traffic on the network and assuming he can break keys (which corresponds to our threat scenario), then breaking N_{Max} old keys may compromise the entire system, even if the compromised keys had not been in use for some time.

We therefore assume that the key management commands for keys of level Max are sent over a private channel. In practice, this could be achieved by several means. For example, we may assume that the key administrator has a physical access to the TRD that needs to be updated. Or we may also assume that the user would connect his/her TRD to a trusted machine, on which a secure channel (e.g. via TLS) is established with the key administrator. Note that the key management commands for keys of level Max are executed only when a key of level Max is lost (or suspected to be lost), or when keys of level Max are updated (e.g. when their validity has expired). We expect these events to occur infrequently.

Note that our assumption does not prevent an adversary from trying to run the UPM command. In particular, in case it has sufficiently many keys of level Max, it may well build a well formed Update Max command and send it to a TRD. This is reflected in the silent transitions defined above.

All the other key management commands can be sent over an insecure network. The corresponding transition system is presented in Figure 4. The fact that we assume the UPM command is sent over a secure channel is reflected in the fact that the set $\mathfrak{M}$ remains unchanged for this transition.

In the remainder of the paper we assume that managers in charge of generating key management commands behave consistently. More precisely, we assume that they only use fresh keys when updating and creating keys and that they never give different attributes to the same key value.

Key compromise.

We model the fact that the attacker may compromise a key by adding a transition lost, which allows an attacker to obtain a key of his choice.

$$\text{(LST)} \quad (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\mathsf{Lost}(k)} (\mathcal{P}, \mathcal{I}, \mathfrak{M} \cup \{k\}, N, K, t)$$

where k is a key that appears on at least one TRD, that is k occurs in the image of $\mathcal{I}$.

4.3 Example

Let us reconsider the example of Figure 2. Let E be the initial global state, we have $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$. To simplify, let us consider that $\mathcal{P} = \{a, b\}$, $K = \{k, k', k_1\}$, t and

(UPM) $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\mathsf{updateMax}} (\mathcal{P}, \mathcal{I}', \mathfrak{M}, N', K', t)$

(NEW) $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\mathsf{create}} (\mathcal{P}, \mathcal{I}', \mathfrak{M} \cup \{m\}, N', K', t)$
with $m = \{\mathsf{create}, N, N', x_1, l_1, v_1, m_1 \ldots, m_p\}_{k_1 \cdots k_n}$

(UPD) $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\mathsf{update}} (\mathcal{P}, \mathcal{I}', \mathfrak{M} \cup \{m\}, N', K', t)$
with $m' = \{\mathsf{update}, x_1, x_1', l_1', v_1', m_1' \ldots, m_p'\}_{k_1 \cdots k_n}$

(RVK) $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\mathsf{revoke}} (\mathcal{P}, \mathcal{I}', \mathfrak{M} \cup \{m\}, N, K, t)$
with $m = \{\mathsf{revoke}, F\}_{k_1 \cdots k_n}$

(BLK) $(\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t) \xrightarrow{\mathsf{blacklist}} (\mathcal{P}, \mathcal{I}', \mathfrak{M} \cup \{m\}, N, K, t)$
with $m = \{\mathsf{blacklist}, (l_1, t_1), \ldots, (l_n, t_n)\}_{k_1 \cdots k_n}$

For each transition, the function $\mathcal{I}'$ and the sets N' and K' are updated as specified by the corresponding command, as for silent transitions.

Figure 4: Key management command transitions

$\mathfrak{M}$ are arbitrary. Then $\mathcal{I} = \{a \mapsto (\Theta_a, H_a, \mathfrak{B}_a, t_a, N_a, K_a), b \mapsto (\Theta_b, H_b, \mathfrak{B}_b, t_b, N_b, K_b)\}$ where $t_a = t_b = t$, $H_a = \{h_1, h_2\}$, $H_b = \{h_1', h_2'\}$, $K_a = K_b = K$. We have $\Theta_a(h_1) = (k, \mathsf{Max}, v, m)$, $\Theta_b(h_1') = (k', \mathsf{Max}, v', m')$ and $\Theta_a(h_2) = \Theta_b(h_2') = (k_1, l_1, v_1, m_1)$.

Using the generationSecret command with TRD a will lead to a global state $E' = (\mathcal{P}, \mathcal{I}', \mathfrak{M}, N, K', t)$ where $\mathcal{I}' = \{a \mapsto (\Theta_a', H_a', \mathfrak{B}_a, t_a, N_a, K_a'), b \mapsto (\Theta_b, H_b, \mathfrak{B}_b, t_b, N_b, K_b)\}$ with $K_a' = K' = K \cup \{k_2\}$, $H_a' = H_a \cup \{h_2\}$ and $\Theta_a' = \Theta_a \cup \{h_3 \mapsto (k_2, l_2, v_2, m_2)\}$.

Then the encrypt command leads to a global state $E'' = (\mathcal{P}, \mathcal{I}, \mathfrak{M}', N, K, t)$ where $\mathfrak{M}' = \mathfrak{M} \cup \{\{k_2, l_2, v_2\}_{k_1}\}$. Finally the decrypt command is used by TRD b on the message $\{k_2, l_2, v_2\}_{k_1}$. That leads to $E''' = (\mathcal{P}, \mathcal{I}''', \mathfrak{M}, N, K, t)$ with $\mathcal{I}''' = \{a \mapsto (\Theta_a', H_a', \mathfrak{B}_a, t_a, N_a, K_a'), b \mapsto (\Theta_b''', H_b''', \mathfrak{B}_b, t_b, N_b, K_b''')\}$ with $K_b''' = K_b \cup \{k_2\}$, $H_b''' = H_b \cup \{h_3'\}$ and $\Theta_b''' = \Theta_b \cup \{h_3' \mapsto (k_2, l_2, v_2, m_2)\}$.

5. SECURITY

We are now ready to formally state the security properties discussed in section 3.6. Intuitively, we would like to show that apart from explicitly lost keys, any key that has not expired is secure, that is, cannot be deduced by the attacker. However, this is not strictly the case: several commands such as encrypt or the administrator commands produce messages of the form $\{\ldots k' \ldots\}_k$ that are sent over the network and possibly stored by the attacker. It could the case that k expires before k'. Clearly, if k becomes known to the attacker then k' will be immediately known as well, even if k has expired and k' has not.

Therefore, we distinguish three kinds of key: *valid* keys, *latent* keys, and *dead* keys. Informally, valid keys are those stored on a device whose validity time has not passed, latent keys are those which are not valid but on which security of some encryption of a valid lower level key might still depend, and dead keys are everything else.

Definition 2. *Let $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ be a global state and $k \in K$. Given $a \in \mathcal{P}$, we denote by Θ_a the key table stored on a and defined by $\mathcal{I}(a) = (\Theta_a, \ldots)$.*

- *k is a **valid** key if $\exists\, a \in \mathcal{P}$, $\exists\, h \in H_a$ s.t. $\Theta_a(h) = (k, l, v, m)$ with $v > t_a$ or if there exists $\{\ldots, k, l, v, m, \ldots\} \in St(\mathfrak{M})$ with $v > t$.*
  We denote by $\mathscr{V}_E$ the set of valid keys in E.
- *k is a **latent** key if $\exists\, a \in \mathcal{P}$, $\exists\, h \in H_a$ s.t. $\Theta_a(h) = (k, l, v, m)$ with $v + \Delta(l) > t$ or if there exists $\{\ldots, k, l, v, m, \ldots\} \in St(\mathfrak{M})$ with $v + \Delta(l) > t$.*
  We denote by $\mathscr{U}_E$ the set of latent keys in E. The notation $\mathscr{U}_E$ comes from the fact that these keys may also be viewed as "undead".
- *Otherwise, k is a **dead** key.*

A key is latent if it expired but less than $\Delta(l)$ ago. The reason why latent keys still influence the security of the whole system can be illustrated with the following scenario. Let k be a key of level l and validity time v and consider $l_1 < \cdots < l_n < l$ such that $\Delta(l) = \sum_{i=1}^{n} \delta(l_i)$. Then it may be the case that k was used to encrypt a key k_n of level l_n just before its expiration time, yielding the message $\{k_n, l_n, v_n, m_n\}_k$ with $v_n = v + \delta(l_n)$. Similarly, each key k_i may have been used to encrypt a key k_{i-1} of level l_{i-1}, yielding a message $\{k_{i-1}, l_{i-1}, v_{i-1}, m_{i-1}\}_{k_i}$ with $v_{i-1} = v_i + \delta(l_{i-1})$. Then it is easy to see that $v_1 = v + \Delta(l)$ and clearly if k is lost at anytime after its expiration date v but before $v + \Delta(l)$ then the security of k_1 (valid) would be immediately compromised as well.

5.1 Initial states

As discussed in section 3.2, before being deployed, a TRD is loaded with some level Max keys so that the administrator can then remotely equip the device with an initial set of working keys.

Definition 3. *A global state $E_\emptyset$ is said to be* originating *if $E_\emptyset = (\mathcal{P}, \mathcal{I}_0, \emptyset, 0, \emptyset, \emptyset)$ with $\mathcal{I}_0 : a \mapsto (\Theta_a, H_a, 0, \emptyset, \emptyset)$ and for all $a \in \mathcal{P}$, we have $H_a = \{h_1^a, \ldots, h_n^a\}$ with $\Theta_a(h_i^a) = (k_i^a, \mathsf{Max}, v_i^a, m_i^a)$. The k_i^a keys are revocation keys initially placed in the memory of each API and should be all distinct: $k_i^a = k_j^a$ implies $i = j$.*

5.2 Keys of level Max

The keys of level Max are crucial since they can be used to update and revoke all the keys, including keys of level Max themselves. We first show that security of these keys is preserved, assuming that not too many of the keys of level Max stored on an API are simultaneously lost.

Hypothesis 1. *Let $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ be a global state such that $E_\emptyset \to^* E$ and let $\mathscr{L}$ the set of keys that have been lost so far (i.e. the set of keys k such that the transition $\mathsf{Lost}(k)$ has been executed). We assume that $\forall a \in \mathcal{P}$, we have*

$$\#\{k \mid k \in \mathscr{U}_E \cap \mathscr{L} \cap K_a \text{ and } \mathsf{Max} \in \mathsf{Level}(k)\} \leq N_{\mathsf{Max}} - 1$$

where K_a is the set of keys associated to a by the function $\mathcal{I}$ (that is $\mathcal{I}(a) = (\ldots, K_a)$).

We assume this hypothesis to hold in all the global states we consider in the remainder of the paper. From a practical point of view, it means that the lifetime of a key of level Max should be set such that during the time when a Max key is latent, it is sufficiently unlikely that an attacker will break more than $N_{\mathsf{Max}} - 1$ such latent keys. More precisely, our

assumption is slightly weaker since the attacker may break more keys provided that no more than $N_{\mathsf{Max}}-1$ of them have occurred on the same TRD.

The design of our API ensures that no Max keys can be learned by an attacker, except for the explicitly lost ones.

THEOREM 1. *Let E be a global state such that $E_\emptyset \longrightarrow^* E$, then :*

$$\forall k \in \mathscr{V}_E^{\mathsf{Max}} \setminus \mathscr{L},\ E \not\vdash k.$$

The proof relies on the fact that keys of level Max only appear in key position in the messages sent over the network and the fact that for any Max key, $\mathsf{Level}(k)$ is a singleton. These two properties are shown to be preserved by application of the rules and rely on Hypothesis 1 that ensures that the attacker never knows sufficiently many Max keys to forge an administrator command.

5.3 Lower level keys

The case of keys of lower level is more difficult than the case of Max keys. First, as soon as a key k of level l is lost, then any key of lower level $l' < l$ is compromised as well. Indeed, for any key k' of level $l' < l$ stored on the same TRD as the key k, the attacker may use the encrypt command to get k' encrypted by k and therefore deduce k'. This attack does not only compromise the keys stored on TRDs that contain the compromised key k. Indeed, assume that the key k is stored on some TRD a_1 and does not appear in some other TRD a_2. Then as soon as a_1 and a_2 share some key k'' of level $l'' > l$, the attacker may use the encrypt command on a_1 to get k encrypted by k''. Sending this encryption to a_2 and executing the decrypt command on a_2, the attacker registers k on a_2 and can now gain any key of level lower then l.

Therefore, as soon as a key of level l is lost, any key of level $l' < l$ should be considered as lost too. We introduce the notion of keys *compatible* with a set of levels, which are latent keys that have an uncompromised level.

DEFINITION 4. *Let $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ be a global state, $\mathscr{L}$ a set of keys (typically lost keys), and $k \in K$. Let $\mathsf{L_v}$ be a set of levels (that typically corresponds to the levels of keys lost so far). Then k is* compatible with $\mathsf{L_v}$ *if $k \in \mathscr{U}_E \setminus \mathscr{L}$, $\mathsf{Max} \notin \mathsf{Level}(k)$, and there exists $l \in \mathsf{Level}(k)$ such that $l \not< \mathsf{L_v}$ (that is $l \not< l'$ for any $l' \in \mathsf{L_v}$).*

We note $\mathcal{G}^{\mathsf{L_v}}_{(E,\mathscr{L})}$ the set of keys compatible with $\mathsf{L_v}$ in the global state E. We may write $\mathcal{G}^{\mathsf{L_v}}_E$ instead of $\mathcal{G}^{\mathsf{L_v}}_{(E,\mathscr{L})}$ when $\mathscr{L}$ is clear from the context.

The first property of our API is that keys of levels not marked as lost are uncompromised. We also show that for such keys, the level and all other attributes are unique (i.e. consistent across all TRDs). We actually need to enforce a stronger property, called *robustness*.

DEFINITION 5. *A global state $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ is* robust up to *a set of levels $\mathsf{L_v}$ and a set $\mathscr{L}$ of (lost) keys if the following properties are satisfied:*

(1) *We consider that (l, v, m) are* attributes *of k if any of these three cases holds :*

   (a) *$\exists\, a \in \mathcal{P}$, $h \in H_a$ s.t. $\Theta_a(h) = (k, l, v, m)$,*

   (b) *$\exists\, q \in K$, $m' \in \mathfrak{M}$ s.t. $\mathsf{Max} \in \mathsf{Level}(q)$ and $\{\mathsf{cmd}, \dots, k, l, v, m, \dots\}_{q_1 \cdots q_j q} \in St(m')$ with $\mathsf{cmd} \neq \mathsf{blacklist}$ and $\mathsf{Max} \in \mathsf{Level}(q_i)$,*

   (c) *$\exists\, q \in K$, $m' \in \mathfrak{M}$ s.t. $\{\dots, k, l, v, m, \dots\}_q \in St(m')$.*

   If (l_1, v_1, m_1) and (l_2, v_2, m_2) are attributes of a key $k \in \mathcal{G}^{\mathsf{L_v}}_E$, then $(l_1, v_1, m_1) = (l_2, v_2, m_2)$.

(2) *$\forall k \in \mathcal{G}^{\mathsf{L_v}}_E$, $\forall m \in \mathfrak{M}$ s.t. $k \in St(m)$, then, any occurrences of k is of one of these three forms :*

   (i) *$\{m'\}_k$,*

   (ii) *$\{\dots, k, \dots\}_{k'}$ with $k' \in K$, $v_k \leq v_{k'} + \delta_k$ and $\mathsf{Level}(k) < \mathsf{Level}(k') \neq \mathsf{Max}$,*

   (iii) *$\{\dots, k, \dots\}_{q_1 \cdots q_n k'}$ with $q_1, \dots, q_n$, $k' \in K$ s.t., for $i \in [1, n]$, $\mathsf{Max} \in \mathsf{Level}(q_i)$, $\mathsf{Max} \in \mathsf{Level}(k')$, $k' \notin \mathscr{L}$ and $v_k \leq v_{k'} + \delta_k$,*

(3) *$\mathsf{L_v} \geq_s \{l \mid \exists k \in \mathscr{L} \cap \mathscr{U}_E$ s.t. $l \in \mathsf{Level}(k)\})$ where $S_1 \geq_s S_2$ if for any $l_1 \in S_1$, there exists $l_2 \in S_2$ such that $l_1 \geq l_2$.*

Robustness is meant to precisely control how compatible keys occur in a global state E. It ensures two main properties. First, compatible keys have a unique attribute in the system, that is, levels and validity times of compatible keys are consistently propagated through the system. Moreover, robustness ensures that compatible keys occur only in key position, except if they are encrypted by a key of greater level or if they occur in an administrator command (thus encrypted by a key of level Max). The third item is a technical condition that ensures that levels corresponding to lost (and not yet dead) keys should always be considered as compromised.

Whenever a state is robust, all its compatible keys remain secret from the attacker.

PROPOSITION 1. *Let $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ be a global state, $\mathscr{L}$ a set of keys (typically lost keys), and $k \in K$. Let $\mathsf{L_v}$ be a set of levels. Assume that E is robust up to $\mathsf{L_v}$ and $\mathscr{L}$ then:*

$$\forall k \in \mathcal{G}^{\mathsf{L_v}}_{(E,\mathscr{L})} \quad \mathfrak{M} \not\vdash k.$$

Our three main results are:

- A global state is always robust up to the levels of lost keys (Theorem 2). This means that losing a key does not compromise any other key apart from those at lower levels than the lost key, i.e. those which are trivially compromised.
- Once a lost key is dead, the corresponding level is protected again (Theorem 3), meaning that even when a key is lost and no special action is taken, the system repairs itself after a certain time.
- When a TRD receives a blacklist command, the corresponding levels are immediately protected again on this TRD (Theorem 4).

THEOREM 2. *Let E be a global state such that $E_\emptyset \longrightarrow^* E$ and let $\mathscr{L}$ be the set of keys lost along the execution $E_\emptyset \longrightarrow^* E$. Then E is robust up to $\mathsf{L_v} = \{l \mid \exists\, \mathsf{Lost}(k) \in \mathscr{L}$ and $\mathsf{Level}(k) = l\}$ and $\mathscr{L}$.*

Theorem 2 is the key theorem of our paper and its proof consists in proving that robustness is invariant under the rules of our API. Theorem 1 ensures that it is impossible for an attacker to break more than $N_{\mathsf{Max}} - 1$ keys of level Max. This, in turn, ensures that all accepted key management commands come from the administrator. Then it remains to show that the robustness property is invariant under application of the other rules of the API. For the deduction rule, the first step is to show that compatible (latent) keys are all non-deducible. For decryption, the robustness property in the global state E ensures that encrypted messages have the right form. Therefore when decrypting a message, the newly stored keys also satisfy the invariant. In case the attacker forges a message, the checks on the levels and the validity times ensure that the newly stored keys are of an incompatible level and therefore their security does not need to be assessed. The arguments for encryption and the other commands are along the same lines.

Theorem 3. *Let E, E' be two global states such that $E_\emptyset \longrightarrow^* E \longrightarrow^* E'$. Assume that E is robust up to $\mathsf{L_v}$ and $\mathscr{L}$ and let $l \in \mathsf{L_v}$ be a compromised level. Let $\mathscr{L}'$ be the set of keys lost along the execution $E \longrightarrow^* E'$. We further assume that there has been no 'lost' event for keys of level greater than l, that is, there is no $k \in \mathscr{L}'$ such that $\mathsf{Level}(k) \geq l$. Let $v = \mathsf{sup}(\{v_k \mid k \in \mathscr{L} \text{ and } \mathsf{Level}(k) = l\})$, where v_k is the validity time associated to k (Theorem 2 ensures v_k is unique). Then:*

$$t' \geq v + \Delta(l) \Longrightarrow E'' \text{ is robust up to } \mathsf{L_v}'' \text{ for } \mathscr{L}'',$$

where t' is the global time of E' and E'' is such that $E' \xrightarrow{\mathsf{Lost}(A)} E''$ with $A = \{k \mid \mathsf{Level}(k) = l' \text{ with } l' < l\}$ and $\mathsf{L_v}'' = (\mathsf{L_v} \cup \bigcup_{k \in \mathscr{L}''} \mathsf{Level}(k)) \setminus \{l\}$ where $\mathscr{L}'' = \mathscr{L}' \cup A$.

Intuitively, Theorem 3 ensures that whenever a key k has been lost then once k is not latent any more (*i.e.* once k is dead), the system is now robust up to $\mathsf{L_v} \setminus \{l\}$ plus the levels $\{l' \mid l' < l\}$ which can still be compromised and the newly lost keys $\mathscr{L}'$, that is, the keys that might have been lost during the execution between E and E'. Theorem 3 is actually a consequence of Theorem 2, which can be seen by carefully analysing the robustness property and noticing that once the lost key k is dead, the set of keys compatible with $\mathsf{L_v}''$ and $\mathsf{L_v}$ coincide (if the keys of lower levels are explicitly lost).

We now state the security gained with the blacklist command. This command only fixes the system locally. We therefore need to define robustness locally to a TRD.

Definition 6. *Let E be a global state, $\mathscr{L}$ be a set of lost keys, and $\mathsf{L_v}$ be a set of levels. Let $a \in \mathcal{P}$ be a TRD. Then a is* locally robust up to *$\mathsf{L_v}$ and $\mathscr{L}$ in E if $E_a = (\{a\}, \mathcal{I}_a, \mathfrak{M}, N, K, t)$ is robust up to $\mathsf{L_v}$ where $\mathcal{I}_a$ is the restriction of $\mathcal{I}$ on $\{a\}$ and where the conditions of robustness are considered for keys that are stored on a (that is, appear in the image of Θ_a).*

Proposition 2 is the analog of Proposition 1 for locally robustness and states that compatible keys on locally robust TRDs are secure.

Proposition 2. *Let $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ be a global state, $\mathscr{L}$ a set of keys (typically lost keys), and $k \in K$. Let $\mathsf{L_v}$ be a set of levels. Assume that a is locally robust up to $\mathsf{L_v}$ and $\mathscr{L}$ in E then:*

$$\forall k \in \mathcal{G}^{\mathsf{L_v}}_{(E,\mathscr{L})} \cap \{k' \mid \exists h \text{ s.t. } \Theta_a(h) = (k', l, v, m)\} \quad \mathfrak{M} \not\vdash k.$$

Finally, we can state our final theorem that ensures that blacklisting a level protects the corresponding keys on a TRD.

Theorem 4. *Let $E = (\mathcal{P}, \mathcal{I}, \mathfrak{M}, N, K, t)$ a be a global state s.t. $E_\emptyset \longrightarrow^* E$. Let $\mathscr{L}$ be a set of (lost) keys. Assume that $a \in \mathcal{P}$ is locally robust up to $\mathsf{L_v}$ and $\mathscr{L}$ in E. Assume that $E \xrightarrow{L}\longrightarrow^* E'$ with L a blacklist command for TRD a, level l, and validity time t_h (with t_h greater or equal to $v_h = \mathsf{sup}(\{v_k \mid k \in \mathscr{L} \text{ and } l \in \mathsf{Level}(k)\})$. Let $\mathscr{L}'$ be the set of keys lost along the execution $E \xrightarrow{L}\longrightarrow^* E'$. We further assume that there has been no lost event for keys of level greater than l, that is, there is no $k \in \mathscr{L}'$ such that $\mathsf{Level}(k) \geq l$.*

Then a is locally robust up to $\mathsf{L_v}'' = (\mathsf{L_v} \cup \bigcup_{k \in \mathscr{L}''} \mathsf{Level}(k)) \setminus \{l\}$ in E'' where E'' is such that $E' \xrightarrow{\mathsf{Lost}(A)} E''$ with $A = \{k \mid \mathsf{Level}(k) = l' \text{ with } l' < l\}$ and $\mathscr{L}'' = \mathscr{L}' \cup A$.

Similarly to Theorem 3, Theorem 4 ensures that once a level is blacklisted, the corresponding level is secure again. More precisely, it states that the system is then robust up to $\mathsf{L_v} \setminus \{l\}$ plus the levels $\{l' \mid l' < l\}$ which can still be compromised and the newly lost keys $\mathscr{L}'$, that is, the keys that might have been lost during the execution between E and E'. The proof follows easily from Theorem 3.

5.4 Application

Suppose we are using our API to implement a protocol similar to Kerberos [13] for granting access to resources (see Figure 5). We consider key distribution centre (KDC), a ticket granting server (TGS) and principals Alice (a) and Bob (b).

We will order the levels using an inclusion ordering, where K_a has level (a), $K_{a,TGS}$ has level (a, TGS, KDC) and $K_{a,b}$ has level (a, b, TGS, KDC) (i.e. the levels correspond to the parties assumed to have access to the key). Other terms are considered public (level 0). For two levels l, l', $l > l' \Leftrightarrow l \subset l'$. The protocol is now implementable using the commands of the API.

In a deployment of the system, we assume that devices for the users a, b,... and for the servers TGS, KDC are loaded with appropriate keys of level Max as part of some personalisation process. The administrator can then load long term keys $K_a, K_{a,TGS}$ etc. on to the user devices and the servers. They can now play out the protocol using just the working key commands encrypt, decrypt, secretGenerate, publicGenerate. Note that when implemented by our API the protocol messages will be tagged following our scheme, e.g. message 2 will appear as $\{tgt, 0, v', \mathsf{TGT}\}_{K_{TGS}}$, $\{K_{a,TGS}, (a, TGS, KDC), v, \mathsf{K_{A,TGS}}\}_{K_a}$ where TGT and $\mathsf{K_{A,TGS}}$ are elements in the miscellaneous field and v, v' are some validity times.

Suppose key $K_{a,TGS}$ is compromised. Then the administrator has two choices. One is to do nothing. Then he knows that after time $v + \Delta((a, TGS, KDC))$, where v is the expiry time of $K_{a,TGS}$, the system returns to a state where keys at level (a, TGS, KDC) are secure. Lower level keys like session keys $K_{a,b}$ might still be compromised, but

1.	a	$\rightarrow$	KDC	:	$request, KDC$
2.	KDC	$\rightarrow$	a	:	$\{tgt\}_{K_{TGS}}, \{K_{a,TGS}\}_{K_a}$
3.	a	$\rightarrow$	TGS	:	$a,b, \{tgt\}_{K_{TGS}}, \{Auth1\}_{K_{a,TGS}}$
4.	TGS	$\rightarrow$	a	:	$\{K_{a,b}\}_{K_b}, \{K_{a,b}\}_{K_{a,TGS}}$
5.	a	$\rightarrow$	b	:	$b, \{K_{a,b}\}_{K_b}, \{Auth2\}_{K_{a,b}}$

Figure 5: A Kerberos-like protocol

new session keys will be safe. Alternatively he can signal to TGS and a to blacklist the key. In this case this level (a, TGS, KDC) becomes unusable. However, to preserve functionality, and administrator can add extra tags t_i to all levels, so level (a) becomes (a, t_1), level (a, TGS, KDC) becomes (a, TGS, KDC, t_1, t_2), and the session keys $K_{a,b}$ will have level $(a, b, TGS, KDC, t_1, t_2, t_3)$. If $K_{a,TGS}$ is compromised the administrator only needs to issue a new key with a new level $a, b, TGS, KDC, t_1, t_2'$ and the device can continue to issue tickets securely while level (a, TGS, KDC, t_1, t_2) is blacklisted.

6. CONCLUSIONS

We have presented our API for key management with revocation and proved its security properties. While the underlying assumptions for our system to be secure are not unrealistic, in future work we intend to weaken them further. To relax the assumption on level Max messages remaining confidential seems to require more cryptographic primitives. In particular, in order to ensure that the loss of long term keys does not lead to the loss of update messages we require perfect forward secrecy, which would seem to imply the use of asymmetric cryptography. Extending our API to asymmetric cryptography is ongoing work. We also assume here that each device has a real time clock. This could be weakened by adding some sort of nonce based freshness test for each new key accepted, as required in the restricted version of the API on which our revocation scheme is based [4]. All our security proofs are made in the symbolic model, but it should be possible to extend them to the standard model of cryptography under the usual assumptions, i.e. IND-CCA2 for the authenticated encryption scheme, since we avoid problematic constructions such as key cycles.

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 258865, project ProSecure, and the Direction Générale de l'Armement under contact no 11810242, Secure Interfaces.

7. REFERENCES

[1] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*, 10(3):1–59, July 2007.

[2] B. Blanchet and A. Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P'08)*, pages 417–431, Oakland, CA, 2008. IEEE.

[3] C. Cachin and N. Chandran. A secure cryptographic token interface. In *Computer Security Foundations (CSF-22)*, pages 141–153, Long Island, New York, 2009. IEEE Computer Society Press.

[4] V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *Lecture Notes in Computer Science*, pages 605–620, Saint Malo, France, September 2009. Springer.

[5] V. Cortier, G. Steel, and C. Wiedling. Revoke and let live: A secure key revocation api for cryptographic devices. Research Report RR-7949, INRIA, 2012.

[6] J. Courant and J.-F. Monin. Defending the bank with a proof assistant. In *Proceedings of the 6th International Workshop on Issues in the Theory of Security (WITS'06)*, pages 87 – 98, Vienna, Austria, March 2006.

[7] L. Eschenauer and V. D. Gligor. A key management scheme for distributed sensor networks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 41–47, 2002.

[8] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, volume 5511 of *Lecture Notes in Computer Science*, pages 92–106, York, UK, 2009. Springer.

[9] F. E. Kargl. Sevecom baseline architecture. Deliverable D2.1-App.A for EU Project Sevecom, 2009.

[10] F. Levy. SAM and key management functional presentation. Available from `http://www.calypsostandard.net/`, December 2010.

[11] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, march 1996.

[12] S. Mödersheim and P. Modesti. Verifying sevecom using set-based abstraction. In *IWCMC*, pages 1164–1169. IEEE, 2011.

[13] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The kerberos network authentication service - rfc 4120. Available at `http://tools.ietf.org/html/rfc4120`.

[14] G. Steel and A. Bundy. Attacking group protocols by refuting incorrect inductive conjectures. *Journal of Automated Reasoning*, 46(1–2):149–176, January 2006. Special Issue on Automated Reasoning for Security Protocol Analysis.

[15] Trusted Computing Group. *TPM Main Part 3 Commands.*, level 2 revision 116 edition, March 2011. Specification Version 1.2.

[16] B. Weyl. Secure on-board architecture specification. Deliverable D3.2 for EU Project EVITA, `http://evita-project.org/Deliverables/EVITAD3.2.pdf`, August 2011.

[17] X. Z. Yong Wan, Byrav Ramamurthy. Keyrev: An effifcient key revocation scheme for wireless sensor networks. In *IEEE International Conference on Communications (ICC)*, pages 1260 – 1265, 2007.