A Secure Key Management Interface with Asymmetric Cryptography

Marion Daubignard[1,2], David Lubicz[2], and Graham Steel[3]

[1] ANSSI, 51 Bvd de La Tour-Maubourg, 75007 Paris, France
[2] DGA.MI, BP 57419, 35174 Bruz Cedex, France
[3] INRIA Team Prosecco, 23 Avenue d'Italie, 75013 Paris, France

Abstract. Cryptographic devices such as Hardware Security Modules are only as secure as their application programming interfaces (APIs) that offer cryptographic functionality to the outside world. Design flaws and implementation errors in security APIs have been shown to cause vulnerabilities that may leak secrets such as keys and PINs. Ideally, we would like to design such interfaces in such a way that we can formally prove security properties, even in the presence of some corrupted keys. In this work, we propose the first such provably secure interface to support asymmetric key operations for key management: Cachin and Chandran's secure token interface supports asymmetric key operations only for encrypting and signing data, but not for managing keys, while Cortier and Steel handle only symmetric keys. Due to the fact that anyone can encrypt under a public key, in order to secure integrity of the keys under management, we must consider confidentiality and integrity properties separately and provide support for classical operations of public key infrastructure (e.g. certification of public keys).

1 Introduction

In a context of constant security threats combined with increasing heterogeneity of platforms and applications, developers are turning more and more to solutions based on secure hardware, whether it be a smartcard, Trusted Platform Module (TPM), or Hardware Security Module (HSM). In a typical architecture, the secure hardware contains cryptographic keys and the ability to perform some basic crypto operations which can be leveraged to ensure security for the whole system. However, designing the application programming interface (API) of such a device is difficult: it must allow the user to manage the keys on the device and access the crypto while preventing an attacker, who may in the worst case be able to make arbitrary calls to the API, from obtaining secrets. Many attacks have been found on the APIs of contemporary devices [2,3,5]. One promising approach to solving this problem is to design APIs such that one can formally prove security properties in the presence of a suitably powerful intruder. Such an approach has been applied both in the standard cryptographic model [4] and the symbolic or Dolev-Yao model [7]. However, neither of these designs present a

scheme for managing keys using asymmetric cryptography, which is widely used in practice for the task since it provides a convenient way to bootstrap security without any pre-shared secrets. The contribution of this paper is to present the design for such an API with security proofs in the symbolic model. For the symmetric key part of the API, we adapt slightly the API designed by Cortier and Steel [7]. For the asymmetric key part, since anyone can encrypt under a public key, we have to add an explicit mechanism for assuring the integrity of keys to be imported to prevent so-called "Trojan key" attacks [6]. We add signature keys for signing encryption under public keys and also separate certification keys, the latter used to manage the public key infrastructure (PKI) of keys and certificates. We show how to adapt the security labels given to keys by Cortier and Steel to this new scenario, with separate labels for confidentiality of the private key and integrity of the corresponding public key. This allows us to account for corruption in our proof. As far as we are aware, this is the first such design to be proposed with security proofs.

In the rest of the paper, we first introduce our symbolic model and explain the features of our API design in Section 2. We describe the API rules formally in Section 3, and then give the security properties and sketch proofs in Section 4. We describe some experiments implementing protocols with the API in Section 5 and draw conclusions in Section 6. Full proofs are given in a technical report [9].

Related Work Cortier and Steel (CS) [7] proposed an API that supports only symmetric key cryptography, but can nonetheless be used to implement any secure symmetric key exchange protocol from the Clark-Jacob corpus. The main principle is that keys are arranged in a hierarchy of levels. Each key is associated to its level and the set of agents who are allowed to use it. This association is made when storing the key on the device, by including it as metadata stored with the key, and when encrypting the key for transfer, by tagging the encrypted key with exactly this information. The API rules are designed such that keys may only be encrypted by other keys which are higher in the hierarchy, i.e. they are at least one level higher and assigned to a set of agents that is equal to or smaller than the payload key. We generalise this notion slightly in our API. The CS API includes a notion of freshness for imported keys enforced by nonces. It has also recently been extended to accommodate key revocation [8]. Although we do not include these mechanisms in our API, we do not foresee any obstacle to these generalizations if needed.

Cachin and Chandran proposed an API with a quite different design [4]. They rely on the fact that all keys are stored on a central key server. Instead of assigning security attributes such as levels and agent identifiers to keys at creation time, they allow the key's role to evolve over time by logging all operations, and then disallowing operations that would be insecure by observing the log. They allow asymmetric keys to be managed by symmetric key cryptography, but do not allow asymmetric keys to be used for key management operations like export and import.

Other work has investigated the foundations of models for secure key management APIs: Kremer, Steel and Warinschi give a model that can be interpreted

in the symbolic and computational cryptography worlds [12]. They show that the possibility of key corruption requires strong assumptions to be made on the key wrapping primitives in the computational model. Recent work by Künnemann, Kremer and Steel investigates composable notions of security for key management [11]. This is an appealing idea because it allows (almost) arbitrary secure cryptographic primitives to be used with the keys under management without having to repeat the security proofs, but currently only management with symmetric keys is supported.

2 Design of the API

We present the design of our API in an abstract 'Dolev-Yao' style symbolic model. We first describe the roles assigned to keys in our API. We then give the syntax and informal semantics for the message algebra and introduce our notion of *key handles* which extends previous designs.

2.1 Key Types

In order to limit the number of key roles in the API we consider that the asymmetric keys are double keys, with one part for encryption/decryption, and one for signature/verification. This means that the same key can be used as an input of both an encryption and a signature scheme. Thus, we have encryption/verification public keys and decryption/signature private keys. It is clear that in practice a double key can simply be obtained by the concatenation of a signature and encryption key and that a simple key can be simulated by a double key. Thus, we do not lose generality with this simplification. Moreover, it makes sense from a security point of view since the encrypt and sign operation is the minimal basic operation which ensures the confidentiality of message and an authentication of the issuer, which is mandatory for the set up of our security policy. Signature keys are used to sign encryptions of other keys or messages. Asymmetric public keys are certified by certification keys (with a signature algorithm). The list of key roles that we are going to manipulate is: symmetric encryption/decryption keys, encryption/verification of signature double public keys, decryption/signature double private keys, verification of certificates public keys, certification private keys.

It is possible that the algorithm used to sign the certificates is the same as the one used to sign the encrypted messages. Nonetheless, it is important to distinguish the key roles to prevent a signature algorithm from being used as a certification oracle by an adversary. The different key roles and their associated types are summarised in the table 1. $\mathcal{T}$ denotes the set of key types.

2.2 Security Levels

The set of key security levels I is a finite set together with a partial strict order relation denoted $<$. We suppose that there is a minimal element in I denoted by

Key	Role	Type
Priv Double	decryption/signature private key	privDecSign
Pub Double	encryption/verification of signature public key	pubEncVerif
Sym	symmetric encryption key	symEncDec
Pub Certif	certificate verification key	pubCertVerif
Priv Certif	certificate signature key	privCertSign

Fig. 1. Table of the set of key roles and types ($\mathcal{T}$)

0. By definition, for all $x \in I \setminus 0$, we have $0 < x$. The 0 element represents the security level of public information. We are given a partition of I in two subsets:

- the levels $I_1 \subset I$ which correspond to the keys which can only deal with regular messages;
- the levels $I_2 \subset I$ which correspond to the keys which can be used to transport keys of level I_1.

Note that for $x \in I_1$ and $y \in I_2$, if x and y are comparable with the relation $<$ then we have necessarily $x < y$. We set $I_{>0} = I_1 \sqcup I_2 = I - \{0\}$ (where $\sqcup$ denotes a disjoint union).

2.3 Message Algebra

Messages are represented by a term algebra. We suppose a given set of agents Agent, a set of nonces Nonce and a set of keys Key. We are also given a set of variables Var in which we distinguish a set of key variables VarKey and a set of nonce variables $\mathsf{VarNonce}$. All these sets are countably infinite. The term algebra is given by:

$$\begin{aligned}
\mathsf{Keyv} &::= \mathsf{Key} \mid \mathsf{VarKey} \mid inv(\mathsf{Keyv}) \\
\mathsf{Noncev} &::= \mathsf{Nonce} \mid \mathsf{VarNonce} \\
\mathsf{Msg} &::= \mathsf{Agent} \mid \mathsf{Keyv} \mid \mathsf{Noncev} \mid I \mid \mathcal{T} \mid \{\!|\mathsf{Msg}|\!\}_{\mathsf{Keyv}} \mid \{\mathsf{Msg}\}_{\mathsf{Keyv}} \\
&\quad \mid \Sigma(\mathsf{Msg}, \mathsf{Keyv}) \mid \mathsf{nhdl}(\mathsf{Msg}) \mid < \mathsf{Msg}, \mathsf{Msg} > \\
\mathsf{Handle} &::= h^{\alpha}_{\mathsf{Agent}}(\mathsf{Noncev}, \mathsf{Noncev}, \mathsf{Msg}, \mathcal{T}, I, \mathscr{S}, \mathscr{S}) \mid h_{\mathsf{Agent}}(\mathsf{Noncev}, \mathsf{Msg})
\end{aligned}$$

where $\mathscr{S}$ is the set of subsets of Agent.

The set Keyv represents the set of keys and variable of keys. A term of the form $inv(k)$ with $k \in \mathsf{Key}$ represents the private key associated to the public key k. The set Noncev is the set of nonces and variable of nonces. The terms of type Msg are made of elements of Agent, Keyv, Noncev together with constructors representing encryption, signature together with sets needed to represent the attributes of the handles. More precisely,

- the term $\{\!|m|\!\}_k$ represents the symmetric encryption of the message m with the key k;
- the term $\{m\}_k$ represents the asymmetric encryption of the message m with the double key k;

– the term $\Sigma(m, k)$ represents the signature of the message m with the double key k;
– the term $\mathsf{nhdl}()$ allows one to encapsulate a regular message which does not correspond to the transportation of a handle (see below);
– the term $< m_1, m_2 >$ represents the pair of the two messages $m_1, m_2 \in \mathsf{Msg}$.

For $n > 0$, $< m_1, < m_2, < \ldots, m_n >>>$ is shortened as $m_1, \ldots, m_n$.

2.4 Handles

The purpose of a key management API is to give access to cryptographic functionalities without giving direct access to sensitive keys stored on the device. Instead, an agent can manipulate the data by calling the API commands and referring to the keys by identifiers called *handles*, of which we define two types in our API:

– *key handles* used to protect integrity and confidentiality of the data on the device. They are typically used for keys and secret nonces.
– *integrity handles* used to protect the integrity of data on the device. They are typically used for certificates that have been verified.

Identifiers are meant to be a public way of referring to keys without revealing their values. Thus, knowing an identifier does not mean knowing the cryptographic value of a key. Then, to represent that an agent *can use a key*, we write that she *owns a handle referring to that key*. Intuitively, it means that there is a part of the memory of a secure device that the agent can make use of, and that contains such a data structure. In our framework, much as in that of [7], there is no mapping between memory owned by agents and secure devices; this is totally abstracted away and translates only in the ownership of handles. As a result, if two agents a and b share a key to communicate with one another, they each own a handle referring to this key, but nothing in our model represents whether they use different physical devices. Neither do we capture that one agent has all its handles on the same physical device. It might be the case that an agent has handles spread over multiple devices. Even then, our abstraction is sound from a security point of view in the sense that we consider operations that may not be functionally possible, but do not overlook any feasible call.

Let us now formally describe the handles that we use. Key handles are terms of the form $h_a^{\alpha}(N_1, N_2, m, T, i, S_1, S_2)$, with:

– the agent $a \in \mathsf{Agent}$ who owns the handle;
– the identifier $N_1 \in \mathsf{Nonce}$ (unique in the whole system) of the handle;
– if m is a double private key, then N_2 is the identifier of the associated certificate of the double public key, else $N_2 = \mathsf{Null}$;
– the message $m \in \mathsf{Msg}$ (usually m is a key or a nonce) associated to the handle;
– the type $T \in \mathcal{T}$ of the message (see table 1 for a list of possible types);

– the triple $(i, S_1, S_2) \in I \times \mathscr{S} \times \mathscr{S}$ is the security level of the handle (the security policy of the API is based on this structure); the first element i gives the role of the key (data encryption or key transport) while the second (respectively third) element gives a set of agents who must be uncorrupted for this key's confidentiality (respectively integrity) to hold - this is explained in details below.
– the label $\alpha \in \{r, g\}$ allows to distinguish the keys which have been generated by a ($\alpha = g$) from the keys which have been received and imported ($\alpha = r$).

Integrity handles are terms of the form $h_a(N_1, m)$ with an identifier N_1 and a message $m \in \mathsf{Msg}$. They are meant to model the preservation of the integrity of data by a signature : given as input a valid signature of a message m, the API produces an integrity handle containing the message m. Public key certificates usually refer to some signed public information. We are more precise than this and distinguish two elements, the pre-certificate and the certificate which is a signed pre-certificate. Indeed, the outcome of the certificate verification operation is a new pre-certificate stored under an integrity handle in the device.

In the following, for clarity, we use the notation $\mathcal{C}(N_1, N_2, N_3, k, T, i, S_1, S_2)$, which is a synonym of the concatenation of the terms $N_1, N_2, N_3, k, T, i, S_1, S_2 \in \mathsf{Msg}$, to represent a pre-certificate of double public key. We emphasize that the notation $\mathcal{C}(N_1, N_2, N_3, k, T, i, S_1, S_2)$ does not imply requirements on the type of the fields. Nonetheless, we say that a pre-certificate is *well-formed* if its fields correspond to the following terms and types (we also give their semantics):

– the identifier $N_1 \in \mathsf{Nonce}$ of the certificate;
– the identifier $N_2 \in \mathsf{Nonce}$ of the associated private key;
– the identifier $N_3 \in \mathsf{Nonce}$ of the certification public key which allows to verify the certificate;
– a double public key $k \in \mathsf{Key}$;
– the type $T \in \mathcal{T}$ of k ;
– the associated private key handle security level $(i, S_1, S_2) \in I \times \mathscr{S} \times \mathscr{S}$.

Thus, matching asymmetric double keys stored in a physical device are typically formalized as:

– a key handle $h_{\cdot}^{\alpha}(N_1, N_2, k, \text{privDecSign}, i, S_1, S_2)$ for the secret part,
– an integrity handle $h_{\cdot}(N_2, \mathcal{C}(N_2, N_1, N_3, k, \text{pubEncVerif}, i, S_1, S_2))$ for the certificate of the public part.

We remark that we choose to trace the association of public and private part of asymmetric key pairs via their identifiers. This requires us to have system-wide identifiers for handles, in the sense that identifiers are independent of the secure hardware they are stored in. As a result, when importing a pre-certificate, the identifier cannot be generated at random. This explains why the pre-certificate contains a field corresponding to its identifier.

2.5 API Rules

The model that we present is a transition system inspired by [7]. It represents the evolution of the knowledge of the adversary and agents with the API calls. We use a set of knowledge predicates $\mathcal{P} = \{P_a | a \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$, where int is a particular element representing the attacker. For term t, $P_a(t)$ means that a knows term t.

The system is formalized as a set of rules of the general form:

$$P_{b_1}(u_1), \ldots, P_{b_k}(u_k) \overset{N_1,\ldots,N_m}{\Longrightarrow} P_{b_{k+1}}(u_{k+1}), \ldots, P_{b_l}(u_l),$$

where u_i are terms, N_i are variables, $b_i \in \mathsf{Agent}$ for $i = 1, \ldots, l$ and the P_{b_i} are predicates. The rules define how to derive knowledge predicates. They are instantiated by substituting the variables by terms of the same type. In order to explain that, let $x_1, \ldots, x_n$ be elements of Var and let $t_1, \ldots, t_n$ be a set of terms. We denote by $\{x_1 \to t_1, \ldots, x_n \to t_n\}$ the substitution σ which replaces the variables x_i by the terms t_i for $i = 1, \ldots, n$. We say that σ is well-typed if the variables x_i and the terms t_i have the same types. In the sequel, we only consider well-typed substitutions. The application of the substitution σ on the term t is denoted by $t\sigma$. Classically, given a set of rules, we say that a state $\mathcal{S}'$ is reachable from a state $\mathcal{S}$ if there exists an instantiated rule in the set allowing to transition from $\mathcal{S}$ to $\mathcal{S}'$. The state therefore represents the set of terms known to each agent (including the intruder) at a moment in time. We then generalize this reachability definition to the transitive closure of a set of rules, which we denote $\Rightarrow^*$.

We let $\mathcal{S}_b$ be the part of a state indexed by b. Then, the state of the system is given by the family $\{\mathcal{S}_b | b \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$. The notations $P_b(t)$ and $t \in \mathcal{S}_b$ are equivalent. In the sequel, we provide two kinds of rules. Firstly, API rules only deal with knowledge of a given agent. As a result, such a rule has the form : $P_a(u_1), \ldots, P_a(u_k) \overset{N_1,\ldots,N_m}{\Longrightarrow} P_a(u_{k+1}), \ldots, P_a(u_l)$ for some agent a different from the intruder int. It models that the inputs provided by an agent to an API call, i.e. $u_1, \ldots, u_k$, result in the output of new terms $u_{k+1}, \ldots, u_l$, which are added to the agent's knowledge. Secondly, other rules involve the adversary : they are rules with at least one predicate $P_{\mathsf{int}}(.)$.

2.6 Adversarial model

As is usual in Dolev-Yao style models, the adversary is assumed to have complete control of the network. We further assume that the host machines (such as a desktop computer) in which the secure device might be embedded is also under the adversary's control. Therefore the interface between our trusted platform and the attacker controlled network is just our API. It can be argued as over-pessimistic, but it is sound from a security point of view to rely only on the trust we place in the tamper-resistant devices. This modeling choice results in rules translating direct transfers from the agent knowledge to the adversary knowledge and vice versa. A consequence is that the intruder can execute any command he

likes on any device and use the result (or part of it) to form a command call to any other device.

On top of network control, we empower the adversary with the ability to statically corrupt agents. Formally, the set of agents is partitioned once and for all into honest and dishonest agents, and every key referred to by a handle owned at some point by a dishonest agent is leaked to the adversary. This models that some keys stored on secure hardware might be lost, perhaps due to side channel attacks or other abstracted events.

These choices are illustrated in Figure 3, in which the perimeter of control of the attacker encompasses all the knowledge of honest agents, the network, and dishonest agent devices. To simplify Figure 3, we have represented one agent per device, which need not be the case in practice. However, all keys of a dishonest agent are indifferently leaked to the adversary. This quite strong corruption model could be relaxed to a key-by-key corruption model.

Our corruption model defines an order relation on the set of keys. To a key k we can associate the set S_k of devices, the corruption of which implies that of a key. A key k_1 is more secure than k_2 if $S_{k_1} \subseteq S_{k_2}$. In other words, a key that relies on the integrity of just a few agents is considered more secure than one that depends on the integrity of a large number of agents.

With such adversary capabilities, we stress that the only elements on which we can state security results are those stored in the secure devices which we have formalized. Indeed, these devices are our only source of trust and the whole point of this security API is to protect the elements stored in these secure areas from unwanted interaction with an adversary. Concretely, this means that the elements on which we can prove security results are elements under handles, and only them, i.e. key or certificate values. Of course, regular data *can* be encrypted and decrypted using our API, but it is never hosted on a secure device : no handle is created to refer to them. In our framework, regular data is thus modeled in the form of messages coming unfettered from the network, on which we do not aim to provide security guarantees. This choice to 'only' protect keys makes total sense. Firstly, there is only a limited amount of space in tamper-resistant devices so that priorities have to be attributed. Secondly, if keys are suitably protected, then so is the data that they in turn protect, because there usually *are* intermediate workstations in which data is treated and hosted.

3 Symbolic Security of the API

3.1 Security ordering

In the rules of our API, we put to use an order relation on the set of triples $(i, S_1, S_2) \in I \times \mathscr{S} \times \mathscr{S}$ (recall that $\mathscr{S}$ is the set of subsets of Agent). Let $(i_1, S_{1,1}, S_{1,2})$ and $(i_2, S_{2,1}, S_{2,2})$ be two elements of $I \times \mathscr{S} \times \mathscr{S}$, we write

$$(i_1, S_{1,1}, S_{1,2}) \prec (i_2, S_{2,1}, S_{2,2}) \text{ if } i_1 < i_2,\ S_{2,1} \subseteq S_{1,1} \text{ and } S_{2,2} \subseteq S_{1,2},$$
$$(i_1, S_{1,1}, S_{1,2}) \preceq (i_2, S_{2,1}, S_{2,2}) \text{ if } i_1 \leq i_2,\ S_{2,1} \subseteq S_{1,1} \text{ and } S_{2,2} \subseteq S_{1,2}.$$

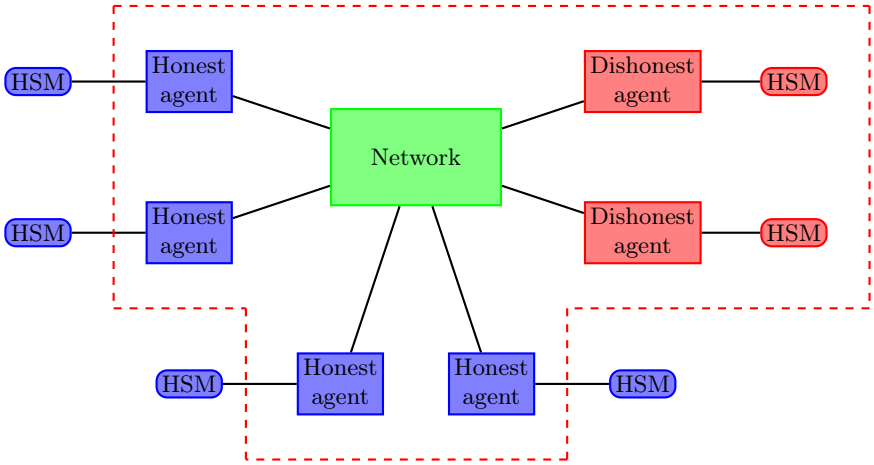

Fig. 2. Corruption model

It is clear that $\preceq$ (resp. $\prec$) is an order relation (resp. a strict order relation). This relation plays an important role in the definition of the security policy of our API and the fact that it is strict ensures that we avoid cycles of encryption, e.g. terms of the form $\{\!\{\{\!\{\ldots\}\!\}_{K_1}\}\!\}_{K_2}\}\!\}_{K_1}$.

This order relation may look complex but is in fact quite natural. The security level of a handle is given by a set of agents S such that the corruption of any member of $S = S_1 \cup S_2$ would imply the corruption of the handle. In the API, we want to guarantee that if a particular set $S = S_1 \cup S_2$ of agents are honest, then a handle cannot be corrupted. In the case of a public key API, the keys are split into a public part (the certificate), whose value is known to everyone but the integrity of which must be guaranteed, and the private part which must be protected in confidentiality and integrity. The security of a key depends on both parts, but still it is important to be able to distinguish between these two aspects of security because we want to control the diffusion of the private key, while the integrity of the public part may depend on a long chain of certification.

For asymmetric keys, it may well be the case that S_1 is a rather large set (e.g. tracing a certification chain back to a root certificate) and yet we still want S_2 to be as small as possible (possibly just the agent who generated the key). Finally, it should be remarked that a key k which is wrapped by another asymmetric key k' should inherit from k' the control sets S_1 and S_2 even if k is symmetric.

Dividing the agent sets into public key and private key parts also affects our security properties. In the Cortier-Steel API, a secret key cannot be sent to an agent $a \in \mathsf{Agent}$ outside of the control set S: indeed, it would be a violation of the security property in the case that a is a corrupt agent. In our setting, the security property guarantees the secrecy of a private key k if none of the agents of $S_1 \cup S_2$ are corrupted. We also want to ensure that no agent in $S_1 - S_2$ actually obtains the value of k, which they should not since they are not legitimate agents

of the key. Both these security requirements appear in the statement of the main result of this paper (see Theorem 1). Identifying legitimate agents constitutes another important motivation for dividing the control set into two parts.

Note that agent identifiers do not need to be known in advance. If identifiers come from a big enough space of possible values, one can always generate a new key referring to an agent identifier that has not been used before. The only restriction is that one cannot add the name of a new agent to the sets S of an existing key, for obvious security reasons.

3.2 The rules of the generic asymmetric API

We describe the transition rules defining the security API. We recall that agents are not supposed to know key values and instead use identifiers to refer to them. However, as explained in 2.4, knowledge of an identifier N differs from ownership of a data structure pointing to a key identified by N. An agent a should not be able to use the value of a key if he does not own a handle $h(N, \dots)$ referring to it, a fact we denote as $P_a(h_a(N, \dots))$. In other words, writing $P_a(h_a(N, \dots))$ on the left-hand side of an API rule formalizes two things : the fact that agent a performs the corresponding API call with input N *and* the fact that there exists a handle $h_a(N, \dots)$ owned by agent a and identified by N. Symmetrically, writing $P_a(h_a(N, \dots))$ on the right-hand side of an API rule means that a new handle is created, owned by a and with identifier N.

When an agent wants to export a key to which he owns a handle, he provides its identifier as an input to the corresponding API function, which replaces this latter by the value of the key and its attributes when computing the real payload value to encrypt. Reciprocally, the injection functions must identify these patterns and create the appropriate handle rather than output the key value as a plaintext. Thus, we emphasize that there has to exist a distinction between handle translations and regular messages, which we materialize by the message container nhdl. Respect of the security ordering is enforced by appropriate checks when encrypting and decrypting payloads.

In the following rules, $N_i \in \mathsf{Noncev}$, $X_k, inv(X_k), Y_k, inv(Y_k) \in \mathsf{Keyv}$, $S_i \subseteq \mathsf{Agent}$ and i (possibly indexed by an agent name) denotes an element in I.

Symmetric key generation This rule allows the generation of key X_k of level i and control sets containing (S_1, S_2) by the agent e for the set of agents S_2, which is modeled by the following handle creation:

$$P_e(i), P_e(S_1), P_e(S_2) \overset{N,X_k}{\Longrightarrow} P_e(h^g_e(N, \mathsf{Null}, X_k, \text{symEncDec}, i, S_1, S_2 \cup \{e\})) \qquad \textbf{(Sym Gen)}$$

Symmetric encryption This rule allows agent b to encrypt with the key X_k (to which he has a handle), a payload consisting of messages and handles $m_1, \dots, m_n$, where handles are translated into key values and attributes.

$$P_b(h^\alpha_b(N, \mathsf{Null}, X_k, \text{symEncDec}, i, S_1, S_2)), P_b(m_1), \dots, P_b(m_n)$$
$$\Longrightarrow P_b(\{\!|m'_1, \dots, m'_n|\!\}_{X_k}), \qquad \textbf{(Sym Encrypt)}$$

with $b \in S_2$, $m_j, m'_j \in \mathsf{Msg}$ and for $j = 1, \ldots, n$:

- if $m_j = h_b^{\alpha}(N_j, N'_j, X_{k,j}, T_j, i_j, S_{j,1}, S_{j,2})$ with $X_{k,j} = \mathsf{Keyv} \cup \mathsf{Noncev}$ then
  - if $i \in I_2$, $b \in \mathsf{Agent}$ and $(i_j, S_{j,1}, S_{j,2}) \prec (i, S_1, S_2)$ then we let $m'_j = N_j, N'_j, X_{k,j}, T_j, i_j, S_{j,1}, S_{j,2}$;
  - else $m'_j = \varnothing$.
- else $m'_j = \mathsf{nhdl}(m_j)$.

Symmetric decryption The following rule lets agent b, provided he knows a handle pointing to key X_k, decrypt a ciphertext. Whenever a pattern consisting of a key and attributes is identified, it results in a suitable handle creation. Otherwise, the plaintext is output.

$$P_b(h_b^{\alpha}(N, \mathsf{Null}, X_k, \mathrm{symEncDec}, i, S_1, S_2)), P_b(\{\!|m_1, \ldots, m_n|\!\}_{X_k})$$
$$\Longrightarrow P_b(m'_1), \ldots, P_b(m'_n), \qquad \textbf{(Sym Decrypt)}$$

with $b \in S_2$, $m_j, m'_j \in \mathsf{Msg}$ and moreover for $j = 1, \ldots, n$:

- if $m_j = N_j, N'_j, X_{k,j}, T_j, i_j, S_{j,1}, S_{j,2}$, then
  - if $i \in I_2$, $(i_j, S_{j,1}, S_{j,2}) \prec (i, S_1, S_2)$ then we set $m'_j = h_b^r(N_j, N'_j, X_{k,j}, T_j, i_j, S_{j,1}, S_{j,2})$;
  - else $m'_j = \varnothing$.
- else
  - if $m_j = \mathsf{nhdl}(t_j)$ with $t_j \in \mathsf{Msg}$ then $m'_j = t_j$;
  - else $m'_j = \varnothing$.

Asymmetric encryption/signature double key generation The following rule allows agent e, given a certification key pair under handles[4], to generate $(X_k, inv(X_k))$ of level i_2 and control sets containing (S_1, S_2) for agent b. Note that generation and certificate issue are part of a single rule. This allows us to eliminate the need for a certification command, for which deciding the key authenticity could raise a problem.

$$P_e(h_e^{\alpha}(N_1, N_2, inv(Y_k), \mathrm{privCertSign}, i_1, S_{e,1}, S_{e,2})),$$
$$P_e(h_e(N_2, \mathcal{C}(N_2, N_1, N_{cert}, Y_k, \mathrm{pubCertVerif}, i_1, S_{e,1}, S_{e,2}))),$$
$$P_e(i_2), P_e(S_1), P_e(S_2), P_e(b) \overset{N_3, N_4, X_k}{\Longrightarrow}$$
$$P_e(h_e^g(N_3, N_4, inv(X_k), \mathrm{privDecSign}, i_2, S_{e,1} \cup S_{e,2} \cup S_1 \cup \{e\}, \{b, e\} \cup S_2)),$$
$$P_e(\Sigma(\mathcal{C}(N_4, N_3, N_2, X_k, \mathrm{pubEncVerif}, i_2, S_{e,1} \cup S_{e,2} \cup S_1 \cup \{e\}, \{b, e\} \cup S_2), inv(Y_k))),$$
(Asym Gen)

with $e \in S_{e,2}$, $i_1, i_2 \in I_{>0}$, $\alpha \in \{r, g\}$ on condition that $i_2 < i_1$.

[4] We require that both parts of the certification key exist in the creating agent's secure hardware. This is not a compulsory security constraint, in the sense that a few modifications can be performed in the rules and proof to get rid of it. However, it seems reasonable in practice to perform such a verification.

Asymmetric encryption with signature This API command enables an agent b, owner of a handle pointing to an asymmetric key Y_k, to encrypt and sign a payload for agents in $S_{c,2}$, provided b has an integrity handle for a public key X_k of agents in $S_{c,2}$. As in the symmetric case, handles in payload $m_1, \ldots, m_n$ are translated into real values and attributes. Encryption and signature needs to be an atomic command to enable the device to control what can be signed.

$$\begin{aligned} &P_b(h_b^\alpha(N_1, N_2, inv(Y_k), \text{privDecSign}, i_b, S_{b,1}, S_{b,2}),\\ &\qquad P_b(h_b(N_3, \mathcal{C}(N_3, N_4, N_5, X_k, \text{pubEncVerif}, i_c, S_{c,1}, S_{c,2}))),\\ &P_b(m_1), \ldots, P_b(m_n) \Longrightarrow P_b(\{m'_1, \ldots, m'_n\}_{X_k}), P_b(\Sigma(\{m'_1, \ldots, m'_n\}_{X_k}, inv(Y_k))), \end{aligned}$$

(Asym SignEncrypt)

with $i_b, i_c \in I_{>0}$, $b \in S_{b,2}$, $m_j, m'_j \in \mathsf{Msg}$ and for $j = 1, \ldots, n$:

- if $m_j = h_b^\alpha(N_j, N'_j, X_{k,j}, T_j, i_j, S_{j,1}, S_{j,2})$ with $X_{k,j} \in \mathsf{Keyv} \cup \mathsf{Noncev}$ then :
  - if $i_b, i_c \in I_2$, $(i_j, S_{j,1}, S_{j,2}) \prec (i_b, S_{b,1}, S_{b,2})$ and $(i_j, S_{j,1}, S_{j,2}) \prec (i_c, S_{c,1}, S_{c,2})$ then $m'_j = N_j, N'_j, X_{k,j}, T_j, i_j, S_{j,1}, S_{j,2}$;
  - else $m'_j = \varnothing$.
- else $m'_j = \mathsf{nhdl}(m_j)$.

Asymmetric decryption with signature verification The following rule allows for decryption by the agent b of an authenticated ciphertext, using an integrity handle pointing to a public key Y_k to verify the signature and a handle pointing to a key $inv(X_k)$ to decrypt the ciphertext.

$$\begin{aligned} &P_b(h_b(N_1, \mathcal{C}(N_1, N_2, N_3, Y_k, \text{pubEncVerif}, i_c, S_{c,1}, S_{c,2}))),\\ &\qquad P_b(h_b^\alpha(N_4, N_5, inv(X_k), \text{privDecSign}, i_b, S_{b,1}, S_{b,2})),\\ &\qquad P_b(\{m_1, \ldots, m_n\}_{X_k}), P_b(\Sigma(\{m_1, \ldots, m_n\}_{X_k}, inv(Y_k)))\\ &\Longrightarrow P_b(m'_1), \ldots, P_b(m'_n), \end{aligned}$$

(Asym VerifDecrypt)

with $i_b, i_c \in I_{>0}$, $b \in S_{b,2}$, $m_j, m'_j \in \mathsf{Msg}$ and for $j = 1, \ldots, n$:

- if $m_j = N_j, N'_j, X_{k,j}, T_j, i_j, S_{j,1}, S_{j,2}$ then
  - if $i_b, i_c \in I_2$, $(i_j, S_{j,2}, S_{j,2}) \prec (i_b, S_{b,1}, S_{b,2})$ and $(i_j, S_{j,2}, S_{j,2}) \prec (i_c, S_{c,1}, S_{c,2})$ then $m'_j = h_b^r(N_j, N'_j, X_{k.j}, T_j, i_j, S_{j,1}, S_{j,2})$;
  - else $m'_j = \varnothing$.
- if $m_j = \mathsf{nhdl}(t_j)$ for $t_j \in \mathsf{Msg}$ then $m'_j = t_j$.

Certification key generation Given a certification key pair under handles, this rule allows agent e to generate a certification key pair $(X_k, inv(X_k))$ for agent b. As for asymmetric generation, generation and certificate issue are part of an atomic call. It eliminates the need for a certification command, for which deciding the key authenticity could raise a problem.

$$P_e(h_e^{\alpha}(N_1, N_2, inv(Y_k), \text{privCertSign}, i_e, S_{e,1}, S_{e,2})),$$
$$P_e(h_e(N_2, \mathcal{C}(N_2, N_1, N_{cert}, Y_k, \text{pubCertVerif}, i_e, S_{e,1}, S_{e,2}))),$$
$$P_e(i_b), P_e(S_1), P_e(S_2) \overset{N_3, N_4, X_k}{\Longrightarrow}$$
$$P_e(h_e^{g}(N_3, N_4, inv(X_k)), \text{privCertSign}, i_b, S_{e,1} \cup S_{e,2} \cup S_1 \cup \{e\}, \{e, b\} \cup S_2)),$$
$$P_e(\Sigma(\mathcal{C}(N_4, N_3, N_2, X_k, \text{pubCertVerif}, i_b, S_{e,1} \cup S_{e,2} \cup S_1 \cup \{e\}, \{e, b\} \cup S_2), inv(Y_k))),$$
(Cert Gen)

with $e \in S_{e,2}$ and $i_b < i_e$.

Verification of a certificate This rule allows an agent b, given an integrity handle pointing to a verification key and a pre-certificate signed by the matching certification key, to create the suitable integrity handle. For $\Theta \in \{\text{EncVerif}, \text{CertVerif}\}$,

$$P_b(\Sigma(\mathcal{C}(N_1, N_2, N_3, X_k, \text{pub}\Theta, i_c, S_{c,1}, S_{c,2}), inv(Y_k))),$$
$$P_b(h_b(N_3, \mathcal{C}(N_3, N_4, N_5, Y_k, \text{pubCertVerif}, i_e, S_{e,1}, S_{e,2}))) \Longrightarrow$$
$$P_b(h_b(N_1, \mathcal{C}(N_1, N_2, N_3, X_k, \text{pub}\Theta, i_c, S_{c,1}, S_{c,2}))),$$ **(Cert Verif)**

with $i_c, i_e \in I_{>0}$ and $(i_c, S_{c,1}, \varnothing) \prec (i_e, S_{e,1} \cup S_{e,2}, \varnothing)$.

3.3 Security rationale

Below we will formally prove security properties for our design, but first we discuss the design features that prevent it from suffering from the kinds of attacks seen in the literature [2,3,5]. First, we maintain consistent attribute values: the attributes of a key are set once and for all when it is generated or imported onto a device, and when transporting keys, we export all attributes along with the value of the key and protect their integrity.

Second, we prevent 'Wrap and Decrypt' attacks [6, Alg.2] by the distinction between the way keys and data are tagged for encryption: either as a concatenation of key and attributes or encapsulated in a container nhdl. In an implementation of our design, a suitable tagging scheme should be used to ensure this distinction.

Key conjuring, i.e. the ability of the adversary to generate any number of (possibly related) keys on the device, is critical to a number of attacks [2]. Careful design of the decrypt command prevents this. The security proof includes an enumeration of the terms which the adversary can successfully submit to a decryption request (see (**Sign**) and (**SymEnc**)). Roughly, suitable terms are either wrapped under compromised keys or result from an honest use of the encrypt command.

Example In Figure 3 we show the 'before' and 'after' states for three agents using the API in a typical configuration. In the 'before' state, there are no

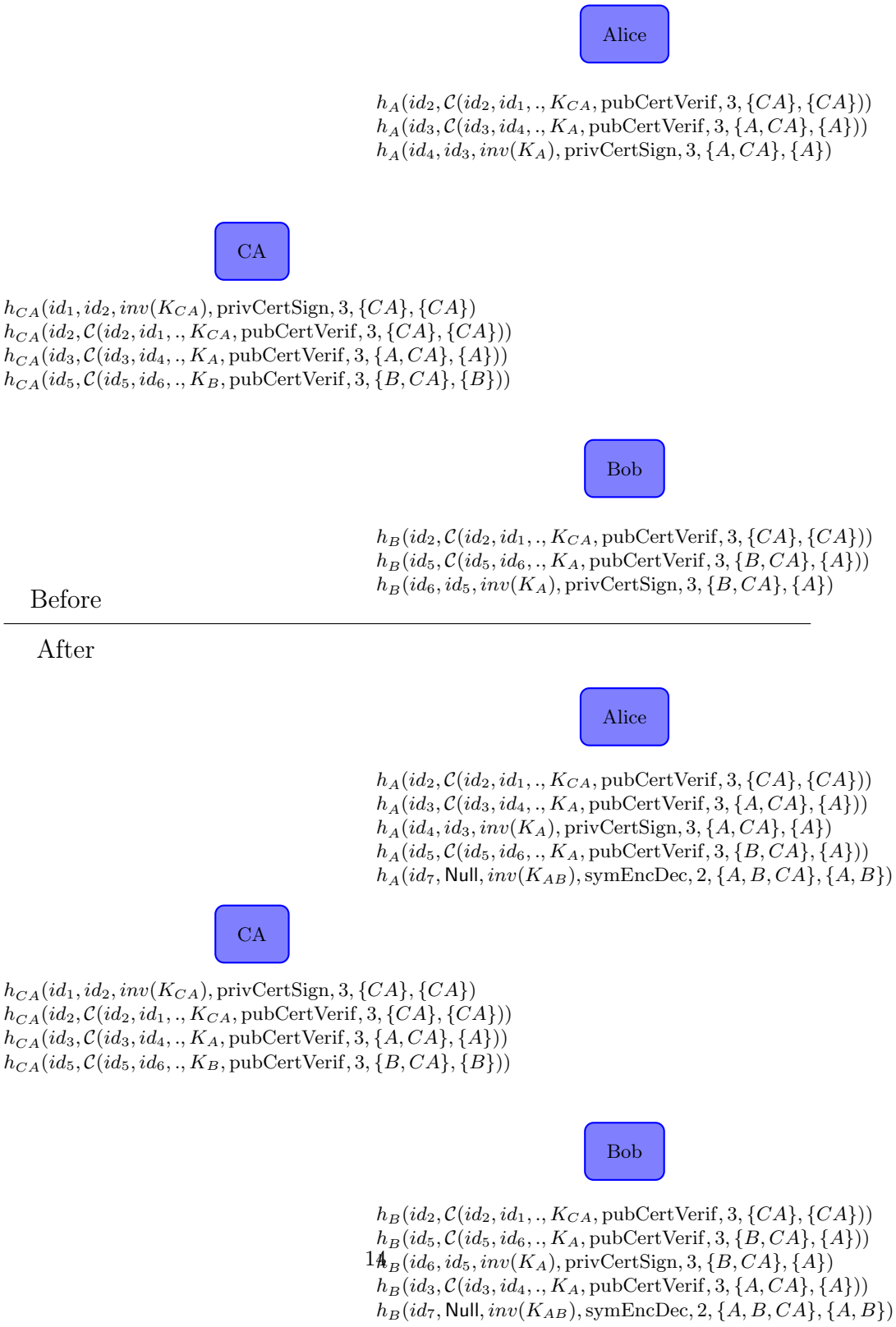

Fig. 3. Operation of the API. See 3.3 for narration.

shared secrets. Alice and Bob both have accepted a copy of the CA's public key certificate and placed it under an integrity handle and they have generated their own public-private keypairs. The CA has accepted public key certificates for each of these pairs. Here we are using integers to label key levels, arbitrarily assigning the long term keys the level 3.

To establish a shared secret, Alice and Bob first need to accept each others public key certificates. This can be done by requesting them from the CA. The CA uses the AsymEncryptSign command to sign the (public) message containing the certificate. Now Alice and Bob can use the certificate verification command to accept the certificates, generating new handles for them.

Now either Alice can generate a symmetric key (handle identified by id_7) and send it to Bob using AsymEncryptSign. Bob will use AsymDecryptVerify and accept the key. Alice and Bob can then exchange messages using the new symmetric key. Note that the new symmetric key is confidential between Alice and Bob, hence has a confidentiality control set S_2 containing only these identifiers, but for integrity it has inherited the dependence on the CA, hence S_1 contains the set of agents CA, Alice and Bob.

4 Security of the API in the symbolic model

4.1 Model of security

In this section, we describe the capacity of the attacker in the spirit of Dolev and Yao [10], as formalized in [1].

Computation of new terms We denote by $\mathsf{INTRUDER}$ the set of rules which allow the attacker to build new terms from the ones that it has already. See figure 4 for a description of the rules.

The transitive reflexive closure of the preceding rules can be interpreted as the set of terms that an attacker can deduce from its knowledge at a certain state. In the following, we say that m is deducible from a set of terms T, which we denote by $T \vdash m$, if starting from the state $\mathcal{S}$ such that $\mathcal{S}_{\mathsf{int}} = T$ and for all $a \in \mathsf{Agent}$, $\mathcal{S}_a = \emptyset$, there exists a state $\mathcal{S}'$ such that $\mathcal{S} \Longrightarrow^*_{\mathsf{INTRUDER}} \mathcal{S}'$ and $m \in \mathcal{S}'_{\mathsf{int}}$. In the sequel, we slightly abuse notations as follows. If t is a term and $\mathcal{S}$ is a state, we write $t \in \mathcal{S}$ (resp. $\mathcal{S} \vdash t$) if $t \in \cup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b$ (resp. if $\cup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \vdash t$).

Control of the network and corruption A couple of rules allows the intruder to control the network (see figure 4). He can intercept and forward or redirect at will messages sent over network channels. Moreover, to formalize corruption of agents (see beginning of Section 2.6), we suppose a given set H of honest agents. The device corruption rule (in figure 4) models the possibility for an adversary to open a device and retrieve all its information. A key-by-key corruption model can also be considered, as is done in [8].

INTRUDER set of rules:

• **Pair rules**

$P_{\mathsf{int}}(m_1), P_{\mathsf{int}}(m_2) \Rightarrow P_{\mathsf{int}}(< m_1, m_2 >)$

$P_{\mathsf{int}}(< m_1, m_2 >) \Rightarrow P_{\mathsf{int}}(m_1), P_{\mathsf{int}}(m_2)$

• **Symmetric cryptography**

$P_{\mathsf{int}}(X_k), P_{\mathsf{int}}(m_1), ..., P_{\mathsf{int}}(m_n) \Rightarrow P_{\mathsf{int}}(\{\!|m_1, ..., m_n|\!\}_{X_k})$

$P_{\mathsf{int}}(X_k), P_{\mathsf{int}}(\{\!|m_1, ..., m_n|\!\}_{X_k}) \Rightarrow P_{\mathsf{int}}(m_1), ..., P_{\mathsf{int}}(m_n)$

• **Asymmetric encryption**

$P_{\mathsf{int}}(X_k), P_{\mathsf{int}}(m_1), ..., P_{\mathsf{int}}(m_n) \Rightarrow P_{\mathsf{int}}(\{m_1, ..., m_n\}_{X_k})$

$P_{\mathsf{int}}(inv(X_k)), P_{\mathsf{int}}(\{m_1, ..., m_n\}_{X_k}) \Rightarrow P_{\mathsf{int}}(m_1), ..., P_{\mathsf{int}}(m_n)$

• **Message container**

$P_{\mathsf{int}}(m) \Rightarrow P_{\mathsf{int}}(\mathsf{nhdl}(m))$

$P_{\mathsf{int}}(\mathsf{nhdl}(m)) \Rightarrow P_{\mathsf{int}}(m)$

• **Signature**

$P_{\mathsf{int}}(\Sigma(m, X_k)) \Rightarrow P_{\mathsf{int}}(m)$

$P_{\mathsf{int}}(X_k), P_{\mathsf{int}}(m) \Rightarrow P_{\mathsf{int}}(\Sigma(m, X_k))$

CONTROL set of rules:

• **Control of the network**

$P_a(m) \Rightarrow P_{\mathsf{int}}(m)$

$P_{\mathsf{int}}(m) \Rightarrow P_a(m)$

• **Device corruption**

$P_a(h_a^\alpha(N_1, N_2, m, T, i, S_1, S_2)) \Rightarrow P_{\mathsf{int}}(m)$, where $a \notin H$

In the above rules, $m, m_i \in \mathsf{Msg}$, $X_k \in \mathsf{Keyv}$ and H is the set of honest agents.

Fig. 4. Rules modeling the adversary abilities

4.2 Initial states

We impose a few requirements on the initial state of a device assuming they are set up in a secure environment. These requirements seem realistic in practice and allow us to start from states compatible with the security policy. In the initial states, we assume that the attacker knows some public information like the set of key levels and the set of agents.

Definition 1. *A state $\mathcal{S}_0$ is said to be* initial *if it satisfies the following hypotheses :*

1. *the set of terms known by the agents and the intruder are atomic : for all $a \in \mathsf{Agent} \cup \{\mathsf{int}\}$, $\mathcal{S}_a \subseteq \mathsf{Handle} \cup \mathsf{Key} \cup \mathsf{Nonce} \cup \mathsf{Agent} \cup \mathcal{T} \cup I \cup \mathscr{S}$ and moreover $\mathcal{T} \cup I \cup \mathscr{S} \subseteq \mathcal{S}_{\mathsf{int}}$.*
2. *all terms stored under handles are secret : for $a \in \mathsf{Agent}$, if $h_a^\alpha(N_1, N_2, m, T, i, S_1, S_2) \in \mathcal{S}_a$ then for $b \in \mathsf{Agent} \cup \{\mathsf{int}\}$, $m \notin \mathcal{S}_b$.*
3. *all key handles known by an agent point to an atomic element : for $a \in \mathsf{Agent}$, if $h_a^\alpha(N_1, N_2, m, T, i, S_1, S_2) \in \mathcal{S}_a$ then $m \in \mathsf{Key} \cup \mathsf{Nonce}$.*
4. *the owner of a key handle is in the set of legitimate agents for this handle. More precisely, we impose that for all $a \in \mathsf{Agent}$, if $h_a^\alpha(N_1, N_2, m, T, i, S_1, S_2) \in \mathcal{S}_a$ then $a \in S_2$.*
5. *any public key certificate under handle corresponds to a private key stored by a rightful agent: $\forall b \in \mathsf{Agent}$, if $h_b(N_1, \mathcal{C}(N_1, N_2, N_3, X_k, \mathrm{pub}\Theta, i, S_1, S_2)) \in \mathcal{S}_b$, then there exists $a \in S_2$ so that*

$$h_a^\alpha(N_2, N_1, inv(X_k), \mathrm{priv}\Theta', i, S_1, S_2) \in \mathcal{S}_a,$$

with $(\Theta, \Theta') \in \{(\mathrm{EncVerif}, \mathrm{DecSign}), (\mathrm{CertVerif}, \mathrm{CertSign})\}$.

6. *the key handles form a* coherent *set: for all* $a, a' \in \mathsf{Agent}$, $h_a^\alpha(N_1, N_2, m, T, i, S_1, S_2) \in \mathcal{S}_a$ *and* $h_{a'}^{\alpha'}(N_1', N_2', m, T', i', S_1', S_2') \in \mathcal{S}_{a'}$ *we have* $N_1 = N_1', N_2 = N_2', T = T', i = i', S_1 = S_1'$ *and* $S_2 = S_2'$.

We can now define the set of states for which we can prove a security property.

Definition 2. *We say that a state* $\mathcal{S}$ *is* accessible *from an initial state* $\mathcal{S}_0$ *if it is reachable by applying a finite number of times the rules of the set* API, INTRUDER *and* CONTROL *to* $\mathcal{S}_0$, *i.e. if* $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API} \cup \mathsf{CONTROL} \cup \mathsf{INTRUDER}} \mathcal{S}$.

4.3 Security properties and sketch of proof

The security of the API should entail that given a state $\mathcal{S}$, secret key values of honest agents should not be known to the intruder. But we would also like to ensure that these values are only used by rightful agents. Secret key values of honest agents are messages $m \in \mathsf{Msg}$ for which there exists a handle of the form $h_a^\alpha(.,.,m,.,.,S_1,S_2)$ with $a \in H$ and $S_1, S_2 \subseteq H$. As the set of legitimate users of m is S_2, the property that we want to prove is formalized as:

$$\forall a \in H, \forall m \in \mathsf{Msg}, \forall i \in I_{>0}, \forall \alpha \in \{r, g\}, \forall S_1, S_2 \subseteq H,$$
$$\mathcal{S} \vdash h_a^\alpha(.,.,m,.,i,S_1,S_2) \Rightarrow \mathcal{S} \not\vdash m \text{ and } a \in S_2 \quad \textbf{(Sec)}$$

If this property is clearly something we want from a security API, it seems legitimate to discuss whether we should require some other security results. Other than confidentiality, security usually also comprises integrity or authenticity aspects. In our framework, this can translate into two different requirements. On one hand, integrity of the attribute values amongst various handles owned by honest agents pointing to the same key seems highly desirable. It can be formalized as :

$$\forall a \in H, \forall b \in \mathsf{Agent}, \forall m \in \mathsf{Msg},$$
$$\forall i, i' \in I_{>0}, \forall \alpha, \alpha' \in \{r, g\}, \forall S_1, S_2 \subseteq H, \forall S_1', S_2' \subseteq \mathsf{Agent},$$
$$\mathcal{S} \vdash h_a^\alpha(N_1, N_2, m, T, i, S_1, S_2) \wedge \mathcal{S} \vdash h_a^\alpha(N_1', N_2', m, T', i', S_1', S_2') \Rightarrow$$
$$N_1 = N_1' \wedge N_2 = N_2' \wedge T = T' \wedge i = i' \wedge S_1 = S_1' \wedge S_2 = S_2' \quad \textbf{(Intg)}$$

On the other hand, since we consider an asymmetric cryptography setting, an agent should be able to trust the value of an integrity handle he owns, on condition it points to a public key certificate whose control sets S_1 and S_2 consist of honest agents. More precisely, if S_1, S_2 contain only honest agents, then there exists a private key handle associated to this certificate the attributes of which are coherent with that of the certificate. This in turn is the meaning of the following property :

$$\forall a \in H, \forall N_1, N_2, N_3 \in \mathsf{Nonce}, \forall i \in I_{>0}, \forall S_1, S_2 \subseteq H \text{ with}$$
$$\mathcal{S} \vdash^* h_a(N_1, \mathcal{C}(N_1, N_2, N_3, X_k, \mathrm{pub}\Theta, i, S_1, S_2)) \Rightarrow \exists b \in S_2 \text{ such that}$$
$$\mathcal{S} \vdash^* h_b^\alpha(N_2, N_1, inv(X_k), \mathrm{priv}\Theta', i, S_1, S_2). \quad \textbf{(Cert)}$$

where $(\Theta, \Theta') \in \{(\text{EncVerif}, \text{DecSign}), (\text{CertVerif}, \text{CertSign})\}$.

We can now give the principal result of this paper, stating the security of our API if it is correctly initialised.

Theorem 1 (Security of the API) *Let $\mathcal{S}_0$ be an initial state and $\mathcal{S}$ be an accessible state from $\mathcal{S}_0$. Then $\mathcal{S}$ satisfies the properties* **Sec**, **Intg** *and* **Cert**.

Proof. We present a sketch of proof (details can be found in [9]). First we consider a more powerful attacker with access to all values stored in compromised hardware as well as to all messages m associated to handles of the form $h_a^\alpha(.,.,m,.,.,S_1,S_2)$ where $S_1, S_2 \subsetneq H$ even if a is honest. The classic adversary can learn these terms anyway, and this extension ensures stability of intruder knowledge when applying rules from $\mathsf{INTRUDER} \cup \mathsf{CONTROL}$.

It yields a generalized deduction definition: we write that $\mathcal{S} \vdash^* t$ when $\cup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \cup \{m, N_1, N_2 | h_a^\alpha(N_1, N_2, m, ., ., S_1, S_2) \in \mathcal{S}, S_1 \subsetneq H \text{ or } S_2 \subsetneq H, a \in \mathsf{Agent}\} \cup \{m, N_1, N_2 | h_a^\alpha(N_1, N_2, m, ., ., ., .) \in \mathcal{S}, a \notin H\} \cup \{m | h_a(., m) \in \mathcal{S}\} \vdash t$.

We then consider a stronger version of the property (**Sec**):

$$\forall a \in H, \forall m \in \mathsf{Msg}, \forall i \in I_{>0}, \forall \alpha \in \{r, g\}, \forall S_1, S_2 \subseteq H,$$
$$\mathcal{S} \vdash^* h_a^\alpha(., ., m, ., i, S_1, S_2) \Rightarrow \mathcal{S} \nvdash^* m, a \in S_2 \text{ and } m \in \mathsf{Key} \cup \mathsf{Nonce}. \qquad (\mathbf{Sec}^*)$$

Intuitively, the property (**Sec***) means that the values stored in the handles of honest agents are always of type Key or Nonce and are not deducible even with the extended deduction rule $\vdash^*$. It is clear that in order to prove the theorem, it is enough to prove the same statement with the stronger version of the property (**Sec**). In the technical report [9], we prove by induction that the property (**Sec***) is invariant under the API rules. To prove this, we introduce four invariants : the first, (**SymEnc**), states that the only well-formed symmetric encryption terms that an adversary can build are either encrypted under a compromised key, or results from an honest and well-formed request to the symmetric encryption command:

$$\forall\, u, k \in \mathsf{Msg}, \mathcal{S} \vdash^* \{\!|u|\!\}_k \Rightarrow \mathcal{S} \vdash^* k$$
$$\textbf{OR } \exists S_1, S_2 \subseteq H\,, a \in S_2 \text{ such that } \mathcal{S} \vdash^* h_a^{\cdot}(., ., k, ., i, S_1, S_2) \text{ and } u = u'_1, \ldots, u'_p$$
$$\text{with} \quad \bullet \text{ either } u'_j = \mathsf{nhdl}(m_j)$$
$$\bullet \text{ or } u'_j = N_{j,1}, N_{j,2}, m_j, T_j, i_j, S_{j,1}, S_{j,2},\ (i_j, S_{j,1}, S_{j,2}) \prec (i, S_1, S_2)$$
$$\text{and } \mathcal{S} \vdash^* h_a^{\cdot}(N_{j,1}, N_{j,2}, m_j, T_j, i_j, S_{j,1}, S_{j,2}) \qquad (\mathbf{SymEnc})$$

The next invariant states that all asymmetric encryption terms deducible from a reachable state have a payload deducible by the attacker or result from an honest request to the asymmetric encryption command.

$$\forall u,\quad K \in \mathsf{Msg}, \mathcal{S} \vdash^* \{u\}_K \Rightarrow \mathcal{S} \vdash^* u$$

OR $\exists S_{c,1}, S_{c,2} \subseteq H, b \in S_{c,2}$ such that

$$\mathcal{S} \quad \vdash^* h_b(.,\mathcal{C}(.,.,.,K,\text{pubEncVerif}, i_c, S_{c,1}, S_{c,2})) \text{ and } u = u'_1, \ldots, u'_p$$

with • either $u'_j = \mathsf{nhdl}(m_j)$

• or $u'_j = N_{j,1}, N_{j,2}, m_j, T_j, i_j, S_{j,1}, S_{j,2}$,

$$(i_j,\quad S_{j,1}, S_{j,2}) \prec (i, S_{c,1}, S_{c,2}) \text{ and } \mathcal{S} \vdash^* h_b(N_{j,1}, N_{j,2}, m_j, T_j, i_j, S_{j,1}, S_{j,2})$$

(AsymEnc)

We need a similar invariant for signed terms the adversary is able to obtain (**Sign**). The invariant here is slightly more involved since we have to deal with both the issue of certificates when generating asymmetric keys and asymmetric wrapping commands:

$$\forall\, u, k \in \mathsf{Msg}, \mathcal{S} \vdash^* \Sigma(u,k) \Rightarrow \mathcal{S} \vdash^* k$$

OR $\exists S'_1, S'_2 \subseteq H, e \in S'_2$ such that

$\mathcal{S} \vdash^* h_e(.,.,k,\text{privCertSign}, i_1, S'_1, S'_2)$

and $u = \mathcal{C}(N_4, N_3, N_2, X_k, \text{pub}\Theta, i_2, S_1 \cup \{e\}, S_2 \cup \{b, e\})$

with $S'_1 \cup S'_2 \subset S_1,\ e \in S'_2, i_2 < i_1, \Theta \in \{\text{EncVerif}, \text{CertSign}\}$

OR $\exists S_{c,1}, S_{c,2} \subseteq H, b \in S_{c,2}$ such that

$\mathcal{S} \vdash^* h_b(.,\mathcal{C}(.,.,.,K,\text{pubEncVerif}, i_c, S_{c,1}, S_{c,2}))$ and $u = \{u'_1, \ldots, u'_p\}_K$

with • either $u'_j = \mathsf{nhdl}(m_j)$

• or $u'_j = N_{j,1}, N_{j,2}, m_j, T_j, i_j, S_{j,1}, S_{j,2}$,

$(i_j, S_{j,1}, S_{j,2}) \prec (i, S_{c,1}, S_{c,2})$ and $\mathcal{S} \vdash^* h_b(N_{j,1}, N_{j,2}, m_j, T_j, i_j, S_{j,1}, S_{j,2})$

(Sign)

To conclude, we remark moreover that from its definition, an initial state satisfies the properties (**Sec***), (**Cert**), (**SymEnc**), (**AsymEnc**), (**Sign**).

5 Experiments

We have used our API to implement some asymmetric key protocols based on well-known examples from the Clark-Jacob corpus. Since we impose a secure encryption and signature scheme, our versions of protocols are secure even when the original is not. For example, our implementation of Needham-Schroeder public key avoids Lowe's attack because all messages are signed. Full details together with a Prolog script for generating API commands from protocols are available at `http://www.lsv.ens-cachan.fr/~steel/genericapi/asym`.

6 Conclusion

We have given the design for a key management API for cryptographic devices that allows the use of asymmetric keys for managing keys, together with security

properties and proofs in the Dolev Yao model. This is the first such design with security proofs as far as we are aware. In future work we will add more flexibility to the API. In particular it should be easy to adapt the design to other security orderings not necessarily based on agent identifiers.

References

1. M. Abadi and P. Rogaway. Reconciling two views of cryptography. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin / Heidelberg, 2000.
2. M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.
3. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269, Chicago, Illinois, USA, Oct. 2010. ACM Press.
4. 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.
5. J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.
6. J. Clulow. On the security of PKCS#11. In *Proceedings of CHES 2003*, pages 411–425, 2003.
7. V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In M. Backes and P. Ning, editors, *Computer Security - ESORICS 2009*, volume 5789 of *Lecture Notes in Computer Science*, pages 605–620. Springer Berlin / Heidelberg, 2009.
8. V. Cortier, G. Steel, and C. Wiedling. Revoke and let live: A secure key revocation API for cryptographic devices. In *19th ACM Conference on Computer and Communications Security (CCS'12)*, Raleigh, USA, October 2012. ACM.
9. M. Daubignard, D. Lubicz, and G. Steel. A secure key management interface with asymmetric cryptography. Technical Report RR8274, INRIA, 2013. Available at http://hal.inria.fr/hal-00805987.
10. D. Dolev and A. C.-C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.
11. S. Kremer, R. Künnemann, and G. Steel. Universally composable key-management. *IACR Cryptology ePrint Archive*, 2012:189, 2012.
12. S. Kremer, G. Steel, and B. Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 266–280, Cernay-la-Ville, France, June 2011. IEEE Computer Society Press.