Synthesising Secure APIs

Véronique Cortier, Graham Steel

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Synthesising Secure APIs

Véronique Cortier — Graham Steel

N° 6882

Mars 2009

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6882--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Synthesising Secure APIs

Véronique Cortier*, Graham Steel

Thème SYM — Systèmes symboliques
Équipes-Projets Cassis et Secsi

Rapport de recherche n° 6882 — Mars 2009 — 24 pages

Abstract: Security APIs are used to define the boundary between trusted and untrusted code. The security properties of existing API are not always clear. In this paper, we give a new generic API for managing symmetric keys on a trusted cryptographic device. We state and prove security properties for the API. In particular, our API offers a high level of security even when the host machine is controlled by an attacker.

Our API is generic in the sense that it can implement a wide variety of (symmetric key) protocols. As a proof of concept, we give an algorithm for automatically instantiating the API commands for a given key management protocol. We demonstrate the algorithm on a set of key establishment protocols from the Clark-Jacob suite.

Key-words: Application Programming Interface, Hardware Security Module, Formal methods

* This work has been pasrtially supported by the ANR project AVOTÉ

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Synthèse d'interfaces de programmation sûres

Résumé : Les interfaces de programmation de sécurité sont utilisées pour définir une frontière entre le code sûr et non sûr. La sécurité de la plupart des interfaces de programmation n'est pas toujours claire. Dans ce papier, nous proposons une nouvelle interface générique qui permet la gestion des clefs symétriques sur un module cryptographique sûr. En particulier, notre interface assure un haut niveau de sécurité même lorsque la machine hôte est contrôlée par un attaquant, à l'aide d'un virus par exemple.

Notre interface est générique dans le sens qu'elle peut implémenter une grande variété de protocoles (à clefs symétrique). Nous proposons ainsi un algorithme qui permet de déduire automatiquement à partir d'un protocole les commandes à effectuer sur l'API. Cet algorithme a été testé sur les protocoles d'établissement de clefs de la libraire Clark-Jacob

Mots-clés : Interface de programmation, Module matériel de sécurité, Méthodes formelles

1 Introduction

Security APIs are used to define the boundary between trusted and untrusted code. They typically arise in systems where certain security-critical fragments of a programe are executed on some tamper resistant device (TRD), such as a smartcard, USB security key or hardware security module (HSM). Though they typically employ cryptography, security APIs differ from regular cryptographic APIs in that they are designed to enforce a policy, i.e. no matter what API commands are received from the (possibly malicious) untrusted code, certain properties will continue to hold, e.g. the secrecy of sensitive cryptographic keys.

The ability of these APIs to enforce their policies has been the subject of formal and informal analysis in recent years. Open standards such as PKCS#11 [15] and proprietary solutions such as IBM's Common Cryptographic Architecture [4] have been shown to have flaws which may lead to breaches of the policy [2, 6, 7, 9, 12]. The situation is complicated by the lack of a clearly specified security policy, leading to disputes over what does and does not constitute an attack [11]. All this leaves the application developer in a confusing position. Since more and more applications are turning to TRD based solutions for enforcing security [1, 14] there is a pressing need for solutions.

In this paper, we set out to tackle this problem from a different direction. We suggest a way to infer functional properties of a security API for a TRD from the security protocols the device is supposed to support. Our first main contribution is to give a generic API for key management protocols. Our API is generic in the sense that it can implement any (executable) symmetric key protocol, while ensuring security of confidential data. The key idea is that confidential data should be stored inside a secure component together with the set of agents that are granted access to it. Then our API will encrypt data only if the agents that are granted access to the encryption key are all also granted access to the encrypted data. In this way, trusted data can be securely shared between APIs. To illustrate the generality of our API, we show how to instantiate the API commands for a given protocol using a simple algorithm that has been implemented in Prolog. In particular, we show that our API supports a suite of well-known key establishment protocols.

Our second main contribution is to state and prove key security properties for the API no matter what protocol has been implemented. We show in particular that our API guarantees the confidentiality of any (non public) data that is meant to be shared between honest agents only. The property holds even when honest agents APIs are controlled by an attacker (in case e.g. an honest machine has been infected by a worm). Considering an even stronger attack scenario, where the attacker is also given old confidential keys, we show that our API still provides security provided it is switched to a restricted mode where the API decrypts a cyphertext only when it is able to perform some freshness test. This restricted mode allows us to implement fewer protocols. In particular, of course it does not allow us to implement protocols subject to replay attacks. In fact, we discovered that any symmetric key establishment protocol of the Clark and Jacob library [5] can be implemented within the restricted mode, except for protocols that are known to suffer from replay attacks.

The paper is organised as follows: in section 2, we define our formal model for APIs and protocols. We then define our API (section 3), and show how it

can be used to implement a protocol (section 4). We state and prove security properties (section 5 and section 6), and then give our results on the Clark-Jacob protocols (section 7). Finally, we discuss related work, further work and conclusions (section 8). Proofs can be found in an Appendix.

2 Model

2.1 Syntax

As usual, messages are represented using a term algebra. We assume a finite set of agents Agent and infinite sets of nonces Nonce and keys Key. We also assume an infinite set of variables Var, among which we distinguish a set VarKey of variables of sort key and a set $\mathsf{VarNonce}$ of sort nonce.

$$\begin{array}{rcl} \mathsf{Keyv} & ::= & \mathsf{Key} \mid \mathsf{VarKey} \\ \mathsf{Noncev} & ::= & \mathsf{Nonce} \mid \mathsf{VarNonce} \\ \mathsf{Msg} & ::= & \mathsf{Agent} \mid \mathsf{Keyv} \mid \mathsf{Noncev} \mid \mathsf{Var} \\ & & \mid \{\mathsf{Msg}\}_{\mathsf{Keyv}} \mid \langle \mathsf{Msg}, \mathsf{Msg} \rangle \\ \mathsf{Handle} & ::= & h_a^\alpha(\mathsf{Nonce}, \mathsf{Msg}, i, S) \end{array}$$

where $i \in \{0,1,2,3\}, S \subseteq \mathsf{Agent}, a \in \mathsf{Agent}, \alpha \in \{r,g\}$. In what follows, we only consider well-sorted substitution. We may write $t_1, t_2, \ldots, t_n$ instead of $\langle t_1, \langle t_2, \langle \ldots, t_n \rangle \ldots \rangle \rangle$.

The API does not give direct access to secret keys but provides the user with a handle that can be used later to indicate to the API to use a specific key. A handle $h_a^\alpha(n,k,i,S)$ represents a reference stored on the API belonging to a for a key k of security level i. The set S represents the set of users that are allowed to access to k. The nonce n is used to avoid confusion between handles that refer to the same data. The label α distinguishes the handles corresponding to values k generated by the API ($\alpha = g$) from values k received by the API ($\alpha = r$). This distinction allows the API to check for freshness. We consider four levels of security:

- 0: public data
- 1: secret data that are not used for encryption (typically nonces)
- 2: short term keys
- 3: long term keys

We consider the set $\mathcal{P} = \{k_a \mid a \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$ of predicates. K_a with $a \in \mathsf{Agent}$ to represent the knowledge of an agent a. The predicate K_{int} is a special predicate that represents the knowledge of the intruder.

2.2 Model

Our model is a state-based transition system. A rule is an expression of the form $P_1(u_1), \ldots, P_k(u_k) \xrightarrow{N_1,\ldots,N_p} Q_1(v_1), \ldots, Q_l(v_l)$ where the u_i, v_i are messages or handles possibly with variables, the N_i are variables and P_i, Q_i are predicates.

Example 1 *The following set* INTRUDER *of rules represents the ability of an attacker to concatenate and pair and to encrypt and decrypt when he knows the key.*

$$\begin{array}{rcl} K_{\mathsf{int}}(x), K_{\mathsf{int}}(y) & \rightarrow & K_{\mathsf{int}}(\langle x, y\rangle) \\ K_{\mathsf{int}}(\langle x, y\rangle) & \rightarrow & K_{\mathsf{int}}(x) \\ K_{\mathsf{int}}(\langle x, y\rangle) & \rightarrow & K_{\mathsf{int}}(y) \\ K_{\mathsf{int}}(x), K_{\mathsf{int}}(y) & \rightarrow & K_{\mathsf{int}}(\{x\}_y) \\ K_{\mathsf{int}}(\{x\}_y), K_{\mathsf{int}}(y) & \rightarrow & K_{\mathsf{int}}(x) \end{array}$$

A state of our execution model is the current knowledge of the intruder and the users. It is formally represented by a family $\{S_b \mid b \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$ where int is a special index representing the intruder. The S_b are sets of messages and handles. Given a family $\mathcal{S}$ of sets and an index $b \in \mathsf{Agent} \cup \{\mathsf{int}\}$, we denote by $\mathcal{S}_b$ the set of $\mathcal{S}$ indexed by b.

The knowledge of the agents evolves following the rules. Given a set of rules $\mathcal{R}$, we say that a state $\mathcal{S}$ is accessible in one step from a state $\mathcal{S}'$, denoted by $\mathcal{S} \rightarrow_{\mathcal{R}} \mathcal{S}'$ if there exists a rule $K_{a_1}(u_1), \ldots, K_{a_k}(u_k) \stackrel{N_1,\ldots,N_p}{\longrightarrow} K_{b_1}(v_1), \ldots, K_{b_l}(v_l)$ of $\mathcal{R}$ and a substitution θ such that

- $u_i\theta \in \mathcal{S}_{a_i}$ for any $1 \leq i \leq k$;
- $N_j\theta$ are fresh nonces (that do not appear in $\mathcal{S}$);
- $\mathcal{S}'$ is the smallest family such that $\mathcal{S}_b \subseteq \mathcal{S}'_b$ for any $b \in \mathsf{Agent} \cup \{\mathsf{int}\}$ and $v_i\theta \in \mathcal{S}'_{b_i}$ for any $1 \leq i \leq l$.

$\rightarrow^*_{\mathcal{R}}$ denotes the reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$. We may omit $\mathcal{R}$ when the set of rules is clear from the context.

Note that we retrieve the usual deducibility notion by saying that a term m is deducible from a set of terms S, which is denoted by $S \vdash m$, whenever there exists $\mathcal{S}'$ such that $\mathcal{S} \rightarrow^*_{\mathsf{INTRUDER}} \mathcal{S}'$ and $m \in \mathcal{S}'_{\mathsf{int}}$ where $\mathcal{S}$ is defined by $\mathcal{S}_a = \emptyset$ for any $a \in \mathsf{Agent}$ and $\mathcal{S}_{\mathsf{int}} = S$.

3 Presentation of the Generic API

We design an API that allows users to store secret data inside a secure component or tamper-resistant device (TRD). A user should never have direct access to the stored secret values but should use the API commands to require the TRD to encrypt and decrypt for him. Our API has simply three commands: generation of new data, encryption and decryption.

3.1 API rules

The encryption command takes as inputs a handle for the key that will be used for encryption and a list of data that are meant to be encrypted. Secret data are not given in clear but are designated through handles. The key idea is that for each encrypted data, the API indicates its level of security and the set of agents

that are granted access to it. The encryption command is formally represented by the set of rules of the form

$$K_a(h_a^\alpha(X_n, X_k, i_0, S_0)), K_a(m_1), \ldots, K_a(m_k) \quad \Rightarrow \quad K_a(\{m'_1, \ldots, m'_k\}_{X_k}) \qquad (\textbf{Encrypt})$$

where

- $\alpha \in \{r, g\}$, $k \in \mathbb{N}$, $a \in S_0 \subseteq \mathsf{Agent}$, $i_0 \in \{2, 3\}$, $X_k \in \mathsf{VarKey}$;
- $m'_j = m_j, 0$ if $m_j \in \mathsf{Var}$ is a variable. The user provides some public data to be encrypted by the API.
- $m'_j = X_{k_j}, i_j, S_j$ if $m_j \in \mathsf{Handle}$ is a handle of the form $h_a^\alpha(X_{n_j}, X_{k_j}, i_j, S_j)$. The user requires the API to encrypt the secret value k_j associated the handle $h_a^\alpha(n_j, k_j, i_j, S_j)$;
- the rule is undefined otherwise.

provided that i_0 is strictly greater than any (defined) i_j (keys only encrypt data of strictly lower security level) and $S_0 \subseteq S_j$ for any (defined) S_j (data should not be transmitted to users that are not allowed to access to).

The encryption command takes as inputs a handle for the key that will be used for decryption, a cyphertext and a (possibly empty) list of handles together with a list L of indices (of the same length). After decrypting the cyphertext, the API recovers a list of component of the form m, i, S where i is a security level and S is the set of agents that are granted access to the data m. For each component whose index is in S, the API checks whether the value m corresponds the handle given in input. Then if all tests succeed, for all other (untested) components, the API either returns the value m in clear if the data has security level 0 or, if the data has security level greater than 0, the API stores the data, generates a fresh handle that refers to this data and returns the handle. The decryption command is formally represented by the set of rules of the form

$$K_a(h_a^\alpha(X_n, X_k, i_0, S_0)), K_a(\{m_1, \ldots, m_p\}_{X_k}), \bigcup_{j \in L} K_a(m'_j) \overset{N_1, \ldots, N_p}{\Rightarrow} \cup_{j \notin L} K_a(m'_j) \qquad (\textbf{Decrypt/Test})$$

where

- $L \subseteq \{1, \ldots, p\}$, $\alpha \in \{r, g\}$, $k \in \mathbb{N}$, $a \in S_0 \subseteq \mathsf{Agent}$, $i_0 \in \{2, 3\}$, $N_1, \ldots, N_k \in \mathsf{VarNonce}$;
- for any $j \in L$, $m'_j = h_a^g(X_{n_j}, X_j, 0, \emptyset)$ if m_j is of the form $X_j, 0$ and $m'_j = h_a^g(X_{n_j}, X_j, i_j, S_j)$ if m_j is of the form i_j, S_j, X_j with $i_j \geq 1$. For any $j \in L$, the API checks that m_j corresponds to a value it generated itself.
- for any $j \notin L$, $m'_j = x_j$ if m_j is of the form $x_j, 0$ (data of security level 0 are given to the user) and $m'_j = h_a^\alpha(N_j, y_{k_j}, i_j, S_j)$ if m_j is of the form y_{k_j}, i_j, S_j with $i_j \geq 1$. Secret data are not transmitted to the user, he is only given a handle to them.

provided again that i_0 is strictly greater than any (defined) i_j and $S_0 \subseteq S_j$ for any (defined) S_j.

The generation command of the API allows a user to generate new secret values of level security 1 or 2, associated to a set of agents provided by the user, in which case he is simply given a handle referring to the new secret value. It also allows a user to generate fresh public data, in which case he is given both the value and a handle to it. The public data generated by the API can be typically used for ensuring freshness. The generation command for secret values is formally represented by the following set of rules.

$$\stackrel{N,K}{\Rightarrow} K_a(h^g_a(N, K, i, S)) \qquad \textbf{(Secure Generate)}$$

where $i \in \{1, 2\}$, $S \subseteq \mathsf{Agent}$ such that $a \in S$, $N \in \mathsf{VarNonce}$, and $K \in \mathsf{VarNonce}$ if $i = 1$, $K \in \mathsf{VarKey}$ if $i = 2$.

The generation command for public data is formally represented by the following set of rules.

$$\stackrel{N,K}{\Rightarrow} K_a(K), K_a(h^g_a(N, K, 0, \emptyset)) \qquad \textbf{(Public Generate)}$$

where $N, K \in \mathsf{VarNonce}$.

Example 2 *Carlsen's Secret Key Initiator Protocol [3, Figure 2]*

1. $A \rightarrow B$: A, Na
2. $B \rightarrow S$: $\mathrm{A}, \mathrm{Na}, \mathrm{B}, \mathrm{Nb}$
3. $S \rightarrow B$: $\{\mathrm{K_{ab}}, \mathrm{N_b}, \mathrm{A}\}\mathrm{K_{bs}}, \{\mathrm{N_a}, \mathrm{B}, \mathrm{K_{ab}}\}\mathrm{K_{as}}$
4. $B \rightarrow A$: $\{\mathrm{N_a}, \mathrm{B}, \mathrm{K_{ab}}\}\mathrm{K_{as}}, \{\mathrm{N_a}\}\mathrm{K_{ab}}, \mathrm{N'_b}$
5. $A \rightarrow B$: $\{\mathrm{N'_b}\}_{\mathrm{Kab}}$

The aim of the protocol is to establish a fresh session key K_{ab} for participants a *and* b *using a key server* s. *In the first message,* a *sends her name and a fresh nonce to* b. *In message 2* b *forwards these values together with his own fresh nonce to the server* s. *The server generates K_{ab} and encrypts it first for* b, *under* b*'s long term key K_{bs}, in a package together with his nonce and* a*'s name, and then for* a, *under her long term key K_{as}, together with her nonce and* b*'s name. The server sends both packets to* b. *In message 4,* b *forwards to* a *her encrypted package,* a*'s nonce N_a encrypted under the session key K_{ab}, and a further fresh nonce N'_b. In message 5* a *returns this nonce encrypted under K_{ab}. Now both* a *and* b *should accept K_{ab} as the session key.*

To implement this protocol using our API, a *should have a handle $h^r_a(n'_{KAS}, k_{as}, 3, \{a, s\})$ to the key k_{as} of level 3. The agent* a *can execute its first protocol's rule by using the following API command:*

$$\stackrel{N,N_A}{\Rightarrow} K_a(N_A), K_a(h^g_a(N, N_A, 0, \emptyset))$$

where N, N_A are nonce variables. a *obtains both a fresh (public) nonce N_A and a handle $h^g_a(N, N_A, 0, \emptyset)$ for it.*

a*'s second step in the protocol (rule 5) can also be performed using the API's commands. Upon receiving a message of the form $\{\mathrm{N_a}, \mathrm{a}, \mathrm{K_{ab}}\}\mathrm{K_{as}}, \{\mathrm{N_a}\}\mathrm{K_{ab}}, \mathrm{N'_b}$,* a *can split it into two parts x_1, x_2 and x_3. Intuitively, x_1 should correspond to $\{\mathrm{N_a}, \mathrm{b}, \mathrm{K_{ab}}\}\mathrm{K_{as}}$, the part x_2 should correspond to $\{\mathrm{N_a}\}\mathrm{K_{ab}}$ and x_3 should correspond to $\mathrm{N'_b}$. Then* a *can decrypt x_1 using the following decryption command*

(with $J = \{1\}$, that is the first component should be checked):

$$\begin{aligned} & K_a(h^r_a(N'_{KAS}, K_{as}, 3, \{a, s\})), \\ & K_a(\{N_A, 0, y, 0, x, 2, \{a, b, s\}\}_{K_{as}}) \\ & [K_a(h^g_a(N, N_A, 0, \emptyset))] \\ \stackrel{N'}{\Rightarrow} & K_a(y), K_a(h^r_a(N', x, 2, \{a, b, s\})) \end{aligned}$$

where $N, N_A, N'_{kas}, K_{as}, x, y$ are variables. a *can check that y is equal to* b *and receives a handle $K_A(h^r_A(N', x, 2, \{a, b, s\})$ that refers to x and should correspond to the inside key K_{ab}. Then* a *can decrypt x_2 using the following decryption command (with again $J = \{1\}$, that is the first component should be checked):*

$$\begin{aligned} & K_a(h^r_a(N', K_{ab}, 2, \{a, b, s\})) \\ & K_a(\{N_A, 0\}_{K_{ab}}) \\ & [K_a(h^g_a(N, N_A, 0, \emptyset))] \Rightarrow \end{aligned}$$

where N, N_A, N', K_{ab} are variables. If the command succeeds, the agent a *knows that the second component x_2 indeed corresponds to* $\{\mathrm{N_a}\}\mathrm{K_{ab}}$*. Then* a *can build her message for* b *by using the following encryption command.*

$$\begin{aligned} & K_a(K_a(h^r_a(N', K_{ab}, 2, \{a, b, s\}))), K_a(x_3) \\ & \Rightarrow K_a(\{x_3, 0\}_{K_{ab}}) \end{aligned}$$

where N', K_{ab} are variables.

4 Using the Generic API to Implement a Protocol

In this section we show how the generic API can be used to implement symmetric key protocols, including in particular symmetric key distribution protocols from the venerable Clark-Jacob survey [5].

To deduce the API commands, we first require the protocol to be specified in a manner following e.g. [16], that is each protocol step is given as a rule

$$A : u \quad \xrightarrow{new\ \mathcal{N}} \quad v$$

A is the agent who plays the role. The u, v are terms in our algebra from section 2, where agent names, keys and nonces are given as variables. The set $\mathcal{N}$ of nonce and key variables represents freshly generated data. In addition we require the terms in the protocol to be tagged with their type (agent, nonce, key or message), and nonces and session keys must be tagged with the name of the agent which generated them, their level (0 for a nonce is sent in the clear, 1 for a nonce only ever sent encrypted, 2 for a session key) and the set of participants expected to share secrets. Everything generated by the participants during the protocol (i.e. keys and nonces) will be assumed to be shared between all participants. We will not attempt to deduce whether a nonce is kept secret from the server, or secret from Bob, etc. Tagged nonces in a protocol will be written $n(A, N_A, L, Set)$, where A is the agent, N_A the name for the nonce, L

the level and *Set* the set. Similarly, we have tagged keys $k(S, K_A, L, Set)$, agent names $a(A)$ and message variables $m(X)$. This tagging can be easily guessed by a user reading the protocol but could also be found automatically (for example, by trying several possible taggings).

Given a tagged term t, $\mathsf{un}(t)$ denotes its untagged version obtained from t by removing all the tags. For example, $\mathsf{un}(n(A, N_A, L, Set)) = N_A$. Moreover, given a term t, we denotes by $\overline{t}$ the term obtained from t by replacing each subterm $\{u\}_v$ of t by the variable $X_{\{u\}_v}$. The function $\overline{\cdot}$ is a one-to-one mapping.

4.1 Algorithm

We give a simple algorithm for constructing API commands for a given protocol below in informal pseudocode. The algorithm relies on a global store H of handles that each participant in the protocol will expect to have when a protocol step is executed. This store has an initial state. For example, for the three-party key exchange protocols, the initial state is

$h_a^r(N_{Kas}, kas, 3, \{a, s\})$ `% A handle for kas`
$h_b^r(N_{Kbs}, kbs, 3, \{b, s\})$ `% B handle for kbs`
$h_s^g(N'_{Kas}, kas, 3, \{a, s\})$ `% S handle for kas`
$h_s^g(N'_{Kbs}, kbs, 3, \{b, s\})$ `% S handle for kbs`

Note that where we give agent names a, b, and s as ground terms these should be interpreted as parameters - it is up to the implementer to equip the TRD with the hanldes and API for the roles of a, b or s as appropriate.

Implementing a single protocol step requires:

1. zero or more Decryption Commands, followed by
2. zero or more Generate commands, followed by
3. zero or more Encryption Commands

To construct the commands for rule $u \xrightarrow{new\ \mathcal{N}} v$ played by agent A:

Decryption

For each encyption $\{m_1, \ldots, m_p\}_{Xk}$ occurring in u:

Retrieve $h_A^\alpha(N, X_k, j, Set)$ from store H. If none exists then the algorithm fails. The protocol is actually not executable since the agent does not have the decryption key (and enrcypted packets for forwarding must be marked as message variables).

Select the first m_i such that $m_i = n(A, X, I, Set)$ and $h_A^g(N', X, I, Set)$ is in the handle store and set $L = [K_A(h_A^g(N', X, I, Set))]$. If no such m_i exists, and $j = 3$ then output the warning "missing freshness test" and set $L = []$. We will see later that tests ensure a higher level of security.

Add decryption command of the form

$$K_A(h_A^\alpha(N, X_k, j, Set)), K_A(\{\overline{\mathsf{un}(m_1)}, \ldots, \overline{\mathsf{un}(m_p)}\}_{X_k}), L \overset{N_1,\ldots,N_p}{\Rightarrow} \bigcup_{j \neq i} K_A(m'_i)$$

where the m'_i are defined from the $\overline{\mathsf{un}(m_i)}$ as in section 3.1.

Generate

For each $n(A, X, 0, Set) \in \mathcal{N}$, add generate command

$$\stackrel{N,X}{\Rightarrow} K_A(X), K_A(h^g_A(N, X, L, Set))$$

Add $h^g_A(N, X, 0, Set)$ to the handle store H.

For each $n(A, X, 1, Set) \in \mathcal{N}$, add generate command

$$\stackrel{N,X}{\Rightarrow} K_A(h^g_A(N, X, 1, Set))$$

Add $h^g_A(N, X, 1, Set)$ to the handle store H.

For each $k(A, X, 2, Set) \in \mathcal{N}$, add generate command

$$\stackrel{N,X}{\Rightarrow} K_A(h^g_A(N, X, 2, Set))$$

Add $h^g_A(N, X, 2, Set)$ to the handle store H.

Encryption

For each encryption $\{m_1, \ldots, m_p\}_{Xk}$ occurring in v:

Retrieve $h^\alpha_A(N, X_k, i, Set)$ from the handle store H.
Add encryption command of the form

$$K_A(h^\alpha_A(N, X_k, i, S)), K_A(m'_1), \ldots, K_A(m'_k) \Rightarrow K_A(\{\overline{\mathsf{un}(m_1)}, \ldots, \overline{\mathsf{un}(m_k)}\}_{X_k})$$

Where m'_i is

- h if $m_i = n(A, Y, 1, S)$ is a level 1 nonce with a handle $h = h^\alpha_A(N', Y, 1, S) \in H$
- h if $m_i = k(A, X, 2, S)$ is a key with a handle $h = h^\alpha_A(n', Y, 2, S) \in H$
- $\overline{\mathsf{un}(m_i)}$ if m_i is an agent name, a nonce of level 0, a message variable or a cyphertext.
- The algorithm fails otherwise, that is, in case m_i is of level security 1 or 2 with no corresponding handle in the store (or if m_i is of higher security level). This corresponds to a case where the agents is enable to build the message thus the protocol is not executable.

We consider encrypted terms to be terms of level 0. In this way we can treat nested encryptions by recursively generating encryption commands, treating the innermost encryption first.

4.2 Example

We consider the Carlsen's Secret Key Initiator Protocol presented in example 2.

In our tagged notation, the protocol is written

$$A: \quad \rightarrow \mathsf{a(A)}, \mathsf{n(A, NA, 0, [])}$$

$$B: \quad \mathsf{a(A)}, \mathsf{m(NA)} \xrightarrow{\mathsf{n(B,NB,0,[])}} \mathsf{a(A)}, \mathsf{m(NA)}, \mathsf{a(B)}, \mathsf{n(B, NB, 0, [])}$$

$$S: \quad \mathsf{a(A)}, \mathsf{m(NA)}, \mathsf{a(B)}, \mathsf{m(NB)} \xrightarrow{\mathsf{k(S,KAB,2,[A,B,S])}} \{\mathsf{k(S, KAB, 2, [A, B, S])}, \mathsf{m(NB)}, \mathsf{a(A)}\}_{\mathsf{k(S,KBS,3,[B,S])}}, \{\mathsf{m(NA)}, \mathsf{a(B)}, \mathsf{k(S, KAB, 2, [A, B, S])}\}_{\mathsf{k(S,KAS,3,[A,S])}}$$

$$B: \quad \{\mathsf{m(KAB)}, \mathsf{n(B, NB, 0, [])}, \mathsf{a(A)}\}_{\mathsf{k(S,KBS,3,[B,S])}}, \mathsf{m(X)} \xrightarrow{\mathsf{n(B,NBB,0,[])}} \mathsf{m(X)}, \{\mathsf{m(NA)}\}_{\mathsf{m(KAB)}}, \mathsf{n(B, NBB, 0, [])}$$

$$A: \quad \{\mathsf{n(A, NA, 0, [])}, \mathsf{a(B)}, \mathsf{m(KAB)}\}_{\mathsf{k(S,KAS,3,[A,S])}}, \{\mathsf{n(A, NA, 0, [])}\}_{\mathsf{m(KAB)}}, \mathsf{m(NBB)} \rightarrow \{\mathsf{m(NBB)}\}_{\mathsf{m(KAB)}}$$

$$B: \quad \{\mathsf{n(B, NBB, 0, [])}\}_{\mathsf{m(KAB)}} \rightarrow$$

Note that this tagged version corresponds to a tagging where variable are typed as little as possible. For example, the server will accept any value (possibly not a nonce) for NA and NB. Another tagging option may be chosen if the implementation guarantees against type flaw attacks for example.

The API commands generated by our algorithm for the rules of A are the commands that have been presented in example 2. The commands obtained by our algorithm for B and S are given below. Variables such as N_A have a local scope within the command for a particular protocol step.

Command for B corresponding to his first step

$$\overset{N,N_B}{\Rightarrow} K_b(N_B), K_b(h_b^g(N, N_B, 0, \emptyset))$$

Commands for S corresponding to his first step

$$\overset{N'',K_{AB}}{\Rightarrow} K_s(h_s^g(N'', K_{AB}, 2, \{a, b, s\}))$$

$$K_s(h_s^g(N'_{Kbs}, K_{BS}, 3, \{b, s\})), K_s(h_s^g(N'', K_{AB}, 2, \{a, b, s\})), K_s(N_B), K_s(a) \Rightarrow K_s(\{K_{AB}, 2, \{a, b, s\}), N_B, 0, a, 0)\}_{K_{BS}})$$

$$\begin{aligned} & K_s(h_s^g(N'_{Kas}, K_{AS}, 3, \{a, s\})), \\ & K_s(N_A), \\ & K_s(b), h_s^g(N'', K_{AB}, 2, \{a, b, s\})) \\ \Rightarrow\ & K_s(\{N_A, 0, b, 0, K_{AB}, 2, \{a, b, s\}\}_{K_{AS}}) \end{aligned}$$

Commands for B corresponding to his second step

$$\stackrel{N', N'_B}{\Rightarrow} K_b(N'_B), K_b(h_b^g(N', N'_B, 0, \emptyset))$$

$$\begin{aligned} & K_b(h_b^r(N_{Kbs}, K_{BS}, 3, \{b, s\})), \\ & K_b(\{K_{AB}, 2, \{a, b, s\}, N_B, 0, a, 0\}_{K_{BS}}), \\ & [K_b(h_b^g(N, N_B, 0, \{\}))] \\ \stackrel{N''}{\Rightarrow}\ & K_b(h_b^r(N'', K_{AB}, 2, \{a, b, s\})), K_b(a) \end{aligned}$$

$$\begin{aligned} & K_b(h_b^r(N'''', K_{AB}, 2, \{a, b, s\})), K_b(N_A) \\ \Rightarrow\ & K_b(\{N_A\}_{K_{AB}}) \end{aligned}$$

Commands for B corresponding to his last step

$$\begin{aligned} & K_b(h_b^r(N'', K_{AB}, 2, \{a, b, s\})), \\ & K_a(\{N'_B, 0\}_{K_{AB}}), \\ & [K_b(h_b^g(N', N'_B, 0, \emptyset))] \\ \Rightarrow & \end{aligned}$$

A Prolog implementation has been tested on all the protocols in section 6.3 of the Clark-Jacob survey, excepting those where freshness is assured by timestamps. The Prolog source and the results are available via http: `http://www.lsv.ens-cachan.fr/GenericAPI/`. We give the results in section 7, after we discuss the security properties of our API.

5 Security of the API

The aim of the API is to protect the confidentiality of secret data for a certain group of users, called *honest agents*. Let H be such a set. Agents that are not in H are said to be *compromised* or *corrupted*.

We assume the intruder to have complete control not only of the network, but also of the machines of the honest users (using viruses or worms for example). We also assume that he has access to the values of corrupted users. The only trusted secure parts are the secure storage components (TRDs) of the honest

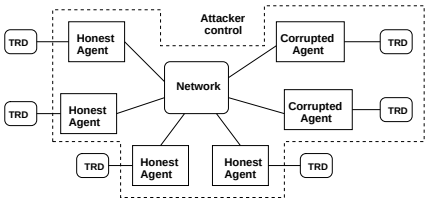

Figure 1: Threat model. The attacker controls the network, all machines, and has obtained access to the memory of some corrupted agents' TRDs

users, managed by the API (see Figure 1). This can be easily modeled by adding the following set $\mathsf{CONTROL}$ of rules

$$K_a(x) \Rightarrow K_{\mathsf{int}}(x) \quad (1)$$

$$K_{\mathsf{int}}(x) \Rightarrow K_a(x) \quad (2)$$

$$K_b(h_b^\alpha(x, y, i, S)) \Rightarrow K_{\mathsf{int}}(y) \quad (3)$$

for any $a, b \in \mathsf{Agent}$ such that $b \notin H$, $i \in \{1, 2, 3\}$, $\alpha \in \{r, g\}$ and $S \subseteq \mathsf{Agent}$. This models the fact that the intruder can access any value known by the user (including handles) and that the intruder can also store messages on users machines in order to then communicate with the API. The last rule indicates the fact that the intruder is given any value that may be stored in a TRD of a corrupted machine. Given a state $\mathcal{S}$ of our execution model and by abuse of notation, we write $t \in \mathcal{S}$ instead of $t \in \bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b$.

When the API is initialized, keys of level 3 are generated and distributed between the secure components managed by APIs and users are given handles to these keys. These keys are initially unknown to the intruder. Thus we say that a state $\mathcal{S}$ is *initial* if $\mathcal{S}_{\mathsf{int}} \subseteq \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key}$ is a set of atomic messages and if for any $a \in \mathsf{Agent}$, the set $\mathcal{S}_a$ only contains handles of the form $h_a^\alpha(n, k, i, S)$ with $n \in \mathsf{Nonce}$, $k \in \mathsf{Nonce} \cup \mathsf{Key}$ and such that n, k do not appear in $\mathcal{S}_{\mathsf{int}}$.

The security of the API can be expressed as follows: given a state $\mathcal{S}$ of the system, secret data of honest users should not be known to the intruder. Secret data of honest users are values k for which there are handles of the form $h_a^\alpha(n, k, i, S)$ where S is a subset of honest users. This is reflected by the following formula:

$$\forall a \in \mathsf{Agent}, \forall x, y \in \mathsf{Msg}, \forall i \in \{1, 2, 3\}, \forall \alpha \in \{r, g\}, \forall S \subseteq H$$
$$\mathcal{S} \vdash h_a^\alpha(x, y, i, S) \Rightarrow \mathcal{S} \not\vdash y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \quad (\mathbf{Sec})$$

We can show that our generic API satisfies the security property **Sec** as the API is correctly initialized. This is an important feature since it guarantees confidentiality of sensitive data for an API which can implement a variety of protocols (cf Section 4) even if the intruder has control of all honest users machines.

Theorem 1 *Let $\mathcal{S}_0$ be an initial state. Then for any state $\mathcal{S}$, accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \rightarrow^*_{\mathsf{API} \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}} \mathcal{S}$, we have that $\mathcal{S}$ satisfies property* **Sec**.

Proof: (sketch) We first start by adding more power to the intruder, providing him access to any value k for which there exists a handle $h_a^\alpha(n, k, i, S)$ where

some participant of S is dishonest, even if a is honest, meaning that the value k is stored on non corrupted API. Formally, we write $\mathcal{S} \vdash^* t$ when $\bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \cup \{k \mid h_a^\alpha(n,k,i,S) \in \mathcal{S}, S \not\subseteq H, a \in \mathsf{Agent}\} \vdash t$.

We then consider a stronger version of property **Sec**.

$$\forall a \in \mathsf{Agent}, \forall x,y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r,g\}, \forall S \subseteq H$$
$$\mathcal{S} \vdash^* h_a^\alpha(x,y,i,S) \Rightarrow \mathcal{S} \not\vdash^* y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \quad (\textbf{Sec*})$$

The key of the proof consists in showing that **Sec*** together with the two following properties are invariant by application of rules of $\mathsf{API} \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$:

$$\forall n,k,m_1,\ldots,m_p \in \mathsf{Msg}, \forall i,i_1,\ldots,i_p \in \{0,1,2,3\},$$
$$\forall \alpha \in \{r,g\}, \forall j \text{ s.t. } i_j \geq 1, \text{ and } S_j \subseteq H, \forall S \subseteq H$$
$$\mathcal{S} \vdash^* \{i_1,S_1,m_1,\ldots,i_p,S_p,m_p\}_k \text{ and } \mathcal{S} \vdash^* h_a^\alpha(n,k,i,S) \Rightarrow$$
$$m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and}$$
$$\exists n_j \in \mathsf{Nonce}, \exists b \in \mathsf{Agent}, \exists \alpha' \in \{r,g\},$$
$$\mathcal{S} \vdash^* h_b^{\alpha'}(n_j,m_j,i_j,S_j) \quad (\textbf{Enc})$$

$$\forall k,m_1,\ldots,m_p \in \mathsf{Msg}, \forall i_1,\ldots,i_p \in \{0,1,2,3\},$$
$$\forall j \text{ s.t. } i_j = 0$$
$$\mathcal{S} \vdash^* \{i_1,S_1,m_1,\ldots,i_p,S_p,m_p\}_k \Rightarrow \mathcal{S} \vdash^* m_j \quad (\textbf{Enc0})$$

Theorem 1 then easily follows since any initial state satisfies the three properties **Sec***, **Enc** and **Enc0** and property **Sec** is an immediate consequence of property **Sec***.

6 Security of the API under compromised handles

We have seen in the previous section that our API protects any data for which there is an honest handle $h_a^\alpha(n,k,i,S)$ with $S \subseteq H$. Imagine that some secret data is accidentally leaked to the attacker, possibly using a brute force attack or some other means. So, the attacker knows both $h_a^\alpha(n,k,i,S)$ and k. Then the attacker can learn any data of security level strictly smaller than the security level i of k, stored by the API of a, for which he has a handle $h_a^{\alpha'}(n',k',j,S')$ with $j < i$, $S \subseteq S'$. Indeed, the attacker can use the encryption command of the API

$$\mathsf{Encrypt} \quad h_a^\alpha(n,k,i,S) \quad h_a^{\alpha'}(n',k',j,S')$$

and obtain the cyphertext $\{j,S',k'\}_k$ thus k'. Note that this attack requires the attacker to control the API of a and only allows handles of strictly lower security level to be compromised. Even so, this situation is not completely satisfactory.

Thus we assume that (honest) agents periodically erase from the API any handle that corresponds to a data of a security level strictly lower than 3.

Since data of security level 2 are typically short-term session key and data of security level 1 are typically nonces, it makes sense to refresh them periodically. Formally, we say that a state $\mathcal{S}$ is *refreshed* if $\mathcal{S}_{\mathsf{int}} \subseteq \mathsf{Msg}$ is any set of messages and if for any $a \in H$, the set $\mathcal{S}_a$ only contains handles of the form $h_a^\alpha(n, k, 3, S)$ with $n \in \mathsf{Nonce}$, $k \in \mathsf{Nonce} \cup \mathsf{Key}$ and such that k only (possibly) appears in $\mathcal{S}$ in key position[1]. Note that we do not make any assumption on the states of compromised agents (besides that keys of level 3 only appear in key position).

This is however still not sufficient to guarantee the security of the API in case the attacker is able to learn old keys. Indeed, assume that an attacker knows a cyphertext $\{j, S', k'\}_k$ where k is a long-term (honest) key (of security level 3) such that he also knows k' (possibly using brute force attacks) of security level 2. For every (honest) agent a that has access to k using some handle of the form $h_a^r(n, k, 3, S)$, the attacker can register k' using the decryption command of the API if$a$.

$$\mathsf{Decrypt} \quad h_a^r(n, k, 3, S) \quad \{j, S', k'\}_k$$

The attacker then learns $h_a^\alpha(n', k', 2, S')$, a fresh handle that refers to k', which allows him to mount the previous attacks, again allowing the attacker to learn any data of security level 1 stored by the TRD of a. This corresponds a classical replay attack. Intuitively, since our API can be used to implement a protocol subject to replay, it suffers from replay attack as well.

To prevent such replay attacks, we reinforce the security of the API by restricting the use of decryption rules: the API should allow decryption with keys of level 3 only if at least one component is checked for freshness. In particular, our restricted API will not allow the implementation of protocols subject to this form of replay attack. Formally this corresponds to considering only decryption rules of the form

$$K_a(h_a^\alpha(X_n, X_k, i_0, S_0)), K_a(\{m_1, \ldots, m_p\}_{X_k}), \bigcup_{j \in L} K_a(m'_j) \overset{N_1,\ldots,N_p}{\Rightarrow} \bigcup_{j \notin L} K_a(m'_j)$$

where $J \neq \emptyset$ if $i_0 = 3$ (and all the other conditions of the decryption rule are fulfilled). Let API^r be the set of rules obtained from API by removing the decryption rules where J is empty when $i_0 = 3$.

Our restricted API preserves secrecy of its confidential values, even when the attacker is able to learn old keys and to control honest APIs, provided honest agents refresh the data in their TRDs.

Theorem 2 *Let $\mathcal{S}_0$ be an refreshed state. Then for any state $\mathcal{S}$, accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \rightarrow^*_{\mathsf{API}^r \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}} \mathcal{S}$, we have that $\mathcal{S}$ satisfies property* **Sec**.

Proof: Let $\mathcal{S}_0$ be an refreshed state. We define Fresh to be the set of *fresh* values, that is the set of nonces and keys that do not occur in $\mathcal{S}_0$. As for the proof of Theorem 1, we first re-enforce the properties that are invariant under

[1] That is, whenever k occurs at position p in a message t of $\mathcal{S}$, then $p = p'.2$ and $t|_{p'} = \{t'\}_k$.

$\mathsf{API}^r \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$. We consider the three following properties.

$$\forall a \in \mathsf{Agent}, \forall x, y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall S \subseteq H, \forall \alpha \in \{r,g\}$$
$$\mathcal{S} \vdash^* h_a^\alpha(x,y,i,S) \quad \Rightarrow \quad \mathcal{S} \not\vdash^* y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce}$$
$$\text{and in case } i \neq 3 \text{ then } y \in \mathsf{Fresh} \quad (\textbf{SecFresh*})$$

$$\forall n, k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i, i_1, \ldots, i_p \in \{0,1,2,3\},$$
$$\forall \alpha \in \{r,g\}, \forall j \text{ s.t. } i_j \geq 1, \text{ and } S_j \subseteq H, \forall S \subseteq H$$
$$\mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \text{ and } \mathcal{S} \vdash^* h_a^\alpha(n,k,i,S) \Rightarrow$$
$$(m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and}$$
$$\exists n_j \in \mathsf{Nonce}, b \in \mathsf{Agent}, \exists \alpha' \in \{r,g\},$$
$$\mathcal{S} \vdash^* h_b^{\alpha'}(n_j, m_j, i_j, S_j))$$
$$\text{or } \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \in \mathcal{S}_0 \quad (\textbf{Enc'})$$

$$\forall k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i_1, \ldots, i_p \in \{0,1,2,3\},$$
$$\forall j \text{ s.t. } i_j = 0$$
$$\mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \Rightarrow$$
$$\mathcal{S} \vdash^* m_j \text{ or } \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \in \mathcal{S}_0 \quad (\textbf{Enc0'})$$

We can show by inspection of the rules that these three properties are invariant under application of the rules of $\mathsf{API}^r \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$. Theorem 2 then easily follows since any refreshed state satisfies the three properties **SecFresh***, **Enc'** and **Enc0'** and property **Sec** is an immediate consequence of property **SecFresh***.

7 Results

We have tested our implementation on all the key distribution protocols in section 6.3 of the Clark-Jacob survey, excepting those which rely on synchronised clocks and timestamps for freshness. We summarise the results in Table 1 - full details are available at `http://www.lsv.ens-cachan.fr/~steel/GenericAPI`. The results illustrate how the properties we are able to guarantee by the use of our API translate to the properties of the protocols that can be implemented. Needham-Schroeder Symmetric Key can be implemented by API but not API^r, and indeed is subject to a replay attack. The amended version can be implemented by API^r, and has no known attack. The Otway-Rees protocol has a known type attack, which would be avoided by the tagged encryption scheme used by our API since in particular agent identities are included in every encryption. Yahalom cannot be implemented by API^r. The missing test is reported for the final message to B. At first sight this would seem to indicate inadequate functionality in our API, since B is supposedly assured the freshness of the session key by the fact that A has used it to encrypt B's nonce in a separate packet. However, this missing test can in fact be exploited by a malicious party playing A's role in the protocol to force B to accept an old key [13]. Carlsen's protocol has no known attack. Woo-Lam has a known parallel session attack, but this exploits a type flaw which our encryption scheme would prevent.

Protocol (section in Clark-Jacob)	API	API^r
Needham-Schroeder SK (6.3.1)	+	-
NSSK amended version (6.3.4)	+	+
Otway-Rees (6.3.3)	+	+
Yahalom (6.3.6)	+	-
Carlsen (6.3.7)	+	+
Woo-Lam Mutual Auth (6.3.11)	+	+

Table 1: Implementation of some protocols. *A + indicates an implementation of the protocol was found by our algorithm in section 4. A - indicates the algorithm reported a missing test.*

8 Conclusions

We have presented a generic API that can be used to implement many symmetric key protocols, and we have proved its security no matter what protocol has been implemented, and no matter how the attacker uses the API. If an attacker might be able to learn old secret values, our API should be switched to a restricted mode, in which case fewer protocols can be implemented, but protection against replay attacks is enforced.

Although our API is limited to symmetric key cryptography and a particular notion of freshness checking which may not accomodate all correct protocols, we believe we have established that it is possible to construct a secure API with a satisfactory level of generality by examining the protocols it is supposed to implement. Extensions to asymmetric cryptography, signatures, PKI certificates, and more exotic properties such as correctness of timestamps (vital for some applications [14]) remain as future work.

As we mentioned in the introduction, most previous work on analysis of security APIs has resulted in the discovery of flaws in existing schemes. Some positive results include the verification of various fixes of the IBM CCA in a bounded model for a particular security property (the secrecy of PINs) [7, 8]. Forthcoming work by the second author currently includes the verification of the secrecy of sensitive keys for a small subset of PKCS#11 (with certain modifications) in an unbounded model [10]. This API includes no freshness checking and no correspondance between keys and agents, so could not hope to enforce the kinds of properties we have specified here. However, it does offer the possibility of updating long-term keys, something we have yet to tackle for our API.

References

[1] Council regulation (ec) no 2252/2004: on standards for security features and biometrics in passports and travel documents issued by member states, December 2004. Available at `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2004:385:0001:0006:EN:PDF`.

[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] U. Carlsen. Optimal privacy and authentication on a portable communications system. *SIGOPS Oper. Syst. Rev.*, 28(3):16–23, 1994.

[4] *CCA Basic Services Reference and Guide*, Oct. 2006. Available online at `www.ibm.com/security/cryptocards/pdfs/bs327.pdf`.

[5] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Available via `http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz`, 1997.

[6] J. Clulow. On the security of PKCS#11. In *Proceedings of the 5th International Worshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425, Cologne, Germany, 2003. Springer.

[7] V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of XOR-based key management schemes. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 538–552, Braga, Portugal, 2007. Springer.

[8] 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.

[9] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[10] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Proceedings of ARSPA-WITS '09*, 2009. To appear.

[11] IBM Comment on "A Chosen Key Difference Attack on Control Vectors", Jan. 2001. Available from `http://www.cl.cam.ac.uk/~mkb23/research.html`.

[12] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

[13] A. Perrig and D. Song. Looking for diamonds in the desert. In *Proc. of the 13th Computer Security Foundations Workshop (CSFW'00)*, pages 64–76. IEEE Computer Society Press, 2000.

[14] M. Raya and J.-P. Hubaux. Securing vehicular ad hoc networks. *Journal of Computer Security*, 15(1):39–68, 2007.

[15] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.

[16] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proc. of the 14th Computer Security Foundations Workshop (CSFW'01)*, pages 174–190, Cape Breton, Nova Scotia, Canada, 2001. IEEE Computer Society Press.

A Proof of Theorem 1

We first remark that the intruder does not really increase when he learns handles or un-decryptable cyphertext.

Lemma 1 *Let S and S' be set of terms such that $S' = S \cup \{\{m\}_k\} \cup \mathsf{Hdls}$ where $\mathsf{Hdls} \subseteq \mathsf{Handle}$ is a set of handles and k is not deducible: $S \not\vdash^* k$. Let $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$ be an atomic value. Then*

$$S' \vdash^* u \quad \textit{iff} \quad S \vdash^* u \textit{ or } u \in \mathsf{Hdls}.$$

Moreover, Let $v \in \mathsf{Msg}$ and $w \in \mathsf{Key}$. Then

$$S' \vdash^* \{v\}_w \textit{ and } S' \not\vdash^* w \quad \textit{iff} \quad S \vdash^* \{v\}_w \textit{ or } \{v\}_w = \{m\}_k.$$

The proof mainly relies on the fact that keys are atomic.

We are now ready to show that properties **Sec***, **Enc** and **Enc0** are invariant under application of the rules of the API. Let $\mathcal{S}$ be a state satisfying **Sec***, **Enc** and **Enc0** and consider a state $\mathcal{S}'$ such that $\mathcal{S} \rightarrow_{\mathsf{API} \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}} \mathcal{S}'$. Let us show that $\mathcal{S}'$ satisfies **Sec***, **Enc** and **Enc0**

Let us first notice that properties **Sec***, **Enc** and **Enc0** are clearly invariant under application of the rules of $\mathsf{INTRUDER} \cup \mathsf{CONTROL}$. Indeed, if $\mathcal{S} \rightarrow_{\mathsf{INTRUDER} \cup \mathsf{CONTROL}} \mathcal{S}'$ then for any term u, we have $S' \vdash^* u$ if and only if $S \vdash^* u$. For the last rule of $\mathsf{CONTROL}$, this is due to the fact that it is easy to notice that whenever an accessible state $\mathcal{S}$ is such that $\mathcal{S} \vdash h_a^\alpha(n, k, i, S)$ then $a \in S$. Thus whenever $h_b^\alpha(n, k, i, S) \in \mathcal{S}$ with $b \notin H$ then $S \not\subseteq H$ thus $\mathcal{S} \vdash^* h_b^\alpha(n, k, i, S)$.

Thus we now assume $\mathcal{S} \rightarrow_{\mathsf{API}} \mathcal{S}'$ and we consider three cases depending on the rule that has been applied.

Generation rule. $a \in S \subseteq \mathsf{Agent}$

$$\begin{array}{ll} \stackrel{N,K}{\Rightarrow} & K_a(h_a^g(N, K, i, S)) \text{ and } i \in \{1, 2\} \\ \stackrel{N,K}{\Rightarrow} & K_a(K), K_a(h_a^g(N, K, 0, \emptyset)) \end{array}$$

$\mathcal{S}' = \mathcal{S} \cup \{h_a^g(n, k, i, S)\}$ and $i \in \{1, 2\}$ or $\mathcal{S}' = \mathcal{S} \cup \{k, h_a^g(n, k, 0, \emptyset)\}$. where n is a fresh nonce and k is a fresh key. Using Lemma 1, we easily check that $\mathcal{S}$ satisfies properties **Sec***, **Enc** and **Enc0** implies $\mathcal{S}'$ satisfies properties **Sec***, **Enc** and **Enc0**.

Encryption rule.

$$K_a(h_a^\alpha(X_n, X_k, i_0, S_0)), K_a(m_1), \ldots, K_a(m_p) \Rightarrow K_a(\{m'_1, \ldots, m'_p\}_{X_k})$$

We have $m_i\theta \in \mathcal{S}$, $h_a^\alpha(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$ and $\mathcal{S}' = \mathcal{S} \cup \{(\{m'_1, \ldots, m'_p\}_{X_k})\theta\}$ where the m_i and m'_i are defined as in rule **Encrypt**. We distinguish between two cases.

- Either $\mathcal{S} \vdash^* X_k\theta$. In that case, since $\mathcal{S} \vdash^* h_a^\alpha(X_n, X_k, i_0, S_0)\theta$, property **Sec*** ensures that $S_0 \not\subseteq H$. We deduce that for any $1 \leq i_j \leq 3$, we have $S_j \not\subseteq H$ thus $m_j\theta = h_a^\alpha(n_j, k_j, i_j, S_j)$ and $\mathcal{S} \vdash^* k_j$. For any $i_j = 0$, we have $m_j\theta \in \mathcal{S}$. Thus in both cases, we deduce that $\mathcal{S} \vdash^* m'_j\theta$ for any $1 \leq i \leq p$. This ensures that $\mathcal{S}' \vdash^* u$ if and only if $\mathcal{S} \vdash^* u$ for any term or handle u. We deduce that $\mathcal{S}$ satisfies properties **Sec***, **Enc** and **Enc0** implies $\mathcal{S}'$ satisfies properties **Sec***, **Enc** and **Enc0**.

- Or $\mathcal{S} \not\vdash^* X_k\theta$. Thus it must be the case that $S_0 \subseteq H$ (otherwise $\mathcal{S} \vdash^* h_a^\alpha(X_n, X_k, i_0, S_0)\theta$ implies $\mathcal{S} \vdash^* X_k\theta$). Moreover, applying Lemma 1, we get that $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$ (*).

  Property **Sec***: Assume $\mathcal{S}' \vdash^* h_a^\alpha(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $E_1 \subseteq H$. Then (*) implies $\mathcal{S} \vdash^* h_a^\alpha(n_1, k_1, l_1, E_1)$. Thus Property **Sec*** implies $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$. Thus applying (*) again, we get $\mathcal{S}' \not\vdash^* k_1$. We conclude that $\mathcal{S}'$ satisfies Property **Sec***.

  Property **Enc**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $\mathcal{S}' \vdash^* h_a^\alpha(n, k, i, S)$ with $S, S_j \subseteq H$ and $i_j \geq 1$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we conclude using Lemma 1 and the fact that $\mathcal{S}$ satisfies Property **Enc**. Otherwise, by Lemma 1, we must have $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k = \{m'_1, \ldots, m'_p\}_{X_k}\theta$. Thus $S_j = S'_j$ and $m_j\theta = h_a^\alpha(n_j, u_j, i_j, S_j) \in \mathcal{S}$. Thus $\mathcal{S}' \vdash^* h_a^\alpha(n_j, u_j, i_j, S_j)$ thus $\mathcal{S}'$ satisfies Property **Enc**.

  Property **Enc0**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i_j = 0$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0**. If $\mathcal{S}' \vdash^* k$ then we of course have $\mathcal{S}' \vdash^* u_j$. Otherwise, by Lemma 1, we must have $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k = (\{m'_1, \ldots, m'_p\}_{X_k})\theta$. Thus $S_j = S'_j$ and $m_j\theta = u_j \in \mathcal{S}$. Thus $\mathcal{S}' \vdash^* u_j$ and $\mathcal{S}'$ satisfies Property **Enc0**.

Decryption rule.

$$K_a(h_a^\alpha(X_n, X_k, i_0, S_0)), K_a(\{m_1, \ldots, m_p\}_{X_k}), \bigcup_{j \in J} K_a(m'_j) \overset{N_1, \ldots, N_k}{\Rightarrow} \bigcup_{j \notin J} K_a(m'_j)$$

We have $\{m_1, \ldots, m_k\}_{X_k}\theta \in \mathcal{S}$, $h_a^\alpha(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$, $m'_j\theta \in \mathcal{S}$ for any $j \in J$ and $\mathcal{S}' = \mathcal{S} \cup \{m'_j\theta \mid j \notin J\}$ where the m_i and m'_i are defined as in rule **Encrypt**. For any j such that $i_j = 0$, Property **Enc0** ensures that $\mathcal{S} \vdash^* m'_j$. For any j such that $i_j \geq 1$, then m'_j is a fresh handle. Thus we can deduce from Lemma 1 that for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$, we have $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ or $u = m'_j$ for some j such that $i_j \geq 1$ (**).

Property **Sec***: Assume $\mathcal{S}' \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $E_1 \subseteq H$. Then (**) implies $\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ or $h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j$ for some j such that $i_j \geq 1$. In the first case ($\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$), Property **Sec*** ensures $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In the second case ($h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j$) then $m_j = k_1$ and property **Enc** ensures that there exists $n' \in \mathsf{Nonce}$, $c \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n', k_1, l_1, E_1)$.

Property **Sec*** on $\mathcal{S}$ ensures that $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In both cases we conclude that $\mathcal{S}'$ satisfies Property **Sec***.

Property **Enc**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $\mathcal{S}' \vdash^* h_b^{\alpha'}(n, k, i, S)$ with $S, S'_j \subseteq H$ and $i_j \geq 1$. Property **Sec*** on $\mathcal{S}'$ ensures that $\mathcal{S}' \not\vdash^* k$. Thus by Lemma 1, we must have $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ Moreover, (**) implies that either $\mathcal{S} \vdash^* h_b^{\alpha'}(n, k, i, S)$ or $h_b^{\alpha'}(n, k, i, S) = m'_j$ for some $l \notin L$. In the first case, we conclude using the fact that $\mathcal{S}$ satisfies Property **Enc** thus $u_j \in \mathsf{Nonce} \cup \mathsf{Key}$ and there exists $n_j \in \mathsf{Nonce}$ and $c \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n_j, u_j, i'_j, S'_j)$, which implies $\mathcal{S}' \vdash^* h_c^{\alpha''}(n_j, u_j, i'_j, S'_j)$ thus $\mathcal{S}'$ satisfies Property **Enc**. In the second case, we have $\{m_1, \ldots, m_k\}_{X_k}\theta \in \mathcal{S}$ and $h_a^{\alpha}(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$. Since $\mathcal{S}$ enjoys Property **Enc**, we deduce that there exists $n_l \in \mathsf{Nonce}$ and $c \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n_l, k, i_l, S_l)$, that is $\mathcal{S} \vdash^* h_c^{\alpha''}(n_l, k, i, S)$. Since $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, Property **Enc** on $\mathcal{S}$ ensures that $u_j \in \mathsf{Nonce} \cup \mathsf{Key}$ and that there exists $n'_j \in \mathsf{Nonce}$ and $d \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_d^{\alpha'''}(n'_j, u_j, i'_j, S'_j)$. We can deduce that $\mathcal{S} \vdash^* h_d^{\alpha'''}(n'_j, u_j, i'_j, S'_j)$ thus $\mathcal{S}'$ satisfies Property **Enc**.

Property **Enc0**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i_j = 0$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0**. Otherwise, by Lemma 1, we must have $\mathcal{S}' \vdash^* k$ thus $\mathcal{S}' \vdash^* u_j$.

B Proof of Theorem 2

Let $\mathcal{S}$ be a state satisfying **SecFresh***, **Enc'** and **Enc0'** and consider a state $\mathcal{S}'$ such that $\mathcal{S} \rightarrow_{\mathsf{API} \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}} \mathcal{S}'$. Let us show that $\mathcal{S}'$ satisfies **SecFresh***, **Enc'** and **Enc0'**

Let us first notice that properties **SecFresh***, **Enc'** and **Enc0'** are clearly invariant under application of the rules of $\mathsf{INTRUDER} \cup \mathsf{CONTROL}$. Indeed, if $\mathcal{S} \rightarrow_{\mathsf{INTRUDER} \cup \mathsf{CONTROL}} \mathcal{S}'$ then for any term u, we have $S' \vdash^* u$ if and only if $S \vdash^* u$. For the last rule of $\mathsf{CONTROL}$, this is due to the fact that it is easy to notice that whenever an accessible state $\mathcal{S}$ is such that $\mathcal{S} \vdash h_a^{\alpha}(n, k, i, S)$ then $a \in S$. Thus whenever $h_b^{\alpha}(n, k, i, S) \in \mathcal{S}$ with $b \notin H$ then $S \not\subseteq H$ thus $\mathcal{S} \vdash^* h_b^{\alpha}(n, k, i, S)$.

Thus we now assume $\mathcal{S} \rightarrow_{\mathsf{API}^r} \mathcal{S}'$ and we consider three cases depending on the rule that has been applied.

Generation rules. $a \in S \subseteq \mathsf{Agent}$

$$\begin{array}{ll} \stackrel{N,K}{\Rightarrow} & K_a(h_a^g(N, K, i, S)) \text{ and } i \in \{1, 2\} \\ \stackrel{N,K}{\Rightarrow} & K_a(K), K_a(h_a^g(N, K, 0, \emptyset)) \end{array}$$

$\mathcal{S}' = \mathcal{S} \cup \{h_a^g(n, k, i, S)\}$ and $i \in \{1, 2\}$ or $\mathcal{S}' = \mathcal{S} \cup \{k, h_a^g(n, k, 0, \emptyset)\}$. where n is a fresh nonce and k is a fresh key. Using Lemma 1, we easily check that $\mathcal{S}$ satisfies properties **SecFresh***, **Enc'** and **Enc0'** implies $\mathcal{S}'$ satisfies properties **SecFresh***, **Enc'** and **Enc0'**.

Encryption rule.

$$K_a(h_a^{\alpha}(X_n, X_k, i_0, S_0)), K_a(m_1), \ldots, K_a(m_p) \Rightarrow K_a(\{m'_1, \ldots, m'_p\}_{X_k})$$

We have $m_i\theta \in \mathcal{S}$, $h_a^\alpha(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$ and $\mathcal{S}' = \mathcal{S} \cup \{(\{m'_1, \ldots, m'_p\}_{X_k})\theta\}$ where the m_i and m'_i are defined as in rule **Encrypt**. We distinguish between two cases.

- Either $\mathcal{S} \vdash^* X_k\theta$. In that case, since $\mathcal{S} \vdash^* h_a^\alpha(X_n, X_k, i_0, S_0)\theta$, property **SecFresh*** ensures that $S_0 \not\subseteq H$. We deduce that for any $1 \leq i_j \leq 3$, we have $S_j \not\subseteq H$ thus $m_j\theta = h_a^\alpha(n_j, k_j, i_j, S_j)$ and $\mathcal{S} \vdash^* k_j$. For any $i_j = 0$, we have $m_j\theta \in \mathcal{S}$. Thus in both cases, we deduce that $\mathcal{S} \vdash^* m'_j\theta$ for any $1 \leq i \leq p$. This ensures that $\mathcal{S}' \vdash^* u$ if and only if $\mathcal{S} \vdash^* u$ for any term or handle u. We deduce that $\mathcal{S}$ satisfies properties **SecFresh***, **Enc'** and **Enc0'** implies $\mathcal{S}'$ satisfies properties **SecFresh***, **Enc'** and **Enc0'**.

- Or $\mathcal{S} \not\vdash^* X_k\theta$. Thus it must be the case that $S_0 \subseteq H$ (otherwise $\mathcal{S} \vdash^* h_a^\alpha(X_n, X_k, i_0, S_0)\theta$ implies $\mathcal{S} \vdash^* X_k\theta$). Moreover, applying Lemma 1, we get that $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$ (*).

  Property **SecFresh***: Assume $\mathcal{S}' \vdash^* h_a^\alpha(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $E_1 \subseteq H$. Then (*) implies $\mathcal{S} \vdash^* h_a^\alpha(n_1, k_1, l_1, E_1)$. Thus Property **Sec*** implies $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ and moreover $k_1 \in \mathsf{Fresh}$ in case $l_3 \neq 3$. Thus applying (*) again, we get $\mathcal{S}' \not\vdash^* k_1$. We conclude that $\mathcal{S}'$ satisfies Property **SecFresh***.

  Property **Enc'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $\mathcal{S}' \vdash^* h_a^\alpha(n, k, i, S)$ with $S, S_j \subseteq H$ and $i_j \geq 1$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we conclude using Lemma 1 and the fact that $\mathcal{S}$ satisfies Property **Enc'**. Otherwise, by Lemma 1, we must have $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k = \{m'_1, \ldots, m'_p\}_{X_k}\theta$. Thus $S_j = S'_j$ and $m_j\theta = h_a^\alpha(n_j, u_j, i_j, S_j) \in \mathcal{S}$. Thus $\mathcal{S}' \vdash^* h_a^\alpha(n_j, u_j, i_j, S_j)$ thus $\mathcal{S}'$ satisfies Property **Enc'**.

  Property **Enc0'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i_j = 0$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0'**. If $\mathcal{S}' \vdash^* k$ then we of course have $\mathcal{S}' \vdash^* u_j$. Otherwise, by Lemma 1, we must have $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k = (\{m'_1, \ldots, m'_p\}_{X_k})\theta$. Thus $S_j = S'_j$ and $m_j\theta = u_j \in \mathcal{S}$. Thus $\mathcal{S}' \vdash^* u_j$ and $\mathcal{S}'$ satisfies Property **Enc0'**.

Decryption rule.

$$K_a(h_a^\alpha(X_n, X_k, i_0, S_0)), K_a(\{m_1, \ldots, m_p\}_{X_k}), \bigcup_{j \in J} K_a(m'_j) \overset{N_1, \ldots, N_k}{\Rightarrow} \bigcup_{j \notin J} K_a(m'_j)$$

We have $\{m_1, \ldots, m_p\}_{X_k}\theta \in \mathcal{S}$, $h_a^\alpha(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$, $m'_j\theta \in \mathcal{S}$ for any $j \in J$ and $\mathcal{S}' = \mathcal{S} \cup \{m'_j\theta \mid j \notin J\}$ where the m_i and m'_i are defined as in rule **Encrypt**. Since the decryption rule belongs to API^r, we must have that if $i_0 = 3$ then J is not empty.

Assume first that $S_0 \subseteq H$. If $i_0 \neq 3$ then Property **SecFresh*** ensures that $k \in \mathsf{Fresh}$, which ensures that $\{m_1, \ldots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. If $i_0 = 3$, then J is not empty. Let $j_0 \in J$, we have $m_{j_0} = i_{j_0}, S_{j_0}, w_{j_0}$. There exists n_{j_0} such that $h_a^g(n_{j_0}, w_{j_0}, i_{j_0}, S_{j_0}) \in \mathcal{S}$. Then Property **SecFresh*** ensures that $w_{j_0} \in \mathsf{Fresh}$, which ensures that $\{m_1, \ldots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. In both cases, if $S_0 \subseteq H$ we can conclude that $\{m_1, \ldots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$.

Moreover, for any j such that $i_j = 0$, Property **Enc0'** ensures that $\mathcal{S} \vdash^* m'_j$. For any j such that $i_j \geq 1$, then m'_j is a fresh handle. Thus we can deduce from Lemma 1 that for any $u \in \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key} \cup \mathsf{Handle}$, we have $\mathcal{S}' \vdash^* u$ iff $\mathcal{S} \vdash^* u$ or $u = m'_j$ for some j such that $i_j \geq 1$ (**).

Assume now that $S_0 \not\subseteq H$. Then $\mathcal{S} \vdash^* X_k\theta$ thus for any $1 \leq i \leq p$, $\mathcal{S} \vdash^* m_i$. In particular, for any j such that $i_j = 0$, $\mathcal{S} \vdash^* m'_j$. Thus we deduce that property (**) also holds.

Property **SecFresh***: Assume $\mathcal{S}' \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ with $l_1 \in \{1, 2, 3\}$, $E_1 \subseteq H$. Then (**) implies $\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$ or $h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j$ for some j such that $i_j \geq 1$. In the first case ($\mathcal{S} \vdash^* h_b^{\alpha'}(n_1, k_1, l_1, E_1)$), Property **SecFresh*** ensures $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ and $k_1 \in \mathsf{Fresh}$ is $l_1 \neq 3$. Thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In the second case ($h_b^{\alpha'}(n_1, k_1, l_1, E_1) = m'_j$) then $m_j = k_1$ and $S_0 \subseteq E_1 \subseteq H$ thus $\{m_1, \ldots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. Property **Enc'** then ensures that there exists $n' \in \mathsf{Nonce}$, $c \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n', k_1, l_1, E_1)$. Property **SecFresh*** on $\mathcal{S}$ ensures that $\mathcal{S} \not\vdash^* k_1$ and $k_1 \in \mathsf{Nonce} \cup \mathsf{Key}$ and $k_1 \in \mathsf{Fresh}$ is $l_1 \neq 3$. Thus (**) implies $\mathcal{S}' \not\vdash^* k_1$. In both cases we conclude that $\mathcal{S}'$ satisfies Property **SecFresh***.

Property **Enc'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $\mathcal{S}' \vdash^* h_b^{\alpha'}(n, k, i, S)$ with $S, S'_j \subseteq H$ and $i_j \geq 1$. Property **SecFresh*** on $\mathcal{S}'$ ensures that $\mathcal{S}' \not\vdash^* k$. Thus by Lemma 1, we must have $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ Moreover, (**) implies that either $\mathcal{S} \vdash^* h_b^{\alpha'}(n, k, i, S)$ or $h_b^{\alpha'}(n, k, i, S) = m'_j$ for some $l \notin L$. In the first case, we conclude using the fact that $\mathcal{S}$ satisfies Property **Enc'** thus $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k \in \mathcal{S}_0$ or $u_j \in \mathsf{Nonce} \cup \mathsf{Key}$ and there exists $n_j \in \mathsf{Nonce}$ and $c \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n_j, u_j, i'_j, S'_j)$, which implies $\mathcal{S}' \vdash^* h_c^{\alpha''}(n_j, u_j, i'_j, S'_j)$ thus $\mathcal{S}'$ satisfies Property **Enc'**. In the second case, we must have $i \neq 3$ and $S_0 \subseteq S \subseteq H$ thus $\{m_1, \ldots, m_p\}_{X_k}\theta \notin \mathcal{S}_0$. we have $\{m_1, \ldots, m_k\}_{X_k}\theta \in \mathcal{S}$ and $h_a^{\alpha}(X_n, X_k, i_0, S_0)\theta \in \mathcal{S}$. Since $\mathcal{S}$ enjoys Property **Enc'**, we deduce that there exists $n_l \in \mathsf{Nonce}$ and $c \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_c^{\alpha''}(n_l, k, i_l, S_l)$, that is $\mathcal{S} \vdash^* h_c^{\alpha''}(n_l, k, i, S)$. Since $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i \neq 3$, Property **SecFresh*** ensures that $f \in \mathsf{Fresh}$ thus $\{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k \notin \mathcal{S}_0$. Property **Enc'** on $\mathcal{S}$ thus ensures that $u_j \in \mathsf{Nonce} \cup \mathsf{Key}$ and that there exists $n'_j \in \mathsf{Nonce}$ and $d \in \mathsf{Agent}$ such that $\mathcal{S} \vdash^* h_d^{\alpha'''}(n'_j, u_j, i'_j, S'_j)$. We can deduce that $\mathcal{S} \vdash^* h_d^{\alpha'''}(n'_j, u_j, i'_j, S'_j)$ thus $\mathcal{S}'$ satisfies Property **Enc'**.

Property **Enc0'**: Assume $\mathcal{S}' \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$ and $i_j = 0$. If $\mathcal{S} \vdash^* \{i'_1, S'_1, u_1, \ldots, i'_p, S'_p, u_p\}_k$, we can easily conclude since $\mathcal{S}$ satisfies Property **Enc0'**. Otherwise, by Lemma 1, we must have $\mathcal{S}' \vdash^* k$ thus $\mathcal{S}' \vdash^* u_j$.

Contents

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399