A Generic Security API for Symmetric Key Management on Cryptographic Devices

Véronique Cortier[1] and Graham Steel[2]

[1] LORIA, Projet Cassis, CNRS & INRIA, cortier@loria.fr
[2] LSV, CNRS & INRIA & ENS de Cachan, Graham.Steel@inria.fr

Abstract. Security APIs are used to define the boundary between trusted and untrusted code. The security properties of existing APIs 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 our 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.

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 program are executed on some tamper resistant device (TRD), such as a smartcard, USB security token 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 a wide class of symmetric key protocols, 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 TRDs. 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 propose a formal model for a threat scenario where TRDs may sometimes be connected to a clean host machine, and sometimes to a corrupted one where the attacker can execute arbitrary code. Additionally, the attacker is assumed to have defeated the tamper resistance on some devices, obtaining the long term keys of some users. 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 (honest agents are those whose TRDs are intact). The property holds even when honest agents APIs are controlled by an attacker (in case e.g. an honest user's 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. It does not cover all notions of freshness, but 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.

2 Formal 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.

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

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 messages but provides the user with a handle that can be used later to indicate to the API to use a specific message. A handle $h_a^\alpha(n, m, i, S)$ represents a reference stored on the API belonging to a for a message m of security level i. The set S represents the set of users that are allowed to access to m. By convention, the special constant All will indicate public data. 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 m generated by the API ($\alpha = g$) from values m received by the API ($\alpha = r$). This distinction allows the API to check for freshness. The values stored inside the TRD will typically be nonces or keys. However, in order to reflect the inability of an TRD to check whether an arbitrary bitstring is a key or not, we *a priori* allow any message to be stored inside the TRD. 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 $\mathcal{P} = \{P_a \mid a \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$ of predicates. P_a with $a \in \mathsf{Agent}$ to represent the knowledge of an agent a. The predicate P_{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 $P_1(u_1), \ldots, P_k(u_k) \overset{N_1,\ldots,N_p}{\Longrightarrow} Q_1(v_1), \ldots, Q_l(v_l)$ 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 $\mathsf{INTRUDER}$ of rules represents the ability of an attacker to pair and project and to encrypt and decrypt when he knows the key.

$$\begin{aligned} P_{\mathsf{int}}(x), P_{\mathsf{int}}(y) &\Rightarrow P_{\mathsf{int}}(\langle x, y \rangle) \\ P_{\mathsf{int}}(\langle x, y \rangle) &\Rightarrow P_{\mathsf{int}}(x) \\ P_{\mathsf{int}}(\langle x, y \rangle) &\Rightarrow P_{\mathsf{int}}(y) \\ P_{\mathsf{int}}(x), P_{\mathsf{int}}(y) &\Rightarrow P_{\mathsf{int}}(\{x\}_y) \\ P_{\mathsf{int}}(\{x\}_y), P_{\mathsf{int}}(y) &\Rightarrow P_{\mathsf{int}}(x) \end{aligned}$$

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 $P_{a_1}(u_1), \ldots, P_{a_k}(u_k) \overset{N_1,\ldots,N_p}{\Longrightarrow} P_{b_1}(v_1), \ldots, P_{b_l}(v_l)$ 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 assume a tamper resistant device with a limited (but for the moment unspecified) amount of memory, capable of symmetric key cryptography. The device is to be deployed to facilitate the execution of symmetric key distribution protocols, and the subsequent use of the session keys established by these protocols. To that end, we design an API that allows users to manage secret data inside the 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, referring to the secrets by their handles. Our API has simply three commands: generation of new data, encryption and decryption. We present the rules in the language of our model (see section 2). To translate them informally to an imperative programming language, imagine each rule as a function, with input parameters on the left of the arrow, and the output returned to the right. Above the arrow are the fresh random values generated during function execution.

3.1 API rules

The API allows a user a to generate a new nonce or key K of security level $i \in \{0, 1, 2\}$ for a group $S \subseteq \mathsf{Agent}$ of agents.

$$\overset{N,K}{\Rightarrow} P_a(h^g_a(N, K, i, S)) \quad i \in \{1, 2\} \qquad \textbf{(Secure Generate)}$$

$$\overset{N,K}{\Rightarrow} P_a(K), P_a(h^g_a(N, K, 0, \mathsf{All})) \qquad \textbf{(Public Generate)}$$

where $N \in \mathsf{VarNonce}$, and $K \in \mathsf{VarNonce}$ if $i = 1$, $K \in \mathsf{VarKey}$ if $i = 2$. The agent gets in return a handle to the new value together with the value itself if the value is public.

An agent a can require the API to encrypt public data $x_1, \ldots, x_k$ together with secret data $y_1, \ldots, y_l$ using a key K. The agent a knows the key K only through a handle $h_a^\alpha(X_n, K, i_0, S_0)$ and the value y_j through a handle $h_a^{\alpha_j}(X_{n_j}, y_j, i_j, S_j)$.

$$\begin{aligned}&P_a(h_a^\alpha(X, K, i_0, S_0)), P_a(x_1), \ldots, P_a(x_n),\\ &\qquad P_a(h_a^{\alpha_1}(X_{n_1}, y_1, i_1, S_1)), \ldots, P_a(h_a^{\alpha_l}(X_{n_l}, y_l, i_l, S_l))\\ &\qquad\qquad \Rightarrow P_a(\{x_1, 0, \ldots, x_n, 0, y_1, i_1, S_1, \ldots, y_l, i_l, S_l\}_K) \quad (\textbf{Encrypt})\end{aligned}$$

The API encrypts the data adding for each data its security level together with the group of agents authorized to access to it. We require moreover that $i_0 > i_j$ (keys only encrypt data of strictly lower security level) and $S_0 \subseteq S_j$ to ensure that data are not transmitted to users that are not allowed to access to.

A user a can also request the API to decrypt messages for him using a key K, passed through the API using a handle $h_a^\alpha(X_n, K, i_0, S_0)$, checking equalities of some of the (public or private) components $x_1, \ldots, x_s, y_1, \ldots, y_r$ previously generated by the API. These tests can be used to ensure freshness of the message, as we will see in Section 6.

$$\begin{aligned}&P_a(h_a^\alpha(X, K, i_0, S_0)), P_a(\{x_1, 0, \ldots, x_k, 0, y_1, i_1, S_1, \ldots, y_l, i_l, S_l\}_K),\\ &\qquad P_a(h_a^g(X_1, x_1, 0, \mathsf{All})), \ldots, P_a(h_a^g(X_s, x_s, 0, \mathsf{All})),\\ &\qquad P_a(h_a^g(Y_1, y_1, i_1, S_1)), \ldots, P_a(h_a^g(Y_r, y_r, i_r, S_r))\\ &\qquad\qquad \overset{N_{r+1},\ldots,N_l}{\Rightarrow} P_a(x_{s+1}) \ldots, P_a(x_k),\\ &P_a(h_a^r(N_{r+1}, y_{r+1}, i_{r+1}, S_{r+1})), \ldots, P_a(h_a^r(N_l, y_l, i_l, S_l)) \quad (\textbf{Decrypt/Test})\end{aligned}$$

The user gets in return the decrypted public data that were not used in tests for equality and handles to the decrypted private data that were not used in tests for equality, provided that $i_0 > i_j$ (keys only encrypt data of strictly lower security level) and that $S_0 \subseteq S_j$ to enforce that data are not transmitted to users that are not allowed to access to.

For the sake of simplicity, we have given above a presentation of the rules such that public data are encrypted first in the **Encrypt** rule, and only the first values are tested for equalities in the **Decrypt/Test** rule. Of course the commands in fact have no restrictions on the order of their arguments. The full family of rules, representing the encryption and decryption commands of the API in complete generality, is displayed in Figure 1. The set of all the rules is denoted by API.

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

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

$$P_a(h_a^\alpha(X_n, X_k, i_0, S_0)), P_a(m_1), \ldots, P_a(m_n) \Rightarrow P_a(\{m'_1, \ldots, m'_n\}_{X_k}) \qquad \textbf{Encrypt}$$

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.
- $m'_j = X_{k_j}, i_j, S_j$ with $i_j < i_0$ and $S_0 \subseteq S_j$ if $m_j \in \mathsf{Handle}$ is a handle of the form $h_a^{\alpha_j}(X_{n_j}, X_{k_j}, i_j, S_j)$.

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

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 $j \in L$, $m'_j = h_a^g(X_{n_j}, X_j, 0, \mathsf{All})$ 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 $j \notin L$, $m'_j = x_j$ if m_j is of the form $x_j, 0$ (data of security level 0 are given to the user) and $m'_j = h_a^r(N_j, y_{k_j}, i_j, S_j)$ if m_j is of the form y_{k_j}, i_j, S_j with $i_j \geq 1$, $i_j < i_0$ and $S_0 \subseteq S_j$.

Fig. 1. Complete description of rules for encryption and decryption.

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 $h_a^r(n'_{KAS}, k_{as}, 3, \{a, s\})$ to the key k_{as} of level 3. The agent a can execute its first protocol's rule by using the following API command:

$$\overset{N, N_A}{\Rightarrow} P_a(N_A), P_a(h_a^g(N, N_A, 0, \mathsf{All}))$$

where N, N_A are nonce variables. a obtains both a fresh (public) nonce N_A and a handle $h_a^g(N, N_A, 0, \mathsf{All})$ 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 $L = \{1\}$, that is the first component should be checked):

$$\begin{aligned} P_a(h^r_a(N'_{KAS}, K_{as}, 3, \{a, s\})), P_a(\{N_A, 0, y, 0, x, 2, \{a, b, s\}\}_{K_{as}}),& \\ P_a(h^g_a(N, N_A, 0, \mathsf{All}))& \\ \overset{N'}{\Rightarrow} P_a(y), P_a(h^r_a(N', x, 2, \{a, b, s\}))& \end{aligned}$$

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 $P_A(h^r_A(N', x, 2, \{a, b, s\}))$ 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 $L = \{1\}$, that is the first component should be checked):

$$P_a(h^r_a(N', K_{ab}, 2, \{a, b, s\})), P_a(\{N_A, 0\}_{K_{ab}}), P_a(h^g_a(N, N_A, 0, \mathsf{All})) \Rightarrow$$

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.

$$P_a(h^r_a(N', K_{ab}, 2, \{a, b, s\})), P_a(x_3) \Rightarrow P_a(\{x_3, 0\}_{K_{ab}})$$

where N', K_{ab} are variables.

3.2 Comparison with PKCS#11

The most widely used API for TRDs is the RSA standard PKCS#11, also known as 'Cryptoki' [15]. PKCS#11-based APIs have been shown to be vulnerable to a variety of attacks whereby sensitive keys are compromised [6, 9]. Our API has several features designed specifically to counter these kinds of threats. Firstly, we insist on an encryption scheme whereby data from the host machine and secret data from inside the TRD are tagged differently when encrypted to avoid confusion. PKCS#11 does not do this, and this confusion is exploited by many of the known attacks. Secondly we insist that keys are stored with specific roles, either as session keys or long term keys, and these roles cannot be changed. Allowing the roles of keys to change (signified by their *attributes* in PKCS#11) is another major source of vulnerabilities in the Cryptoki API. Finally, we store the identities of agents for whom a key is intended to be used inside the TRD, and include these identities as tags in our encryption scheme. PKCS#11 makes no such provision, but it seems necessary in order to obtain security properties which are preserved when some TRDs are compromised.

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

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

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 $\bar{t}$ the term obtained from t by replacing each subterm $\{u\}_v$ of t by the variable $X_{\{u\}_v}$. The function $\bar{\cdot}$ 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

$h^r_a(N_{Kas}, kas, \mathit{3}, \{a, s\})$ `% A handle for kas`
$h^r_b(N_{Kbs}, kbs, \mathit{3}, \{b, s\})$ `% B handle for kbs`
$h^g_s(N'_{Kas}, kas, \mathit{3}, \{a, s\})$ `% S handle for kas`
$h^g_s(N'_{Kbs}, kbs, \mathit{3}, \{b, s\})$ `% 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 $\{m_1, \dots, m_p\}_{Xk}$ occurring in u:

Retrieve $h^{\alpha}_{A}(N, X_k, j, \mathit{Set})$ 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 $m_i = n(A, X, I, \mathit{Set})$ and $h^{g}_{A}(N', X, I, \mathit{Set})$ is in the handle store and set $L = [P_A(h^{g}_{A}(N', X, I, \mathit{Set}))]$. 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

$$P_A(h^{\alpha}_{A}(N, X_k, j, \mathit{Set})), P_A(\{\overline{\mathsf{un}(m_1)}, \dots, \overline{\mathsf{un}(m_p)}\}_{X_k}), L \overset{N_1,\dots,N_p}{\Rightarrow} \bigcup_{j \neq i} P_A(m'_i)$$

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

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

$$\overset{N,X}{\Rightarrow} P_A(X), P_A(h^{g}_{A}(N, X, L, \mathit{Set}))$$

Add $h^{g}_{A}(N, X, 0, \mathit{Set})$ to the handle store H.

For each $n(A, X, i, \mathit{Set}) \in \mathcal{N}$, $i \in \{1, 2\}$, add generate command

$$\overset{N,X}{\Rightarrow} P_A(h^{g}_{A}(N, X, i, \mathit{Set}))$$

Add $h^{g}_{A}(N, X, i, \mathit{Set})$ to the handle store H.

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

Retrieve $h^{\alpha}_{A}(N, X_k, i, \mathit{Set})$ from the handle store H.

Add encryption command of the form

$$P_A(h^{\alpha}_{A}(N, X_k, i, S)), P_A(m'_1), \dots, P_A(m'_k) \Rightarrow P_A(\{\overline{\mathsf{un}(m_1)}, \dots, \overline{\mathsf{un}(m_k)}\}_{X_k})$$

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

Consider the role of A in the Carlsen's Secret Key Initiator Protocol. Using our algorithm, we retrieve the API commands presented in example 2.

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

Recall that our API is designed to be used on a device which may sometimes be connected to a corrupted host machine, and sometimes to a 'clean' machine. When all machines involved in a run of a protocol are 'clean' the formal threat model reduces to the so-called Dolev-Yao model: all network traffic goes through the intruder, but computations on honest users' machines remain secure. In this case, our API merely implements the protocol, and does not provide extra security. We are interested in what guarantees our API offers when one or more of the machines involved in a protocol run are corrupted, but the TRDs are still intact. If a host machine is corrupted, then all the public data on the machine (level 0 terms in our model) is assumed to be lost. We want to show that secret terms stored on the device (level ≥ 1) remain secret. Further, we want to show that session keys established while the device was connected to the corrupted machine can still be trusted, even if some (other) session keys have been lost. These simple to state properties give (we claim) an intuitively easy to understand security policy for the API. Furthermore, they are precisely the properties that are violated by previously discovered attacks on existing APIs [9, 10]. We will prove that our API preserves these properties.

We first give a precise formal model of the threat scenario. 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*.

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 long-term secret values of some compromised users (by defeating the tamper resistance of their devices or some other means). The only trusted secure parts are the secure storage components (TRDs) of the honest users, managed by the API (see Figure 2). This can be easily modeled by adding the following set $\mathsf{CONTROL}$ of rules

$$P_a(x) \Rightarrow I(x) \tag{1}$$

$$I(x) \Rightarrow P_a(x) \tag{2}$$

$$P_b(h_b^\alpha(x, y, i, S)) \Rightarrow I(y) \tag{3}$$

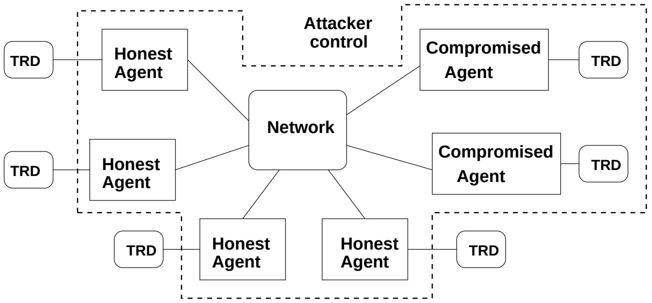

Fig. 2. Threat model. The attacker controls the network, all machines, and has obtained access to the memory of some compromised agents' TRDs

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 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 compromised agent. Given a state $\mathcal{S}$ of our execution model and by abuse of notation, we write $t \in \mathcal{S}$ (resp. $\mathcal{S} \vdash t$) instead of $t \in \bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b$ (resp. $\bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \vdash t$).

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:

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

This also ensures that whenever a value k is stored for a set S of honest users, then k is indeeed a key or a nonce.

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

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

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

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

$$\begin{gathered}\forall n, k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i, i_1, \ldots, i_p \in \{0, 1, 2, 3\}, \forall \alpha \in \{r, g\}, \forall j\\ i_j \geq 1, S_j \subseteq H, \forall S \subseteq H, \mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k, \mathcal{S} \vdash^* h_a^\alpha(n, k, i, S) \Rightarrow\\ m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and}\\ \exists n_j \in \mathsf{Nonce}, \exists b \in \mathsf{Agent}, \exists \alpha' \in \{r, g\}, \mathcal{S} \vdash^* h_b^{\alpha'}(n_j, m_j, i_j, S_j) \quad (\textbf{Enc})\end{gathered}$$

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

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[3] whenever $S \subseteq H$. Note that we do not make any assumption on the states of compromised agents (besides that honest 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 of 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

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

where J must not be the emptyset whenever $i_0 = 3$ (and all the other conditions of the decryption rule of Figure 1 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 have refreshed the data in their TRDs.

Theorem 2. *Let $\mathcal{S}_0$ be a 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 a 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

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

proof of Theorem 1, we first re-enforce the properties that are invariant under $\mathsf{API}^r \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$. We consider the three following properties.

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

$$\begin{aligned}&\forall n, k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i, i_1, \ldots, i_p \in \{0,1,2,3\}, \forall \alpha \in \{r, g\}, \forall j \\ &i_j \geq 1, S_j \subseteq H, \forall S \subseteq H, \mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k, \mathcal{S} \vdash^* h_a^\alpha(n, k, i, S) \Rightarrow \\ &(m_j \in \mathsf{Key} {\cup} \mathsf{Nonce} \text{ and } \exists n_j \in \mathsf{Nonce}, b \in \mathsf{Agent}, \exists \alpha' \in \{r, g\}, \mathcal{S} \vdash^* h_b^{\alpha'}(n_j, m_j, i_j, S_j)) \\ &\qquad\qquad \text{or } \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \in \mathcal{S}_0 \quad (\textbf{Enc'})\end{aligned}$$

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

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***.

Note that our freshness condition does not require agents to erase their data after each session. Intuitively, refreshment should occur only when a leak from an honest user is suspected or when keys have been stored and used for a sufficient time to allowing brute force attacks. Thus refreshment could occur every hour, day, week, month or year depending on application-specific factors.

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 $\mathsf{API}^{\mathsf{r}}$, and indeed is subject to a replay attack. The amended version can be implemented by $\mathsf{API}^{\mathsf{r}}$, 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 $\mathsf{API}^{\mathsf{r}}$. 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	$\mathsf{API}^{\mathsf{r}}$
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. API is the original API (see section 3), and $\mathsf{API}^{\mathsf{r}}$ is the restricted API where we insist on at least one test for every new session key (see section 6). 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 for a tamper-resistant device that can be used to implement many symmetric key protocols. We have proved vital security properties of the API 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 accommodate 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, etc. remain as future work. Recently, proposals have emerged for TRDs containing real-time clocks, e.g. for automotive applications [14]. Modelling these with an API allowing timestamp-based freshness to be checked would be an interesting future project. Note also that all our proofs are in the so-called 'symbolic model', where encryption is treated as a black box function on terms. We intend to investigate the extension of our results to more precise computational models of security.

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 correspondence 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. IEEE Computer Society Press, 2001.