Analysing PKCS#11 Key Management APIs with Unbounded Fresh Data

Sibylle Fröschle[2] and Graham Steel[1]

[1] LSV, ENS Cachan & CNRS & INRIA, France
[2] School of Informatics, University of Oldenburg, Germany

Abstract. We extend Delaune, Kremer and Steel's framework for analysis of PKCS#11-based APIs from bounded to unbounded fresh data. We achieve this by: formally defining the notion of an *attribute policy*; showing that a well-designed API should have a certain class of policy we call *complete*; showing that APIs with complete policies may be safely abstracted to APIs where the attributes are fixed; and proving that these *static* APIs can be analysed in a small bounded model such that security properties will hold for the unbounded case. We automate analysis in our framework using the SAT-based security protocol model checker SATMC. We show that a symmetric key management subset of the Eracom PKCS#11 API, used in their ProtectServer product, preserves the secrecy of sensitive keys for unbounded numbers of fresh keys and *handles*, i.e. pointers to keys. We also show that this API is not robust: if an encryption key is lost to the intruder, SATMC finds an attack whereby all the keys may be compromised.

1 Introduction

RSA Laboratories standard PKCS#11 defines Cryptoki, a general purpose API for cryptographic devices such as smartcard security tokens and cryptographic hardware security modules (HSMs) [7]. It has been widely adopted in industry. As well as providing cryptographic functionality, PKCS#11 is also supposed to enforce certain security properties. In particular, it is stated in the standard that even if the device is connected to an untrusted machine where the operating system and device drivers might be compromised, keys marked "sensitive" cannot be compromised [7, §7]. However, in 2003, Clulow presented a number of attacks, i.e. sequences of valid PKCS#11 commands, which result in sensitive keys being revealed in the clear [2]. A typical one is the so-called 'key separation attack'. The name refers to the fact that a key may have conflicting roles. Clulow gives the example of a key configured for decryption of ciphertexts, and for 'wrapping', i.e. encryption of other keys for secure transport. To determine the value of a sensitive key, the attacker calls the 'wrap' command and then the 'decrypt' command, as shown in Figure 1. Here (and subsequently) $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ is what we call a *handle binding*, i.e. a function application modelling the fact that n_1 is a *handle* for (or pointer to) key k_1 on the device. The symmetric encryption of k_1 under key k_2 is represented by the term $\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$. We model commands by giving the inputs on the left of the arrow, and the output on the right. The result of the attack is to reveal the value of k_1 in the clear.

Delaune, Kremer and Steel have proposed a formal model for the operation of the PKCS#11 API [5], which together with a model checker allowed various configurations of the API to be examined and several new attacks to be found. However, their model requires a bound on the number of freshly generated names, used to model new keys and handles. This means that when the model is shown to be secure up to the bound, we cannot be sure that there is not an attack on a larger model of the API. Moreover, they were only able to find secure configurations by severely restricting the functionality of the interface - in effect restricting the update of long-term keys to an operation requiring the device to be connected to a trusted machine. In this paper, we motivate a restriction to a certain class of APIs, showing how APIs outside this class are likely to be problematic because of the wrapping and unwrapping operations intrinsic to the API. We prove theorems showing that if there is an attack on an API in our class in the unbounded model, then there is an attack in a particular bounded model of a size suitable for treatment with a model checker. We use version 3.0 of the SAT-based model checker for security protocols SATMC [1] to carry out experiments on these APIs.

The rest of the paper is organised as follows. We first define our formal model (§2). We make explicit the idea of an *attribute policy*, which specifies what roles a key may have, and insist that API rules give rise to a certain class of attribute policies, called *complete* policies (§3). We explain why this is a reasonable demand, and show that an API with a complete policy can be mapped to an equivalent (with respect to security) API with a static policy, i.e. one where the attributes are non-mutable. We then show how security analysis of APIs with static policies for unbounded numbers of keys and handles can be performed by model checking carefully designed small bounded models (§4). Finally, we prove by model checking the secrecy of sensitive keys for a small PKCS#11 based API, for unbounded numbers of keys and handles (§5). This API is a subset of the implementation of PKCS#11 used by Eracom/SafeNet in their ProtectServer product [6]. We also show how the loss of a key with the $\mathsf{encrypt}$ attribute set can lead to the loss of all the keys, thanks to an

Initial knowledge: The intruder knows $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ and $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$. The name n_2 has the attributes wrap and $\mathsf{decrypt}$ set whereas n_1 has the attribute $\mathsf{sensitive}$ and $\mathsf{extract}$ set.

Trace:

Wrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) \rightarrow \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$
SDecrypt:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2) \rightarrow \mathsf{k}_1$

Fig. 1. Decrypt/Wrap attack

attack found by SATMC. We show how to modify the API to prevent the attack. Thus finally we are able to prove some robust guarantees of security for a functional subset of PKCS#11, albeit in our symbolic model. We conclude in section 6.

2 Formal Model

In Cryptoki's logical view a "cryptographic token" (or simply "token") is a device that stores *objects* and can perform cryptographic functions with these objects. For the subset of Cryptoki functions that we will model, the objects will always be cryptographic *keys*. Each object will be referenced by a *handle*, which (like a file handle) can be thought of as a pointer to the object. Several instances of the same object are possible. *Attributes* are characteristics that distinguish a particular instance of an object. They enable or restrict the way functions can be applied to the instance of the object. For example, the instance of a key can only be used to decrypt a ciphertext (provided by the application programmer) if the attribute $\mathsf{decrypt}$ is set. The combinations of attributes that an instance of an object is allowed to have, and how the attributes can be set and unset, depends on the particular configuration of the token. We will model this by defining the concept of an *attribute policy*. The API commands and their semantics will as usual be modelled by a *rewriting system* over a *term algebra*.

Our model has two main differences to that of Delaune, Kremer and Steel (DKS) [5]. First, justified by results therein [5, Theorem 1] we will work with a *typed* term algebra, silently insisting that all APIs considered are *well-moded* with respect to our types. Second, our concept of defining our APIs relative to a *attribute policy* is new. Following DKS [5] we will model handles by nonces, and the reference of an object k by a nonce n

by the function application $h(n, k)$, where h is a symbol unknown to the attacker. However, to avoid ambiguity in informal discussions, we will call such a function application a *handle binding* rather than just a *handle* as in [5].

2.1 Term algebra

We consider the following types: $\mathsf{Cipher}, \mathsf{Key}, \mathsf{Nonce}$, and $\mathsf{HBinding}$ where Key and Nonce are atomic types. We assume four countably infinite sets: a set $\mathcal{N}$ of ground terms of type Nonce; a set $\mathcal{X}_{\mathcal{N}}$ of variables of type Nonce; a set $\mathcal{K}$ of ground terms of type Key; and a set $\mathcal{X}_{\mathcal{K}}$ of variables of type Key. All other terms are built up from the atomic terms by the following operators:

- encryption $\mathsf{enc} : \mathsf{Key} \times \mathsf{Key} \rightarrow \mathsf{Cipher}$, and
- handle binding $h : \mathsf{Nonce} \times \mathsf{Key} \rightarrow \mathsf{HBinding}$.

We denote the set of *terms* by $\mathcal{T}$, and the set of *ground terms*, i.e., terms that do not contain any variables, by $\mathcal{GT}$. We define the set of *template terms*, denoted by $\mathcal{TT}$, to be the set of terms that do not contain any subterm of $\mathcal{N} \cup \mathcal{K}$. Thus, a template term is a term that does not contain any ground nonce or key but may use variables to represent them. We define *key terms* as $\mathcal{T}_{\mathcal{K}} = \mathcal{K} \cup \mathcal{X}_{\mathcal{K}}$, and *nonce terms* as $\mathcal{T}_{\mathcal{N}} = \mathcal{N} \cup \mathcal{X}_{\mathcal{N}}$ respectively. Given a term $t \in \mathcal{T}$, let $\mathit{nonces}(t)$ be the set of nonce terms occurring in t. We extend this notation to sets of terms in the obvious way.

We also consider a *finite* set $\mathcal{A}$ of predicate symbols, disjoint from the other symbols, which we call *attributes.* In this paper we work with the following restricted set of attributes:

$$\mathcal{A} = \{\mathsf{encrypt}, \mathsf{decrypt}, \mathsf{wrap}, \mathsf{unwrap}\}.$$

The set of *attribute terms* is defined as

$$\mathcal{AT} = \{\mathit{att}(n) \mid \mathit{att} \in \mathcal{A} \ \& \ n \in \mathcal{T}_{\mathcal{N}}\}.$$

Attribute terms will be interpreted as propositions. A *literal* is an expression a or $\neg a$ where $a \in \mathcal{AT}$. *Ground attribute terms* and *literals*, and *template attribute terms* and *literals* are defined analogously to above. We denote template literals by $\mathcal{TL}$.

2.2 Attribute policies and attribute states

An *attribute valuation* is a partial function $a : \mathcal{A} \to \{\bot, \top\}$. If an attribute valuation is total then it is an *object state*. We write $\mathcal{S}$ for the set of object states. An *attribute policy* is a finite directed graph $P = (S_P, \to_P)$ where $S_P \subseteq \mathcal{S}$ is the set of *allowable object states*, and $\to_P \subseteq S_P \times S_P$ is the set of *allowable transitions* between the object states.

A *token state* is a partial function $A : \mathcal{N} \to \mathcal{S}$ which assigns an object state to each nonce in $dom(A)$, and thus to the keys represented by the nonces in $dom(A)$. Given an attribute policy P, we say A *conforms to* P iff $ran(A) \subseteq S_P$. Every token state induces a valuation V_A for the set of attribute terms over $dom(A)$, defined by:

$$V_A(att(n)) = \begin{cases} \top & \text{if } A(n)(att) = \top \\ \bot & \text{otherwise,} \end{cases} \qquad V_A(\neg att(n)) = \begin{cases} \top & \text{if } A(n)(att) = \bot \\ \bot & \text{otherwise.} \end{cases}$$

2.3 API rewrite systems

We model two types of API commands: *key management commands* and the command `SetAttributeValue`, which allows the API programmer to manipulate the token state. The actions of the intruder will be modelled by a fixed set of *intruder rules*.

Syntax and informal semantics. Key management commands may generate new objects, in which case the token state will be extended by assigning an object state to each new object. The execution of a key management command may be subject to whether the objects referenced have certain attributes set or unset. Thus, formally, such commands are described by *key management rules* of the form

$$R : \quad T \; ; \; L \xrightarrow{\mathsf{new}\, \tilde{x}} T' \; ; \; A_{new}$$

where $T \subseteq \mathcal{TT}$ is a set of template terms, $L \subseteq \mathcal{TL}$ is a set of template literals such that $vars(L) \subseteq vars(T)$, $\tilde{x} \subseteq \mathcal{X}_\mathcal{N} \cup \mathcal{X}_\mathcal{K}$ is a set of key and nonce variables such that $\tilde{x} \cap vars(T) = \emptyset$, $T' \subseteq \mathcal{TT}$ is a set of template terms such that $vars(T') \subseteq vars(T) \cup \tilde{x}$, and A_{new} is a template token state with $dom(A_{new}) = nonces(\tilde{x})$.

The `SetAttributeValue` command is modelled by the parameterized *attribute rule* $\mathsf{set}(n, a)$ where $n \in \mathcal{N}$, and a is an attribute valuation.

Whether an instance of this rule can indeed be applied at a given token state will depend on the attribute policy.

An *API rewrite system* is a pair $(P, \mathcal{R})$ where P is an attribute policy and $\mathcal{R}$ is a set of *key management rules* that conform to P in the following way: every rule that generates a new handle is parameterized by an object state $a \in S_P$ such that the new handle is assigned object state a in A_{new}. Note that the attribute rule $\mathsf{set}(n, a)$ is assumed to be part of all APIs modelled here and does not need to be specified explicitly.

Example 1. As an example consider the rewrite system $(P, \mathcal{R})$ where $\mathcal{R}$ is given by the rules in Figure 2, and P is defined by:

- $S_P = \{\{\mathsf{encrypt}, \mathsf{decrypt}\}, \{\mathsf{wrap}, \mathsf{unwrap}\}\}$, and
- $\rightarrow_P = (S_P, S_P \times S_P)$.

As in this example we often specify the allowed object states concisely as sets of attributes rather than functions. Note that due to our concept of attribute policy we do not need to specify rules for setting or unsetting attributes.

$$\mathsf{Wrap}: \mathsf{h}(\mathsf{n_w}, \mathsf{k_w}), \mathsf{h}(\mathsf{n}, \mathsf{k});\ \mathsf{wrap}(\mathsf{n_w}) \rightarrow \mathsf{senc}(\mathsf{k}, \mathsf{k_w})$$

For all $a \in S_P$:

$$\mathsf{Unwrap(a)}: \quad \mathsf{h}(\mathsf{n_w}, \mathsf{k_w}), \mathsf{senc}(\mathsf{k}, \mathsf{k_w});\ \mathsf{unwrap}(\mathsf{n_w}) \xrightarrow{\mathsf{new\ n}} \mathsf{h}(\mathsf{n}, \mathsf{k});\ \{\mathsf{n} \mapsto \mathsf{a}\}$$

$$\mathsf{KeyGenerate(a)}: \quad \xrightarrow{\mathsf{new\ n,k}} \mathsf{h}(\mathsf{n}, \mathsf{k});\ \{\mathsf{n} \mapsto \mathsf{a}\}$$

$$\mathsf{SEncrypt}: \quad \mathsf{h}(\mathsf{n_e}, \mathsf{k_e}),\ \mathsf{k};\ \mathsf{encrypt}(\mathsf{n_e}) \rightarrow \mathsf{senc}(\mathsf{k}, \mathsf{k_e})$$

$$\mathsf{SDecrypt}: \quad \mathsf{h}(\mathsf{n_e}, \mathsf{k_e}), \mathsf{senc}(\mathsf{k}, \mathsf{k_e});\ \mathsf{decrypt}(\mathsf{n_e}) \rightarrow \mathsf{k}$$

Fig. 2. PKCS#11 symmetric key management subset relative to an attribute policy P

Intruder capability. The following two rules represent the deduction capabilities of the intruder. We will denote the intruder theory by $\mathcal{I}$.

$$\mathsf{I\text{-}SEncrypt}: \quad \mathsf{k_e},\ \mathsf{k} \rightarrow \mathsf{senc}(\mathsf{k}, \mathsf{k_e})$$

$$\mathsf{I\text{-}SDecrypt}: \quad \mathsf{senc}(\mathsf{k}, \mathsf{k_e}),\ \mathsf{k_e} \rightarrow \mathsf{k}$$

Note that the intruder rules can be considered to be in the same format as the key management rules. We will make use of this when defining the semantics.

Semantics. Let $(P, \mathcal{R})$ be an API rewrite system. A *state* of $(P, \mathcal{R})$ is a pair (Q, A) where $Q \subseteq \mathcal{GT}$ is a set of ground terms, and A is a token state such that $dom(A) = nonces(Q)$ and A conforms to P. Given a state (Q, A) and a rule $R \in \mathcal{R} \cup \mathcal{I}$ of the form $T; L \xrightarrow{\mathsf{new}\,\tilde{x}} T', A_{new}$, we say R *can be applied* at (Q, A) under substitution θ if

1. θ is grounding for R and assigns to the variables in $\tilde{x}$ distinct nonces and keys that do not occur in Q,
2. $T\theta \subseteq Q$, and
3. $V_A(L\theta) = \top$.

The *successor state* of (Q, A) under R, θ is then defined as (Q', A') where $Q' = Q \cup T'\theta$, and $A' = A \cup A_{new}\theta$. This gives rise to a transition $(Q, A) \rightsquigarrow_{R,\theta} (Q', A')$.

Given a state (Q, A) and an instance of the attribute rule R of the form $\mathsf{set}(n, a)$, we say R *can be applied* at (Q, A) if

1. $n \in nonces(Q)$, and
2. $A(n) \rightarrow_P a'$, where a' is defined by

$$a'(att) = \begin{cases} a(att) & \text{if } att \in dom(a), \\ A(n)(att) & \text{otherwise.} \end{cases}$$

The *successor state* of (Q, A) under R is then (Q, A') where A' is defined by $A'(n) = a'$, and $A'(n') = A(n')$ for all $n' \in dom(A)$ such that $n' \neq n$. This gives rise to a transition $(Q, A) \rightsquigarrow_R (Q', A')$.

We write $(Q, A) \rightsquigarrow (Q', A')$ when there is some transition from (Q, A) to (Q', A'). We write $\rightsquigarrow^*$ for the reflexive and transitive closure of $\rightsquigarrow$. We call *derivation* a sequence of rule applications $D = (S_0, A_0) \rightsquigarrow (S_1, A_1) \cdots \rightsquigarrow (S_n, A_n)$. We call (S_0, A_0) the initial state of D and (S_n, A_n) the final state of D.

Queries. A *query* is a pair (T, L) where T is a set of template terms and L is a set of template literals. A state (Q, A) satisfies a query (T, L) iff there exists a substitution θ grounding for (T, L) and such that $T\theta \subseteq Q$ and $V_A(L\theta) = \top$. A derivation D satisfies a query (T, L) relative to an initial state (T_0, L_0) iff the initial state of D satisfies (T_0, L_0) and the final state of D satisfies (T, L).

3 Towards static attribute policies

An attribute policy is called *static* if it rules out any change of object state. Formally, an attribute policy $P = (S, \rightarrow)$ is static if $\rightarrow = \emptyset$. It is clear that, in general, a given set of API rules is much easier to analyse and verify when the associated attribute policy is static: in an informal analysis one does not need to consider any side effects that may arise from moving between allowable object states; in a formal analysis one does not need to consider any attribute set/unset rules; thereby the state space is reduced and remains monotonic. Even though static attribute policies may seem very restrictive, we will argue that well-designed attribute policies should satisfy a criterion we call *completeness*, and that complete attribute policies may be safely abstracted to static attribute policies.

Due to limitation of space we will only consider API rewrite systems whose rewrite rules are positive in that all the tests of attributes are positive. In a future version of this paper we will show how our results can be carried over to all APIs.

API rewrite systems in positive form. A rule $T; L \xrightarrow{\mathsf{new}\,\tilde{x}} T', A_{new}$ is *positive* if all literals $l \in L$ are positive. An API rewrite system $(P, \mathcal{R})$ is *in positive form* if all the rules in $\mathcal{R}$ are positive.

Given an API rewrite system in positive form we adopt the following simplifications. An object state $a : \mathcal{A} \to \{\top, \bot\}$ can be viewed as the set of attributes $\{att \mid a(att) = \top\}$, and a token state $A : \mathcal{T}_{\mathcal{N}} \to \mathcal{S}$ as the set of attribute terms $\{att(n) \mid n \in dom(A) \;\&\; att \in A(n)\}$. The valuation induced by an attribute state A, V_A, then simplifies to $V_A(att(n)) = \top$ if and only if $att(n) \in A$. (We never need to consider valuations of negative literals.)

One could adopt similar conventions for API rewrite systems in general. However, due to our restriction to positive form we have: for two object states $a \subseteq a'$ the object state a' enables at least as many commands as a, and similarly for token states.

Complete attribute policies. In the following let $\mathcal{R}$ be a set of positive API rules. Assume we wish to obtain a secure attribute policy for the commands modelled by $\mathcal{R}$. Typically we will start out with an idea of which attributes are conflicting. For example, the attack in Figure 1 tells us that no object should ever have both wrap and decrypt set. In this way we can induce the set of allowable object states. It is clear from previously discovered attacks, however, that defining a safe transition relation between allowable states is non-trivial. For example, one might try to prevent the attack in Figure 1 by disallowing the attributes wrap and decrypt from being set on the same handle (which is illustrated by

Initial knowledge: The intruder knows $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ and $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$. The handle n_2 has the attribute wrap set.

Trace:

Wrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) \to \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$
Unset wrap:	$\mathsf{set}(n_2, [\mathsf{wrap} \mapsto \bot])$
Set decrypt:	$\mathsf{set}(n_2, [\mathsf{decrypt} \mapsto \top])$
SDecrypt:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2) \to \mathsf{k}_1$

Fig. 3. Decrypt/Wrap attack II [8]

the policy of Example 1). The attack in Figure 3 shows that this will not suffice. To address this, one might decide to declare wrap and $\mathsf{decrypt}$ as 'sticky' attributes, i.e. attributes which cannot be unset. However, the fact that the intruder can generally wrap and unwrap keys in order to obtain multiple handles for the same key means the attack can still be performed [5, Fig. 4].

For the rest of this paper, we will consider a restricted class of policies:

Definition 1. *An attribute policy $P = (S, \to)$ is* complete *if P consists of a collection of disjoint, disconnected cliques, and for each clique C, $c_0, c_1 \in C \Rightarrow c_0 \cup c_1 \in C$.*

The intuition behind this is that we do not expect to be able to prevent the intruder from making copies of an object via the import and export mechanisms, as in certain known attacks [5, Fig. 4]. This means that for policy $P = (S, \to)$, $\forall s_1, s_2, s_3 \in S$, if $s_1 \to s_2$ and $s_1 \to s_3$ are in P, then by copying a handle in state s_1, the intruder can obtain what is effectively a handle for an object in state $s_2 \cup s_3$. A well designed policy should take this into account. We further require the transition relation to be symmetric. We believe that the results in this paper could be extended to relax this restriction, but observe that our current definition has the advantage of giving a simple and intuitive rule for attribute policy design. Of course not all complete policies will be secure: consider a trivial policy in which all object states, including conflicting ones, are connected (such as the policy of Example 1). However, complete policies are amenable to analysis, as we will now show.

End point abstractions. Let $P = (S, \to)$ be a complete attribute policy. We call the object states of S that are maximal in S with respect to set inclusion *end points* of P, denoted by $\varepsilon(P)$. Such object states are end points in the following sense: once an attacker has reached an end point for an object he does not gain anything by leaving it: any object state that can be reached from an end point will have less enabling power.

P naturally gives rise to a static attribute policy where the allowable object states are taken to be the end points of P. Formally, we define the *end point abstraction of* P, denoted by $EP(P)$, to be the attribute policy $(\varepsilon(P), \emptyset)$. In Theorem 1 below we prove that $EP(P)$ provides a sound and complete abstraction of P.

Given $a \in S$, define $\varepsilon(a)$ to be the uniquely given end point e such that $a \subseteq e$. Given an object state A, define $\varepsilon(A)$ to be the object state that results when replacing every map $[n \mapsto a] \in A$ by $[n \mapsto \varepsilon(a)]$. If a rule $R \in \mathcal{R}$ generates a new handle with object state a then $\varepsilon(R)$ is the rule that results from R by replacing a by $\varepsilon(a)$. (Recall that $\varepsilon(R) \in \mathcal{R}$ by definition of API rewrite systems.)

Proposition 1. *1. For all standard rules R, we have:*
if $(Q_1, A_1) \rightsquigarrow_R (Q_2, A_2)$ *then* $(Q_1, \varepsilon(A_1)) \rightsquigarrow_{\varepsilon(R)} (Q_2, \varepsilon(A_2))$.
2. For all attribute rule instances R, we have:
if $(Q_1, A_1) \rightsquigarrow_R (Q_2, A_2)$ *then* $\varepsilon(A_1) = \varepsilon(A_2)$ *(and* $Q_1 = Q_2$ *as usual).*

Proof. (1) is a consequence of the definition of $\varepsilon(a)$ and our restriction to positive rules. (2) follows since by definition of complete attribute policies if $a_1 \to a_2$ is a transition in P then $\varepsilon(a_1) = \varepsilon(a_2)$.

Theorem 1. *If there is a derivation under P from* (Q_0, A_0) *to* (Q_m, A_m) *then there is a derivation under $EP(P)$ from* $(Q_0, \varepsilon(A_0)$ *to* $(Q_m, \varepsilon(A_m))$.
Conversely, if there is a derivation under $EP(P)$ from (Q_0, A_0) *to* (Q_m, A_m) *then there is a derivation under P from* (Q_0, A_0) *to* (Q_m, A_m).

Proof. To prove the first part assume a derivation D under P. By Prop. 1 we can transform D into a derivation under $EP(P)$ in the following way: replace each state (Q_i, A_i) occurring in D by $(Q_i, \varepsilon(A_i))$, and remove all attribute rule transitions from D. The converse direction is immediate since $EP(P)$ is a subgraph of P.

Altogether our results mean that for our class of APIs it suffices to analyse security under a static attribute policy. As we will demonstrate in the next section, together with further insights, this will lead us to reducing two variants of the PKCS#11 API to a bounded model.

4 Reducing APIs with static attribute policies to bounded models

We consider the symmetric key fragment of the PKCS#11 key management API as modelled in Figure 2, and the Eracom version of the API to be introduced in this section. We show that for both these rewrite systems, when we assume a static attribute policy, it is sufficient to work with a bounded number of freshly generated values.

In the following we assume an API rewrite system $(P, \mathcal{R})$ where $P = (\mathcal{E}, \emptyset)$ is any static attribute policy and $\mathcal{R}$ will be specified in each paragraph. Derivations will always be assumed to start from a given initial state (Q_0, A_0). Since we work with static attribute policies only, the object state of every handle stays constant:

Proposition 2. *Let* $(Q_0, A_0) \rightsquigarrow (Q_1, A_1) \cdots \rightsquigarrow (Q_m, A_m)$ *be a derivation. Let* n *be a nonce, and* i *such that* n *is generated by the* i*th transition. Then for all* $j \in [i, m]$*,* $A_j(n)$ *is defined and* $A_i(n) = A_j(n)$*.*

Justified by this, in the context of a derivation as above, for every nonce n occurring in D we write $A(n)$ for the one object state it can assume. Also note that we no longer need to consider any attribute rule instances.

Atom substitutions. One key insight behind our reductions is that we can eliminate freshly generated keys and handles whenever we are able to replace them by already existing keys and handles that provide the same functionality. Formally, we will require the concept of *atom substitutions*, following [**?**]. We use *Atoms* to denote the set of ground atomic terms, i.e., $\mathcal{N} \cup \mathcal{K}$.

An *atom substitution* is a partial function $\delta : \mathit{Atoms} \to \mathit{Atoms}$ that respects the type of the atoms. We extend atom substitutions to sets, relations, etc. in the usual way. Given a token state A, we say atom substitution δ, is *defined for* A if for all $n_1, n_2 \in \mathit{dom}(A)$ we have:

1. $n_1 \mapsto n_2 \in \delta \;\&\; n_2 \notin \mathit{dom}(\delta) \implies A(n_1) = A(n_2)$, and
2. $n_1 \mapsto n, n_2 \mapsto n \in \delta$ for some $n \implies A(n_1) = A(n_2)$.

Proposition 3. *If* A *is a token state and* δ *is an atom substitution defined for* A *then we have:*

1. $A\,\delta$ *is a token state with* $\mathit{dom}(A\,\delta) = \mathit{dom}(A)\,\delta$*, and*

2. for all literals l, if $V_A(l)$ is defined then $V_{A\delta}(l\delta)$ is defined and $V_A(l) = V_{A\delta}(l\delta)$.

The following two propositions show that atom substitutions preserve rule applications as well as queries in a natural way. Both propositions are routine to prove by Prop. 3 and inspection of the definitions.

Proposition 4. *Let R be a rule of the form $T; L \xrightarrow{\text{new}\,\tilde{x}} T', A_{new}$. If $(Q, A) \rightsquigarrow_{R,\theta} (Q', A')$ and δ is an atom substitution defined for A such that $\theta(\tilde{x}) \cap (dom(\delta) \cup ran(\delta)) = \emptyset$ then we have $(Q, A)\delta \rightsquigarrow_{R,\theta\delta} (Q', A')\delta$.*

Proposition 5. *If a state (Q, A) satisfies a query (T, L) by a substitution θ, and δ is an atom substitution defined for A, then $(Q, A)\delta$ satisfies (T, L) by $\theta\delta$.*

PKCS#11 rewrite system. Consider the symmetric key fragment of the standard PKCS#11 API as modelled in Figure 2. There are only two rules that generate fresh values: KeyGenerate and Wrap.

It is intuitive that a pair k, n generated by KeyGenerate with, say, the object state of n set to ε provides the same functionality as any other pair k', n' generated by KeyGenerate with n' set to the same object state ε. Thus, it is plausible that to check whether a query is satisfied we need to consider at most one instance of KeyGenerate for each allowable object state. The following proposition gives us the means to successively delete instances of KeyGenerate from any derivation until we arrive at a derivation that is bounded in this way.

Proposition 6. *Assume a derivation*

$$(Q_0, A_0) \cdots \rightsquigarrow_{R_i} (Q_i, A_i) \cdots \rightsquigarrow_{R_j} (Q_j, A_j) \cdots \rightsquigarrow_{R_n} (Q_n, A_n)$$

such that for some n_i, n_j, k_i, k_j,

1. *R_i is an instance of* KeyGenerate *producing $h(n_i, k_i)$,*
2. *R_j is an instance of* KeyGenerate *producing $h(n_j, k_j)$, and*
3. *$A(n_i) = A(n_j)$.*

Then

$$\begin{aligned}(Q_0, A_0) \cdots \rightsquigarrow_{R_i} (Q_i, A_i) \cdots \rightsquigarrow_{R_{j-1}} (Q_{j-1}, A_{j-1}) \rightsquigarrow_{R_{j+1}\delta} (Q_{j+1}, A_{j+1})\delta \cdots \\ \cdots \rightsquigarrow_{R_n\delta} (Q_n, A_n)\delta\end{aligned}$$

is a derivation, where $\delta = [n_j \mapsto n_i, k_j \mapsto k_i]$.

Proof. This follows by Prop. 4 when considering that $(Q_{j-1}, A_{j-1}) = (Q_j, A_j)\delta$, and that δ satisfies the conditions of Prop. 4 with respect to each transition $\rightsquigarrow_{R_m}$, $m \geq j$.

It can be read from the rules that, given a key k, whenever two handles for k, say n_1 and n_2, have the same object state then they can be employed in exactly the same way. Hence, it is plausible that for each key we need at most one handle per allowable object state. The following proposition gives us the means to eliminate instances of Unwrap until we arrive at a derivation that is reduced in this way.

Proposition 7. *Assume a derivation*

$$(Q_0, A_0) \cdots \rightsquigarrow_{R_i} (Q_i, A_i) \cdots \rightsquigarrow_{R_n} (Q_n, A_n)$$

such that for some n_1*,* n_2*,* k

1. $h(n_1, k) \in Q_{i-1}$ *, and*
2. R_i *is an instance of* Unwrap *producing* $h(n_2, k)$ *with* $A(n_1) = A(n_2)$*.*

$$(Q_0, A_0) \cdots \rightsquigarrow_{R_{i-1}} (Q_{i-1}, A_{i-1}) \rightsquigarrow_{R_{i+1}\delta} (Q_{i+1}, A_{i+1})\delta \cdots \rightsquigarrow_{R_n\delta} (Q_n, A_n)\delta$$

is a derivation, where $\delta = [n_2 \mapsto n_1]$*.*

Proof. This follows similarly to Prop. 6.

Altogether we obtain:

Lemma 1. *If there is a derivation of a query* (T, L) *then there is a derivation* D' *of the same query such that, in* D'*, the following holds:*

1. *for all pairs* k_1, n_1 *and* k_2, n_2 *generated by* KeyGenerate*,* $k_1 \neq k_2$ *implies* $A(n_1) \neq A(n_2)$*;*
2. *for every key* k *and all handles* n_1*,* n_2 *with bindings* $h(n_1, k)$*,* $h(n_2, k)$*,* $n_1 \neq n_2$ *implies* $A(n_1) \neq A(n_2)$*.*

Proof. The lemma easily follows by successively applying the above two propositions and because by Prop. 5 the transformations preserve the query.

The lemma implies that for the PKCS#11 rewrite rules we can reduce the static model to a bounded model. We also make use of the fact that in a model with no disequalities on keys, an attacker is as powerful when given one key (without a handle) in his initial knowledge as when given many such keys.

Theorem 2. *In the PKCS#11 rewrite system with attribute policy $(\mathcal{E}, \emptyset)$, if there is a derivation of a query (T, L) then there is a derivation of the same query using*

1. *at most* $1+|\mathcal{E}|$ *keys: at most* 1 *key in* Q_0*, and at most* $|\mathcal{E}|$ *keys generated by* $\mathsf{KeyGenerate}$*, and*
2. *at most* $|\mathcal{E}| \times (1 + |\mathcal{E}|)$ *handles.*

For all $e \in \mathcal{E}$:

$$\mathsf{KeyGenerate(e)} : \xrightarrow{\mathsf{new\ n,k}} \mathsf{h(n,k); e(n)}$$

$\mathsf{Wrap(e)}$:

$$\mathsf{h(n_w, k_w), h(n, k);\ wrap(n_w), e(n)} \xrightarrow{\mathsf{new\ k_m}} \mathsf{enc(k, k_w), enc(m_k, k_w), hmac_{k_m}(k, e)}$$

$\mathsf{Unwrap(e)}$:

$$\mathsf{h(n_w, k_w), enc(k, k_w), enc(k_m, k_w), hmac_{k_m}(k, e);\ unwrap(k_w)} \xrightarrow{\mathsf{new\ n}} \mathsf{h(n, k);\ e(n)}$$

$$\mathsf{SEncrypt} : \quad \mathsf{h(n_e, k_e), k;\ encrypt(n_e)} \rightarrow \mathsf{enc(k, k_e)}$$

$$\mathsf{SDecrypt} : \mathsf{h(n_e, k_e), enc(k, k_e);\ decrypt(k_e)} \rightarrow \mathsf{k}$$

Fig. 4. Symmetric Key Management subset of the Eracom PKCS#11 API. $e(n)$ is a short-hand for "n is in object state e".

Eracom rewrite system. Consider the API rules given in Figure 4, derived from the symmetric key management commands of the Eracom ProtectServer [6]. Here, an HMAC is used to bind the attributes to the wrapped key, to prevent an attacker from re-importing several copies of a key with different attributes. Formally, we have the new type mac and the function symbol $\mathsf{hmac} : \mathsf{Key} \times \mathsf{Key} \times \mathit{att} \rightarrow \mathsf{mac}$. Although the proof of Lemma 1 carries over to this set of rules, Theorem 2 no longer holds, since the wrap rule gives an additional way of generating fresh keys (the key to be used as MAC-key). However, we can recover the result with a simple abstraction: if there is an attack, then there is an attack where the MAC-key generated by the wrap command is constant:

Proposition 8. *Given a derivation under the rules of Figure 4, we can map in the obvious way to a derivation under rules that are like those of Figure 4 apart from that the wrap rule always uses a constant mac-key* $\mathsf{m_K}$.

This gives us an abstraction, not an equivalence. It may in theory lead to false attacks, but it is sound for proofs since in our queries we have no disequalities on keys. In this model, we have:

Theorem 3. *In the* abstracted *Eracom rewrite system with attribute policy* $(\mathcal{E}, \emptyset)$*, if there is a derivation of a query* (T, L) *then there is a derivation of the same query using*

1. *at most* $1 + |\mathcal{E}| + 1$ *keys:*
   at most 1 *key in* Q_0*,*
   at most $|\mathcal{E}|$ *keys generated by* KeyGenerate*, and*
   at most 1 *key used by* Wrap*; and*
2. *at most* $|\mathcal{E}| \times (2 + |\mathcal{E}|)$ *handles.*

In fact we can recover an exact version (i.e. without the abstraction) of Theorem 2 for the Eracom rewrite system involving a slightly larger number of keys and handles, but due to lack of space we leave this for a future longer version of the paper.

5 Experiments

Having established the validity of model checking a small bounded model of our Eracom-based PKCS#11 API (Fig. 4), we can now investigate security properties for unbounded keys and handles. We assume an end-point attribute policy $P = (\{ed, wu\}, \emptyset)$, where ed represents encrypt and decrypt and wu represents wrap and unwrap. All keys are assumed to be sensitive. SATMC includes the usual rules for encryption and decryption by known keys. First we investigate the stated property required of PKCS#11 in the specification [7, §7].

Definition 2. *A* known key *is a key* k *such that the intruder knows the plaintext value* k *and the intruder has a handle* $\mathsf{h}(\mathsf{n}, \mathsf{k})$*.*

Property 1. If an intruder starts with no known keys, he cannot obtain any known keys.

In the API of Figure 4, this property is verified by SATMC in 0.4 seconds. A further desirable property of such an API is that if a session key (i.e. an encryption/decryption key) is lost to the intruder by some means beyond the scope of the model, then no further keys are compromised.

Property 2. If an intruder starts with a known key $\mathsf{k_i}$ with handle $\mathsf{h}(\mathsf{n_i}, \mathsf{k_i})$, and $\mathsf{ed}(\mathsf{n_i})$ is true, then he cannot obtain any further known keys.

Unfortunately this does not hold for the Eracom-based API. An attack is found by SATMC in 0.4 seconds. We give the attack in Figure 5. The intruder first wraps his key $\mathsf{k_i}$, then fakes an HMAC for it using $\mathsf{k_i}$ as the MAC key. This allows him to re-import $\mathsf{k_i}$ as a wrap/unwrap key. One way to prevent this attack is to add the wrapping key inside the HMAC (see Figure 6). SATMC verifies this property in about 0.5 seconds.

Full details of our model checking experiments, including all relevant model files, are available at `http://www.lsv.ens-cachan.fr/~steel/pkcs11/`.

Initial knowledge: Handles $\mathsf{h(n_1, k_1)}$, $\mathsf{h(n_2, k_2)}$, and $\mathsf{h(n_i, k_i)}$. Key $\mathsf{k_i}$. Attributes $\mathsf{ed(n_1), wu(n_2), ed(n_i)}$. The HMAC key is $\mathsf{k_3}$.

Trace:

Wrap: (ed)	$\mathsf{h(n_2, k_2),\ h(n_i, k_i) \rightarrow senc(k_i, k_2),\ senc(k_3, k_2),\ hmac_{k_3}(k_i, ed)}$
Unwrap: (wu)	$\mathsf{h(n_2, k_2),\ senc(k_i, k_2),\ senc(k_i, k_2),\ hmac_{k_i}(k_i, wu) \rightarrow h(n_2, k_i)}$
Wrap: (ed)	$\mathsf{h(n_2, k_i),\ h(n_1, k_1) \rightarrow senc(k_1, k_i),\ senc(k_3, k_i),\ hmac_{k_3}(k_1, ed)}$
Decrypt:	$\mathsf{k_i,\ senc(k_1, k_i) \rightarrow k_1}$

Fig. 5. Lost session key attack

Wrap :

$$\mathsf{h(x_1, y_1), h(x_2, y_2);\ wrap(x_1), A(x_2) \xrightarrow{\text{new } m_k} enc(y_2, y_1), enc(m_k, y_1), hmac_{m_k}(y_2, A, y_1)}$$

Unwrap :

$$\mathsf{h(x_1, y_2), enc(y_1, y_2), enc(x_m, y_2), hmac_{x_m}(y_1, A, y_2);\ unwrap(x_1) \xrightarrow{\text{new } n_1} h(n_1, y_1);\ A(n_1)}$$

Fig. 6. Revised Wrap/Unwrap Mechanism for the Eracom API

6 Conclusions

We have presented our framework for analysing PKCS#11 based APIs in an unbounded model. We described an attack on a version of the API used by Eracom, discovered using our model and the model checker SATMC. We suggested a fix and proved the secrecy of sensitive keys for the fixed version of the API, for unbounded numbers of keys, handles and command calls. An extension to asymmetric cryptography is our first priority, and then experiments with further proprietary implementations of PKCS#11.

We have explained how our work extends previous work by Delaune, Kremer and Steel [5]. There have been other attempts to analyse PKCS#11, but these were also for bounded models [8, 9], and included further approximations such as monotonic global state. With our endpoint abstractions, we also obtain a monotonic global state, however we have formally justified this in the presence of a complete attribute policy. In fact, our analysis suggests that robust attribute policies can be reduced to static ones. There have also been proofs of security for other APIs, such as revised versions of the IBM Common Cryptographic Architecture, again in bounded models [3, 4]. We plan to adapt our method to this API.

References

1. A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008. Software available at `http://www.ai-lab.it/satmc`. Currently developed under the AVANTSSAR project, `http://www.avantssar.eu`.
2. 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.
3. 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.
4. 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.
5. 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.
6. Sibylle Fröschle. The insecurity problem: tackling unbounded data. In *IEEE Computer Security Foundations Symposium 2007*. IEEE Computer Society, 2007.
7. J. Krhovják. PKCS #11 based APIs. Talk given at the Analysis of Security APIs Workshop (ASA-1), July 2007. Slides available at `http://homepages.inf.ed.ac.uk/gsteel/asa/slides/`.
8. RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.
9. E. Tsalapati. Analysis of PKCS#11 using AVISPA tools. Master's thesis, University of Edinburgh, 2007.
10. P. Youn. The analysis of cryptographic APIs using the theorem prover Otter. Master's thesis, Massachusetts Institute of Technology, 2004.