Formal Analysis of PKCS#11[*]

Stéphanie Delaune, Steve Kremer and Graham Steel
LSV, CNRS & INRIA & ENS de Cachan
61, avenue du Président Wilson
94235 CACHAN Cedex, FRANCE

Abstract

PKCS#11 defines an API for cryptographic devices that has been widely adopted in industry. However, it has been shown to be vulnerable to a variety of attacks that could, for example, compromise the sensitive keys stored on the device. In this paper, we set out a formal model of the operation of the API, which differs from previous security API models notably in that it accounts for non-monotonic mutable global state. We give decidability results for our formalism, and describe an implementation of the resulting decision procedure using a model checker. We report some new attacks and prove the safety of some configurations of the API in our model.

1 Introduction

RSA Laboratories Public Key Standards (PKCS) #11 defines the 'Cryptoki' API, designed to be an interface between applications and cryptographic devices such as smartcards, Hardware Security Modules (HSMs), and PCMCIA and USB key tokens. It has been widely adopted in industry, promoting interoperability of devices. However, the API as defined in the standard gives rise to a number of serious security vulnerabilities, [4]. In practice, vendors try to protect against these by restricting the functionality of the interface, or by adding extra features, the details of which are often hard to determine. This has lead to an unsatisfactory situation in which widely deployed security solutions are using an interface which is known to be insecure if implemented naïvely, and for which there are no well established fixes. The situation is complicated by the variety of scenarios in which PKCS#11 is used: an effective security patch for one scenario may disable functionality that is vital for another.

In this paper, we aim to lay the foundations for an improvement in this situation by defining a formal model for the operation of PKCS#11 key management commands, proving the decidability of certain security properties in this model, and describing an automated framework for proving these properties for different configurations of the API. The organisation of the paper is as follows: in Section 2, we first describe PKCS#11 and some of the known vulnerabilities. We define our model in Section 3, give some decidability results in Section 4, and detail our experiments in proving the (in)security of particular configurations in Section 5. Finally we conclude with a discussion of related work in Section 6.

Background. API level attacks were first identified by Longley and Rigby [12]. Anderson and Bond discovered many more [1]. Clulow revealed the existence of such attacks on PKCS#11 [4]. Since then, efforts have been made to formally analyse APIs using model checkers, theorem provers, and customised decision procedures [15, 19, 7, 6, 5, 17]. None of these models account for non-monotonic mutable global state, which was identified by Herzog [11] as a major barrier to the application of security protocol analysis tools to the problem.

2 An introduction to PKCS#11

The PKCS#11 API is designed to allow multiple applications to access multiple cryptographic devices through a number of *slots*. Each slot represents a socket or device reader in which a device may or not be present. To talk to a device, an application must establish a *session* through the appropriate slot. Once a session has been established, an application can authenticate itself to a token as one of two distinct types of user: the security officer (SO) and the normal user.

[*] Partially supported by project PFC ("plateforme de confiance"), pôle de compétitivité System@tic Paris-région Ile-de-France.

Authentication is by means of a PIN: a token is typically supplied with a default SO PIN, and it is up to the SO to set himself and the user a new PIN. As seen under PKCS#11, the token contains a number of *objects*, such as keys and certificates. Objects are referenced in the API via *handles*, which can be thought of as pointers to or names for the objects. In general, the value of the handle e.g. for a secret key does not reveal any information about the actual value of the key. Objects are marked as public or private. Once authenticated, the normal user can access public and private objects. The SO can only access public objects, but can perform functions not available to the user, such as setting the user's PIN. A session can also be unauthenticated, in which case only public objects and functions are available. In addition to being public or private, objects have other attributes, which may be general, such as the attribute sensitive which is true of objects which cannot be exported from the token unencrypted, or specific to certain classes of object, such as modulus or exponent for RSA keys.

Note that if malicious code is running on the host machine, then the user PIN may easily be intercepted, e.g. by a tampered device driver, allowing an attacker to create his own sessions with the device. Indeed the PKCS#11 standard recognises this: it states that this kind of attack cannot "compromise keys marked 'sensitive', since a key that is sensitive will always remain sensitive", [14, p. 31]. Clulow presented a number of attacks which violate this property, [4]. A typical one is the so-called 'key separation attack'. The name refers to the fact that the attributes of a key can be set and unset in such a way as to give a key conflicting roles. Clulow gives the example of a key with the attributes set 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 simply wraps it and then decrypts it, as shown in Figure 1. Here (and in subsequent boxes showing attacks), $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ represents the handle n_1 of key k_1, where h is a symbol not known to the attacker. Hence, if the attacker knows, $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$, he can't immediately deduce the value of k_1, and if he knows the value of some k_3, he can't *a priori* create himself a handle $\mathsf{h}(\mathsf{n}_3, \mathsf{k}_3)$ for that key. The symmetric encryption of k_1 under key k_2 is represented by $\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$.

As Clulow observes, it is not easy to prevent these kind of attacks, since there are a large number of possible attributes a key might have, and it is not clear which combinations are conflicting. The standard itself gives no advice on the subject, perhaps because to give incomplete advice might lead designers into a false sense of security. Additionally, if safeguards are added to the commands for setting and unsetting attributes, an attacker can subvert this by importing two copies of a key onto a device, and setting one of the conflicting attributes on each copy. Clulow also presented a pair of attacks he called 'Trojan key attacks', whereby the intruder introduces a wrapping key that he knows the true value of using a public unwrapping key. He then wraps the sensitive key under this known wrapping key and decrypts the result himself. Other vulnerabilities Clulow found include an attack based on the use of ECB mode to wrap double length 3DES keys, and the use of single length DES keys to wrap double length 3DES keys. Finally, he presented a series of attacks relying on particular details of the algorithms supported by PKCS#11, specifically the use of small exponents in RSA keys when using the X.509 mechanism to wrap symmetric keys, and the use of mechanisms that permit a set of related symmetric keys to be generated, making them susceptible to a parallel key search.

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 decrypt set whereas n_1 has the attribute sensitive and 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:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$, $\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$	$\rightarrow$	k_1

Figure 1. Decrypt/Wrap attack

The aim of our work described in this paper was to formally model the core key management operations of PKCS#11 and analyse them to learn more about which configurations are secure and insecure. In particular, we were interested in controlling key attributes to prevent key separation attacks. Note that attributes can be set and subsequently unset, which gives rise to non-monotonic mutable state (i.e. loops) in the model. We make the usual 'Dolev-Yao' assumptions, [9], used for protocol analysis, i.e. we abstract bit strings to terms, and assume the attacker can decompose and recompose terms arbitrarily, with the restriction that he can only decrypt encrypted packets if he has the correct key. This means that Clulow's final two attacks (small exponent X.509 and parallel key search) are out of scope for our model.

3 Formal model

3.1 Term algebra

We assume a given *signature* Σ, i.e. a finite set of *function symbols*, with an arity function $ar : \Sigma \to \mathbb{N}$, a (possibly infinite) set of *names* $\mathcal{N}$ and a (possibly infinite) set of *variables* $\mathcal{X}$. Names represent keys, data values, nonces, etc. and function symbols model cryptographic primitives, e.g. encryption. Function symbols of arity 0 are called constants. The set of *plain terms* $\mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})$ is defined by the following grammar:

$$\begin{array}{rcll} t, t_i & := & x & x \in \mathcal{X} \\ & \mid & n & n \in \mathcal{N} \\ & \mid & f(t_1, \ldots, t_n) & f \in \Sigma \text{ and } ar(f) = n \end{array}$$

We also consider a finite set $\mathcal{A}$ of unary function symbols, disjoint from Σ which we call *attributes*. The set of *attribute terms* is defined as

$$\mathcal{AT}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X}) = \{att(t) \mid att \in \mathcal{A}, t \in \mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})\}.$$

In our model, attribute terms will be interpreted as propositions. We define the set of *terms* as follows

$$\mathcal{T}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X}) = \mathcal{AT}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X}) \cup \mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X}).$$

The set $\mathcal{PT}(\Sigma, \mathcal{N}, \emptyset)$, also written $\mathcal{PT}(\Sigma, \mathcal{N})$, is the set of *ground terms* and similarly for plain and attribute terms. We write $vars(t)$ for the set of variables that occur in the term t and extend $vars$ to sets of terms in the expected way. A *position* is a finite sequence of positive integers. The empty sequence is denoted ϵ. The set of positions $pos(t)$ of a term t is defined inductively as $pos(u) = \{\epsilon\}$ for $u \in \mathcal{N} \cup \mathcal{X}$ and $pos(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \bigcup_{1 \leq i \leq n} i \cdot pos(t_i)$ for $f \in \Sigma \cup \mathcal{A}$. If p is a position of t then the expression $t|_p$ denotes the subterm of t at the position p, i.e. $t|_\epsilon = t$ and $f(t_1, \ldots, t_n)|_{i \cdot p} = t_i|_p$. The set of *subterms* of a term t, written $st(t)$, is defined as $\{t|_p \mid p \in pos(t)\}$. We denote by top the function that associates to each term t its root symbol, i.e. $\mathsf{top}(u) = u$ for $u \in \mathcal{N} \cup \mathcal{X}$ and $\mathsf{top}(f(t_1, \ldots, t_n)) = f$.

A *substitution* σ is a mapping from a finite subset of $\mathcal{X}$ called its domain, written $dom(\sigma)$, to $\mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})$. Substitutions are extended to endomorphisms of $\mathcal{T}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X})$ as usual. We use a postfix notation for their application. A substitution σ is *grounding for* a term t if $t\sigma$ is ground. This notation is extended as expected to sets of terms.

Example 1 *We consider the signature* $\Sigma_{\mathsf{enc}} = \{\mathsf{senc}, \mathsf{aenc}, \mathsf{pub}, \mathsf{priv}, \mathsf{h}\}$ *which we will use in the following to model PKCS#11. The symbols* senc *and* aenc *of arity 2 represent respectively symmetric and asymmetric encryption whereas* pub *and* priv *of arity 1 are constructors to obtain public and private keys respectively. Lastly,* h *is a symbol of arity 2 which allows us to model handles to keys. We will use it with a nonce as the first argument and a key as the second argument. Adding a nonce to the arguments of* h *allows us to model several distinct handles to the same key.*

We model the attributes that are associated to handles by the means of the set $\mathcal{A}$*. For the sake of simplicity our running example only considers these attributes:* extract, wrap, unwrap, encrypt, decrypt, sensitive.

We illustrate notations for manipulating terms on the plain term $t = \mathsf{senc}(\mathsf{aenc}(n_1, \mathsf{pub}(n_2)), x)$*. We have that* $vars(t) = \{x\}$ *and* $\mathsf{top}(t) = \mathsf{senc}$*. The set of positions of* t *is* $pos(t) = \{\epsilon, 1, 1.1, 1.2, 1.2.1, 2\}$ *and* $t|_{1.2}$*, the subterm of* t *at position* 1.2*, is* $\mathsf{pub}(n_2)$*.*

By abuse of notation, sometimes we will use term instead of plain term.

3.2 Description language

To model PKCS#11 and attacker capabilities we define a rule-based description language. It is close to a guarded command language *à la Dijkstra* (see [8]) and to the multi-set rewriting framework (e.g. [13]). One particular point is that it makes a clean separation between the intruder knowledge part, i.e. the monotonic part, and the current system state which is formalized by the attributes that may be set or unset. The semantics will be defined in a classical way as a transition system.

Syntax and informal semantics. As already mentioned attribute terms will be interpeted as propositions. A *literal* is an expression a or $\neg a$ where $a \in \mathcal{AT}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X})$. The description of a system is given as a finite set of rules of the form

$$T; L \xrightarrow{\mathsf{new}\ \tilde{n}} T'; L'$$

where T and T' are sets of plain terms in $\mathcal{T}(\Sigma, \mathcal{N}, \mathcal{X})$, L and L' are sets of literals and $\tilde{n}$ is a set of names in $\mathcal{N}$. The intuitive meaning of such a rule is the following. The rule can be fired if all terms in T are in the intruder knowledge and if all the literals in L are evaluated to true in the current state. The effect of the rule is that terms in T' are added to the intruder knowledge and the valuation of the attributes is updated to satisfy L'. The $\mathsf{new}\ \tilde{n}$ means that all the names in $\tilde{n}$ need to be replaced by fresh names in T' and L'. This allows us to model nonce or key generation: if the rule is executed several times, the effects are different as different names will be used each time.

We always suppose that L' is satisfiable, i.e. it does not contain both a and $\neg a$. We also suppose that any variable appearing in T' also appears in T, i.e. $\mathit{vars}(T') \subseteq \mathit{vars}(T)$, and any variable appearing in L' also appears in L, i.e. $\mathit{vars}(L') \subseteq \mathit{vars}(L)$. These conditions were easily verified in all of our experiments with PKCS#11.

Example 2 *As an example consider the rules given in Figure 2. They model a part of PKCS#11. We detail the first rule which allows wrapping of a symmetric key with a symmetric key. Intuitively the rule can be read as follows: if the attacker knows the handle* $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$*, a reference to a symmetric key* y_1*, and a second handle* $\mathsf{h}(\mathsf{x}_2, \mathsf{y}_2)$*, a reference to a symmetric key* y_2*, and if the attribute* wrap *is set for the handle* $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$ *(note that the handle is uniquely identified by the nonce* x_1*) and the attribute* extract *is set for the handle* $\mathsf{h}(\mathsf{x}_2, \mathsf{y}_2)$ *then the attacker may learn the wrapping* $\mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1)$*, i.e. the encryption of* y_2 *with* y_1*.*

Semantics. The formal semantics of our description language is given in terms of a *transition system*. We assume a given signature Σ, a set of attributes $\mathcal{A}$, a set of names $\mathcal{N}$, a set of variables $\mathcal{X}$, a set of rules $\mathcal{R}$ defined over $\mathcal{T}(\Sigma, \mathcal{N}, \mathcal{X})$. A partial valuation V of ground attribute terms is a partial function $V : \mathcal{AT}(\mathcal{A}, \Sigma, \mathcal{N}) \to \{\top, \bot\}$. We extend valuations to literals as

$$V(\ell) \quad = \quad \begin{cases} \;\; V(a) & \text{if } \ell = \;\; a \\ \neg V(a) & \text{if } \ell = \neg a \end{cases}$$

and to sets of literals (interpreted as a conjunction over the literals) as

$$V(L) = \bigwedge_{\ell \in L} V(\ell)$$

when $\{a \mid a \in L \text{ or } \neg a \in L\} \subseteq \mathit{dom}(V)$. Moreover, we assume a given set of ground terms $S_0 \subseteq \mathcal{T}(\Sigma, \mathcal{N})$ and a partial valuation of ground attribute terms to represent the initial state. In the following we say that the rule

$$t_1, \ldots, t_n; L \to v_1, \ldots, v_p; L''$$

is a fresh renaming of a rule

$$t_1, \ldots, t_n; L \xrightarrow{\mathsf{new}\ n_1, \ldots, n_k} u_1, \ldots, u_p; L'$$

if $v_i = u_i[n_1 \to n'_1, \ldots, n_k \to n'_k]$ $(1 \leq i \leq p)$, $L'' = L'[n_1 \to n'_1, \ldots, n_k \to n'_k]$ and $n'_1, \ldots, n'_k$ are fresh names of $\mathcal{N}$. The transition system $(Q, q_0, \rightsquigarrow)$ is defined as follows:

- Q is the set of *states*: each state is a pair (S, V), such that $S \subseteq \mathcal{T}(\Sigma, \mathcal{N})$ and V is any partial valuation of $\mathcal{AT}(\mathcal{A}, \Sigma, \mathcal{N})$.
- $q_0 = (S_0, V_0)$ is the *initial state*. S_0 is the initial attacker knowledge and V_0 defines the initial valuation of some attributes.
- $\rightsquigarrow \subseteq Q \times Q$ is the transition relation defined as follows. For each fresh renaming R of a rule in $\mathcal{R}$,

  $$\mathsf{R} := T; L \to T'; L'$$

  we have that $(S, V) \rightsquigarrow (S', V')$ if there exists a grounding substitution θ for R such that

  - $T\theta \subseteq S$, and
  - $V(L\theta) = \top$.

  Then, we have that $S' = S \cup T'\theta$, and the function V' is defined as follows:

  $$\mathit{dom}(V') = \mathit{dom}(V) \cup \{a \mid a \in L' \text{ or } \neg a \in L'\}$$

  $$V'(a) = \begin{cases} \top & \text{if } \;\; a \in L' \\ \bot & \text{if } \neg a \in L' \\ V(a) & \text{otherwise} \end{cases}$$

Some of the rules, e.g. the unwrap and key generation rules in Figure 2, allow the creation of new handles for which attributes are set and unset. We therefore dynamically extend the domain of the valuation whenever such new handles are created. Note also that when $(S, V) \rightsquigarrow (S', V')$ we always have that $S \subseteq S'$ and $\mathit{dom}(V) \subseteq \mathit{dom}(V')$.

3.3 Queries

Security properties are expressed by the means of *queries*.

Definition 1 *A* query *is a pair* (T, L) *where* T *is a set of terms and* L *a set of literals (both are not necessarily ground).*

Intuitively, a query (T, L) is satisfied if there exists a substitution θ such that we can reach a state where the adversary knows all terms in $T\theta$ and all literals in $L\theta$ are evaluated to true.

Definition 2 *A transition system* $(Q, q_0, \rightsquigarrow)$ *satisfies a query* (T, L) *iff there exists a substitution* θ *grounding for* Q *and a state* $(S, V) \in Q$ *such that* $q_0 \rightsquigarrow^* (S, V)$*,* $T\theta \subseteq S$*, and* $V(L\theta) = \top$*.*

$$\begin{array}{rrcl}
\mathsf{Wrap\,(sym/sym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1),\mathsf{h}(\mathsf{x}_2,\mathsf{y}_2);\ \mathsf{wrap}(\mathsf{x}_1),\mathsf{extract}(\mathsf{x}_2) & \rightarrow & \mathsf{senc}(\mathsf{y}_2,\mathsf{y}_1)\\
\mathsf{Wrap\,(sym/asym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{priv}(\mathsf{z})),\mathsf{h}(\mathsf{x}_2,\mathsf{y}_2);\ \mathsf{wrap}(\mathsf{x}_1),\mathsf{extract}(\mathsf{x}_2) & \rightarrow & \mathsf{aenc}(\mathsf{y}_2,\mathsf{pub}(\mathsf{z}))\\
\mathsf{Wrap\,(asym/sym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1),\mathsf{h}(\mathsf{x}_2,\mathsf{priv}(\mathsf{z}));\ \mathsf{wrap}(\mathsf{x}_1),\mathsf{extract}(\mathsf{x}_2) & \rightarrow & \mathsf{senc}(\mathsf{priv}(\mathsf{z}),\mathsf{y}_1)
\end{array}$$

$$\begin{array}{rrcl}
\mathsf{Unwrap\,(sym/sym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_2),\mathsf{senc}(\mathsf{y}_1,\mathsf{y}_2);\ \mathsf{unwrap}(\mathsf{x}_1) & \xrightarrow{\mathsf{new}\ \mathsf{n}_1} & \mathsf{h}(\mathsf{n}_1,\mathsf{y}_1);\ \mathsf{extract}(\mathsf{n}_1),\ \mathsf{L}\\
\mathsf{Unwrap\,(sym/asym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{priv}(\mathsf{z})),\mathsf{aenc}(\mathsf{y}_1,\mathsf{pub}(\mathsf{z}));\ \mathsf{unwrap}(\mathsf{x}_1) & \xrightarrow{\mathsf{new}\ \mathsf{n}_1} & \mathsf{h}(\mathsf{n}_1,\mathsf{y}_1);\ \mathsf{extract}(\mathsf{n}_1),\ \mathsf{L}\\
\mathsf{Unwrap\,(asym/sym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_2),\mathsf{senc}(\mathsf{priv}(\mathsf{z}),\mathsf{y}_2);\ \mathsf{unwrap}(\mathsf{x}_1) & \xrightarrow{\mathsf{new}\ \mathsf{n}_1} & \mathsf{h}(\mathsf{n}_1,\mathsf{priv}(\mathsf{z}));\ \mathsf{extract}(\mathsf{n}_1),\ \mathsf{L}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{KeyGenerate}: & \xrightarrow{\mathsf{new}\ \mathsf{n}_1,\mathsf{k}_1} & \mathsf{h}(\mathsf{n}_1,\mathsf{k}_1);\neg\mathsf{extract}(\mathsf{n}_1),\ \mathsf{L}\\
\mathsf{KeyPairGenerate}: & \xrightarrow{\mathsf{new}\ \mathsf{n}_1,\mathsf{s}} & \mathsf{h}(\mathsf{n}_1,\mathsf{s}),\mathsf{pub}(\mathsf{s});\ \neg\mathsf{extract}(\mathsf{n}_1),\ \mathsf{L}
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{SEncrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1),\mathsf{y}_2;\ \mathsf{encrypt}(\mathsf{x}_1) & \rightarrow & \mathsf{senc}(\mathsf{y}_2,\mathsf{y}_1)\\
\mathsf{SDecrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1),\mathsf{senc}(\mathsf{y}_2,\mathsf{y}_1);\ \mathsf{decrypt}(\mathsf{x}_1) & \rightarrow & \mathsf{y}_2
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{AEncrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{priv}(\mathsf{z})),\mathsf{y}_1;\ \mathsf{encrypt}(\mathsf{x}_1) & \rightarrow & \mathsf{aenc}(\mathsf{y}_1,\mathsf{pub}(\mathsf{z}))\\
\mathsf{ADecrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{priv}(\mathsf{z})),\mathsf{aenc}(\mathsf{y}_2,\mathsf{pub}(\mathsf{z}));\ \mathsf{decrypt}(\mathsf{x}_1) & \rightarrow & \mathsf{y}_2
\end{array}$$

$$\begin{array}{lrcl}
\mathsf{Set\_Wrap}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \neg\mathsf{wrap}(\mathsf{x}_1) & \rightarrow & \mathsf{wrap}(\mathsf{x}_1)\\
\mathsf{Set\_Encrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \neg\mathsf{encrypt}(\mathsf{x}_1) & \rightarrow & \mathsf{encrypt}(\mathsf{x}_1)\\
\vdots & & & \vdots\\
\mathsf{UnSet\_Wrap}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{wrap}(\mathsf{x}_1) & \rightarrow & \neg\mathsf{wrap}(\mathsf{x}_1)\\
\mathsf{UnSet\_Encrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{encrypt}(\mathsf{x}_1) & \rightarrow & \neg\mathsf{encrypt}(\mathsf{x}_1)\\
\vdots & & & \vdots
\end{array}$$

where $\mathsf{L} = \neg\mathsf{wrap}(\mathsf{n}_1), \neg\mathsf{unwrap}(\mathsf{n}_1), \neg\mathsf{encrypt}(\mathsf{n}_1), \neg\mathsf{decrypt}(\mathsf{n}_1), \neg\mathsf{sensitive}(\mathsf{n}_1)$. The ellipsis in the set and unset rules indicates that similar rules exist for some other attributes.

Figure 2. PKCS#11 key management subset.

Example 3 *To illustrate our formal model, we will describe how the Decrypt/Wrap attack of Figure 1 is reflected in this model. We consider the signature* Σ_{enc} *and the set of attributes* $\mathcal{A}$ *given in Example 1, a set of names* $\mathcal{N} \supseteq \{\mathsf{n}_1, \mathsf{n}_2, \mathsf{k}_1, \mathsf{k}_2\}$*, a set of variables* $\mathcal{X} \supseteq \{\mathsf{x}_1, \mathsf{y}_1, \mathsf{x}_2, \mathsf{y}_2\}$*. We only use the rules* Wrap (sym/sym) *and* SDecrypt *of Figure 2. Suppose that*

- $S_0 = \{\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1), \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)\}$*, and*
- V_0 *is such that* $V_0(\mathsf{wrap}, \mathsf{n}_2) = V_0(\mathsf{decrypt}, \mathsf{n}_2) = V_0(\mathsf{sensitive}, \mathsf{n}_1) = V_0(\mathsf{extract}, \mathsf{n}_1) = \top$ *and all other attributes of* n_1 *and* n_2 *are mapped to* $\bot$.

Then we have that

$$\begin{array}{rcl}
(S_0, V_0) & \rightsquigarrow & (S_0 \cup \{\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)\}, V_0) \overset{def}{=} (S_1, V_1)\\
& \rightsquigarrow & (S_1 \cup \{\mathsf{k}_1\}, V_1)
\end{array}$$

which implies that the query $(\{\mathsf{h}(\mathsf{x}, \mathsf{y}), \mathsf{y}\}, \mathsf{sensitive}(\mathsf{x}))$ *is satisfied with the substitution* $\theta = \{\mathsf{x} \rightarrow \mathsf{n}_1,\ \mathsf{y} \rightarrow \mathsf{k}_1\}$.

4 Decidability

In this section, we first define the class of *well-moded* rules, using the notation introduced in Section 3.2. In this class, when checking the satisfiability of a query, we show that it is correct to restrict the search space and only consider well-moded terms (Theorem 1). The notion of mode is inspired from [2]. It is similar to the idea of having well-typed rules, but we prefer to call them well-moded to emphasize that we do not have a typing restriction. For the rules given in Figure 2 that model a part of PKCS#11, the notion of mode we consider allows us to bound the size of terms involved in an attack. Unfortunately, the secrecy problem is still undecidable in this setting. Therefore we restrict ourselves to a bounded number of nonces (see Section 4.3).

4.1 Preliminaries

In the following we consider a set of modes Mode and we assume that there exists a *mode* function:

$$\mathsf{M} : \Sigma \cup \mathcal{A} \times \mathbb{N} \rightarrow \mathsf{Mode}$$

such that $\mathsf{M}(f, i)$ is defined for every symbol $f \in \Sigma \cup \mathcal{A}$ and every integer i such that $1 \leq i \leq ar(f)$. We also assume that a function $\mathsf{sig} : \Sigma \cup \mathcal{A} \cup \mathcal{X} \cup \mathcal{N} \rightarrow \mathsf{Mode}$ returns the mode to which a symbol f belongs. Note that variables and nonces are also uniquely moded. As usual, we extend the function sig to terms as follows:

$$\mathsf{sig}(t) = \mathsf{sig}(\mathsf{top}(t)).$$

We will use a rule-based notation $f : \mathsf{m}_1 \times \ldots \mathsf{m}_n \to \mathsf{m}$ for each $f \in \Sigma \cup \mathcal{A}$ and $u : \mathsf{m}$ for $u \in \mathcal{N} \cup \mathcal{X}$ to define the functions M and sig: $\mathsf{M}(f, i) = \mathsf{m}_i$ for $1 \leq i \leq n$, $\mathsf{sig}(f) = \mathsf{m}$, and $\mathsf{sig}(u) = \mathsf{m}$.

We say that a position $p \neq \epsilon$ of a term t is *well-moded* if $p = p'.i$ and $\mathsf{sig}(t|_p) = \mathsf{M}(\mathsf{top}(t|_{p'}), i)$. In other words the position in a term is well-moded if the subterm at that position is of the expected mode w.r.t. to the function symbol immediately above it. If a position is not well-moded, it is *ill-moded*. By convention, the root position of a term is ill-moded. A term is well-moded if all its non root positions are well-moded. A literal ℓ such that $\ell = a$ or $\ell = \neg a$ is well-moded if a is well-moded. This notion is extended as expected to sets of terms, rules, queries and states. For a state (S, V), we require that the valuation V is well-moded, i.e. for every $a \in dom(V)$, we have that a is well-moded.

Note that any term can be seen as a well-moded term if there is a unique mode, e.g. Msg, and any symbol $f \in \Sigma \cup \mathcal{A} \cup \mathcal{X} \cup \mathcal{N}$ is such that

$$f : \mathsf{Msg} \times \ldots \times \mathsf{Msg} \to \mathsf{Msg}.$$

However, we will use modes which imply that the message length of well-moded terms is bounded which will allow us to reduce the search space.

Example 4 *We consider the following set of modes:*

$$\mathsf{Mode} = \{\mathsf{Cipher}, \mathsf{Key}, \mathsf{Seed}, \mathsf{Nonce}, \mathsf{Handle}, \mathsf{Attribute}\}.$$

The following rules define the mode and signature functions of the associated function symbol:

$$\begin{array}{rcl}
\mathsf{h} & : & \mathsf{Nonce} \times \mathsf{Key} \to \mathsf{Handle} \\
\mathsf{senc} & : & \mathsf{Key} \times \mathsf{Key} \to \mathsf{Cipher} \\
\mathsf{aenc} & : & \mathsf{Key} \times \mathsf{Key} \to \mathsf{Cipher} \\
\mathsf{pub} & : & \mathsf{Seed} \to \mathsf{Key} \\
\mathsf{priv} & : & \mathsf{Seed} \to \mathsf{Key} \\
\mathsf{att} & : & \mathsf{Nonce} \to \mathsf{Attribute} \quad \textit{for all } \mathsf{att} \in \mathcal{A} \\
\mathsf{x}_1, \mathsf{x}_2, \mathsf{n}_1, \mathsf{n}_2 & : & \mathsf{Nonce} \\
\mathsf{y}_1, \mathsf{y}_2, \mathsf{k}_1, \mathsf{k}_2 & : & \mathsf{Key} \\
\mathsf{z}, \mathsf{s} & : & \mathsf{Seed}
\end{array}$$

The rules described in Figure 2 are well-moded w.r.t. the mode and signature function described above. This is also the case of the following rules which represent the deduction capabilities of the attacker:

$$\begin{array}{rcl}
\mathsf{y}_1,\ \mathsf{y}_2 & \to & \mathsf{senc}(\mathsf{y}_1, \mathsf{y}_2) \\
\mathsf{senc}(\mathsf{y}_1, \mathsf{y}_2),\ \mathsf{y}_2 & \to & \mathsf{y}_1 \\
\mathsf{y}_1,\ \mathsf{y}_2 & \to & \mathsf{aenc}(\mathsf{y}_1, \mathsf{y}_2) \\
\mathsf{aenc}(\mathsf{y}_1, \mathsf{pub}(\mathsf{z})),\ \mathsf{priv}(\mathsf{z}) & \to & \mathsf{y}_1 \\
\mathsf{aenc}(\mathsf{y}_1, \mathsf{priv}(\mathsf{z})),\ \mathsf{pub}(\mathsf{z}) & \to & \mathsf{y}_1 \\
\mathsf{z} & \to & \mathsf{pub}(\mathsf{z})
\end{array}$$

The derivation $(S_0, V_0) \leadsto^* (S_1 \cup \{\mathsf{k}_1\}, V_1)$ described in Example 3 is well-moded in the sense that each of its states is well-moded. However, let $S_0 = \{\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_1\}$ and V_0 be a function such that $dom(V_0) = \emptyset$. Even if the state (S_0, V_0) and the rule involved in this step are well-moded, the following derivation is not:

$$(S_0, V_0) \leadsto (S_0 \cup \{\mathsf{senc}(\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_1), V_0).$$

The term $\mathsf{senc}(\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_1)$ is not well-moded because of its subterm $\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$.

4.2 Existence of a well-moded derivation

We now show that in a system induced by well-moded rules, only well-moded terms need to be considered when checking for the satisfiability of a well-moded query. In the following, we assume a given mode and signature function.

The key idea to reduce the search space to well-moded terms is to show that whenever a state (S, V) is reachable from an initial well-moded state, we have that:

- the partial valuation V is necessarily well-moded (Lemma 1), and
- any ill-moded term v' occurring in a term in S is itself deducible (Lemma 2).

Lemma 1 *Let $\mathcal{R}$ be a set of well-moded rules and (S_0, V_0) be a well-moded state. Let (S, V) be a state such that $(S_0, V_0) \leadsto^* (S, V)$. We have that V is well-moded.*

Proof. We prove this result by induction on the length k of the derivation:

$$(S_0, V_0) \leadsto (S_1, V_1) \leadsto \ldots \leadsto (S_k, V_k) = (S, V).$$

Base case: $k = 0$. In this case, we have that $V = V_0$ and we easily conclude thanks to the fact that (S_0, V_0) is a well-moded state.

Induction step: $k > 0$. We assume that $V_0, \ldots, V_{k-1}$ are well-moded. Now, assume that $a \in dom(V_k)$. If a is already in $dom(V_{k-1}) \subseteq dom(V_k)$ then we can conclude by applying our induction hypothesis. Otherwise, there exists a fresh renaming of a well-moded rule in $\mathcal{R}$, say

$$\mathsf{R} : \ t_1, \ldots, t_n, L \to u_1, \ldots, u_p, L'$$

and a substitution θ grounding for R such that for all $\ell \in L\theta$, if $\ell = a$ or $\ell = \neg a$ we have $a \in dom(V_{k-1})$.

Since $a \notin dom(V_{k-1})$, we have that $a \in L'\theta$ or $\neg a \in L'\theta$. In both cases, we have to show that a is well-moded. We prove this by contradiction. Assume that there exists $t \in st(a)$ such that $t \neq a$ and t occurs at an ill-moded position in a. Since $vars(L') \subseteq vars(L)$ and R is well-moded, we have that t occurs in $L\theta$ at an ill-moded position. This contradicts the fact that V_{k-1} is well-moded and allows us to conclude. □

Lemma 2 *Let $\mathcal{R}$ be a set of well-moded rules and (S_0, V_0) be a well-moded state. Let (S, V) be a state such that $(S_0, V_0) \leadsto^* (S, V)$. Let $v \in S$ and v' be a subterm of v which occurs at an ill-moded position. Then, we have that $v' \in S$.*

Proof. We prove this result by induction on the length k of the derivation:

$$(S_0, V_0) \leadsto (S_1, V_1) \leadsto \ldots \leadsto (S_k, V_k) = (S, V).$$

Base case: $k = 0$. In this case, we have $v \in S_0$ and the term v' occurs at an ill-moded position in v. By hypothesis, the term v is well-moded. Hence, we have that $v' = v \in S$.

Induction step: $k > 0$. Assume that $v \in S_k$. If v is already in $S_{k-1} \subseteq S_k$ then we can conclude by applying our induction hypothesis. Otherwise, there exists a fresh renaming of a well-moded rule in $\mathcal{R}$, say

$$\mathsf{R}: \; t_1, \ldots, t_n, L \rightarrow u_1, \ldots, u_p, L'$$

and a substitution θ grounding for R such that: $\{t_i\theta \mid 1 \leq i \leq n\} \subseteq S_{k-1}$. Let $v' \in st(v)$. Either $v' = v$ and hence $v' \in S_k = S$ or by well-modedness of the rule, $v' \in st(x\theta)$ for some variable $x \in vars(\{u_1, \ldots, u_p\})$. Since by hypothesis we have that $vars(\{u_1, \ldots, u_p\}) \subseteq vars(\{t_1, \ldots, t_n\})$, we easily deduce that v' is a subterm which occurs in S_{k-1} at an ill-moded position. By induction hypothesis, we deduce that $v' \in S_{k-1}$ and hence $v' \in S_k$. In both cases, we have that $v' \in S$. □

Before we prove our main result, that states that only well-moded terms need to be considered when checking for satisfiability of well-moded queries, we introduce a transformation which transforms any term to a well-moded term. We show that when we apply this transformation to a derivation, we obtain again a derivation.

We define for each mode $\mathsf{m} \in \mathsf{M}$ a function $\overline{\cdot}^{\mathsf{m}}$ over ground terms that replaces any ill-moded subterm by a well-moded term, say t_{m}, of the expected mode. In the remainder, we assume given those terms t_{m} (one per mode).

Definition 3 ($\overline{\cdot}^{\mathsf{m}}$, $\overline{\cdot}$) *For each mode $\mathsf{m} \in \mathsf{M}$ we define inductively a function $\overline{\cdot}^{\mathsf{m}}$ as follows:*

- $\overline{n}^{\mathsf{m}} = \begin{cases} n & \textit{if } n \in \mathcal{N} \textit{ and } \mathsf{sig}(n) = \mathsf{m} \\ t_{\mathsf{m}} & \textit{otherwise} \end{cases}$

- $\overline{f(v_1, \ldots, v_n)}^{\mathsf{m}} = \begin{cases} f(\overline{v_1}^{\mathsf{m}_1}, \ldots, \overline{v_n}^{\mathsf{m}_n}) & \\ \qquad \textit{if } f : \mathsf{m}_1 \times \ldots \mathsf{m}_n \rightarrow \mathsf{m} & \\ t_{\mathsf{m}} & \textit{otherwise} \end{cases}$

The function $\overline{\cdot}$ is defined as $\overline{v} = \overline{v}^{\mathsf{sig}(v)}$.

Those functions are extended to sets of terms as expected. Note that, by definition, we have that $\overline{u}^{\mathsf{m}}$ is a well-moded term of mode m and $\overline{u}$ is a well-moded term of mode $\mathsf{sig}(u)$.

In Proposition 1 we show that this transformation allows us to map any derivation to a well-moded derivation. This well-moded derivation is obtained by applying at each step the same rule, say R. However, while the original derivation may use an instance $\mathsf{R}\theta$ of this rule the transformed derivation will use the instance $\mathsf{R}\theta'$, where θ' is obtained from θ as described in the following lemma.

Lemma 3 *Let v be a well-moded term and θ be a grounding substitution for v. Let θ' be the substitution defined as follows:*

- *$dom(\theta') = dom(\theta)$, and*

- *$x\theta' = \overline{x\theta}^{\mathsf{sig}(x)}$ for $x \in dom(\theta')$.*

We have that $\overline{v\theta}^{\mathsf{sig}(v)} = v\theta'$.

Proof. The proof is done by structural induction on v. *Base cases:* If v is a name the result is obvious. If v is a variable x then, by definition of θ', we have that:

$$\overline{v\theta}^{\mathsf{sig}(v)} = \overline{x\theta}^{\mathsf{sig}(x)} = x\theta' = v\theta'.$$

Induction step: $v = f(v_1, \ldots, v_n)$ for some function symbol $f \in \Sigma \cup \mathcal{A} \cup \mathcal{N}$. We assume w.l.o.g. that $f : \mathsf{m}_1, \ldots, \mathsf{m}_n \rightarrow \mathsf{m}$. Moreover, since v is well-moded, we have that $\mathsf{sig}(v_i) = \mathsf{m}_i$ for every i such that $1 \leq i \leq n$. Hence, we have that:

$$\begin{array}{lll} \overline{v\theta}^{\mathsf{sig}(v)} & = \overline{f(v_1, \ldots, v_n)\theta}^{\mathsf{sig}(f)} & \\ & = f(\overline{v_1\theta}^{\mathsf{m}_1}, \ldots, \overline{v_n\theta}^{\mathsf{m}_n}) & \text{by def. of } \overline{\cdot}^{\mathsf{sig}(f)} \\ & = f(v_1\theta', \ldots, v_n\theta') & \text{by induction hyp.} \\ & = f(v_1, \ldots, v_n)\theta' & \\ & = v\theta' & \end{array}$$

□

Proposition 1 *Let $\mathcal{R}$ be a set of well-moded rules. Let (S_0, V_0) be a well-moded state and consider the derivation:*

$$(S_0, V_0) \leadsto (S_1, V_1) \leadsto \ldots \leadsto (S_k, V_k).$$

Moreover, for each mode $\mathsf{m} \in \{\mathsf{sig}(t) \mid t \in \mathcal{PT}(\Sigma, \mathcal{N})\}$*, we assume that there exists a term of mode* m *such that* $t_\mathsf{m} \in S_0$ *and* $\overline{\cdot}$ *is defined w.r.t. to these* t_m*'s.*

We have that $(\overline{S_0}, V_0) \rightsquigarrow (\overline{S_1}, V_1) \rightsquigarrow \ldots \rightsquigarrow (\overline{S_k}, V_k)$ *by using the same rules (but different instances).*

Proof. We show this result by induction on k.
Base case: $k = 0$. In such a case the result is obvious.
Induction step: $k > 0$. In such a case, we have that

$$(S_0, V_0) \rightsquigarrow^* (S_{k-1}, V_{k-1}) \rightsquigarrow (S_k, V_k)$$

By induction hypothesis, we know that

$$(\overline{S_0}, V_0) \rightsquigarrow^* (\overline{S_{k-1}}, V_{k-1}).$$

To conclude, it remains to show that

$$(\overline{S_{k-1}}, V_{k-1}) \rightsquigarrow (\overline{S_k}, V_k)$$

By hypothesis, we have that $(S_{k-1}, V_{k-1}) \rightsquigarrow (S_k, V_k)$ with a fresh renaming of a well-moded rule in $\mathcal{R}$, say:

$$\mathsf{R}: \;\; t_1, \ldots, t_n, L \to u_1, \ldots, u_p, L'.$$

This means that there exists a substitution θ such that

- $\{t_1\theta, \ldots, t_n\theta\} \subseteq S_{k-1}$, and
- $V_{k-1}(L\theta) = \top$.

We show that $(\overline{S_{k-1}}, V_{k-1}) \rightsquigarrow (\overline{S_k}, V_k)$ by using the rule R and the substitution θ' obtained from θ as in Lemma 3, i.e. $dom(\theta') = dom(\theta)$ and $x\theta' = \overline{x\theta}^{\mathsf{sig}(x)}$ for any $x \in dom(\theta)$.

Thanks to Lemma 3, for any well-moded term v, we have that $\overline{v\theta}^{\mathsf{sig}(v)} = v\theta'$. Moreover, the only case where $\overline{v\theta} \neq \overline{v\theta}^{\mathsf{sig}(v)}$ is when v is a variable, say x, and $\mathsf{sig}(x) \neq \mathsf{sig}(x\theta)$. Thus, we are in one of the following cases

- either $\overline{v\theta} = v\theta'$, or
- v is a variable say x and $x\theta' = t_{\mathsf{sig}(x)} \in S_0$.

Now, it is easy to see that:

- for each t_j, since $t_j\theta \in S_{k-1}$ and $t_{\mathsf{sig}(x)} \in S_{k-1}$, we have $\overline{t_j\theta} \in \overline{S_{k-1}}$ and thus $t_j\theta' \in \overline{S_{k-1}}$;
- for each $a \in L$ (resp. $\neg a \in L$), since a cannot be a variable, we have that $\overline{a\theta} = a\theta'$. Thanks to Lemma 1, we know that V_{k-1} and V_k are necessarily well-moded. Hence we have that $a\theta = a\theta'$ for any $(\neg)a \in L \cup L'$.

We deduce that we can apply the same rule R with the substitution θ'. Let (S', V') be the resulting state. It remains to show that $(S', V') = (\overline{S_k}, V_k)$.

Since we have that $a\theta = a\theta'$ for any $(\neg)a \in L \cup L'$, the valuation is updated in the same way, hence $V' = V_k$. To show that $S' = \overline{S_k}$, the only problematic case is when u_j is a variable, say x, and $\mathsf{sig}(x) \neq \mathsf{sig}(x\theta)$. By hypothesis we have that $vars(\{u_1, \ldots, u_p\}) \subseteq vars(\{t_1, \ldots, t_n\})$. This allows us to deduce that $x \in vars(\{t_1, \ldots, t_n\})$. Hence $x\theta \in st(v)$ at an ill-moded position in v for some $v \in S_{k-1}$. By Lemma 2, we deduce that $x\theta \in S_{k-1}$, hence $\overline{x\theta} \in \overline{S_{k-1}}$ and thus $\overline{u_j\theta} \in S'$ since $\overline{S_{k-1}} \subseteq S'$. □

By relying on Proposition 1, it is easy to prove the following result.

Theorem 1 *Let* $\mathcal{R}$ *be a set of well-moded rules. Let* $q_0 = (S_0, V_0)$ *be a well-moded state such that for each mode* $\mathsf{m} \in \{\mathsf{sig}(t) \mid t \in \mathcal{PT}(\Sigma, \mathcal{N})\}$*, there exists a term* $t_\mathsf{m} \in S_0$ *of mode* m*. Let* Q *be a well-moded query that is satisfiable. Then there exists a well-moded derivation witnessing this fact.*

Note that we do not assume that an implementation enforces well-modedness. We allow an attacker to yield derivations that are not well-formed. Our result however states that whenever there exists an attack that uses a derivation which is not well-formed there exists another attack that is. Our result is arguably even more useful than an implementation that would enforce typing: it seems unreasonable that an implementation could detect whether a block has been encrypted once or multiple times, while our result avoids consideration of multiple encryptions, as such a term is ill-moded with the modes given in Example 4.

4.3 Decidability result

Unfortunately, Theorem 1 by itself is not very informative. As already noted, it is possible to have a single mode Msg which implies that all derivations are well-moded. However, the modes used in our modelling of PKCS# 11 (see Example 4) imply that all well-moded terms have bounded message length. It is easy to see that well-moded terms have bounded message length whenever the graph on modes that is defined by the functions M and sig is acyclic (the graph whose set of vertices is Mode with edges between modes m_i $(1 \leq i \leq n)$ and m whenever there exists a rule $f : \mathsf{m}_1 \times \ldots \times \mathsf{m}_k \to \mathsf{m}$). Note that for instance a rule which contains nested encryption does not yield a bound on the message size.

However, bounded message length is not sufficient for decidability. Indeed, undecidability proofs [10, 16] for security protocols with bounded message length and unbounded number of nonces are easily adapted to our setting. We only need to consider rules of the form $T \xrightarrow{\mathsf{new}\ \tilde{n}} T'$ (no literal) to realize their encodings of the Post Correspondence Problem. Therefore we bound the number of atomic data of each mode, and obtain the following corollary of Theorem 1:

Corollary 1 *Let $\mathcal{R}$ be a set of well-moded rules such that well-modedness implies a bound on the message length. Let $q_0 = (S_0, V_0)$ be a well-moded state such that for each mode $\mathsf{m} \in \{\mathsf{sig}(t) \mid t \in \mathcal{PT}(\Sigma, \mathcal{N})\}$, there exists a term $t_{\mathsf{m}} \in S_0$ of mode m. The problem of deciding whether the query Q is satisfiable is decidable when the set of names $\mathcal{N}$ is finite.*

Our main application is the fragment of PKCS#11 described in Figure 2. Thanks to Corollary 1, we are able to bound the search space and to realize some experiments with a well-known model-checker, NuSMV, [3].

5 Analysing PKCS#11

In this section, we describe the implementation of the decision procedure arising from the decidability result (Corollary 1) for a bounded number keys and handles. As explained in Section 1, our formal work was primarily motivated by the example of RSA PKCS#11, which is widely deployed in industry, but other APIs such as the API of the Trusted Platform Module (TPM) will also require global mutable state to be modelled.

PKCS#11 is described in a large and complex specification, running to 392 pages. We model here only the key management operations at the core of the API. We omit the `DeriveKey` command, all of the commands from the session, object, slot and token management function sets, the digest, signing and verification functions, and the random number generating functions. We include key generation, import and export, encryption and decryption of data, and setting and unsetting of key attributes. We assume, as suggested in PKCS#11 [14, p. 31] that the intruder is able to freely hijack user sessions, and is thus able to send arbitrary sequences of commands to the interface with arbitrary parameters from his knowledge set. Following on from our theoretical work, we further assume only a fixed bounded number of handles are available, and a bounded number of atomic keys. We do not *a priori* bound the number of times each command may be executed, but this is implicitly bounded by the finite vocabulary of well-moded terms available, since a rule will not be executed twice with exactly the same state and intruder knowledge inputs. Finally, note that we model the setting of attributes of keys stored on the device via a series of rules: one to set and one to unset each attribute. In the real API, there is a single command `C_SetAttributeValues`, to which the new values for the attributes are supplied as parameters. We found it more convenient to encode this in separate commands to facilitate the addition of constraints to certain attribute setting and unsetting operations.

5.1 Methodology

As we described in Section 1, PKCS#11 is a standard designed to promote interoperability, not a tightly defined protocol with a particular goal. As such, the aim of our experiments was to analyse a number of different configurations in order to validate our approach. Roughly speaking, our methodology was to start with a configuration involving only symmetric keys, and continue to restrict the API until a secure configuration was found. We then added asymmetric keypairs, and repeated the process. Finally we carried out some experiments modelling the algebraic properties of the ECB mode of encryption.

5.2 Generating propositional models

By Theorem 1, once we have bounded the number of handles and keys, we only have to consider a finite set of possible terms in the intruder's knowledge. Our approach is to encode each possible term as a propositional variable, which will be true just when the term is in the intruder's knowledge set. In addition to this, we have the attributes that constitute the state of the system. Since we have bounded the number of handles, and we need only consider attributes applied to handles by our well-modedness result, we can also encode the state as a finite number of propositional variables: one for each attribute applied to each handle. A variable is true when the attribute is set for that handle.

We can now generate a propositional model for the API by generating all the ground instances of the API rules, and compiling these to our propositional encoding. This is currently done by a Perl script, which accepts parameters defining the exact configuration to be modelled, but it should be easy enough to produce a general-purpose compiler for our class of rules. We define a configuration by:

1. the number of symmetric keys, the number of

asymmetric key pairs, and the number of available handles for each key;

2. the initial knowledge of the intruder;

3. the initial state of the attributes.

Note that having set the number of handles available, we are able to pre-compute the names of the handles rather than having to generate fresh names during the unwrap and generate commands. Since all commands which generate fresh handles return their values unencrypted, we can be sure that a handle is fresh when is it is generated in a command simply by checking that it is not yet known to the intruder. As a further optimisation, we include handles for all the keys the intruder can generate in a particular configuration in his initial knowledge, and remove the key generation commands.

To facilitate the generation of the models for our program of experiments, our scripts also accept the following parameters:

1. A list of sticky attributes, i.e. those which, once set, cannot be unset, and those which once unset, cannot be set. Footnotes 11 and 12 in the PKCS standard mark these attributes [14, Table 15]. We add further attributes to the list during our experiments, as detailed below. Adding an attribute to the list causes the generation script to omit the appropriate Set or Unset commands from the model.

2. A list of conflicting attributes, i.e. for each attribute a a list of conflicting attributes $\mathsf{a}_1, \mathsf{a}_2, \ldots$ such that for a given handle $\mathsf{h}(\mathsf{n}, \mathsf{k})$, attribute a may not be set on that handle if any of the a_i are also set. Adding attributes to this list causes the script to add appropriate conditions to the left hand side of the Set rules.

The propositional encoding of the API model is generated in a syntax suitable for the model checker NuSMV, [3]. We then ask NuSMV to check whether a security property holds, which is a reachability property in our transition system. In all our experiments we are concerned with a single security property, the secrecy of sensitive keys.

5.3 Experiments with PKCS#11

All the files for our experiments are available via http from `http://www.lsv.ens-cachan.fr/~steel/pkcs11`. We describe each experiment below and summarise in Table 1. In the figures describing attacks, we sometimes omit the values of attributes whose value is inconsequential to the attack, for the sake of clarity. Similarly, we omit unused terms from the intruder's initial knowledge.

Experiment 1. In our first four experiments, we model a PKCS#11 configuration with 3 symmetric keys: one is a sensitive key, k_1, stored on the device, for which the intruder knows the handle but not the true value of the key. The second, k_2, is also loaded onto the device, and the intruder has a handle but not the true value. The third is the intruder's own key, k_3, which is not loaded onto the device in the initial state. We start with a configuration in which the only restrictions on attribute setting and unsetting are those described in the manual. As expected, we immediately rediscover Clulow's key separation attack for the attributes decrypt and wrap (see Figure 1).

Experiment 2. We modify the configuration from Experiment 1 by applying Clulow's first suggestion: that attribute changing operations be prevented from allowing a stored key to have both wrap and decrypt set. Note that in order to do this, it is not sufficient merely to check that decrypt is unset before setting wrap, and to check wrap in unset before setting decrypt. One must also add wrap and decrypt to the list of sticky attributes which once set, may not be unset, or the attack is not prevented, [17]. Having applied these measures, we discovered a previously unknown attack, given in Figure 3. The intruder imports his own key k_3 by first encrypting it under k_2, and then unwrapping it. He can then export the sensitive key k_1 under k_3 to discover its value.

Initial state: The intruder knows the handles $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$, $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$ and the key k_3; n_1 has the attributes sensitive and extract set whereas n_2 has the attributes unwrap and encrypt set.

Trace:

SEncrypt:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$, k_3	$\rightarrow$	$\mathsf{senc}(\mathsf{k}_3, \mathsf{k}_2)$
Unwrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$, $\mathsf{senc}(\mathsf{k}_3, \mathsf{k}_2)$	$\xrightarrow{\text{new } \mathsf{n}_3}$	$\mathsf{h}(\mathsf{n}_3, \mathsf{k}_3)$
Set_wrap:	$\mathsf{h}(\mathsf{n}_3, \mathsf{k}_3)$	$\rightarrow$	$\mathsf{wrap}(\mathsf{n}_3)$
Wrap:	$\mathsf{h}(\mathsf{n}_3, \mathsf{k}_3)$, $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$	$\rightarrow$	$\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3)$
Intruder:	$\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3)$, k_3	$\rightarrow$	k_1

Figure 3. Attack discovered in Experiment 2

Experiment 3. To prevent the attack shown in Figure 3, we add encrypt and unwrap to the list of conflicting attribute pairs. Another new attack is discovered (see Figure 4) of a type discussed by Clulow, [4, Section 2.3]. Here the key k_2 is first wrapped under k_2 itself, and then unwrapped, gaining a new handle $h(n_3, k_2)$. The intruder then wraps k_1 under k_2, and sets the decrypt attribute on handle $h(n_3, k_2)$, allowing him to obtain k_1.

Initial state: The intruder knows the handles $h(n_1, k_1)$, $h(n_2, k_2)$; n_1 has the attributes sensitive, extract and whereas n_2 has the attribute extract set.

Trace:

Set_wrap:	$h(n_2, k_2)$	$\rightarrow$	$wrap(n_2)$
Wrap:	$h(n_2, k_2), h(n_2, k_2)$	$\rightarrow$	$senc(k_2, k_2)$
Set_unwrap:	$h(n_2, k_2)$	$\rightarrow$	$unwrap(n_2)$
Unwrap:	$h(n_2, k_2), senc(k_2, k_2)$	$\xrightarrow{\text{new } n_4}$	$h(n_4, k_2)$
Wrap:	$h(n_2, k_2), h(n_1, k_1)$	$\rightarrow$	$senc(k_1, k_2)$
Set_decrypt:	$h(n_4, k_2)$	$\rightarrow$	$decrypt(n_4)$
SDecrypt:	$h(n_4, k_2), senc(k_1, k_2)$	$\rightarrow$	k_1

Figure 4. Attack discovered in Experiment 3

Experiment 4. We attempt to prevent the attack in Figure 4 by adding wrap and unwrap to our list of conflicting attribute pairs. We find no attack with respect to our model, even when the model is extended up to 4 possible handles for each key. We now proceed to add asymmetric keypairs.

Experiment 5. We now add two asymmetric keypairs to the model. One, $(pub(s_1), priv(s_1))$, is loaded onto the device and is unknown to the intruder (apart from the handle). The other, $(pub(s_2), priv(s_2))$, is the intruder's own keypair, but is not loaded on to the device. We now rediscover Clulow's Trojan Wrapped Key attack, [4, Section 3.5]. Note that in a model with explicit destructors for decryption, a similar attack would be found on the configuration in experiment 4. Extending our model to deal with this is an area for further work (see section 6). We also note that Clulow's other Trojan Key attack, [4, Section 3.4], is now no longer possible: Clulow analysed version 2.01 of the standard, and observed that the Wrap command accepts a clear public key as input, allowing a Trojan Public Key attack - the intruder generates his own keypair, and then supplies the public key as a wrapping key. In the current version of the standard (2.20), the command accepts only a handle for a public key, which must be loaded on to the device.

Experiment 6. Version 2.20 of the PKCS#11 standard includes a new feature intended to improve security: trusted keys. Two more attributes are introduced: wrap_with_trusted and trusted. In addition to testing that a key to be wrapped is extractable, Wrap now tests that if the key to be wrapped has wrap_with_trusted set, then the wrapping key must have trusted set. Only the security officer (SO) can mark a key as trusted. Additionally, wrap_with_trusted is a sticky attribute - once set, it may not be unset.

This mechanism would appear to have some potential: as long as the security officer only logs into the device when it is connected to a trusted terminal, he should be able to keep his PIN secure, and so be able to control which keys are marked as trusted. We took our configuration from Experiment 5, and added the trusted key features, marking n_1 as wrap_with_trusted, and n_2 as trusted. We discover another attack, given in Figure 5. Here, the intruder first attacks the trusted wrapping key, and then obtains the sensitive key.

Initial state: The intruder knows the handles $h(n_1, k_1)$, $h(n_2, k_2)$ and the key k_3; n_1 has the attributes sensitive, extract and wrap_with_trusted whereas n_2 has the attributes extract and trusted set. The intruder also knows the public key $pub(s_1)$ and its associated handle $h(n_3, priv(s_1))$; n_3 has the attribute unwrap set.

Trace:

Intruder:	$k_3, pub(s_1)$	$\rightarrow$	$aenc(k_3, pub(s_1))$
Set_unwrap:	$h(n_3, priv(s_1))$	$\rightarrow$	$unwrap(n_3)$
Unwrap:	$aenc(k_3, pub(s_1))$ $h(n_3, priv(s_1))$	$\xrightarrow{\text{new } n_4}$	$h(n_4, k_3)$
Set_wrap:	$h(n_4, k_3)$	$\rightarrow$	$wrap(n_4)$
Wrap:	$h(n_4, k_3), h(n_2, k_2)$	$\rightarrow$	$senc(k_2, k_3)$
Intruder:	$senc(k_2, k_3)$, k_3	$\rightarrow$	k_2
Set_wrap:	$h(n_2, k_2)$	$\rightarrow$	$wrap(n_2)$
Wrap:	$h(n_2, k_2), h(n_1, k_1)$	$\rightarrow$	$senc(k_1, k_2)$
Intruder:	$senc(k_1, k_2)$, k_2	$\rightarrow$	k_1

Figure 5. Attack discovered in Experiment 6

Experiment 7. In Experiment 7 we prevent the attack in Figure 5 by marking n_2 as wrap_with_trusted. We obtain a configuration which is secure in our model.

Experiment 8. We extend the initial state from Experiment 7 by setting the attributes trusted and wrap_with_trusted for the public keypair $(priv(s_1), pub(s_1))$. Our security property continues to hold in this model.

Experiment 9. Clulow has shown vulnerabilities in PKCS#11 arising from the use of double length 3DES keys and electronic code book (ECB) mode encryption. We extended our model to capture the properties of ECB using the rules in Figure 6, introducing the symbol spenc for encryption of pairs, and modelling the assumption that single-length DES keys can be cracked by brute force. Note that these rules are still well-moded. In a model with three atomic keys and two double-length keys, we rediscover Clulow's weaker key attack, [4, Section 2.4].

Additional Intruder Rules:

$$\begin{array}{rcl}
\mathsf{y}_1,\, \mathsf{y}_2 & \rightarrow & \mathsf{pair}(\mathsf{y}_1, \mathsf{y}_2) \\
\mathsf{pair}(\mathsf{y}_1, \mathsf{y}_2) & \rightarrow & \mathsf{y}_1 \\
\mathsf{pair}(\mathsf{y}_1, \mathsf{y}_2) & \rightarrow & \mathsf{y}_2 \\
\mathsf{senc}(\mathsf{y}_1, \mathsf{y}_3), \mathsf{senc}(\mathsf{y}_2, \mathsf{y}_3) & \rightarrow & \mathsf{spenc}(\mathsf{pair}(\mathsf{y}_1, \mathsf{y}_2), \mathsf{y}_3) \\
\mathsf{spenc}(\mathsf{pair}(\mathsf{y}_1, \mathsf{y}_2), \mathsf{y}_3) & \rightarrow & \mathsf{senc}(\mathsf{y}_1, \mathsf{y}_3) \\
\mathsf{spenc}(\mathsf{pair}(\mathsf{y}_1, \mathsf{y}_2), \mathsf{y}_3) & \rightarrow & \mathsf{senc}(\mathsf{y}_2, \mathsf{y}_3) \\
\mathsf{senc}(\mathsf{y}_1, \mathsf{y}_2), \mathsf{y}_1 & \rightarrow & \mathsf{y}_2 \\
\mathsf{senc}(\mathsf{y}_1, \mathsf{y}_1) & \rightarrow & \mathsf{y}_1
\end{array}$$

Modes:

$$\begin{array}{llcl}
\mathsf{pair} & \mathsf{Key} \times \mathsf{Key} & \rightarrow & \mathsf{Pair} \\
\mathsf{spenc} & \mathsf{Pair} \times \mathsf{Key} & \rightarrow & \mathsf{CipherPair}
\end{array}$$

Figure 6. Additional Intruder Rules (ECB)

Experiment 10. In Experiment 10, we prevent Clulow's weaker key attack by disallowing the wrap command for accepting a single length key to wrap a double length key. We now rediscover Clulow's ECB based attack, [4, Section 2.2].

As Table 1 shows, run times vary from a few seconds to a few minutes. However, we note that increasing the size of the model by adding more handles often leads to a model larger than NuSMV can store in our compute server's 4Gb of RAM. The largest models in Table 1 (no. s 5, 6, 7 and 8) have 128 variables. This is an area for future work: see Section 6.

Our experiments give an idea of how difficult the secure configuration of a PKCS#11 based API is. Although none of the attacks are particularly subtle or complex, there are so many variations that without some formal work, it is very hard to be sure that a certain configuration is secure. We have seen that for the particular models under investigation here, a secure configuration of PKCS#11 must include $\mathsf{wrap/decrypt}$, $\mathsf{unwrap/encrypt}$, and $\mathsf{wrap/unwrap}$ as conflicting attributes. However, one can easily imagine changes in the scenarios giving rise to more attacks. The interested reader is invited to download the scripts from `http://www.lsv.ens-cachan.fr/~steel/pkcs11` and try out her own configurations. The output from NuSMV for each experiment is available at the same address. The standard NuSMV output gives the details of variables whose value has changed at each step, i.e. changes in attribute values and newly acquired intruder knowledge. Attack traces can be easily extracted from this output.

As a final caveat, we note that Clulow's paper gives a number of vulnerabilities that take account of particular details of the cryptographic algorithms in use. Our free algebra model does not take into account equational theories of cryptographic functions. Security at the abstract level of our models is no guarantee of security from these lower level vulnerabilities.

6 Conclusion

In summary, we have presented a model for the analysis of security APIs with mutable global state, such as PKCS#11. We have given a well-modedness condition for the API that leads to decidability of secrecy properties when the number of fresh names is bounded. We have formalised the core of PKCS#11 in our model, and used an automated implementation of our decision procedure to both discover some new attacks, and to show security (in our bounded model) of a number of configurations.

Related work. We are aware of three previous efforts to formally analyse configurations of PKCS#11. Youn [18] used the first-order theorem prover Otter, and included only the commands needed for Clulow's first attack (Figure 1). This model had one handle for each key, preventing the discovery of attacks like that in Figure 4, and a monotonic model of state, since predicates $T(x, s)$ specifying that x is true in state s persist after state changes. This would allow an intruder to take two mutually exclusive steps from the same state, permitting false attacks. Tsalapati [17] used the AVISPA protocol analysis tools, included all the key management commands, but also used a monotonic model of state, and one handle for each key. She rediscovered a number of Clulow's attacks, but the limitations of the model prevented the discovery of the attacks we have shown here. In unpublished work, Steel and Carbone adapted Tsalapati's models to account correctly for non-monotonic state, using the sets supported by the AVISPA modelling language. However, the number of sessions and hence command calls had to be bounded, and for non-trivial bounds, the performance of the AVISPA back-ends rapidly deteriorated. The main problem is that we were unable to specify our well-modedness condition to the AVISPA tools, and hence the combinatorial possibilities for the intruder actions are too great. The model we have described in this paper accounts for non-monotonic mutable global state, allows analysis for unbounded sessions (although we bound the number of atomic data), and has shown reasonable performance as evidenced by the discovery of new attacks and (bounded) verifications for non-trivial models.

In future work, we aim to prove results allowing us to draw conclusions about security of the unbounded model while analysing a bounded number of keys and handles. We also plan to cover more commands, more attributes, and more algebraic properties of the operations used in PKCS#11, in particular the explicit decryption operator. This will require more theoretical work as well as optimisations to our implementation to combat the combinatorial explosion of possible intruder terms: adapting an existing protocol analysis tool (preferably one of the AVISPA tools which already support state) to unbounded sessions and well-moded terms is one possible approach.

Acknowledgments. We would like to thank Mathieu Baudet for his detailed comments on an early version of this paper.

Exp.	no. sym keys	handles each sym key	no. asym keypairs	handles each asym key	dec/ wrap	enc/ unwrap	wrap/ unwrap	Trusted keys	ECB	Attack	Time
1	3	2	0	0	-	-	-	-	-	Fig 1	4.5s
2	3	2	0	0	×	-	-	-	-	Fig 3	7.6s
3	3	2	0	0	×	×	-	-	-	Fig 4	1.9s
4	3	4	0	0	×	×	×	-	-	-	1m1s
5	3	2	2	1	×	×	×	-	-	[4, §3.5]	10m30s
6	3	2	2	1	×	×	×	×	-	Fig 5	3m28s
7	3	2	2	1	×	×	×	×	-	-	1m21s
8	3	2	2	1	×	×	×	×	-	-	1m21s
9	3	1	0	0	×	×	×	-	×	[4, §2.4]	3s
10	3	1	0	0	×	×	×	-	×	[4, §2.2]	5s

Table 1. Summary of Experiments. Times taken on a Linux 2.6.20 box with a 3.60GHz processor.

References

[1] M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, pages 67–75, October 2001.

[2] Y. Chevalier and M. Rusinowitch. Hierarchical combination of intruder theories. In *Proc. 17th International Conference on Term Rewriting and Applications (RTA'06)*, volume 4098 of *LNCS*, pages 108–122, Seattle, USA, 2006. Springer.

[3] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 359–364, Copenhagen, Denmark, July 2002. Springer.

[4] 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.

[5] V. Cortier, S. Delaune, and G. Steel. A formal theory of key conjuring. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 79–93, Venice, Italy, 2007.

[6] 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.

[7] 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.

[8] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[9] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[10] N. A. Durgin, P. Lincoln, and J. C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[11] J. Herzog. Applying protocol analysis to security device interfaces. *IEEE Security & Privacy Magazine*, 4(4):84–87, July-Aug 2006.

[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] J. C. Mitchell. Multiset rewriting and security protocol analysis. In *Proc. 13th International Conference on Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *LNCS*, pages 19–22, Copenhagen, Denmark, 2002. Springer.

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

[15] G. Steel. Deduction with XOR constraints in security API modelling. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *LNCS*, pages 322–336, Tallinn, Estonia, 2005. Springer.

[16] F. L. Tiplea, C. Enea, and C. V. Birjoveanu. Decidability and complexity results for security protocols. In *Proceedings of the Verification of Infinite-State Systems with Applications to Security (VISSAS'05)*, volume 1 of *NATO Security through Science Series D: Information and Communication Security*, pages 185–211. IOS Press, 2005.

[17] E. Tsalapati. Analysis of PKCS#11 using AVISPA tools. Master's thesis, University of Edinburgh, 2007.

[18] P. Youn. The analysis of cryptographic APIs using the theorem prover Otter. Master's thesis, Massachusetts Institute of Technology, 2004.

[19] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, August 2005.