Formal Security Analysis of PKCS#11 and Proprietary Extensions*

Stéphanie Delaune Steve Kremer Graham Steel

LSV, ENS Cachan & CNRS & INRIA, 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 the model checker NuSMV. We report some new attacks and prove the safety of some configurations of the API in our model. We also analyse proprietary extensions proposed by nCipher (Thales) and Eracom (Safenet), designed to address the shortcomings of PKCS#11.

Keywords Security APIs, PKCS#11, cryptographic devices, decidability, model checking

Corresponding Author

Graham Steel

61, avenue du Président Wilson, 94235 CACHAN Cedex, France

Tél +33 (0)1 47 40 77 80, Fax +33 (0)1 47 40 75 21

E-Mail graham.Steel@lsv.ens-cachan.fr

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

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 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. We analyse some proprietary extensions to the standard in Section 6. Finally we conclude with a discussion of future work in Section 7.

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 [5, 6, 7, 15, 17, 19]. None of these models account for mutable global state, which was identified by Herzog [11] as a major barrier to the application of security protocol analysis tools to verify APIs. In security protocol analysis, it is standard to assume that independent runs or sessions of the protocol

share no mutable (or changeable) state. They might share non-mutable state such as long term keys, and they might have local mutable state (they might create session keys, for example), but independent sessions are assumed not to affect each other. Security APIs do not have this characteristic. Indeed, the fact that calls to some functions, for example key import commands, do indeed change the internal state of the device, is vital for the way they function. In the work in this paper, we specifically aim to model mutable global state.

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. 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. The idea is that the SO login will be used to initialise the device, and from then on only user login will be used. 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

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

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 however "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 exactly 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. Even 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 $\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$. Note also that, crucially, PKCS#11 cannot tell whether an arbitrary bitstring is a cryptographic key or some other piece of plaintext. Thus when it executes the decrypt command, it has no way off telling that the packet it is decrypting contains a key.

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.

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 in his model only the commands needed for Clulow's first attack (Figure 1). This model had one handle for each key, preventing the discovery of some attacks, and a monotonic model of state which 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 exhibit in this paperm. 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. We comment on how this could be improved in Section 7.

All of these approaches (and our work) follow the 'Dolev-Yao' assumptions used for protocol analysis, [9], i.e. bit strings are abstracted to terms, and it is assumed that 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 for example Clulow's final two

attacks (small exponent X.509 and parallel key search) are out of scope for these models.

Our contributions. The contributions of our work are: a formal model for the core key management operations of PKCS#11 which accounts for non-monotonic mutable state; a decidabilitity result for this formalism given a bound on fresh terms, based on a practical syntactic pruning rule ('well-modedness'); a simple implementation that allows model checking in our formalism, together with results on various PKCS#11 configurations, including some new attacks; a formalisation of the proprietary extansions to the standard used in nCipher and Eracom devices; and the results of an analysis of the security of these measures in our model.

3 Formal model

3.1 Term algebra

We use classical notations on terms. For the sake of clarity, we recall here some important definitions and we illustrate them in Example 1. 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:

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

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

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

We define the set of *terms* as $\mathcal{T}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X}) = \mathcal{AT}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X}) \cup \mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})$. The set $\mathcal{T}(\mathcal{A}, \Sigma, \mathcal{N}, \emptyset)$, also written $\mathcal{T}(\mathcal{A}, \Sigma, \mathcal{N})$, is the set of *ground terms* and similarly for plain and attribute terms. We write $vars(t)$, resp. $names(t)$, for the set of variables, resp. names, that occur in the term t and extend *vars* and *names* to sets of terms in the expected way. By abuse of notation, sometimes we will write term instead of plain term.

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 subterm of t at position p is written $t|_p$, 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 the term $t\sigma$ 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 (formally a name) 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. For instance having two handles* $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ *and* $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_1)$ *models the fact that two distinct memory locations* n_1 *and* n_2 *hold the same key* k_1*. 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 $t = \mathsf{senc}(\mathsf{aenc}(n_1, \mathsf{pub}(n_2)), x)$*. We have that* $vars(t) = \{x\}$, $\mathsf{top}(t) = \mathsf{senc}$ *and* $pos(t) = \{\epsilon, 1, 1.1, 1.2, 1.2.1, 2\}$*. The term* $t|_{1.2}$*, i.e. the*

subterm of t *at position* 1.2*, is* $\mathsf{pub}(n_2)$.

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 for protocol analysis (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 of the form 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 $\mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})$, 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 $\neg a$. Moreover, we require that $\mathit{names}(T \cup L) = \emptyset$ and $\mathit{names}(T' \cup L') \subseteq \tilde{n}$. 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*

$$
\begin{array}{lrcl}
\text{Wrap (sym/sym)}: & \mathsf{h}(x_1, y_1), \mathsf{h}(x_2, y_2);\ \mathsf{wrap}(x_1), \mathsf{extract}(x_2) & \rightarrow & \mathsf{senc}(y_2, y_1) \\
\text{Wrap (sym/asym)}: & \mathsf{h}(x_1, \mathsf{priv}(z)), \mathsf{h}(x_2, y_2);\ \mathsf{wrap}(x_1), \mathsf{extract}(x_2) & \rightarrow & \mathsf{aenc}(y_2, \mathsf{pub}(z)) \\
\text{Wrap (asym/sym)}: & \mathsf{h}(x_1, y_1), \mathsf{h}(x_2, \mathsf{priv}(z));\ \mathsf{wrap}(x_1), \mathsf{extract}(x_2) & \rightarrow & \mathsf{senc}(\mathsf{priv}(z), y_1)
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Unwrap (sym/sym)}: & \mathsf{h}(x_1, y_2), \mathsf{senc}(y_1, y_2);\ \mathsf{unwrap}(x_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1, y_1);\ \mathsf{extract}(n_1),\ \mathsf{L} \\
\text{Unwrap (sym/asym)}: & \mathsf{h}(x_1, \mathsf{priv}(z)), \mathsf{aenc}(y_1, \mathsf{pub}(z));\ \mathsf{unwrap}(x_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1, y_1);\ \mathsf{extract}(n_1),\ \mathsf{L} \\
\text{Unwrap (asym/sym)}: & \mathsf{h}(x_1, y_2), \mathsf{senc}(\mathsf{priv}(z), y_2);\ \mathsf{unwrap}(x_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1, \mathsf{priv}(z));\ \mathsf{extract}(n_1),\ \mathsf{L}
\end{array}
$$

$$
\begin{array}{lrl}
\text{KeyGenerate}: & \xrightarrow{\text{new } n_1, k_1} & \mathsf{h}(n_1, k_1); \neg\mathsf{extract}(n_1),\ \mathsf{L} \\
\text{KeyPairGenerate}: & \xrightarrow{\text{new } n_1, s} & \mathsf{h}(n_1, s), \mathsf{pub}(s);\ \neg\mathsf{extract}(n_1),\ \mathsf{L}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{SEncrypt}: & \mathsf{h}(x_1, y_1), y_2;\ \mathsf{encrypt}(x_1) & \rightarrow & \mathsf{senc}(y_2, y_1) \\
\text{SDecrypt}: & \mathsf{h}(x_1, y_1), \mathsf{senc}(y_2, y_1);\ \mathsf{decrypt}(x_1) & \rightarrow & y_2
\end{array}
$$

$$
\begin{array}{lrcl}
\text{AEncrypt}: & \mathsf{h}(x_1, \mathsf{priv}(z)), y_1;\ \mathsf{encrypt}(x_1) & \rightarrow & \mathsf{aenc}(y_1, \mathsf{pub}(z)) \\
\text{ADecrypt}: & \mathsf{h}(x_1, \mathsf{priv}(z)), \mathsf{aenc}(y_2, \mathsf{pub}(z));\ \mathsf{decrypt}(x_1) & \rightarrow & y_2
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Set\_Wrap}: & \mathsf{h}(x_1, y_1);\ \neg\mathsf{wrap}(x_1) & \rightarrow & \mathsf{wrap}(x_1) \\
\text{Set\_Encrypt}: & \mathsf{h}(x_1, y_1);\ \neg\mathsf{encrypt}(x_1) & \rightarrow & \mathsf{encrypt}(x_1) \\
\vdots & & \vdots & \\
\text{UnSet\_Wrap}: & \mathsf{h}(x_1, y_1);\ \mathsf{wrap}(x_1) & \rightarrow & \neg\mathsf{wrap}(x_1) \\
\text{UnSet\_Encrypt}: & \mathsf{h}(x_1, y_1);\ \mathsf{encrypt}(x_1) & \rightarrow & \neg\mathsf{encrypt}(x_1) \\
\vdots & & \vdots &
\end{array}
$$

where $\mathsf{L} = \neg\mathsf{wrap}(n_1), \neg\mathsf{unwrap}(n_1), \neg\mathsf{encrypt}(n_1), \neg\mathsf{decrypt}(n_1), \neg\mathsf{sensitive}(n_1)$. The ellipsis in the set and unset rules indicates that similar rules exist for some other attributes.

Figure 2: PKCS#11 key management subset.

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* $\mathsf{extract}$ *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 $\mathcal{X}$, and a set of rules $\mathcal{R}$ defined over $\mathcal{T}(\mathcal{A}, \Sigma, \mathcal{N}, \mathcal{X})$. 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

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

and to sets of literals (interpreted as a conjunction over the literals) as $V(L) = \bigwedge_{\ell \in L} V(\ell)$ when $\{a \mid a \in L \text{ or } \neg a \in L\} \subseteq dom(V)$. Moreover, we assume a given set of ground terms $S_0 \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ 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 w.r.t. a set of names $N \subseteq \mathcal{N}$ 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 distinct names not in N. These fresh renamings allow us to get rid of the new in the rule. 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 $S \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ 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. We have that $(S, V) \rightsquigarrow (S', V')$ if

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

is a fresh renaming w.r.t. $names(S \cup dom(V))$ of a rule in $\mathcal{R}$ and 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:

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

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

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. When an attribute is not in the domain of the valuation it is considered to be undefined. An instance of a rule with a literal containing such an undefined attribute on the left-hand side cannot be fired. Real devices vary in the default values they assign to newly created objects, and our semantics allows us to model underspecified rules for the creation of objects, where some attributes are not given a value.

Note also that when $(S, V) \rightsquigarrow (S', V')$ we always have that $S \subseteq S'$ and $dom(V) \subseteq dom(V')$.

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 and a state $(S, V) \in Q$ such that $q_0 \rightsquigarrow^* (S, V)$, $T\theta \subseteq S$, and $V(L\theta) = \top$.*

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* $\mathsf{Wrap}\,(\mathsf{sym}/\mathsf{sym})$ *and* $\mathsf{SDecrypt}$ *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

$$\begin{array}{rcll} (S_0, V_0) & \rightsquigarrow & (S_0 \cup \{\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)\}, V_0) \stackrel{def}{=} (S_1, V_1) & \mathsf{Wrap}\,(\mathsf{sym}/\mathsf{sym}) \\ & \rightsquigarrow & (S_1 \cup \{\mathsf{k}_1\}, V_1) & \mathsf{SDecrypt} \end{array}$$

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

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 by considering only 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 assumption. Informally, we do not assume that a device is able discriminate between bitstrings that were generated e.g. as keys and random data, we simply show that if there is an attack, then there is an attack where bitstrings are used for the purpose they were originally created. 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 $\mathsf{M} : \Sigma \cup \mathcal{A} \times \mathbb{N} \to \mathsf{Mode}$ 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 $\mathsf{sig} : \Sigma \cup \mathcal{A} \cup \mathcal{X} \cup \mathcal{N} \to \mathsf{Mode}$ 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: $\mathsf{sig}(t) = \mathsf{sig}(\mathsf{top}(t))$. 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 such that $\mathsf{M}(f, i) = \mathsf{m}_i$ 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:

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

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 $(S_0, V_0) \rightsquigarrow^* (S_1 \cup \{\mathsf{k}_1\}, V_1)$ *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:*

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

The term $\mathsf{senc}(\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_1)$ *is not well-moded because of its subterm* $\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$*. However, note that such a derivation (that is not well-moded) is not forbidden a priori by any typing mechanism. We consider it in our semantics. By relying on the fact that the API we consider is well-moded, we simply show that it is not necessary to consider such a derivation when we are looking for an attack.*

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). Indeed any instance of a well-moded rule that can be triggered from a state (S, V) where the valuation V is well-moded, leads to a state (S', V') in which V' is also well-moded.
- Any ill-moded term v' occurring in a term in S is itself deducible (Lemma 2). Note that an instance of a well-moded rule can be triggered from a well-moded state (S, V) to reach an ill-moded state (S', V') (see Example 4). However, in such a situation, the subterm that occurs at an ill-moded position also occurs in S. This is due to the fact that the rules we consider are well-moded.

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* $(S_0, V_0) \rightsquigarrow^* (S, V)$*. We have that* V *is well-moded.*

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 $(S_0, V_0) \rightsquigarrow^* (S, V)$. Let $v \in S$ and v' be a subterm of v which occurs at an ill-moded position. Then, we have that $v' \in S$.*

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 turns any term into 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:*

- $\overline{n}^{\mathsf{m}} = \begin{cases} n & \text{if } n \in \mathcal{N} \text{ and } \mathsf{sig}(n) = \mathsf{m} \\ t_{\mathsf{m}} & \text{otherwise} \end{cases}$
- $\overline{f(v_1, \ldots, v_n)}^{\mathsf{m}} = \begin{cases} f(\overline{v_1}^{\mathsf{m}_1}, \ldots, \overline{v_n}^{\mathsf{m}_n}) & \\ \qquad \text{if } f : \mathsf{m}_1 \times \ldots \mathsf{m}_n \to \mathsf{m} & \\ t_{\mathsf{m}} & \text{otherwise} \end{cases}$

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

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: $(S_0, V_0) \rightsquigarrow (S_1, V_1) \rightsquigarrow \ldots \rightsquigarrow (S_k, V_k)$. Moreover, for each mode $\mathsf{m} \in \{\mathsf{sig}(t) \mid t \in \mathcal{PT}(\Sigma, \mathcal{N})\}$, we assume that there exists a well-moded term t_{m} of mode m such that $t_{\mathsf{m}} \in S_0$ (i.e. t_{m} is known by the attacker) 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).*

Example 5 *Let S_0 be a set of terms that contains $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$, $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$ and $\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$. Let V_0 be a function such that $V_0(\mathsf{unwrap}(\mathsf{n}_2)) = V_0(\mathsf{encrypt}(\mathsf{n}_2)) = \top$. Consider the following derivation D that is not well-moded:*

$$
\begin{array}{rcll}
(S_0, V_0) & \rightsquigarrow & (S_0 \cup \mathsf{senc}(\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_2), V_0) \stackrel{\mathsf{def}}{=} (S_1, V_1) & \mathsf{SEncrypt} \\
& \rightsquigarrow & (S_1 \cup \mathsf{senc}(\mathsf{senc}(\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_2), \mathsf{k}_2), V_1) \stackrel{\mathsf{def}}{=} (S_2, V_2) & \mathsf{SEncrypt} \\
& \rightsquigarrow & (S_2 \cup \{\mathsf{h}(\mathsf{n}_3, \mathsf{senc}(\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_2))\}, V_3) & \mathsf{Unwrap}
\end{array}
$$

for some valuation V_3.

It is easy to see that the following sequence $(\overline{S_0}, V_0) \rightsquigarrow (\overline{S_1}, V_1) \rightsquigarrow (\overline{S_2}, V_2) \rightsquigarrow (\overline{S_3}, V_3)$ is a well-moded derivation as soon as $t_{\mathsf{Key}} \in S_0$ and $\mathsf{sig}(t_{\mathsf{Key}}) = \mathsf{Key}$. Note that this derivation uses the same rule as D (but not the same intances). The second step keeps the state unchanged since the term $\mathsf{senc}(\mathsf{t}_{\mathsf{Key}}, \mathsf{k}_2)$ is already in the current state.

$$
\begin{array}{rcll}
(S_0, V_0) & \rightsquigarrow & (S_0 \cup \mathsf{senc}(\mathsf{t_{Key}}, \mathsf{k_2}), V_0) & \mathsf{SEncrypt} \\
& \rightsquigarrow & (S_0 \cup \mathsf{senc}(\mathsf{t_{Key}}, \mathsf{k_2}), V_0) & \mathsf{SEncrypt} \\
& \rightsquigarrow & (S_0 \cup \{\mathsf{h}(\mathsf{n_3}, \mathsf{t_{Key}})\}, V_3) & \mathsf{Unwrap}
\end{array}
$$

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 well-moded 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-moded. Our result however states that whenever there exists an attack that uses a derivation which is not well-moded 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.

Example 6 *Continuing Example 5, we consider the query $Q = \{\mathsf{h}(\mathsf{x}, \mathsf{y}),\ \mathsf{y}\}$ and the well-moded state $q_0 = (S_0, V_0)$ as defined in Example 5. We also assume that $S_0 \supseteq \{t_\mathsf{m} \mid \mathsf{m} \in \{\mathsf{Nonce}, \mathsf{Key}, \mathsf{Handle}, \mathsf{Cipher}, \mathsf{Seed}\}\}$. The query Q is a well-moded query that is satisfiable in q_0. The derivation D described in Example 5 witnesses this fact ($\mathsf{x} \mapsto \mathsf{n_3}$, and $\mathsf{y} \mapsto \mathsf{senc}(\mathsf{senc}(\mathsf{k_1}, \mathsf{k_2}), \mathsf{k_2})$). This derivation is not well-moded. However, there exists a well-moded derivation witnessing this fact. Indeed the well-moded derivation described in Example 5 witnesses this fact: $\mathsf{x} \mapsto \mathsf{n_3}$, and $\mathsf{y} \mapsto t_{\mathsf{Key}}$.*

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 well-moded 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 of 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, and hence our experiments focus on PKCS#11 based APIs. Other APIs such as the API of the Trusted Platform Module (TPM) will also require global mutable state to be modelled.

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. We take as our security property the secrecy of sensitive keys, stated in the manual as a property of the interface, [14, p. 31]. Roughly speaking, our methodology was to start with a configuration involving only symmetric keys, and to try to restrict the API until a secure configuration was found. We then added asymmetric keypairs, and repeated the process. A secure configuration proved only to be possible by adding the 'trusted keys' mechanism introduced in PKCS#11 v2.20. We carried out some experiments modelling the algebraic properties of the ECB mode of encryption. Finally we added the proprietary extensions to the standard used by two commercial providers of cryptographic hardware, and analysed the security of these. The proprietary extensions require some explanation, and are described in a separate section (Section 6).

PKCS#11 is described in a large and complex specification [14], running to 392 pages. We model here only the key management operations at the core of the API. With the respect to the full list of commands in the standard [14, p. 89], 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 that 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 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 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#11 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.

5.2 Experiments with PKCS#11

All the files for our experiments are available at `http://www.lsv.ens-cachan.fr/~steel/pkcs11`. 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.

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: attribute changing operations are 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 is 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:

$$
\begin{array}{lrcl}
\text{SEncrypt:} & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \mathsf{k}_3 & \rightarrow & \mathsf{senc}(\mathsf{k}_3, \mathsf{k}_2) \\
\text{Unwrap:} & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \mathsf{senc}(\mathsf{k}_3, \mathsf{k}_2) & \xrightarrow{\text{new } \mathsf{n}_3} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) \\
\text{Set\_wrap:} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) & \rightarrow & \mathsf{wrap}(\mathsf{n}_3) \\
\text{Wrap:} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3),\ \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) & \rightarrow & \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3) \\
\text{Intruder:} & \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3),\ \mathsf{k}_3 & \rightarrow & \mathsf{k}_1
\end{array}
$$

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 $h(n_4, k_2)$. The intruder then wraps k_1 under k_2, and sets the decrypt attribute on handle $h(n_4, k_2)$, 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:	$h(n_2, k_2)$	$\rightarrow$	$\mathsf{wrap}(n_2)$
Wrap:	$h(n_2, k_2), h(n_2, k_2)$	$\rightarrow$	$\mathsf{senc}(k_2, k_2)$
Set_unwrap:	$h(n_2, k_2)$	$\rightarrow$	$\mathsf{unwrap}(n_2)$
Unwrap:	$h(n_2, k_2), \mathsf{senc}(k_2, k_2)$	$\xrightarrow{\text{new } n_4}$	$h(n_4, k_2)$
Wrap:	$h(n_2, k_2), h(n_1, k_1)$	$\rightarrow$	$\mathsf{senc}(k_1, k_2)$
Set_decrypt:	$h(n_4, k_2)$	$\rightarrow$	$\mathsf{decrypt}(n_4)$
SDecrypt:	$h(n_4, k_2), \mathsf{senc}(k_1, k_2)$	$\rightarrow$	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. In addition to the initial knowledge from the first three experiments, we give the intruder an unknown k3 encrypted under k2. Again he is able to affect an attack similar to the one above, this time by unwrapping $\mathsf{senc}(k_3, k_2)$ twice (see Figure 5).

Experiment 5. We now add two asymmetric keypairs to the model. One, $(\mathsf{pub}(s_1), \mathsf{priv}(s_1))$, is loaded onto the device and is unknown to the intruder (apart from the handle). The other, $(\mathsf{pub}(s_2), \mathsf{priv}(s_2))$, is the intruder's own keypair, but is not loaded onto the device. We now rediscover Clulow's Trojan Wrapped Key attack, [4, Section 3.5], shown in Figure 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 -

Initial state: The intruder knows the handles $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$, $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$; n_1 has the attributes sensitive, extract and whereas n_2 has the attribute extract set. The intruder also knows $\mathsf{senc}(\mathsf{k}_3, \mathsf{k}_2)$.

Trace:

$$
\begin{array}{lrcl}
\mathsf{Set\_unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2) & \rightarrow & \mathsf{unwrap}(\mathsf{n}_2) \\
\mathsf{Unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \mathsf{senc}(\mathsf{k}_3, \mathsf{k}_2) & \xrightarrow{\text{new } \mathsf{n}_3} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) \\
\mathsf{Unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \mathsf{senc}(\mathsf{k}_3, \mathsf{k}_2) & \xrightarrow{\text{new } \mathsf{n}_4} & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3) \\
\mathsf{Set\_wrap}: & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) & \rightarrow & \mathsf{wrap}(\mathsf{n}_3) \\
\mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3), \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) & \rightarrow & \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3) \\
\mathsf{Set\_decrypt}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3) & \rightarrow & \mathsf{decrypt}(\mathsf{n}_4) \\
\mathsf{SDecrypt}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3), \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3) & \rightarrow & \mathsf{k}_1
\end{array}
$$

Figure 5: Attack discovered in Experiment 4

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 onto the device.

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, n_2 has the attribute extract. The intruder also knows the public key $\mathsf{pub}(\mathsf{s}_1)$ and its associated handle $\mathsf{h}(\mathsf{n}_3, \mathsf{priv}(\mathsf{s}_1))$;

Trace:

$$
\begin{array}{lrcl}
\mathsf{Intruder}: & \mathsf{k}_3, \mathsf{pub}(\mathsf{s}_1) & \rightarrow & \mathsf{aenc}(\mathsf{k}_3, \mathsf{pub}(\mathsf{s}_1)) \\
\mathsf{Set\_unwrap}: & \mathsf{h}(\mathsf{n}_3, \mathsf{priv}(\mathsf{s}_1)) & \rightarrow & \mathsf{unwrap}(\mathsf{n}_3) \\
\mathsf{Unwrap}: & \mathsf{aenc}(\mathsf{k}_3, \mathsf{pub}(\mathsf{s}_1)) \; \mathsf{h}(\mathsf{n}_3, \mathsf{priv}(\mathsf{s}_1)) & \xrightarrow{\text{new } \mathsf{n}_4} & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3) \\
\mathsf{Set\_wrap}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3) & \rightarrow & \mathsf{wrap}(\mathsf{n}_4) \\
\mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3), \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) & \rightarrow & \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3) \\
\mathsf{Intruder}: & \mathsf{senc}(\mathsf{k}_1, \mathsf{k}_3), \; \mathsf{k}_3 & \rightarrow & \mathsf{k}_1
\end{array}
$$

Figure 6: Clulow's Trojan Wrapped Key attack rediscovered in Experiment 5

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 7. Here, the intruder first attacks the trusted wrapping key, and then obtains the sensitive key.

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, extract and wrap_with_trusted whereas n_2 has the attributes extract and trusted set. The intruder also knows the public key $\mathsf{pub}(\mathsf{s}_1)$ and its associated handle $\mathsf{h}(\mathsf{n}_3, \mathsf{priv}(\mathsf{s}_1))$;

Trace:

Intruder:	$\mathsf{k}_3, \mathsf{pub}(\mathsf{s}_1)$	$\rightarrow$	$\mathsf{aenc}(\mathsf{k}_3, \mathsf{pub}(\mathsf{s}_1))$
Set_unwrap:	$\mathsf{h}(\mathsf{n}_3, \mathsf{priv}(\mathsf{s}_1))$	$\rightarrow$	$\mathsf{unwrap}(\mathsf{n}_3)$
Unwrap:	$\mathsf{aenc}(\mathsf{k}_3, \mathsf{pub}(\mathsf{s}_1))$, $\mathsf{h}(\mathsf{n}_3, \mathsf{priv}(\mathsf{s}_1))$	$\xrightarrow{\text{new } \mathsf{n}_4}$	$\mathsf{h}(\mathsf{n}_4, \mathsf{k}_3)$
Set_wrap:	$\mathsf{h}(\mathsf{n}_4, \mathsf{k}_3)$	$\rightarrow$	$\mathsf{wrap}(\mathsf{n}_4)$
Wrap:	$\mathsf{h}(\mathsf{n}_4, \mathsf{k}_3), \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$	$\rightarrow$	$\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_3)$
Intruder:	$\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_3)$, k_3	$\rightarrow$	k_2
Set_wrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$	$\rightarrow$	$\mathsf{wrap}(\mathsf{n}_2)$
Wrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$	$\rightarrow$	$\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$
Intruder:	$\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_2)$, k_2	$\rightarrow$	k_1

Figure 7: Attack discovered in Experiment 6

Experiment 7. In Experiment 7 we prevent the attack in Figure 7 by marking n_2 as wrap_with_trusted. Finally 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 $(\mathsf{priv}(\mathsf{s}_1), \mathsf{pub}(\mathsf{s}_1))$. 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 bring these attacks in scope. The symbol pair captures pairing of keys. To preserve well-modedness, we introduce new function symbols for encrypting a single key under a double length key (sp_senc), encrypting a double length key under a single key (ps_senc), and encrypting a double length key under a double length key (pp_senc). We give the symbols and their associated modes in Figure 8. We introduce extra intruder rules to capture the intruder's ability to make pairs and deduce exploit the properties of ECB mode encryption of pairs. To capture Clulow's attacks, we model the assumption that single-length DES keys can be cracked by brute force. For this we require rules $\mathsf{senc}(\mathsf{y}_1, \mathsf{y}_2), \mathsf{y}_1 \rightarrow \mathsf{y}_2$ (and the equivalent for plaintext pairs), and $\mathsf{senc}(\mathsf{y}_1, \mathsf{y}_1) \rightarrow \mathsf{y}_1$ - the intruder can crack single length keys if he knows the plaintext, or if he knows the ciphertext consists of the key encrypted under itself. All the intruder rules are given in Figure 8. Note that the model also contains all the rules in Figure 2 with versions for single and double length keys as expected.

In a model with three atomic keys and two double-length keys, we search for attacks on a double length key. We first rediscover Clulow's weaker key attack, [4, Section 2.4], see Figure 9. Note that this attack does not use any of the specific properties of ECB encryption: it merely succeeds because the attacker is able to first brute-force crack a single-length DES key (k3) and then ask to have the double length key $\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2)$ encrypted under k3. This attack is discovered first because it is shorter than the ECB attack.

Experiment 10. In Experiment 10, we prevent Clulow's weaker key attack by removing the rule whereby the wrap command uses a single length key to encrypt a double length key. We now rediscover Clulow's ECB based attack, [4, Section 2.2], see Figure 10. Here, the attacker first wraps a double length key under a double length key, then, by exploiting the properties of ECB encryption, he obtains two single length keys, which he can re-import and crack individually. Notice that this attack requires a handle for a key with both wrap and unwrap set, which we have already discovered can give rise to attacks (see Experiment 3). We re-enabled this combination only to test that Clulow's attacks would be found.

Additional Intruder Rules:

$$
\begin{array}{rcl}
y_1,\ y_2 & \rightarrow & \mathsf{pair}(y_1, y_2) \\
\mathsf{pair}(y_1, y_2) & \rightarrow & y_1 \\
\mathsf{pair}(y_1, y_2) & \rightarrow & y_2 \\
\mathsf{senc}(y_1, y_3), \mathsf{senc}(y_2, y_3) & \rightarrow & \mathsf{ps\_senc}(\mathsf{pair}(y_1, y_2), y_3) \\
\mathsf{ps\_senc}(\mathsf{pair}(y_1, y_2), y_3) & \rightarrow & \mathsf{senc}(y_1, y_3) \\
\mathsf{ps\_senc}(\mathsf{pair}(y_1, y_2), y_3) & \rightarrow & \mathsf{senc}(y_2, y_3) \\
\mathsf{sp\_senc}(y_1, \mathsf{pair}(y_3, y_4)), \mathsf{sp\_senc}(y_2, \mathsf{pair}(y_3, y_4)) & \rightarrow & \mathsf{pp\_senc}(\mathsf{pair}(y_1, y_2), \mathsf{pair}(y_3, y4)) \\
\mathsf{pp\_senc}(\mathsf{pair}(y_1, y_2), \mathsf{pair}(y_3, y_4)) & \rightarrow & \mathsf{sp\_senc}(y_1, \mathsf{pair}(y_3, y_4)) \\
\mathsf{pp\_senc}(\mathsf{pair}(y_1, y_2), \mathsf{pair}(y_3, y_4)) & \rightarrow & \mathsf{sp\_senc}(y_2, \mathsf{pair}(y_3, y_4)) \\
y_1, \mathsf{pair}(y_2, y_3) & \rightarrow & \mathsf{sp\_senc}(y_1, \mathsf{pair}(y_2, y_3)) \\
y_1, \mathsf{pair}(y_2, y_3) & \rightarrow & \mathsf{ps\_senc}(\mathsf{pair}(y_2, y_3), y_1) \\
\mathsf{pair}(y_1, y_2), \mathsf{pair}(y_3, y_4) & \rightarrow & \mathsf{pp\_senc}(\mathsf{pair}(y_1, y_2), \mathsf{pair}(y_3, y_4)) \\
\mathsf{sp\_senc}(y_1, \mathsf{pair}(y_2, y_3)), \mathsf{pair}(y_2, y_3) & \rightarrow & y_1 \\
\mathsf{ps\_senc}(\mathsf{pair}(y_2, y_3), y_1), y_1 & \rightarrow & \mathsf{pair}(y_2, y_3) \\
\mathsf{pp\_senc}(\mathsf{pair}(y_1, y_2), \mathsf{pair}(y_3, y_4)), \mathsf{pair}(y_3, y_4) & \rightarrow & \mathsf{pair}(y_1, y_2) \\
\mathsf{senc}(y_1, y_2), y_1 & \rightarrow & y_2 \\
\mathsf{ps\_senc}(\mathsf{pair}(y_1, y_2), y_3), \mathsf{pair}(y_1, y_2) & \rightarrow & y_3 \\
\mathsf{senc}(y_1, y_1) & \rightarrow & y_1
\end{array}
$$

Modes:

pair	Key × Key	→	Pair
sp_senc	Key × Pair	→	CipherPair
pp_senc	Pair × Pair	→	CipherPairPair
ps_senc	Pair × Key	→	PairCipher

Figure 8: Additional Intruder Rules (ECB)

As Table 1 shows, run times vary from a few seconds to a few minutes. However, 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 (i.e. Exp. 5, 6, 7 and 8) have 128 variables.

6 Analysing Proprietary Extensions

We obtained details of some extensions to PKCS#11 employed by Hardware Security Module (HSM) manufacturers nCipher[1] and Eracom[2]. HSMs typically consist of a small computer capable of carrying out cryptographic operations embedded in a tamper-resistant unit. They

[1]Now owned by Thales, `http://www.ncipher.com`

[2]Now owned by Safenet INC, `http://www.safenet-inc.com/`

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	×	×	×	-	-	Fig 5	1m1s
5	3	2	2	1	×	×	×	-	-	[4, §3.5]	10m30s
6	3	2	2	1	×	×	×	×	-	Fig 7	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.

Initial state: The intruder knows the handles $\mathsf{h}(\mathsf{n}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$ (for double-length key $\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2)$) and $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_3)$. n_1 has the attributes sensitive and extract set.

Trace:

Set_wrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_3)$	$\rightarrow$	$\mathsf{wrap}(\mathsf{n}_2)$
Set_extract:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_3)$	$\rightarrow$	$\mathsf{extract}(\mathsf{n}_2)$
Wrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_3), \mathsf{h}(\mathsf{n}_2, \mathsf{k}_3)$	$\rightarrow$	$\mathsf{senc}(\mathsf{k}_3, \mathsf{k}_3)$
Crack:	$\mathsf{senc}(\mathsf{k}_3, \mathsf{k}_3)$	$\rightarrow$	k_3
Wrap:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_3), \mathsf{h}(\mathsf{n}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$	$\rightarrow$	$\mathsf{ps\_senc}(\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_3)$
Intruder:	$\mathsf{ps\_senc}(\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{k}_3), \ \mathsf{k}_3$	$\rightarrow$	$\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2)$

Figure 9: Clulow's weaker key attack rediscovered in Experiment 9

are used in security critical environments such as the banking sector, and have a typical unit price of several thousand dollars. The aim of our experiments was not to 'attack' these manufacturer's security measures, but rather to validate our approach for analysing particular configurations of a PKCS#11 API including these measures. This is because it is clear that even with the proprietary measures, it is still possible to write a PKCS#11 compatible API that will leak sensitive keys. The aim of the measures is to provide extra functionality that makes it easier to produce secure configurations. Therefore our methodology for the experiments was to try various configurations for each measure, both secure and insecure, and evaluate the performance of our tools on these models.

6.1 nCipher SAM

The nCipher nShield device has a library called the 'Security Assurance Mechanism' (SAM), which is intended to warn you when you use features of PKCS#11 that may lead to attacks. The SAM is designed to throw exceptions when 'questionable' behaviour occurs, where questionable behaviour is designed by a set of checks given below. Intuitively, questionable behaviour is something that might lead to the value of a key being discovered outside the device. The intention is that the application developer examines the warning messages when the application is under test. Then, he can turn off certain rules of the SAM to allow commands which the SAM considers to be questionable to pass, after he has reviewed the error messages to make sure they are safe for his particular application scenario.

Initial state: The intruder knows the handles $\mathsf{h}(\mathsf{n}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$ (for double-length key $\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2)$) and $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_3)$. n_1 has the attributes sensitive, extract,wrap and unwrap.

Trace:

Wrap:	$\mathsf{h}(\mathsf{n}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2)),\ \mathsf{h}(\mathsf{n}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$	$\rightarrow$	$\mathsf{pp\_senc}(\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$
ECB:	$\mathsf{pp\_senc}(\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$	$\rightarrow$	$\mathsf{sp\_senc}(\mathsf{k}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$
Unwrap:	$\mathsf{sp\_senc}(\mathsf{k}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$ $\mathsf{h}(\mathsf{n}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$	$\xrightarrow{\text{new } \mathsf{n}_3}$	$\mathsf{h}(\mathsf{n}_3, \mathsf{k}_1)$
ECB:	$\mathsf{pp\_senc}(\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2), \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$	$\rightarrow$	$\mathsf{sp\_senc}(\mathsf{k}_2, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$
Unwrap:	$\mathsf{sp\_senc}(\mathsf{k}_2, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$ $\mathsf{h}(\mathsf{n}_1, \mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2))$	$\xrightarrow{\text{new } \mathsf{n}_4}$	$\mathsf{h}(\mathsf{n}_4, \mathsf{k}_2)$
Set_wrap:	$\mathsf{h}(\mathsf{n}_3, \mathsf{k}_1)$	$\rightarrow$	$\mathsf{wrap}(\mathsf{n}_3)$
Wrap:	$\mathsf{h}(\mathsf{n}_3, \mathsf{k}_1), \mathsf{h}(\mathsf{n}_3, \mathsf{k}_1)$	$\rightarrow$	$\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_1)$
Crack:	$\mathsf{senc}(\mathsf{k}_1, \mathsf{k}_1)$	$\rightarrow$	k_1
Wrap:	$\mathsf{h}(\mathsf{n}_3, \mathsf{k}_1), \mathsf{h}(\mathsf{n}_3, \mathsf{k}_2)$	$\rightarrow$	$\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_1)$
Intruder:	$\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_1),\ \mathsf{k}_1$	$\rightarrow$	k_2
Intruder:	$\mathsf{k}_1,\ \mathsf{k}_2$	$\rightarrow$	$\mathsf{pair}(\mathsf{k}_1, \mathsf{k}_2)$

Figure 10: Clulow's ECB attack rediscovered in Experiment 10

The description of the SAM is given informally in the nCipher nShield manual, and contains a certain amount of ambiguity. We have tried to rationally reconstruct the operation of the mechanism here, but note that we cannot be absolutely sure that the real device functions in the same way as we have not tested one.

In its default behaviour, the SAM considers several operations to be 'questionable'. Some of these are out-of-scope for our abstract model - we do not consider key length, key lifetime or particular cryptographic algorithms. Our experiments cover only the following 'questionable' behaviour:

1. importing keys from external sources;

2. creating or importing wrapping keys;

3. creating or importing unwrapping keys.

The parameter `OVERRIDE_SECURITY_ASSURANCES` can be set to various values in order to prevent exceptions from being thrown when these behaviour occur. Values relevant to the scope of our model are:

all setting this flag causes all the SAM checks to be turned off.

import By default, keys imported via `Unwrap` command are considered insecure. Setting this parameter allows them to be considered secure, as long as there is no other active SAM check which rules them insecure.

unwrap_mech Setting this parameter allows keys transferred into the device using insecure encryption to be treated as if they had been protected by secure encryption. When the unwrap_mech parameter is set, unwrap keys automatically have the decrypt attribute set. This is because abuse of the unwrap mechanism in the nCipher implementation of PKCS#11 would allow the key to be used for decryption anyway.

unwrap_kek Imported keys are only considered secure if the unwrapping key (or key encrypting key - kek) has only the wrap and unwrap permissions set, not the decrypt attribute. As noted above, in the nCipher implementation, if unwrap_mech is set then all unwrap keys have the decrypt attribute set. Setting this parameter causes the device to ignore that, and consider the imported keys to be secure anyway.

The nCipher implementation considers all of PKCS#11's key transport mechanisms to be insecure. So, in order to unwrap keys without them being considered insecure by the SAM, you need to set at least unwrap_mech, unwrap_kek, and import. Otherwise, any unwrapping operation will cause an error. There seems to be some confusion here: you have to switch on unwrap_mech to allow key import, which gives unwrapping keys the decrypt attribute, immediately rendering key import insecure, which you then have to ignore with the unwrap_kek setting. It would seem more sensible to set an attribute which disallows the abuse of the unwrapping mechanism to decrypt a key, but we report only what we found on reading the nCipher manual. Possibly these rules were designed for some future version of PKCS#11 where some wrapping mechanisms are considered secure, and some not.

As we have explained, the idea of the SAM is not to produce a secure configuration, but to warn the developer when a key may not be secure. To model this, we add an attribute considered_secure to the model described in section 3, denoted cs. Note that there are no set/unset rules for this attribute. We give below the model we used in our formal language

(Figure 11). As noted, we cannot be sure that the nCipher SAM functions exactly in this way having not tested a device, but this seemed the most logical interpretation of the informal specification. During the experiments, the following further changes to the model are made to account for the different values of `OVERRIDE_SECURITY_ASSURANCES`:

1. If all is set, the model functions just as before (see section 3), and the cs attribute is ignored.

2. If all is unset, but all the other parameters under consideration are set, then the only difference is that the wrap and unwrap attributes are sticky off, i.e. cannot be set, preventing the creation of (un)wrapping keys (the second and third 'questionable behaviour' mentioned above).

3. If any of unwrap_mech, unwrap_kek, import are unset, then all unwrapped keys have considered_secure set to false - the unwrap rules marked v1 are included. If they are all set, then an unwrapped key is considered secure, as long as the unwrapping key is also considered_secure - unwrap rules marked v2 are included. Unwrap rules marked v0 are always included.

Summary of Experiments The aim of the nCipher SAM is not to prevent sensitive keys from becoming known, but to alert the application when this might be the case. Therefore, in our experiments, the security property is not that all sensitive keys must remain unknown to the intruder, as before, but that any key with the attribute considered_secure set must remain unknown to the intruder, i.e., an attack is valid only when the intruder learns the value of a key and the SAM still considers it to be secure. All model files are available from `http://www.lsv.ens-cachan.fr/~steel/pkcs11`.

Experiment 11 With all the SAM measures in use, no attacks are found in a model with 2 symmetric keys, 2 asymmetric keypairs, and 8 handles, in 10 minutes time.

$$
\begin{array}{lrcl}
\text{Wrap (sym/sym)} : & \mathsf{h}(x_1,y_1), \mathsf{h}(x_2,y_2); \ \mathsf{wrap}(x_1), \mathsf{extract}(x_2), \mathsf{cs}(x_1) & \rightarrow & \mathsf{senc}(y_2,y_1) \\
 & \mathsf{h}(x_1,y_1), \mathsf{h}(x_2,y_2); \ \mathsf{wrap}(x_1), \mathsf{extract}(x_2), \neg\mathsf{cs}(x_1) & \rightarrow & \mathsf{senc}(y_2,y_1);\ \neg\mathsf{cs}(x_2) \\
\text{Wrap (sym/asym)} : & \mathsf{h}(x_1,\mathsf{priv}(z)), \mathsf{h}(x_2,y_2); \ \mathsf{wrap}(x_1), \mathsf{cs}(x_1), \mathsf{extract}(x_2) & \rightarrow & \mathsf{aenc}(y_2,\mathsf{pub}(z)) \\
 & \mathsf{h}(x_1,\mathsf{priv}(z)), \mathsf{h}(x_2,y_2); \ \mathsf{wrap}(x_1), \neg\mathsf{cs}(x_1), \mathsf{extract}(x_2) & \rightarrow & \mathsf{aenc}(y_2,\mathsf{pub}(z));\ \neg\mathsf{cs}(x_2) \\
\text{Wrap (asym/sym)} : & \mathsf{h}(x_1,y_1), \mathsf{h}(x_2,\mathsf{priv}(z)); \ \mathsf{wrap}(x_1), \mathsf{cs}(x_1), \mathsf{extract}(x_2) & \rightarrow & \mathsf{senc}(\mathsf{priv}(z),y_1) \\
 & \mathsf{h}(x_1,y_1), \mathsf{h}(x_2,\mathsf{priv}(z)); \ \mathsf{wrap}(x_1), \neg\mathsf{cs}(x_1), \mathsf{extract}(x_2) & \rightarrow & \mathsf{senc}(\mathsf{priv}(z),y_1);\ \neg\mathsf{cs}(x_2)
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Unwrap (sym/sym)v0} : & \mathsf{h}(x_1,y_2), \mathsf{senc}(y_1,y_2); \ \mathsf{unwrap}(x_1), \neg\mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{extract}(n_1),\ \neg\mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (sym/asym)v0} : & \mathsf{h}(x_1,\mathsf{priv}(z)), \mathsf{aenc}(y_1,\mathsf{pub}(z)); \ \mathsf{unwrap}(x_1), \neg\mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{extract}(n_1),\ \neg\mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (asym/sym)v0} : & \mathsf{h}(x_1,y_2), \mathsf{senc}(\mathsf{priv}(z),y_2); \ \mathsf{unwrap}(x_1), \neg\mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,\mathsf{priv}(z));\ \mathsf{extract}(n_1),\ \neg\mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (sym/sym)v1} : & \mathsf{h}(x_1,y_2), \mathsf{senc}(y_1,y_2); \ \mathsf{unwrap}(x_1), \mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{extract}(n_1),\ \neg\mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (sym/asym)v1} : & \mathsf{h}(x_1,\mathsf{priv}(z)), \mathsf{aenc}(y_1,\mathsf{pub}(z)); \ \mathsf{unwrap}(x_1), \mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{extract}(n_1),\ \neg\mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (asym/sym)v1} : & \mathsf{h}(x_1,y_2), \mathsf{senc}(\mathsf{priv}(z),y_2); \ \mathsf{unwrap}(x_1), \mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,\mathsf{priv}(z));\ \mathsf{extract}(n_1),\ \neg\mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (sym/sym)v2} : & \mathsf{h}(x_1,y_2), \mathsf{senc}(y_1,y_2); \ \mathsf{unwrap}(x_1), \mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{extract}(n_1),\ \mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (sym/asym)v2} : & \mathsf{h}(x_1,\mathsf{priv}(z)), \mathsf{aenc}(y_1,\mathsf{pub}(z)); \ \mathsf{unwrap}(x_1), \mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{extract}(n_1),\ \mathsf{cs}(n_1),\ \mathsf{L} \\
\text{Unwrap (asym/sym)v2} : & \mathsf{h}(x_1,y_2), \mathsf{senc}(\mathsf{priv}(z),y_2); \ \mathsf{unwrap}(x_1), \mathsf{cs}(n_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,\mathsf{priv}(z));\ \mathsf{extract}(n_1),\ \mathsf{cs}(n_1),\ \mathsf{L}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{KeyGenerate} : & & \xrightarrow{\text{new } n_1,k_1} & \mathsf{h}(n_1,k_1); \neg\mathsf{extract}(n_1), \mathsf{cs}(n_1),\ \mathsf{L} \\
\text{KeyPairGenerate} : & & \xrightarrow{\text{new } n_1,s} & \mathsf{h}(n_1,s), \mathsf{pub}(s);\ \neg\mathsf{extract}(n_1), \mathsf{cs}(n_1),\ \mathsf{L}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{SEncrypt} : & \mathsf{h}(x_1,y_1), y_2;\ \mathsf{encrypt}(x_1) & \rightarrow & \mathsf{senc}(y_2,y_1) \\
\text{SDecrypt} : & \mathsf{h}(x_1,y_1), \mathsf{senc}(y_2,y_1);\ \mathsf{decrypt}(x_1) & \rightarrow & y_2 \\
\text{AEncrypt} : & \mathsf{h}(x_1,\mathsf{priv}(z)), y_1;\ \mathsf{encrypt}(x_1) & \rightarrow & \mathsf{aenc}(y_1,\mathsf{pub}(z)) \\
\text{ADecrypt} : & \mathsf{h}(x_1,\mathsf{priv}(z)), \mathsf{aenc}(y_2,\mathsf{pub}(z));\ \mathsf{decrypt}(x_1) & \rightarrow & y_2 \\
\text{Set\_Encrypt} : & \mathsf{h}(x_1,y_1);\ \neg\mathsf{encrypt}(x_1) & \rightarrow & \mathsf{encrypt}(x_1) \\
\vdots & & \vdots & \\
\text{UnSet\_Encrypt} : & \mathsf{h}(x_1,y_1);\ \mathsf{encrypt}(x_1) & \rightarrow & \neg\mathsf{encrypt}(x_1) \\
\vdots & & \vdots &
\end{array}
$$

where $\mathsf{L} = \neg\mathsf{wrap}(n_1), \neg\mathsf{unwrap}(n_1), \neg\mathsf{encrypt}(n_1), \neg\mathsf{decrypt}(n_1), \neg\mathsf{sensitive}(n_1)$. The ellipsis in the set and unset rules indicates that similar rules exist for some other attributes.

Figure 11: PKCS#11 key management subset with nCipher SAM

Experiment 12 If we turn off the key import restrictions by setting unwrap_mech, unwrap_kek, and import, and give the intruder a wrapping key in his initial knowledge, we rediscover the original wrap/decrypt attack (Fig 1) in just over a minute.

Experiment 13 If we now re-impose the restrictions on attribute setting developed over experiments 1-3 with the original PKCS#11, no attack is found. Note however that this includes declaring unwrap and decrypt as a conflicting pair.

Experiment 14 Finally if we set the parameter all, the model functions as before and we rediscover the Trojan Wrapped Key attack, [4, Section 3.5].

6.2 Eracom Wrap mechanism

The Eracom protectServer API supports an additional key import-export mechanism which is intended to bind attributes to the key. The Eracom mechanism, `Wrapkey_DES3_CBC` is a combination of the 3DES encryption of the key using Cipher Block Chaining mode, and the HMAC of the attributes and key. Given a handle to a wrapping key $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ and the handle for a key to be wrapped $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$, it proceeds as follows, where $\mathsf{atts}(\mathsf{n}_2)$ represents the concatenation of all the attributes of $\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2)$:

1. Generate fresh MAC key M_k;
2. Return $\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_1), \mathsf{atts}(\mathsf{n}_2), \mathsf{hmac}(\mathsf{atts}(\mathsf{n}_2), \mathsf{k}_2, \mathsf{M}_\mathsf{k}), \mathsf{senc}(\mathsf{M}_\mathsf{k}, \mathsf{k}_1)$.

When a key is unwrapped under this mechanism, the MAC key is extracted from the final packet, and used to check that the HMAC matches up with the attributes and key supplied. If the HMAC does not match, key import will fail.

To analyse the Eracom measure we used the model in Figure 12. Note that only symmetric keys are supported, since the Eracom mechanism only applies to 3DES. In our model, the Eracom key wrapping method is the only one available. Note also that unfortunately the introduction of the hash function for the HMAC produces an explosion in the number of variables in the NuSMV model for our naïve encoding, so we cannot carry out the analysis without making some abstractions. In Figure 12, we give a simplified model where we

allow keys to have only the combined attributes 'encrypt and decrypt' and 'wrap and unwrap' (ed and wu) respectively. All keys are considered sensitive and extractable. The two attributes are declared to be a conflicting pair, and are both sticky on. Note that in our experiments with the PKCS#11 standard in Section 5, we identified wrap and unwrap as conflicting attributes, since they allowed a key to be re-imported with different attributes. With the Eracom wrap mechanism, it seems these attributes are no longer in conflict, since the HMAC forces the a key to be imported with the same attributes. We consider three additional function symbols, two constants cst_ed and cst_wu of mode CstAttr and the HMAC function of mode $\mathsf{hmac} : \mathsf{CstAttr} \times \mathsf{Key} \times \mathsf{Key} \rightarrow \mathsf{Hash}$. We only consider two combined attributes function symbols: $\mathsf{wu}, \mathsf{ed} : \mathsf{Nonce} \rightarrow \mathsf{Attribute}$

Note that all HMACS are assumed to be generated by the same secret key $\mathsf{M_k}$. In a model with no disequality tests such as ours, such an abstraction may introduce false attacks but is safe for proofs, since any attack involving different HMAC keys can be immediately mapped to one using a single HMAC key.

Experiment 15 For our simplified abstract model, NuSMV is able to show security in a model with 3 keys and 9 handles in 1 minute 45 seconds.

Experiment 16 With the conflict between the attributes removed, the wrap/decrypt attack (Figure 1) is found in just under 8 seconds.

6.3 Evaluation

In our analysis of the nCipher and Eracom measures, we have seen that both allow secure configurations to be produced, but neither is secure without additional careful restriction of attribute values. The nCipher approach is to flag up operations that may result in keys becoming insecure. If all warnings are on, functionality is quite limited - no new keys can be imported. If some warnings are turned off, attacks are possible. The Eracom measures involve using a MAC to ensure integrity of keys and attributes, but only apply to a symmetric key API. There is clearly scope for more work in engineering secure configurations of PKCS#11.

$$
\begin{array}{lrcl}
\text{Wrap}: & \mathsf{h}(x_1,y_1),\mathsf{h}(x_2,y_2);\ \mathsf{wu}(x_1),\mathsf{ed}(x_2) & \to & \mathsf{senc}(y_2,y_1),\mathsf{hmac}(\mathsf{ed},y_2) \\
 & \mathsf{h}(x_1,y_1),\mathsf{h}(x_2,y_2);\ \mathsf{wu}(x_1),\mathsf{wu}(x_2) & \to & \mathsf{senc}(y_2,y_1),\mathsf{hmac}(\mathsf{wu},y_2) \\
\text{Unwrap}: & \mathsf{h}(x_1,y_2),\mathsf{senc}(y_1,y_2), \mathsf{hmac}(\mathsf{ed},y_1);\ \mathsf{wu}(x_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{ed}(n_1) \\
 & \mathsf{h}(x_1,y_2),\mathsf{senc}(y_1,y_2), \mathsf{hmac}(\mathsf{wu},y_1);\ \mathsf{wu}(x_1) & \xrightarrow{\text{new } n_1} & \mathsf{h}(n_1,y_1);\ \mathsf{wu}(n_1) \\
\text{KeyGenerate}: & & \xrightarrow{\text{new } n_1,k_1} & \mathsf{h}(n_1,k_1);\mathsf{ed}(n_1) \\
 & & \xrightarrow{\text{new } n_1,k_1} & \mathsf{h}(n_1,k_1);\mathsf{wu}(n_1) \\
\text{SEncrypt}: & \mathsf{h}(x_1,y_1),y_2;\ \mathsf{ed}(x_1) & \to & \mathsf{senc}(y_2,y_1) \\
\text{SDecrypt}: & \mathsf{h}(x_1,y_1),\mathsf{senc}(y_2,y_1);\ \mathsf{ed}(x_1) & \to & y_2
\end{array}
$$

Figure 12: Abstract model of Eracom version of PKCS#11

The interested reader is invited to download the scripts from `http://www.lsv.ens-cachan.fr/~steel/pkcs11` and try out her own configurations.

7 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.

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 no matter how many pairs of conflicting attributes are identified, the fact that the attacker can re-import the same key several times means he can always circumvent the restrictions, as in the attack in Figure 5. To prevent this, we investigated three solutions: the trusted keys mechanism, which is part of the standard, and

the proprietary measures of nCipher and Eracom.

While our combination of a naïve propositional encoding and NuSMV proved capable of discovering some new attacks, there is clearly room for improvement here. We were forced to make further abstractions when dealing with the Eracom modifications, because of the combinatorial possibilities introduced by the HMAC function. One approach to tackling this would be to adapt a protocol analysis tool, already optimised for the Dolev-Yao intruder. Currently, no tool supports global mutable state, unbounded sessions, and customisable types (our 'well-modedness' condition). We are involved in ongoing work to adapt the AVISPA backend SATMC to support all three, since this was the tool that performed best in initial experiments with PKCS#11 [17].

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

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 tools to unbounded sessions and well-moded terms is one possible approach.

Acknowledgements

We would like to thank Mathieu Baudet for his detailed comments on an early version of this paper, and the anonymous reviewers of both CSF and JCS for many helpful suggestions.

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 *Proceedings of the 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 *Proceedings of the 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 *Proceeding of the 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.

A Proofs Pertaining to Section 4

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 $(S_0, V_0) \rightsquigarrow^* (S, V)$. We have that V is well-moded.*

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

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

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

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

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

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. By well-modedness of the rule, $t \in st(x\theta)$ for some variable $x \in vars(L')$. Since by hypothesis we have that $vars(L') \subseteq vars(L)$, we easily deduce that there exists $att(u) \in L$ (or $\neg att(u) \in L$) such that $x \in vars(u)$. Hence, t is a strict subterm of $att(u\theta)$ that occurs at an ill-moded position. This implies that $att(u\theta)$ is not well-moded. Since, by hypothesis, we have that $att(u\theta) \in dom(V_{k-1})$, 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 $(S_0, V_0) \rightsquigarrow^* (S, V)$. 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:

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

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

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

and a substitution θ 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 $v' \in S$. □

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 $f \in \Sigma \cup \mathcal{A}$. We assume w.l.o.g. that $f : \mathsf{m}_1, \ldots, \mathsf{m}_n \to \mathsf{m}$. 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:

$$\begin{aligned}
\overline{v\theta}^{\mathsf{sig}(v)} &= \overline{f(v_1,\ldots,v_n)\theta}^{\mathsf{sig}(f)} \\
&= f(\overline{v_1\theta}^{\mathsf{m}_1},\ldots,\overline{v_n\theta}^{\mathsf{m}_n}) \qquad \text{by definition of } \bar{\cdot}^{\mathsf{sig}(f)} \\
&= f(v_1\theta',\ldots,v_n\theta') \qquad \text{by induction hypothesis} \\
&= f(v_1,\ldots,v_n)\theta' \\
&= v\theta'
\end{aligned}$$

□

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: $(S_0, V_0) \rightsquigarrow (S_1, V_1) \rightsquigarrow \ldots \rightsquigarrow (S_k, V_k)$. Moreover, for each mode $\mathsf{m} \in \{\mathsf{sig}(t) \mid t \in \mathcal{PT}(\Sigma, \mathcal{N})\}$, we assume that there exists a well-moded term t_{m} of mode m such that $t_{\mathsf{m}} \in S_0$ (i.e. t_{m} is known by the attacker) and $\bar{\cdot}$ 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:

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

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, by definition of $\overline{\cdot}$, we have that $\overline{v\theta} = \overline{v\theta}^{\mathsf{sig}(v\theta)}$. Thus, the only case where $\overline{v\theta} \neq \overline{v\theta}^{\mathsf{sig}(v)}$ is when $\mathsf{sig}(v\theta) \neq \mathsf{sig}(v)$, i.e. 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, we have that $t_j\theta \in S_{k-1}$, and for each mode m, we have that $t_{\mathsf{m}} \in S_0 \subseteq S_{k-1}$, thus we have that $\overline{t_j\theta} \in \overline{S_{k-1}}$ and thus $t_j\theta' \in \overline{S_{k-1}}$;
- for each $a \in L$ (resp. $\neg a \in L$), we can use the same reasoning as before to deduce that either $\overline{a\theta} = a\theta'$, or a is a variable. By hypothesis, a is of the form $att(t)$ (i.e. a is not a variable), hence 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 $\overline{a\theta} = a\theta$. Altogether this allows us to conclude that $a\theta = a\theta'$ for any $(\neg)a \in L$. The reasoning also holds for any $(\neg)a \in L'$.

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 $V' = V_k$. It remains to show that $S' = \overline{S_k}$. We have that:

- $S' = \overline{S_{k-1}} \cup \{u_1\theta', \ldots, u_p\theta'\}$, and
- $\overline{S_k} = \overline{S_{k-1}} \cup \{\overline{u_1\theta}, \ldots, \overline{u_p\theta}\}$.

Hence, the only problematic case is when $u_j\theta' \neq \overline{u_j\theta}$, i.e. 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 $\overline{S_{k-1}} \subseteq S'$. $\square$