Attacking and Fixing PKCS#11 Security Tokens

Matteo Bortolozzo
Università Ca' Foscari
Venezia, Italy
mbortolo@dsi.unive.it

Matteo Centenaro
Università Ca' Foscari
Venezia, Italy
centenaro@dsi.unive.it

Riccardo Focardi
Università Ca' Foscari
Venezia, Italy
focardi@dsi.unive.it

Graham Steel
LSV, INRIA & CNRS &
ENS-Cachan
Cachan, France
graham.steel@inria.fr

ABSTRACT

We show how to extract sensitive cryptographic keys from a variety of commercially available tamper resistant cryptographic security tokens, exploiting vulnerabilities in their RSA PKCS#11 based APIs. The attacks are performed by Tookan, an automated tool we have developed, which reverse-engineers the particular token in use to deduce its functionality, constructs a model of its API for a model checker, and then executes any attack trace found by the model checker directly on the token. We describe the operation of Tookan and give results of testing the tool on 17 commercially available tokens: 9 were vulnerable to attack, while the other 8 had severely restricted functionality. One of the attacks found by the model checker has not previously appeared in the literature. We show how Tookan may be used to verify patches to insecure devices, and give a secure configuration that we have implemented in a patch to a software token simulator. This is the first such configuration to appear in the literature that does not require any new cryptographic mechanisms to be added to the standard. We comment on lessons for future key management APIs.

Categories and Subject Descriptors:
K.6.m [Miscellaneous]: Security

General Terms: Experimentation, Security, Verification

Keywords: Security APIs, key management, PKCS#11, model checking

1. INTRODUCTION

Tamper-resistant cryptographic security tokens such as smartcards and USB keys are an increasingly common component of distributed systems deployed in insecure environments. They are designed, for example, to enable authentication, to protect cryptographic values from malware, and to facilitate secure login for a variety of applications ranging from door entry to online banking. In this paper, we focus on tokens that achieve their goals by using internally stored cryptographic values. A token must offer an API to the outside world that allows the keys to be used for cryptographic functions and permits key management operations. This API is critical: it must be designed so that even if the device comes into contact with malicious applications, perhaps on a compromised host machine, the cryptographic values stored remain secret. It is difficult to design such an interface, and several key recovery attacks on so-called 'security APIs' have appeared in the literature [3, 5, 12]. The most commonly used standard for designing token interfaces is RSA PKCS#11 [14]. The API described by this standard, 'Cryptoki', is known to have vulnerabilities [6, 8], but since different devices implement different subsets of the standard, it was not previously known to what extent these vulnerabilities affected real devices.

In this paper we describe Tookan[1], an automated tool that reverse engineers the particular functionality offered by a device, constructs a formal model of this functionality, calls a model checker to search for possible attacks, and executes any attack trace found directly on the device. Our model is based on previous work by Delaune, Kremer and Steel, [8], but enriched significantly to better match the functionality we found on real devices. We describe optimisations to the model building process that result in models which can be handled efficiently by the model checker. We also contribute a meta-language for describing PKCS#11 configurations, used by the reverse-engineering part of our tool.

The results of testing the tool on commercially available devices are disquieting: every device that offered the functionality necessary to import and export sensitive keys in an encrypted form, a standard key management operation, did so in an insecure way allowing the key value to be recovered after a few calls to the API. Those not vulnerable to these attacks have very limited functionality (e.g. just asymmetric keypair generation and signing).

We then show how to use our tool to verify patched tokens. We present CryptokiX, a fiXed variant of the openCryptoki [13] software token simulator, which is configurable by selectively enabling different patches. This has allowed us to test our reverse-engineering framework on (simulated) devices implementing various combinations of security patches.

[1] Tool for cryptoki analysis

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

Among its patches, CryptokiX includes the first secure configuration to appear in the literature that does not require any new cryptographic mechanisms to be added to the standard.

Finally, we comment on the lessons for the next generation of standards for cryptographic key management such as IEEE 1619.3 and the OASIS Key Management Interoperability Protocol, currently in the draft stage.

The paper is organized as follows. We first briefly review the PKCS#11 API and some of its known problems which can lead to vulnerabilities (section 2). We then describe our formal model of the API and show how our tool extracts information from the token to allow us to build the model for a particular device (section 3). We give our experimental results on various commercially available devices in section 4. We describe how to use the tool to find secure configurations in section 5. We conclude with a discussion of open problems and future key management APIs in section 6.

2. THE PKCS#11 API

RSA PKCS#11 describes its 'Cryptoki' API in just under 400 pages [14]. We only have room here for a brief description, and we will concentrate on the details that give rise to the category of vulnerabilities found by our tool. In a PKCS#11-based API, applications initiate a *session* with the cryptographic token, by supplying a PIN. Note that if malicious code is running on the host machine, then the user PIN may easily be intercepted, e.g. by a keylogger or by a tampered device driver, allowing an attacker to create his own sessions with the device, a point conceded in the security discussion in the standard [14, p. 31]. PKCS#11 is intended to protect its sensitive cryptographic keys even when connected to a compromised host.

Once a session is initiated, the application may access the *objects* stored on the token, such as keys and certificates. However, access to the objects is controlled. 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 have *attributes*, which may be bitstrings e.g. the value of a key, or Boolean flags signalling properties of the object, e.g. whether the key may be used for encryption, or for encrypting other keys. New objects can be created by calling a key generation command, or by 'unwrapping' an encrypted key packet. In both cases a fresh handle is returned.

When a function in the token's API is called with a reference to a particular object, the token first checks that the attributes of the object allow it to be used for that function. For example, if the encrypt function is called with the handle for a particular key, that key must have its encrypt attribute set. To protect a key from being revealed, the attribute sensitive must be set to true. This means that requests to view the object's key value via the API will result in an error message. Once the attribute sensitive has been set to true, it cannot be reset to false. This gives us the principal security property stated in the standard: attacks, even if they involve compromising the host machine to obtain the PIN, cannot "compromise keys marked 'sensitive', since a key that is sensitive will always remain sensitive", [14, p. 31]. Such a key may be exported outside the device if it is encrypted by another key, but only if its extractable attribute is set to true. An object with an extractable attribute set to false may not be read by the API, and additionally, once set to false, the extractable attribute cannot be set to true. Protection of the keys essentially relies on the sensitive and extractable attributes.

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 extractable set.

Trace:

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

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

Figure 1: Wrap/Decrypt attack

Attacks on PCKS#11

A number of recent papers have shown attacks which compromise sensitive keys [6, 8, 11]. Many of these are 'key separation' attacks, where the attributes of a key are set 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 [6]. To determine the value of a sensitive key, the attacker simply wraps it and then decrypts it, as shown in Figure 1. Here we introduce our notation for PKCS#11 based APIs, defined more formally in the next section: $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ is a predicate stating that there is a handle n_1 for a key k_1 stored on the device. The symmetric encryption of k_1 under key k_2 is represented by $\{\mathsf{k}_1\}_{\mathsf{k}_2}$. Note also that according to the wrapping formats defined in PKCS#11, the device 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 of telling that the packet it is decrypting contains a key.

Delaune, Kremer and Steel proposed a Dolev-Yao style abstract model for PKCS#11 APIs, and showed how difficult it is to prevent these kinds of attacks: the commands can be restricted to prevent certain conflicting attributes from being set on the same object, but still more attacks arise [8]. However, it was not known whether any real devices following the standard actually implement key management like this, since much of the functionality is optional. This was a motivation for the tool we describe in this paper. Furthermore, no previous analysis of PKCS#11 gives a configuration that is proven secure without adding new mechanisms to the standard. This was another motivation for our work.

Note that our tool is focused on these attacks that involve no cryptanalysis. There are further known vulnerabilities in PKCS#11 APIs exploiting particular details of the cryptographic algorithms supported [6]. Covering these remains as further work for our tool development project.

3. MODEL

Our model follows the approach used by Delaune, Kremer and Steel (DKS) [8]. The idea is to model the device as being connected to a host under the complete control of an intruder, representing a malicious piece of software. The intruder can call the commands of the API in any order he likes using any values that he knows. We abstract

away details of the cryptographic algorithms in use, following the classical approach of Dolev and Yao [10]. Bitstrings are modelled as terms in an abstract algebra and the rules of the API and the abilities of an attacker are written as deduction rules in the algebra. The intruder is assumed not to be able to crack the encryption algorithm by brute-force search or similar means, thus he can only read an encrypted message if he has the correct key. We analyse security as reachability, in the model, of *attack* states, i.e. states where the intruder knows the value of a key stored on the device with the sensitive attribute set to true, or the extractable attribute set to false.

3.1 Basic Notions

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. $\{\mathsf{x}\}_\mathsf{y}$ representing symmetric encryption of plaintext x under key y, and $\{\mathsf{x}\}_\mathsf{y}$ representing public key encryption of x under y. Function symbols of arity 0 are called *constants*. This includes the Boolean constants true ($\top$) and false ($\bot$). The set of *plain terms* $\mathcal{PT}$ 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 $\mathcal{F}$ of predicate symbols, disjoint from Σ, from which we derive a set of *facts*. The set of *facts* is defined as

$$\mathcal{FT} = \{p(t, b) \mid p \in \mathcal{F}, t \in \mathcal{PT}, b \in \{\top, \bot\}\}$$

In this way, we can explicitly express the Boolean value b of an attribute p on a term t by writing $p(t, b)$. For example, to state that the key referred to by n has the wrap attribute set we write $\mathsf{wrap}(\mathsf{n}, \top)$. This is a difference in the syntax of our model compared to DKS, where attributes are expressed as literals ($\mathsf{wrap}(\mathsf{n})$ or $\neg\mathsf{wrap}(\mathsf{n})$).

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 $T, T' \subseteq \mathcal{PT}$ are sets of plain terms $L, L' \subseteq \mathcal{F}$ are sets of facts and $\tilde{n} \subseteq \mathcal{N}$ is a set of names. 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 facts in L hold 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.

Example The following rule models wrapping:

$$\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1), \mathsf{h}(\mathsf{x}_2, \mathsf{y}_2);\ \mathsf{wrap}(\mathsf{x}_1, \top), \mathsf{extract}(\mathsf{x}_2, \top) \to \{\mathsf{y}_2\}_{\mathsf{y}_1}$$

Intuitively, $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$ is a handle x_1 for key y_1 while term $\{\mathsf{y}_2\}_{\mathsf{y}_1}$ represents a key y_2 wrapped with y_1. Since the attribute wrap for key y_1 is set, noted as $\mathsf{unwrap}(\mathsf{x}_1, \top)$, and key y_2 is extractable, written $\mathsf{extract}(\mathsf{x}_2, \top)$, then we can wrap y_2 with y_1, creating $\{\mathsf{y}_2\}_{\mathsf{y}_1}$.

Figure 2: Tookan system diagram

The semantics of the model is defined in a standard way in terms of a transition system. Each state in the model consists of a set of terms in the intruder's knowledge, and a set of global state predicates. The intruder's knowledge increases monotonically with each transition, but the global state is non-monotonic. For a formal semantics, we refer to the literature [8].

3.2 Modelling Real Tokens

The motivation for our work was to try to model the PKCS#11 implementations of real tokens. Our experiments on the tokens proceed following the system diagram in figure 2. First, Tookan extracts the capabilities of the token following a reverse engineering process (1). The results of this task are written in a meta-language for PKCS#11 models, described below. Tookan uses this information to generate a model in the above described style (2), which is given as input to the SATMC model checker [1]. Model checker output (3) is sent to Tookan for testing on the token (4).

In table 1 we give the syntax for the model meta-language. The language describes the functions and attributes supported by the token. It is also designed to capture the restrictions on functionality the token imposes. In table 2 we give our model for PKCS#11 showing how it is parametrised by the meta-model. We describe this relationship in more detail below. Note that due to space constraints, the model we give here is slightly simplified: in Tookan we construct separate sets of `Attribute_Restrictions` and `Templates` for asymmetric and symmetric keys, since many tokens impose quite different policies for these two different types. The full syntax and all the configurations derived during our experiments on real tokens can be viewed online[2].

Cryptographic Keys and Key Attributes

Tookan tests to see if a token supports the generation of asymmetric or symmetric keys, and returns the results, respectively, in the Booleans `supports_symmetric_keys` and `supports_asymmetric_keys`. By trying successive key generation commands, Tookan extracts the list of attributes in use for key objects and delivers these as the list `attributes`. These are used throughout the construction of the model and are noted as $\mathcal{A}$ in table 2. Note that as shown in the BNF in table 1, we restrict attention at the moment to a subset of PKCS#11 attributes. We do not consider signing and verification capabilities for example.

Functions

Tookan returns a list of `functions` supported, including one important function not modelled in the DKS work: CreateObject. This function allows the application to directly set the value of a new key on the device. Only the functions on the list are included in the final model.

[2] `http://secgroup.ext.dsi.unive.it/pkcs11-security`

```
PKCS11_CONFIG      = Key_Types
                     Functions
                     Attributes
                     Attribute_Restrictions
                     Templates
                     Flags

Key_Types          = supports_symmetric_keys(BOOL);
                     supports_asymmetric_keys(BOOL);

Functions          = functions( FunctionList );

FunctionList       = nil |  Function, FunctionList

Function           = wrap | unwrap | encrypt | decrypt | create_object

Attributes         = attributes( AttributeList );

AttributeList      = nil | Attribute, AttributeList

Attribute          =  sensitive | extract | always_sensitive |
                      never_extract | wrap | unwrap | encrypt |  decrypt

Attribute_Restrictions = Sticky_On
                         Sticky_Off
                         Conflicts
                         Tied

Sticky_On          = sticky_on( AttributeList );

Sticky_Off         = sticky_off( AttributeList );

Conflicts          = conflict( AttributePairList );

Tied               = tied( AttributePairList );

AttributePairList  = nil | ( Attribute,Attribute ) , AttributePairList

Templates          = generate_templates(TemplateList);
                     create_templates(TemplateList);
                     unwrap_templates(TemplateList);

TemplateList       = nil | (Template) , TemplateList

Template           = nil | ( Attribute , BOOL ) , Template

Flags              = sensitive_prevents_read(BOOL);
                     unextractable_prevents_read(BOOL);

BOOL               = true | false
```

Table 1: Syntax of Meta-language for describing PKCS#11 configurations

$$\text{KeyGenerate}: \quad \xrightarrow{\text{new n,k}} \quad \mathsf{h}(\mathsf{n},\mathsf{k}); \mathcal{A}(\mathsf{n},\mathsf{B}) \qquad (\text{with } \mathsf{B} \in \mathcal{G})$$

$$\text{KeyPairGenerate}: \quad \xrightarrow{\text{new n,s}} \quad \mathsf{h}(\mathsf{n},\mathsf{s}), \mathsf{pub}(\mathsf{s}); \mathcal{A}(\mathsf{n},\mathsf{B}) \quad (\text{with } \mathsf{B} \in \mathcal{G})$$

$$\begin{array}{lrcl}
\text{Wrap (sym/sym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1), \mathsf{h}(\mathsf{x}_2,\mathsf{y}_2);\ \mathsf{wrap}(\mathsf{x}_1,\top), \mathsf{extract}(\mathsf{x}_2,\top) & \rightarrow & \{\!|\mathsf{y}_2|\!\}_{\mathsf{y}_1} \\
\text{Wrap (sym/asym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{priv}(\mathsf{z})), \mathsf{h}(\mathsf{x}_2,\mathsf{y}_2);\ \mathsf{wrap}(\mathsf{x}_1,\top), \mathsf{extract}(\mathsf{x}_2,\top) & \rightarrow & \{\mathsf{y}_2\}_{\mathsf{pub}(\mathsf{z})} \\
\text{Wrap (asym/sym)}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1), \mathsf{h}(\mathsf{x}_2,\mathsf{priv}(\mathsf{z}));\ \mathsf{wrap}(\mathsf{x}_1,\top), \mathsf{extract}(\mathsf{x}_2,\top) & \rightarrow & \{\!|\mathsf{priv}(\mathsf{z})|\!\}_{\mathsf{y}_1}
\end{array}$$

$$\begin{array}{lrcll}
\text{Unwrap (sym/sym)}: & \mathsf{h}(\mathsf{x},\mathsf{y}_2), \{\!|\mathsf{y}_1|\!\}_{\mathsf{y}_2};\ \mathsf{unwrap}(\mathsf{x},\top), & \xrightarrow{\text{new n}_1} & \mathsf{h}(\mathsf{n}_1,\mathsf{y}_1);\ \mathcal{A}(\mathsf{n}_1,\mathsf{B}) & (\text{with } \mathsf{B} \in \mathcal{U}) \\
\text{Unwrap (sym/asym)}: & \mathsf{h}(\mathsf{x},\mathsf{priv}(\mathsf{z})), \{\mathsf{y}_1\}_{\mathsf{pub}(\mathsf{z})};\ \mathsf{unwrap}(\mathsf{x},\top), & \xrightarrow{\text{new n}_1} & \mathsf{h}(\mathsf{n}_1,\mathsf{y}_1);\ \mathcal{A}(\mathsf{n}_1,\mathsf{B}) & (\text{with } \mathsf{B} \in \mathcal{U}) \\
\text{Unwrap (asym/sym)}: & \mathsf{h}(\mathsf{x},\mathsf{y}_2), \{\!|\mathsf{priv}(\mathsf{z})|\!\}_{\mathsf{y}_2};\ \mathsf{unwrap}(\mathsf{x},\top), & \xrightarrow{\text{new n}_1} & \mathsf{h}(\mathsf{n}_1,\mathsf{priv}(\mathsf{z}));\ \mathcal{A}(\mathsf{n}_1,\mathsf{B}) & (\text{with } \mathsf{B} \in \mathcal{U})
\end{array}$$

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

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

$$\text{SetAttribute}: \quad \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{a}(\mathsf{x}_1,\bot), \mathcal{A}^{\mathsf{conf(a)}}(\mathsf{x}_1,\bot) \rightarrow\ ;\ \mathsf{a}(\mathsf{x}_1,\top), \mathcal{A}^{\mathsf{tied(a)}}(\mathsf{x}_1,\top) \quad (\text{with } \mathsf{a} \in \mathcal{A} \setminus \texttt{sticky\_off\_attributes})$$

$$\text{UnsetAttribute}: \quad \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{a}(\mathsf{x}_1,\top) \rightarrow\ ;\ \mathsf{a}(\mathsf{x}_1,\bot), \mathcal{A}^{\mathsf{tied(a)}}(\mathsf{x}_1,\bot) \quad (\text{with } \mathsf{a} \in \mathcal{A} \setminus \texttt{sticky\_on\_attributes})$$

$$\text{CreateObject}: \quad \mathsf{x}; \quad \xrightarrow{\text{new n}} \quad \mathsf{h}(\mathsf{n},\mathsf{x});\ \mathcal{A}(\mathsf{n},\mathsf{B}) \quad (\text{with } \mathsf{B} \in \mathcal{C})$$

$$\text{GetAttribute}: \quad \mathsf{h}(\mathsf{n},\mathsf{x}); \mathsf{extract}(\mathsf{n},\mathsf{b}_\mathsf{e}), \mathsf{sensitive}(\mathsf{n},\mathsf{b}_\mathsf{s}) \rightarrow \mathsf{x} \quad \left(\begin{array}{l} \text{with } \mathsf{b}_\mathsf{e}, \mathsf{b}_\mathsf{s} \in \{\bot,\top\} \text{ and} \\ \texttt{sensitive\_prevents\_read}(\top) \Rightarrow \mathsf{b}_\mathsf{s} = \bot \text{ and} \\ \texttt{unextractable\_prevents\_read}(\top) \Rightarrow \mathsf{b}_\mathsf{e} = \top \end{array} \right)$$

Notation:
- $\mathcal{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{m}\}$ denotes the (ordered) set of `attributes`
- $\mathsf{B} = \{\mathsf{b}_1, \ldots, \mathsf{b}_\mathsf{m}\}$ denotes a *template*, i.e. a set of Boolean values for attributes $\mathcal{A}$
- $\mathcal{A}(\mathsf{n},\mathsf{B})$ stands for $\mathsf{a}_1(\mathsf{n},\mathsf{b}_1), \ldots, \mathsf{a}_\mathsf{m}(\mathsf{n},\mathsf{b}_\mathsf{m})$ while $\mathcal{A}(\mathsf{n},\mathsf{b})$ stands for $\mathsf{a}_1(\mathsf{n},\mathsf{b}), \ldots, \mathsf{a}_\mathsf{m}(\mathsf{n},\mathsf{b})$
- $\mathcal{B}(\mathsf{n},\mathsf{B})$, with $\mathcal{B} = \{\mathsf{a}_{\mathsf{j}_1}, \ldots, \mathsf{a}_{\mathsf{j}_\mathsf{k}}\} \subseteq \mathcal{A}$ denotes $\mathsf{a}_{\mathsf{j}_1}(\mathsf{n},\mathsf{b}_{\mathsf{j}_1}), \ldots, \mathsf{a}_{\mathsf{j}_\mathsf{k}}(\mathsf{n},\mathsf{b}_{\mathsf{j}_\mathsf{k}})$, i.e., the projection of $\mathcal{A}(\mathsf{n},\mathsf{B})$ on $\mathcal{B}$
- $\mathcal{A}^{\mathsf{conf(a)}}$ is the subset of attributes $\mathsf{a}' \in \mathcal{A}$ conflicting with a, i.e., such that $\texttt{conflict}(\mathsf{a}',\mathsf{a})$
- $\mathcal{A}^{\mathsf{tied(a)}}$ is the subset of attributes $\mathsf{a}' \in \mathcal{A}$ tied to a, i.e., such that $\texttt{tied}(\mathsf{a}',\mathsf{a})$

Table 2: PKCS#11 key management subset with side conditions from the meta-language of table 1

Key Generation Templates

A major difference between our model and the DKS model is that we take into account *key templates*. In DKS, the key generation commands create a key with all its attributes unset [8, Fig. 2]. Attributes are then be enabled one by one using the SetAttribute command. In our experiments with real devices, we discovered that some tokens do not allow attributes of a key to be changed. Instead, they use a key template specifying settings for the attributes which are given to freshly generated keys. Templates are used for the import of encrypted keys (unwrapping), key creation using CreateObject and key generation. The template to be used in a specific command instance is specified as a parameter, and must come from a set of valid templates, which we label $\mathcal{G}, \mathcal{C}$ and $\mathcal{U}$ for the valid templates for key generation, creation and unwrapping respectively. Tookan can construct the set of templates in two ways: the first, by exhaustively testing the commands using templates for all possible combinations of attribute settings, which may be very time consuming, but is necessary if we aim to verify the security of a token. The second method is to construct the set of templates that should be allowed based on the reverse-engineered attribute policy (see next paragraph). This is an approximate process, but can be useful for quickly finding attacks. Indeed, in our experiments, we found that these models reflected well the operation of the token, i.e. the attacks found by the model checker all executed on the tokens without any 'template invalid' errors.

Attribute Policies

Most tokens we tested attempt to impose some restrictions on the combinations of attributes that can be set on a key and how these may be changed. Some restrictions are listed as mandatory in the standard, though we found that not all tokens actually implement them. In our meta-model language, we describe four kinds of restriction that Tookan can infer from its reverse engineering process:

`Sticky_on` These are attributes that once set, may not be unset. The PKCS#11 standard lists some of these [14, Table 15]: sensitive for secret keys, for example. As shown in table 2, the UnsetAttribute rule is only included for attributes which are not sticky on. To test if a device treats an attribute as sticky on, Tookan attempts to create a key with the attribute on, and then calls SetAttribute to change the attribute to off.

`Sticky_off` These are attributes that once unset may not be set. In the standard, extractable is listed as such an attribute. As shown in table 2, the SetAttribute rule is only included for attributes which are not sticky off. To test if a device treats an attribute as sticky on, Tookan attempts to create a key with the attribute off, and then calls SetAttribute to change the attribute to on.

`Conflicts` Many tokens (appear to) disallow certain pairs of attributes to be set, either in the template or when changing attributes on a live key. For example, some tokens do not allow sensitive and extractable to be set on the same key. As shown in table 2, the SetAttribute rule is adjusted to prevent conflicting attributes from being set on an object or on the template. When calculating the template sets $\mathcal{C}, \mathcal{G}, \mathcal{U}$ (see above), we forbid templates which have both the conflicting attributes set. To test if a device treats an attribute pair as a conflict, Tookan attempts to generate a key with the the pair of attributes set, then if no error is reported, it calls GetAttribute to check that the token really has created a key with the desired attributes set.

`Tied` Some tokens automatically set the value of some attributes based on the value of others. For example, many tokens set the value of always_sensitive based on the value of the attribute sensitive. As shown in table 2, the SetAttribute and UnsetAttribute rules are adjusted to account for tied attributes. The template sets $\mathcal{C}, \mathcal{G}, \mathcal{U}$ are also adjusted accordingly. To test if a device treats an attribute pair as tied, Tookan attempts to generate a key with some attribute a on and all other attributes off. It then uses GetAttribute to examine the key as it was actually created, and tests to see if any other attributes were turned on.

Respecting the Standard

Tookan checks two vital aspects of the token's behaviour: footnote 7 in table 15 of the standard specifies that certain attributes of an object may not be revealed via a GetAttribute query if either the object's sensitive attribute is set to true, or the extractable attribute is set to false. We test these conditions independently by attempting to read the attribute giving the true value of a secret key. The results are respectively stored in `sensitive_prevents_read` and `unextractable_prevents_read`. Clearly if either of these are false for a real token, we have a vulnerability, since these are two of the critical security properties the token is supposed to provide. Nevertheless, we include them in our model since several of the tokens we tested fail to enforce these restrictions.

Optimising the Template Set

For tokens which allow a large number of different templates, the sets $\mathcal{C}, \mathcal{G}, \mathcal{U}$ can get very large, which creates a model that is very slow to search. We apply some simple optimisations to the template set that make a significant improvement to performance. Specifically, we construct a set of attributes $\mathcal{A}^+$ which only appear in the model set to true and do not appear in any conflicts. It is easy to see that if there are no rules that test this attribute is false, and it does not affect the value of any other attributes, then we need only construct templates where these attributes are set to true. Likewise, we construct a set of attributes $\mathcal{A}^-$ which only appear in the model set to false. We need not construct templates where this attribute is true.

Implementing Abstractions for Proving Security

In previous work [11], we proved that for models where attributes are static (i.e. they are all both sticky on and sticky off), we can make an over-approximation for the generation of fresh handles and keys that allows us to prove security for an unbounded number of handles and keys using a small finite model. Intuitively, the idea is to generate one key for each template, and to allocate one handle for each template. If a template is used twice, the same handle is generated, even if the key is different. Tookan has an option that builds a model following this abstraction. Since it is an over approximation, the abstract model may suggest false attacks. In this case, the user can switch back to the concrete, bounded model, where a user defined number of fresh handles and keys are used.

3.3 Limitations of Reverse Engineering

Our reverse engineering process is not complete: it may result in a model that is too restricted to find some attacks possible on the token, and it may suggest false attacks which cannot be executed on the token. This is because in theory, no matter what the results of our finite test sequence, the token may be running any software at all, perhaps even behaving randomly. However, if a token implements its attribute policy in the manner in which we can describe it, i.e. as a combination of sticky on, sticky off, conflict and tied attributes, then our process is complete in the sense that the model built will reflect exactly what the token can do (modulo the usual Dolev-Yao abstractions for cryptography).

In our testing, the model performed very well: the Tookan consistently found true attacks on flawed tokens, and we were unable to find 'by hand' any attacks on tokens which the model checker deemed secure. This suggests that real devices do indeed implement their attribute policies in a manner similar to our model.

4. RESULTS

In this section, we report experimental results from using our tool to find attacks on commercially available devices. We acquired as many tokens as we could subject to our lab budgets, and the retail or loan availability of single tokens and cards. Tokens cost anything from 20 to 400 USD, with the global market estimated at 5 billion USD[3]. We also tested our tool on two software simulators, intended for development purposes. Table 3 summarises the outcome of the analysis. For each token, we give a summary of the configuration information obtained from the token and a core subset of the attacks we found. Our testing on tokens is ongoing. Latest results can be viewed at the project website[4].

4.1 Implemented functionality

Columns 'sym' and 'asym' respectively indicate whether or not symmetric and asymmetric key cryptography are supported, i.e. the values of `supports_symmetric_keys` and `supports_asymmetric_keys` from the extracted configuration. We do not attempt to distinguish which particular cryptographic algorithms are supported in our analysis, since it is not relevant to the kinds of attacks we are looking for. Both kinds of cryptography are available on all the devices except three: the Eutron Crypto Identity ITSEC, Gemalto Smart Enterprise Guardian and the Gemalto SafeSite Classic TPC IS V1, which only provide asymmetric key cryptography. This last device should implement both symmetric and asymmetric cryptography according to its specification, but the one we tested could not generate and use symmetric keys. This may be a hardware issue with the specific token we possess.

Column 'cobj' refers to the possibility of inserting external, unencrypted, keys on the device via `C_CreateObject` PKCS#11 function, i.e. whether `create_object` is included in the list of functions in the extracted configuration. This is allowed by almost all of the analysed tokens. Although this command does not directly violate a security property, allowing known keys onto a device is generally a dangerous thing: an attacker might import an untrusted wrapping key from outside and ask the device to wrap a sensitive internal key with it [8].

The next column, 'chan', refers to the possibility of changing key attributes through `C_SetAttributeValue`. This functionality can easily be abused if not limited in some way. For example, it is clear (and stated in the standard) that it should never be possible to make a sensitive key nonsensitive. The behaviour of the `C_SetAttributeValue` command for a particular token is reported to the model checker via the `sticky_on` and `sticky_off` lists. A tick in this column indicates that at least one attribute was found that was not both `sticky_on` and `sticky_off`. The three Feitian devices correctly limit `C_SetAttributeValue` so that a sensitive key can never be changed into nonsensitive. However, this is of no use, since these tokens let any user directly access sensitive and unextractable keys (see attacks a3 and a4), disregarding the standard. The Sata and the Gemalto SafeSite Classic V2 devices are the only ones which allow the sensitive attribute to be unset with no limitation; this is in a perverse sense coherent, as just like the Feitian devices, they let any user access sensitive/unextractable keys. An interesting case is the Eracom HSM simulator, which allows attribute change, but correctly implements the above mentioned policy, i.e., it disallows making a sensitive key nonsensitive, while also making sensitive keys unreadable: in this way, once a key is set as sensitive it will never become directly accessible. Subtler attacks on the keys are still possible by exploiting wrap/unwrap functions (see below attacks a1 and a2).

The following two columns, 'w' and 'ws', respectively indicate whether the token permits wrapping of nonsensitive and sensitive keys. It is discouraging to observe that every device providing 'ws', i.e., the wrapping of sensitive keys, is also vulnerable to attack. All the other devices avoid attacks at the price of removing such functionality. Forbidding the wrapping of sensitive keys is a quite limiting design choice since it compromises any proper management of sensitive keys among different devices. Wrapping sensitive keys is necessary in order to export/import those keys in a secure way. Most of these 'limited' tokens simply remove the whole wrapping functionality, i.e., both 'w' and 'ws'. There are however two devices which allow the wrapping of nonsensitive keys only: SafeNet iKey and Siemens CardOS. Although this choice is less restrictive than removing the whole wrapping functionality, it seems difficult to think of an application where this would be a useful functionality. As we will discuss in the next section, it is indeed possible to produce a secure token configuration which allows wrapping (and unwrapping) of sensitive keys.

4.2 Attacks

Attack a1 is a wrap/decrypt attack as discussed in section 2. The attacker exploits a key k_2 with attributes wrap and decrypt and uses it to attack a sensitive key k_1. Using our notation from section 3:

$$\begin{array}{ll} \mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2),\ \mathsf{h}(\mathsf{n}_1,\mathsf{k}_1) \rightarrow \ \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} \\ \mathsf{SDecrypt}: & \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2),\ \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} \rightarrow \ \mathsf{k}_1 \end{array}$$

As we have discussed above, the possibility of inserting new keys in the token (column 'cobj') might simplify further the attack. It is sufficient to add a known wrapping key:

$$\begin{array}{ll} \mathsf{CreateObject}: & \mathsf{k}_2 \xrightarrow{\text{new } \mathsf{n}_2} \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2) \\ \mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2),\ \mathsf{h}(\mathsf{n}_1,\mathsf{k}_1) \rightarrow \ \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} \end{array}$$

[3]InfoSecurity Magazine February 2010, `http://fanaticmedia.com/infosecurity/archive/Feb10/AuthenticationTokensstory.htm`

[4]`http://secgroup.ext.dsi.unive.it/pkcs11-security`

	Device		Supported Functionality						Attacks found					
	Company	Model	sym	asym	cobj	chan	w	ws	a1	a2	a3	a4	a5	mc
USB	Aladdin	eToken PRO	✓	✓	✓	✓	✓	✓	✓	✓				a1
	Athena	ASEKey	✓	✓	✓									
	Bull	Trustway RCI	✓	✓	✓	✓	✓	✓	✓	✓				a1
	Eutron	Crypto Id. ITSEC		✓	✓									
	Feitian	StorePass2000	✓	✓	✓	✓	✓	✓		✓	✓	✓		a3
	Feitian	ePass2000	✓	✓	✓	✓	✓	✓		✓	✓	✓		a3
	Feitian	ePass3003Auto	✓	✓	✓	✓	✓	✓		✓	✓	✓		a3
	Gemalto	Smart Enterprise Guardian		✓		✓								
	MXI Security	Stealth MXP Bio	✓	✓		✓								
	SafeNet	iKey 2032	✓	✓	✓		✓							
	Sata	DKey	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	a3
Card	ACS	ACOS5	✓	✓	✓	✓								
	Athena	ASE Smartcard	✓	✓	✓									
	Gemalto	Cyberflex V2	✓	✓	✓		✓	✓		✓				a2
	Gemalto	SafeSite Classic TPC IS V1		✓		✓								
	Gemalto	SafeSite Classic TPC IS V2	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	a3
	Siemens	CardOS V4.3 B	✓	✓	✓		✓					✓		a4
Soft	Eracom	HSM simulator	✓	✓		✓	✓	✓	✓	✓		✓		a1
	IBM	opencryptoki 2.3.1	✓	✓	✓	✓	✓	✓	✓	✓		✓		a1

	Acronym	Description
Supported functionality	sym	symmetric-key cryptography
	asym	asymmetric-key cryptography
	cobj	inserting new keys via `C_CreateObject`
	chan	changing key attributes
	w	wrapping keys
	ws	wrapping sensitive keys
Attacks	a1	wrap/decrypt attack based on symmetric keys
	a2	wrap/decrypt attack based on asymmetric keys
	a3	sensitive keys are directly readable
	a4	unextractable keys are directly readable (forbidden by the standard)
	a5	sensitive/unextractable keys can be changed into nonsensitive/extractable
	mc	first attack found by Tookan

Table 3: Summary of results on devices

The attacker can then decrypt $\{\mathsf{k}_1\}_{\mathsf{k}_2}$ since he knows key k_2. SATMC discovered this variant of the attack on vulnerable tokens. We note that despite its apparent simplicity, this attack has not appeared before in the PKCS#11 security literature [6, 8].

Attack a2 is a variant of the previous ones in which the wrapping key is a public key $\mathsf{pub}(\mathsf{z})$ and the decryption key is the corresponding private key $\mathsf{priv}(\mathsf{z})$:

$$\begin{array}{ll} \mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{pub}(\mathsf{z})),\ \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) \rightarrow \ \{\mathsf{k}_1\}_{\mathsf{pub}(\mathsf{z})} \\ \mathsf{ADecrypt}: & \mathsf{h}(\mathsf{n}_2, \mathsf{priv}(\mathsf{z})),\ \{\mathsf{k}_1\}_{\mathsf{k}_2} \rightarrow \ \mathsf{k}_1 \end{array}$$

In this case too, the possibility of importing key pairs simplifies even more the attacker's task by allowing him to import a public wrapping key while knowing the corresponding private key. Once the wrap of the sensitive key has been performed, the attacker can decrypt the obtained ciphertext using the private key.

Attack a3 is a clear flaw in the PKCS#11 implementation. It is explicitly required that the value of sensitive keys should never be communicated outside the token. In practice, when the token is asked for the value of a sensitive key, it should return some "value is sensitive" error code. Instead, we found that some of the analysed devices just return the plain key value, ignoring this basic policy. Attack a4 is similar to a3: PKCS#11 requires that keys declared to be unextractable should not be readable, even if they are nonsensitive. If they are in fact readable, this is another violation of PKCS#11 security policy.

Finally, attack a5 refers to the possibility of changing sensitive and unextractable keys respectively into nonsensitive and extractable ones. Only the Sata and Gemalto SafeSite Classic V2 tokens allow this operation. However, notice that this attack is not adding any new flaw for such devices, given that attacks a3 and a4 are already possible and sensitive or unextractable keys are already accessible.

4.3 Model-checking results

Column 'mc' reports which of the attacks has been automatically rediscovered via model-checking. SATMC terminates once it has found an attack, hence we report the attack that was found first. Run-times for finding the attacks vary from a couple of seconds to just over 3 minutes. We evaluate the performance of the model checker further in section 6.

5. FINDING SECURE CONFIGURATIONS

As we noted in the last section, none of the tokens we tested are able to import and export sensitive keys in a secure fashion. In particular, all the analysed tokens are either insecure or have been drastically restricted in their functionality, e.g. by completely disabling wrap and unwrap. Intermediate approaches are in fact possible: the standard can be patched without necessarily removing the wrapping functionality [9]. In this section, we present CryptokiX, a software (fiXed) implementation of a Cryptoki token, whose security is configurable by selectively enabling different patches. As well as providing Tookan with test data, this proof-of-concept of a secure token has also been adopted for educational purposes in a security lab class at the University of Venice, during which students are challenged to extract a sensitive key from a token which has only a subset of the patches turned on, so as to be insecure but not easy to attack [2].

Our starting point is openCryptoki [13], an open-source PKCS#11 implementation for Linux including a software token for testing. As shown in Table 3, the analysis of openCryptoki software token has revealed that it is subject to all the non-trivial attacks. This is in a sense expected, as it implements the standard 'as is', i.e., with no security patches. We have thus extended openCryptoki with:

Conflicting attributes. We have seen, for example, that it is insecure to allow the same key to be used for wrapping and decrypting. In CryptokiX it is possible to specify a set of conflicting attributes.

Sticky attributes. We know that some attributes should always be sticky, such as sensitive. This is also useful when combined with the 'conflicting attributes' patch above: if wrap and decrypt are conflicting, we certainly want to avoid that the wrap attribute can be unset so as to allow the decrypt attribute to be set.

Wrapping formats. It has been shown that specifying a non-conflicting attribute policy is not sufficient for security [6, 8]. A wrapping format should also be used to correctly bind key attributes to the key. This prevents attacks where the key is unwrapped twice with conflicting attributes. Some existing devices already include such wrapping formats; an example is the Eracom ProtectServer [9].

Secure templates. We limit the set of admissible attribute combinations for keys in order to avoid that they ever assume conflicting roles at creation time. This is configurable at the level of the specific PKCS#11 operation. For example, we can define different secure templates for different operations such as key generation and unwrapping.

A way to combine the first three patches with a wrapping format that binds attributes to keys in order to create a secure token has already been demonstrated [11]. Here we show how the fourth patch works, as it is an original idea for a configuration that has not yet appeared in the literature. This patch does not require any new cryptographic mechanisms to be added to the standard, making it quite simple and cheap to incorporate into existing devices. We consider here a set of templates with attributes sensitive and extractable always set. Other attributes wrap, unwrap, encrypt and decrypt are set as follows:

Key generation: we allow three possible templates:

1. wrap and unwrap, for exporting/importing other keys;
2. encrypt and decrypt, for cryptographic operations;
3. neither of the four attributes set, i.e. the default template if none of the above is specified.

Key creation/import: we allow two possible templates for any key created with `CreateObject` or imported with `Unwrap`:

1. unwrap,encrypt set and wrap,decrypt unset;
2. none of the four attributes set.

The templates for key generation are rather intuitive and correspond to a clear separation of key roles, which seems a sound basis for a secure configuration. The rationale behind the single template for key creation/import, however, is less obvious and might appear rather restrictive. The idea is to allow wrapping and unwrapping of keys while 'halving' the functionality of created/unwrapped keys: these latter keys can only be used to unwrap other keys or to encrypt data, wrapping and decrypting under such keys are forbidden. This, in a sense, offers an asymmetric usage of imported keys: to achieve full-duplex encrypted communication two devices will each have to wrap and send a freshly generated key to the other device. Once the keys are unwrapped and imported in the other devices they can be used to encrypt outgoing data in the two directions. Notice that imported keys can never be used to wrap sensitive keys. Note also that we require that all attributes are sticky on and off, and that we assume for bootstrapping that any two devices that may at some point wish to communicate have a shared long term symmetric key installed on them at personalisation time. This need only be used once in each direction. Our solution works well for pairwise communication, where the overhead is just one extra key, but would be more cumbersome for group key sharing applications.

We analysed the developed solution by extracting the model using Tookan. A model for SATMC was constructed using the abstraction option (see section 3.2). Given the resulting model, SATMC terminates with no attacks in a couple of seconds, allowing us to conclude the patch is safe in our abstract model for unbounded numbers of fresh keys and handles. Note that although no sensitive keys can be extracted by an intruder, there is of course no integrity check on the wrapped keys that are imported. Indeed, without having an encryption mode with an integrity check this would seem to be impossible. This means that one cannot be sure that a key imported on to the device really corresponds to a key held securely on the intended receipient's device. This limitation would have to be taken into account when evaluating the suitability of this configuration for an application. CryptokiX is available online[5].

6. CONCLUSION

We conclude by evaluating the state of commercial security tokens, the performance of Tookan, and lessons for future key management APIs.

The state of the art in PKCS#11 security tokens seems rather poor. In our sample of 17 devices, we found 5 tokens that trivially gave up their sensitive keys in complete disregard of the standard, 3 that were vulnerable to a variety of

[5] `http://secgroup.ext.dsi.unive.it/cryptokix`

key separation attacks, and a further smartcard that allowed unextractable keys to be read in breach of the standard. The remainder provide no functionality for secure transport of sensitive keys. We sent vulnerability reports to the manufacturers concerned at least 5 months before publication. Their responses can be viewed at the project website[6].

The tokens we have encountered so far have not provided much of a challenge for Tookan. At the start of the project, we hoped to encounter tokens that were patched in an effort to mitigate the attacks. Instead we found tokens with simple flaws or minimal functionality. Attacks were found on all the vulnerable tokens, usually in just a few seconds. The potential value of the tool is perhaps best indicated by the work in section 5, where we implement patches on a software token simulator obtaining a fully featured software prototype of a secure (at least in our model) token, capable of wrapping and unwrapping keys. The software token can be reverse-engineered accurately by our automated framework, indicating that Tookan is ready to analyse more sophisticated devices as soon as they become available on the market. Our software token might be useful as a reference to develop such next-generation devices. In future work we will be extending our model to more cryptographic detail. We would also like to try Tookan on PKCS#11 based devices currently outside our budgets, such as Hardware Security Modules (HSMs).

Finally, there are at least two new standards which address key management currently at the draft stage: IEEE 1619.3[7] (for secure storage) and OASIS Key Management Interoperability Protocol (KMIP)[8]. Although neither is aimed at cryptographic tokens, it is clear there is a move towards better standards for key management in general. Given the apparent difficulty of constructing a secure interface based on PKCS#11, this seems a timely intervention. Our conclusions based on the research in this paper are that the new standards should:

- Specify clearly what security properties an interface complying to the standard should uphold. Our experimental evidence suggests that the security goals in PKCS#11, i.e. protection of sensitive or unextractable keys, are apparently too well hidden for some implementers to notice. A clear set of security properties would make life substantially easier for application developers as well.

- Include a format for key wrapping that securely preserves key metadata (i.e. attributes etc.). This has already been noted by recent proposals for secure interfaces [4, 7].

- Treat explicitly the problem of key roles, and give guidance to avoid conflicting roles. Again this issue has been treated by recent proposals for APIs in the academic literature [4, 7].

- Make provision for compliance testing, to weed out poorly implemented tokens, and make testing results publicly available.

[6] `http://secgroup.ext.dsi.unive.it/pkcs11-security`
[7] `https://siswg.net`
[8] `http://www.oasis-open.org/committees/kmip/`

7. REFERENCES

[1] A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008. Software available at `http://www.ai-lab.it/satmc`. Currently developed under the AVANTSSAR project, `http://www.avantssar.eu`.

[2] L. Baloci and A. Vianello. Un sistema per lo studio della sicurezza. Baccalaureate Thesis, University of Venice, Italy, April 2010.

[3] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.

[4] C. Cachin and N. Chandran. A secure cryptographic token interface. In *Computer Security Foundations (CSF-22)*, pages 141–153, Long Island, New York, 2009. IEEE Computer Society Press.

[5] R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System - CHES 2002*, pages 579–592, 2002.

[6] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 411–425, 2003.

[7] V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In M. Backes and P. Ning, editors, *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *Lecture Notes in Computer Science*, pages 605–620, Saint Malo, France, Sept. 2009. Springer.

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

[9] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 2009. To appear.

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

[11] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In P. Degano and L. Viganò, editors, *Revised Selected Papaers of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, volume 5511 of *Lecture Notes in Computer Science*, pages 92–106, York, UK, Aug. 2009. Springer.

[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] openCryptoki. `http://sourceforge.net/projects/opencryptoki/`.

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