An Introduction to Security API Analysis *

Riccardo Focardi[1], Flaminia L. Luccio[1] and Graham Steel[2]

[1] DAIS, Università Ca' Foscari Venezia, Italy
{focardi,luccio}@dsi.unive.it
[2] LSV, ENS Cachan & CNRS & INRIA, France
graham.steel@lsv.ens-cachan.fr

1 Introduction

A security API is an *Application Program Interface* that allows untrusted code to access sensitive resources in a secure way. Examples of security APIs include the interface between the tamper-resistant chip on a smartcard (trusted) and the card reader (untrusted), the interface between a cryptographic *Hardware Security Module*, or HSM (trusted) and the client machine (untrusted), and the Google maps API (an interface between a server, trusted by Google, and the rest of the Internet).

The crucial aspect of a security API is that it is designed to enforce a policy, i.e. no matter what sequence of commands in the interface are called, and no matter what the parameters, certain security properties should continue to hold. This means that if the less trusted code turns out to be malicious (or just faulty), the carefully designed API should prevent compromise of critical data.

Designing such an interface is extremely tricky even for experts. A number of security flaws have been found in APIs in use in deployed systems in the last decade. In this tutorial paper, we introduce the subject of security API analysis using formal techniques. This approach has recently proved highly successful both in finding new flaws and verifying security properties of improved designs. We will introduce the main techniques, many of which have been adapted from language-based security and security protocol verification, by means of two case studies: cryptographic key management, and *Personal Identification Number* (PIN) processing in the cash machine network. We will give plenty of examples of API attacks, and highlight the areas where more research is needed.

2 Background

The first security vulnerability that may properly be called an 'API attack', and thus highlighted the security API as a critical design point, was discovered by Longley and Rigby in the early 1990s [41]. Their article showed how the logic programming language Prolog could be used to analyse a key management interface of a cryptographic device. Although the device was not identified at

* Work partially supported by the RAS Project "*TESLA: Techniques for Enforcing Security in Languages and Applications*".

the time, it later became known that it was an HSM manufactured by Eracom and used in the cash machine network. In 2000, Anderson published an attack on key loading procedures on another similar module manufactured by Visa [5], and the term 'security API' was coined by Bond and Anderson [12, 13] in two subsequent papers giving more attacks. Clayton and Bond showed how one of their more computationally intensive attacks could be implemented against a real IBM device using programmable FPGA hardware [21]. Independently from the Cambridge group, an MSc thesis by Clulow gave more examples of attacks, mostly specific to the PIN translation and verification commands offered by the API of Prism HSMs [22]. Clulow also published attacks on the industry standard for cryptographic key management APIs, RSA PKCS#11 [23].

Up until this point all the attacks had been discovered by manual analysis or by ad-hoc semi-formal techniques specific to the particular API under consideration. A first effort to apply more general formal tools, specifically the automatic first-order theorem prover Otter, was not especially successful, and the results remain unpublished (though they are available in a technical report [52]). The researchers were unable to discover any new attacks, and because the modelling lacked formal groundwork, when no attacks were found they were unable to conclude anything about the security of the device. One member of the team later remarked that "It ended up being more about how to use the tools than about analysing the device." [35].

Meanwhile, the formal analysis of protocols for e.g. cryptographic key exchange and authentication had become a mature field. A particularly successful approach had centred around the so-called Dolev-Yao (DY) abstract model, where bitstrings are modelled as terms in an abstract algebra, and cryptographic functions are functions on these terms [27]. Together with suitable abstractions, lazy evaluation rules and other heuristics, this model has proved highly amenable to automated analysis, by model checking or theorem proving techniques [8, 11, 28, 42]. Modern automated tools can check secrecy and authentication properties of (abstract models of) widely-used protocols such as Kerberos and TLS in a few seconds.

The idea of applying protocol analysis techniques to the analysis of security APIs seemed very attractive. However, initial experiments showed that existing tools were not suitable for the problem [14, 38] for a number of reasons: In particular many of the attacks pertinent to security APIs are outside the scope of the normal DY model. For example, they might involve an attacker learning a secret value, such as a cash machine PIN, by observing error messages returned by the API (a so-called error oracle attack). Furthermore, the functionality of security APIs typically depends on global mutable state which may loop, a feature which invalidates many abstractions and optimisations made by protocol analysis tools, particularly when freshly generated nonces and keys are considered. There are also problems of scale - a protocol might describe an exchange of 5 messages between two participants, while an API will typically offer dozens of commands.

In the sections that follow, we will show how formal analysis techniques have been adapted to security API problems. The aim of this paper is to serve as an introduction to the field. As such, the technical material is introduced by means of two case studies. The first is based around the RSA standard for cryptographic interfaces PKCS#11, while the second concerns commands for PIN processing using HSMs in the cash machine network. We follow these two case studies with a discussion of open problems.

3 Key Management with RSA PKCS#11

Cryptographic key management, i.e. the secure creation, storage, backup, use and destruction of keys has long been identified as a major challenge in applied cryptography. Indeed, Schneier calls it "the hardest part of cryptography and often the Achilles' heel of an otherwise secure system." [48]. In real-world applications, key management often involves the use of HSMs or cryptographic tokens, since these are considered easier to secure than commodity hardware, and indeed are mandated by standards in certain sectors [37]. There is also a growing trend towards enterprise-wide schemes based around key management servers offering cryptographic services over open networks [18]. All these solutions aim to enforce security by dividing the system into trusted parts (HSM, server) and untrusted parts (host computer, the rest of the network). The trusted part makes cryptographic functions available via an API. Hence we arrive at our first security API problem: how to design an interface for a key management device so that we can create, delete, import and export keys from the device, as well as permitting their use for encryption, decryption, signature and verification, all so that if the device comes into contact with a malicious application we can be sure the keys stay secure. This is far from trivial, as well will see from our case study of the most widely used standard for such interfaces, RSA PKCS#11.

3.1 The PKCS#11 standard

RSA Public Key Cryptography Standards (PKCS) aim to standardise various aspects of cryptography to promote interoperability and security. PKCS#1, for example, defines the RSA asymmetric encryption and signing algorithm. PKCS#11 describes the 'Cryptoki' API for cryptographic hardware. Version 1.0 was published in 1995. The latest official version is v2.20 (2004) which runs to just under 400 pages [47]. Adoption of the standard is almost ubiquitous in commercial cryptographic tokens and smartcards, even if other interfaces are frequently offered in addition.

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 [47,

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 new 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, cannot "compromise keys marked 'sensitive', since a key that is sensitive will always remain sensitive", [47, 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.

3.2 The Wrap-Decrypt Attack

Clulow first published attacks on PKCS#11 based APIs in 2003 [23], where he gave many examples of ways in which keys with the sensitive attribute set to true could be read in clear outside the device. The most straightforward of these is the 'key separation' attack, 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.

To determine the value of a sensitive key, the attacker simply wraps it and then decrypts it, as shown in Fig. 1. The attack is described with a notation for PKCS#11 based APIs briefly defined in the following and more formally in the next section: $\mathsf{h}(\mathsf{n}_1, \mathsf{k}_1)$ is a predicate stating that there is a handle encoded by 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

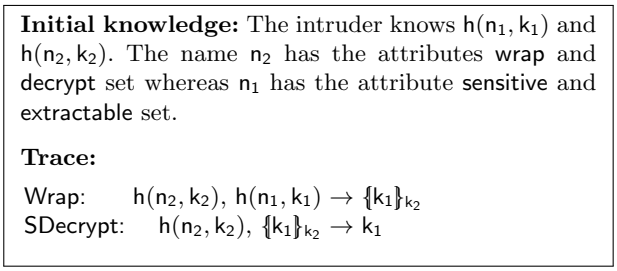

Fig. 1. Wrap/Decrypt attack

the decrypt command, it has no way of telling that the packet it is decrypting contains a key.

It might appear easy to prevent such an attack, but as we shall see, it is in fact rather difficulty within the confines of PKCS#11. Before treating this in detail, we will introduce our language for formal modelling of the API.

3.3 Formal Model

Our model follows the approach used by Delaune, Kremer and Steel (DKS) [25]. The device is assumed to be 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 [27]. Bitstrings are modelled as terms in an abstract algebra. The rules of the API and the abilities of an attacker are written as deduction rules in the algebra.

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:

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

We also consider a finite set $\mathcal{F}$ 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 $\mathsf{wrap}(\mathsf{n}, \top)$.

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 1. The following rule models wrapping:

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

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 $\mathsf{wrap}(\mathsf{x}_1, \top)$, and key y_2 is extractable, written $\mathsf{extract}(\mathsf{x}_2, \top)$, then we can wrap y_2 with y_1, creating $\{\!|\mathsf{y}_2|\!\}_{\mathsf{y}_1}$.

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 (he can only learn more things), but the global state is non-monotonic (attributes may be set on and then off). For a formal semantics, we refer to the literature [25]. We now present in Fig. 2 a subset of PKCS#11 commands sufficient for some basic symmetric key management commands (the asymmetric key has also been treated in the literature [26]). This will suffice to demonstrate the modelling technique and some attacks.

3.4 Using the Formal Model

We first add some rules to the model for the intruder that allow him to encrypt and decrypt using his own keys (see Fig. 3). 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 $\mathsf{sensitive}$ attribute set to true, or the $\mathsf{extractable}$ attribute set to false. We give the intruder some initial knowledge, typically just some key $\mathsf{k_i}$ that is not loaded on the device,

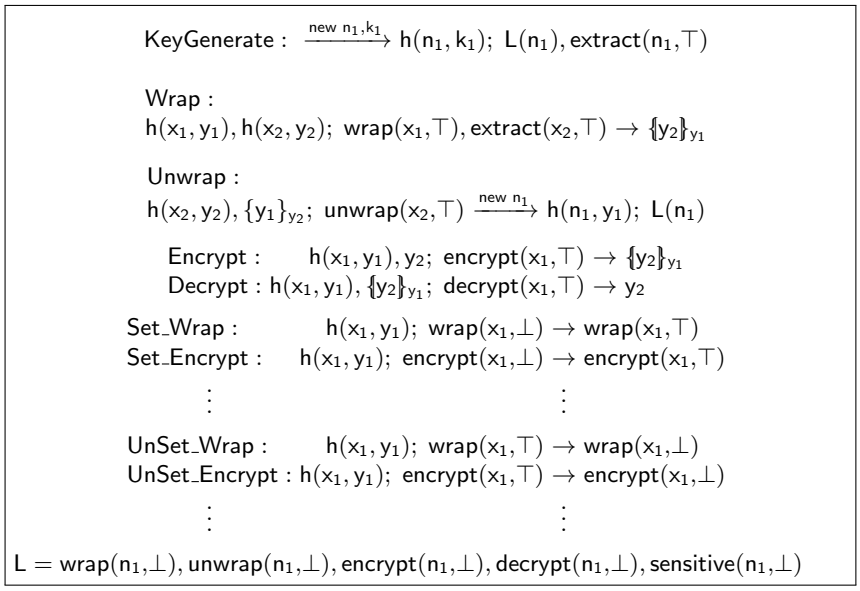

Fig. 2. PKCS#11 Symmetric Key Fragment.

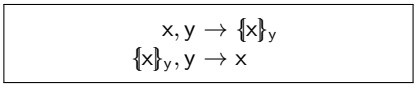

Fig. 3. Intruder rules for symmetric key cryptography.

and then use a tool such as a model checker to search the model for a chain of commands and intruder steps that leads to an attack state.

Unfortunately we immediately encounter both theoretical and practical problems. First, we know that the reachability problem in general for languages like this is undecidable: even with fixed message size there is a reduction to the Post correspondence problem [29, table 12, page 298]. Even so we might hope to find some attacks in practice. But in fact the model checker is quickly swamped by the combinatorial possibilities, many of which at first sight seem unlikely to lead to an attack. For example, the intruder can take an already encrypted term $\{\!|k_1|\!\}_{k_2}$, and use it as input to the encrypt command along with some handle $h(n, k_3)$ to obtain $\{\!|\{\!|k_1|\!\}_{k_2}|\!\}_{k_3}$. Unfortunately one cannot in general just delete these terms from the intruder knowledge: they may be a necessary step for some fiendish attack. Fortunately we can address both the the theoretical and practical problems at once, by means of a *well-modedness* result [26]. There it is shown that each function symbol such as $h(.,.)$, $\{\!|.|\!\}_.$ can be given a unique interpretation in terms of *modes*. For example, $h(.,.)$ has mode $\mathsf{Nonce} \times \mathsf{Key} \rightarrow \mathsf{Handle}$. We assign to each constant symbol a mode. A term is well-moded if all the function symbols

in it are applied to symbols such that the modes are respected. Furthermore, any reachability query that can be expressed with well-moded terms is satisfiable if and only if it it is reachable by a sequence of steps where the intruder learns only well-moded terms. This allows us to prune the search space dramatically, since any branch that results in the creation of an ill-moded term can be ignored. If the function symbols can be moded acyclically, we can also show decidability provided we bound fresh handles and keys. Consult the paper for details and proofs [26].

3.5 A Suite of Attacks

Equipped with a suitable formal model and a model checker, we can attempt to find secure configurations of the standard. Clulow's first suggestion for preventing the attack in Fig. 1 is to prevent attribute changing operations 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 a list of 'sticky' attributes which once set, may not be unset, or the attack is not prevented, [51]. Effectively this means the unset rules will be omitted from the model for these attributes. Having applied these measures, we discover the attack given in Fig. 4. 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:

$$
\begin{array}{lrcl}
\textsf{Encrypt:} & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \mathsf{k}_3 & \rightarrow & \{\!|\mathsf{k}_3|\!\}_{\mathsf{k}_2} \\
\textsf{Unwrap:} & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2),\ \{\!|\mathsf{k}_3|\!\}_{\mathsf{k}_2} & \xrightarrow{\text{new } \mathsf{n}_3} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) \\
\textsf{Set\_wrap:} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) & \rightarrow & \mathsf{wrap}(\mathsf{n}_3, \top) \\
\textsf{Wrap:} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3),\ \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) & \rightarrow & \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_3} \\
\textsf{Intruder:} & \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_3},\ \mathsf{k}_3 & \rightarrow & \mathsf{k}_1
\end{array}
$$

Fig. 4. Attack using encrypt and unwrap.

To prevent the attack shown in Fig. 4, we add encrypt and unwrap to the list of conflicting attribute pairs. Another new attack is discovered (see Fig. 5) of a type discussed by Clulow, [23, Section 2.3]. Here the key k_2 is first wrapped under k_2 itself, and then unwrapped, gaining a new handle $\mathsf{h}(\mathsf{n}_4, \mathsf{k}_2)$. The intruder then wraps k_1 under k_2, and sets the decrypt attribute on handle $\mathsf{h}(\mathsf{n}_4, \mathsf{k}_2)$, allowing him to obtain k_1.

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.

Trace:

$$
\begin{array}{lrcl}
\mathsf{Set\_wrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2) & \to & \mathsf{wrap}(\mathsf{n}_2, \top) \\
\mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2) & \to & \{\!|\mathsf{k}_2|\!\}_{\mathsf{k}_2} \\
\mathsf{Set\_unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2) & \to & \mathsf{unwrap}(\mathsf{n}_2, \top) \\
\mathsf{Unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \{\!|\mathsf{k}_2|\!\}_{\mathsf{k}_2} & \xrightarrow{\text{new } \mathsf{n}_4} & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_2) \\
\mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) & \to & \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} \\
\mathsf{Set\_decrypt}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_2) & \to & \mathsf{decrypt}(\mathsf{n}_4, \top) \\
\mathsf{Decrypt}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_2), \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} & \to & \mathsf{k}_1
\end{array}
$$

Fig. 5. Re-import attack 1.

One can attempt to prevent the attack in Fig. 5 by adding wrap and unwrap to our list of conflicting attribute pairs. Now in addition to the initial knowledge from the first three experiments, we give the intruder an unknown key k3 encrypted under k2. Again he is able to affect an attack similar to the one above, this time by unwrapping $\{\!|\mathsf{k}_3|\!\}_{\mathsf{k}_2}$ twice (see Fig. 6).

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 $\{\!|\mathsf{k}_3|\!\}_{\mathsf{k}_2}$.

Trace:

$$
\begin{array}{lrcl}
\mathsf{Set\_unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2) & \to & \mathsf{unwrap}(\mathsf{n}_2, \top) \\
\mathsf{Unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \{\!|\mathsf{k}_3|\!\}_{\mathsf{k}_2} & \xrightarrow{\text{new } \mathsf{n}_3} & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) \\
\mathsf{Unwrap}: & \mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \{\!|\mathsf{k}_3|\!\}_{\mathsf{k}_2} & \xrightarrow{\text{new } \mathsf{n}_4} & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3) \\
\mathsf{Set\_wrap}: & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3) & \to & \mathsf{wrap}(\mathsf{n}_3, \top) \\
\mathsf{Wrap}: & \mathsf{h}(\mathsf{n}_3, \mathsf{k}_3), \mathsf{h}(\mathsf{n}_1, \mathsf{k}_1) & \to & \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_3} \\
\mathsf{Set\_decrypt}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3) & \to & \mathsf{decrypt}(\mathsf{n}_4, \top) \\
\mathsf{Decrypt}: & \mathsf{h}(\mathsf{n}_4, \mathsf{k}_3), \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_3} & \to & \mathsf{k}_1
\end{array}
$$

Fig. 6. Re-import attack 2.

This sample of the attacks found show how difficult PKCS#11 is to configure in a safe way, and indeed there are several more attacks documented in the literature [17, 23, 26, 33] and perhaps more to discover. Another line of research has consisted of trying to propose secure subsets of the API together with a suitable security proof. An obstacle here is the fresh generation of keys and handles: if there are no attacks in the model after generating n fresh keys, how do we know there are no attacks after generating $n + 1$? To address this prob-

Fig. 7. Tookan system diagram

lem, Fröschle and Steel proposed abstractions for handles and keys that allow security proofs for unbounded fresh data [33]. In particular, they showed the security of a symmetric key management subset based around the proprietary extensions to PKCS#11 made by Eracom, where keyed hashes are used to bind attributes to keys under wrapping, ensuring that they are re-imported with the same attributes.

3.6 Finding Attacks on Real Devices

Attacks on the standard are interesting in themselves, but in reality every device implements a different subset of PKCS#11 with different restrictions on the use of each command. How can one know whether a particular device is vulnerable? To address this, Bortolozzo, Centenaro, Focardi and Steel developed the Tookan[3] tool [17]. Tookan functions as shown in Fig. 7. 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. Tookan uses this information to generate a model the language described above (2), which is encoded for input to the SATMC model checker [9]. If SATMC finds an attack, the attack trace (3) is sent to Tookan for testing directly on the token (4).

Changes to the model There are several differences between Tookan's model and the model of section 3.3. One is that Tookan takes into account *key templates*. In section 3.3, the key generation commands create a key with all attributes unset (see Fig. 2). Attributes are then 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.

[3] Tool for cryptoki analysis.

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 tested attempt to impose some restrictions on the combinations of attributes that can be set on a key and how these may be changed. There are 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 [47, Table 15]: sensitive for secret keys, for example. 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. 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. 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. 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.

Limitations of Reverse Engineering Tookan's 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

	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	SEG		✓		✓								
	MXI Security	Stealth MXP Bio	✓	✓		✓								
	RSA	SecurID 800	✓	✓	✓	✓					✓	✓	✓	a3
	SafeNet	iKey 2032	✓	✓	✓		✓							
	Sata	DKey	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	a3
Card	ACS	ACOS5	✓	✓	✓	✓								
	Athena	ASE Smartcard	✓	✓	✓									
	Gemalto	Cyberflex V2	✓	✓	✓		✓	✓		✓				a2
	Gemalto	Classic TPC IS V1		✓		✓								
	Gemalto	Classic TPC IS V2	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	a3
	Siemens	CardOS V4.3 B	✓	✓	✓		✓					✓		a4
Soft	Eracom	HSM simulator	✓	✓		✓	✓	✓	✓	✓		✓		a1
	IBM	opencryptoki 2.3.1	✓	✓	✓	✓	✓	✓	✓	✓		✓		a1

Table 1. Summary of results on devices.

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.

3.7 Results

Table 1 summarises the results obtained by Tookan on a number of devices as well as two software simulators.

Implemented functionality Columns 'sym' and 'asym' respectively indicate whether or not symmetric and asymmetric key cryptography are supported. Column 'cobj' refers to the possibility of inserting external, unencrypted, keys

	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 2. Key for table 1.

on the device via `C_CreateObject` PKCS#11 function. This is allowed by almost all of the analysed tokens, although it wasn't included in the original model of the standard used by Delaune, Kremer and Steel [26]. The next column, 'chan', refers to the possibility of changing key attributes through `C_SetAttributeValue`. The following two columns, 'w' and 'ws', respectively indicate whether the token permits wrapping of nonsensitive and sensitive keys.

Attacks Attack a1 is a wrap/decrypt attack as discussed in section 3.2. 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 3.3:

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

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:

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

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 several vulnerable tokens. Despite its apparent simplicity, this attack had not appeared before in the PKCS#11 security literature [23, 25].

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 $\mathsf{priv}(\mathsf{z})$:

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

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.

Model-checking results Column 'mc' reports which of the attacks was 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.

3.8 Finding Secure Configurations

As we observed 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. In this section, we present CryptokiX, a software implementation of a Cryptoki token which can be fiXed by selectively enabling different patches. The starting point is openCryptoki [45], an open-source PKCS#11 implementation for Linux including a software token for testing. As shown in Table 1, 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. CryptokiX extends 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 [23, 25]. 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.

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 [33] and discussed in section 3.5. One can also use the fourth patch to produce a secure configuration that 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.

The developed solution has been implemented and analysed by extracting a model using Tookan. A model for SATMC was constructed using the abstractions described above (see section 3.5). 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 recipient'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[4].

3.9 Summary

We have seen how RSA PKCS#11 describes an API for key management where the usage policy for each key is described by a set of attributes. We have seen how this interface, if not configured carefully, can be vulnerable to a variety of attacks, and we have seen that these vulnerabilities affect not just theoretical models but real devices. We have also seen how to use formal modelling techniques to systematically find attacks and then verify the security of models. However there are still many open questions about designs of a secure API and cryptographic soundness of models, for example, which we will discuss in section 5.

4 PIN Processing

We now consider our second case study, which addresses the problem of protecting a user's PIN when withdrawing some money at an *Automated Teller Machine* (ATM). International bank networks are structured in such a way that an access to an ATM implies that the user's PIN is sent from the ATM to the issuing bank for the verification. While travelling, the PIN is decrypted and re-encrypted by special tamper-resistant HSMs which are placed on each network switch, as illustrated in Fig. 8. Indeed, international standards mandate the use of these devices to keep PINs secure [37]. The first PIN encryption is performed by the ATM keypad which is an HSM itself, using a symmetric key k_1 shared with the neighbour acquiring bank. While travelling from node to node, the encrypted PIN is decrypted and re-encrypted by the HSM located in the switch with another key shared with the destination node. This is done using a so called *translation* API. The final verification and acceptance/refusal of the PIN is done by the issuing bank. This check is performed via a *verification* API.

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

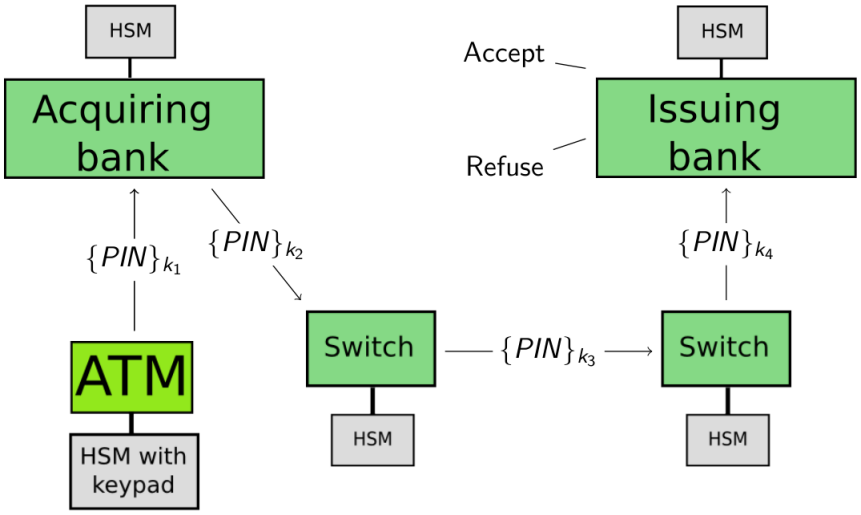

Fig. 8. Bank network.

At first sight this setup seems highly secure. However, several API-level attacks have been discovered on these HSMs in recent years [10, 15, 22]. These attacks work by assuming that the attacker is an insider who has gained access to the HSM at some bank switch, a threat considered in-scope by the standard [37]. The attacker performs some unexpected sequence of API calls from which he is able to deduce the value of a PIN. API-level attacks on PINs have recently attracted attention from the media [1, 3]. This has increased the interest in developing formal methods for analysing PIN recovery attacks. We will briefly survey the literature here.

Examining the PIN verification API, Bond and Zielinski [15] and Clulow [22] discovered the so called *decimalization* attack that we will discuss in section 4.1 at more or less the same time. The original version of the attack requires an average of 16.5 API calls to obtain a 4-digit PIN [15]. Later variations proposed by Steel [49] and Focardi and Luccio [31] reduced that value to 16.145 and 14.47 respectively. The decimalisation table attack is similar to the Mastermind game, and in section 4.2, we will illustrate how to reuse known techniques to solve Mastermind in order to find efficient attacks on bank PINs. Bond and Clulow have also demonstrated an attack that works even if the decimalisation table is fixed [16], although it requires many more calls to the API (some 10s of thousands).

Attacks on the PIN translation API have been presented by Clulow [22] and subsequently by Berkman and Ostrovsky [10]. These attacks often exploit details of the formats used to pad a PIN into a 64 bit block for encryption. Berkman and Ostrovsky's attacks also exploit commands for changing a customer PIN,

```
PIN_V(EPB, len, offset, vdata, dectab) {
    x1 := enc_pdk(vdata);
    x2 := left(len, x1);
    x3 := decimalize(dectab, x2);
    x4 := sum_mod10(x3, offset);
    x5 := dec_k(EPB);
    x6 := fcheck(x5);
    if (x6 = ⊥) then return("format wrong");
    if (x4 = x6) then return("PIN correct");
                 else return("PIN wrong")}
```

Table 3. The *verification* API.

though they generally require a known plaintext-ciphertext pair (i.e. a clear PIN and an encryption of the same PIN) to be introduced into the system.

The first formal analysis work on PIN recovery attacks is due to Steel [49], who showed hos a combination of logic programming and probabilistic model checking could be used to find new attacks and analyse possible patches to the APIs. Recently Centenaro and the authors of this survey presented a language-based setting for analysing PIN processing API via a type-system [20]. We have formally modelled existing attacks, proposed some fixes and proved them correct via type-checking. These fixes typically require to reduce and modify the HSM functionality by, e.g., deciding on a single format of the transmitted PIN or adding MACs for the integrity of user data. This will be the subject of section 4.3. As upgrading the bank network HSMs worldwide is complex and very expensive in [30] the authors have proposed a method for the secure upgrade of HSMs in wide networked systems. This method incrementally upgrades the network so to obtain upgraded, secure subnets, while preserving the compatibility towards the legacy system. Finally, in [32] we have also have proposed a low-impact, easily implementable fix that involves very little change to the existing ATM network infrastructure. Attacks are still possible, but they are 50000 times slower.

4.1 API-level attacks on PIN verification

In this section we show in detail a real API-level attack on cash machine PINs. We focus our attention on the PIN verification API (PIN_V for short) and we specify its code in table 3. It takes as input the encrypted PIN block *EPB*, i.e. the encrypted PIN arriving from the ATM, the PIN length *len* and offset *offset*, the validation data *vdata* and the decimalization table *dectab*. The API returns the result of the verification or an error code. PIN_V behaves as follows:

- The user PIN of length *len* is computed by first encrypting validation data *vdata* with the PIN derivation key *pdk* (x_1) and obtaining a 16 hexadecimal digit string. Then, the first *len* hexadecimal digits are chosen (x_2), and

decimalized through *dectab* (x_3), obtaining the "natural" PIN assigned by the issuing bank to the user. decimalize is a function that associates to each possible hexadecimal digit (of its second input) a decimal one as specified by its first parameter (*dectab*). Finally, if the user wants to choose her own PIN, an *offset* is calculated by digit-wise subtracting (modulo 10) the natural PIN from the user-selected one (x_4).

- To recover the trial PIN *EPB* is first decrypted with key k (x_5), then the PIN is extracted by the formatted decrypted message (x_6). This last operation depends on the specific PIN format adopted by the bank. In some cases, for example, the PIN is padded with random digits so to make its encryption immune from codebook attacks. In this case, extracting the PIN involves removing this random padding.
- If the format in incorrect ($\perp$ represents failure) then an error message is returned. Otherwise, the equality between the user PIN and the trial PIN is verified.

Example 2. For simplicity we consider the 'default' decimalisation table:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
```

We will write it as $dectab = 0123456789012345$. Let also EPB $= \{|5997, r|\}_k$, len=4, $offset$ =4732 and $x_1 = \mathsf{enc}_{pdk}(vdata)$ = BC6595FDE32BA101 then:

$$\begin{array}{|l|l|}
x_2 = \mathsf{left}(4, BC6595FDE32BA101) & = \text{BC65} \\
x_3 = \mathsf{decimalize}(dectab, BC65) & = 1265 \\
x_4 = \mathsf{sum_mod10}(1265, 4732) & = 5997 \\
x_5 = \mathsf{dec}_k(\{|5997, r|\}_k) & = (5997, r) \\
x_6 = \mathsf{fcheck}(5997, \mathsf{r}) & = 5997 \\
x_6 \neq \perp & \\
x_4 = x_6 & \mathsf{return}(''PIN\ \ correct'')
\end{array}$$

Since x_6 is different from $\perp$ and $x_4 = x_6$, the API returns $''PIN\ \ correct''$.

The decimalization attack We now illustrate a real attack on PIN_V first reported in [15, 22], called the decimalization table attack. It consists of deducing the PIN digits by modifying the way numbers are decimalized and by observing whether this affects the result of PIN_V. The position of the guessed PIN digits is reconstructed by manipulating the offset, which is a public parameter. By combining all this information the attacker is able to reconstruct the whole PIN.

More specifically, the attack works by iterating the following two steps, until the whole PIN is recovered:

1. To discover whether or not a decimal digit d is present in the intermediate value contained in x_3 the intruder picks digit d, changes the *dectab* function so that values previously mapped to d now map to $d + 1$ mod 10, and then checks whether the system still returns $''PIN\ \ correct''$. If this is the case digit d does not appear in x_3.

2. It the API call returns *"PIN wrong"* the attacker discovers that d is one of the digits of x_3. To locate the position of the digit the intruder also decreases the *offset* digits by 1, position by position, until the API returns again that the PIN is correct.

We illustrate the attack through a simple example.

Example 3. Consider again Example 2. The attacker, unaware of the value of the PIN, first tries to discover whether or not 0 appears in x_3, so it changes the dectab as

$$dectab' = \underline{1}123456789\underline{1}12345$$

i.e., he replaces the two 0's by 1's. Invoking the API with $dectab'$ he obtains

$$\mathsf{decimalize}(dectab', BC65) = \mathsf{decimalize}(dectab, BC65) = 1265$$

that is x_3 remains unchanged and so the attacker deduces 0 does not appear in the PIN. The attacker proceeds by replacing the 1's of *dectab* by 2's:

$$dectab'' = 0\underline{2}23456789\underline{0}\underline{2}2345$$

which gives

$$\begin{array}{|ll|}
x_3 = \mathsf{decimalize}(dectab'', BC65) & = 2265 \neq 1265 \\
x_4 = \mathsf{sum_mod10}(2265, 4732) & = 6997 \neq 5997 \\
x_5 = \mathsf{dec}_k(\{\lfloor 5997, r \rfloor\}_k) & = (5997, r) \\
x_6 = \mathsf{fcheck}(5997, \mathsf{r}) & = 5997 \\
x_6 \neq \perp & \\
x_4 \neq x_6 & \mathsf{return}(\textit{"PIN wrong"})
\end{array}$$

The intruder now knows that digit 1 occurs in x_3. To discover its position and multiplicity, he now varies the offset so as to "compensate" for the modification of the *dectab*. In particular, he tries to decrement each offset digit by 1. For example, testing the position of one occurrence of one digit amounts to trying the following offset variations:

$$\underline{3}732, 4\underline{6}32, 47\underline{2}2, 473\underline{1}$$

Notice that, in this specific case, offset value 3732 makes the API return again *"PIN correct"*.

The attacker now knows that the first digit of x_3 is 1. Given that the *offset* is public, he also calculates the first digit of the user PIN as $1 + 4 \bmod 10 = 5$. He then proceeds using the same technique to discover the other PIN digits.

4.2 PIN attacks as a Mastermind game

As observed in [14, 31], the above attack resembles the Mastermind game, invented in 1970 by the Israeli postmaster and telecommunication expert Mordecai Meirowitz [46, page 442]. The game is shown in Fig. 9 which is taken from [2]. It is

Fig. 9. Guessing a code of a Mastermind game is similar to guessing a PIN.

played as a board game between two players, a *codebreaker* and a *codemaker* [50]. The codemaker chooses a linear sequence of coloured pegs and conceals them behind a screen. Duplicates of coloured pegs are allowed. The codebreaker has to guess, in different trials, both the colour and the position of the pegs. During each trial he learns something and based on this he decides the next guess: in particular, a response consisting of a *red marker* represents a correct guess of the colour and the position of a peg (but the marker does not indicate which one is correct), a response consisting of a *white marker* represents only the correct guess of a colour but at the wrong position.

The analogy between the game and the attack is as follows: each API call represents a trial of the codebreaker and the API return value is the corresponding answer of the codemaker. Modifying the dectab corresponds to disclosing the presence of certain digits in the PIN, like the white marker in the Mastermind Game. On the other hand, manipulating the dectab and the offset together is similar to asking the codemaker to confirm a guess of both the value and the position of one PIN digit, like the red marker of the game.

An extended Mastermind game To make the above analogy more precise, it is in fact necessary to extend the Mastermind game so as to allow the codebreaker to pose a guess of *sets* of coloured pegs, instead of just single pegs. Intuitively, the sets represent alternative guesses, i.e., it is sufficient that one of the pegs in the set is correct to get a red or a white marker. E.g., given the secret $(1, 5, 3, 1)$ and the (extended) guess $(\{1\}, \{3, 4, 5\}, \{1, 2\}, \{0\})$, then the result is two red markers for the first two positions of the guess and one white marker for the third position of the guess. The correspondence between the values of the secret and the values in the guess are shown as underlined values in $(\{\underline{1}\}, \{3, 4, \underline{5}\}, \{\underline{1}, 2\}, \{0\})$.

In [31] it is proved that the decimalization attack is equivalent to playing an *extended* Mastermind game, only focusing on answers containing 4 red markers. Intuitively, given a modified *dectab* and *offset*, we construct the sets for the extended guess by picking all digits whose modification in the offset are correctly compensated by the variation in the dectab. The first set corresponds to the first offset digit, the second set to the second digit and so on. For example, consider an original offset $(1, 5, 4, 2)$ modified as $(0, 3, 1, 0)$, giving a variation of $(-1, -2, -3, -2)$. Suppose now that the modified dectab turns the mapping of digit 4 into 5, with a variation of $+1$, then digit 4 will be in the first set of the extended guess, since the offset has the correct (opposite) variation -1 in the fist position.

As an example, consider the following default *dectab* and a modified one, named *dectab′*:

dectab	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
dectab′	0	2	4	3	5	8	6	9	1	9	0	2	4	3	5	8
variation		+1	+2		+1	+3		+2	+3			+1	+2		+1	+3

Since the variation on the first offset digit is -1 we collect, in the first set of the guess, all digits with variation $+1$ in the dectab, i.e., $\{1, 4\}$. For the second and fourth offset digits, both with variation -2, we obtain set $\{2, 7\}$. Finally, for the third offset digit, with variation -3, we have set $\{5, 8\}$. The resulting extended guess is $(\{1, 4\}, \{2, 7\}, \{5, 8\}, \{2, 7\})$.

Let us now see how the answer to this guess in the Mastermind game relates to the result of an API call with the above modified offset and dectab. Recall that we start from a dectab and offset that give *"PIN correct"*. Thus, we will still have *"PIN correct"* only if the variations in the dectab are correctly compensated for by the ones in the offset. For example, if digit 1 originally occurred in the first place after decimalization (x_3), with *dectab′* it will now be a 2. Summing 1 with the first digit of the original offset we had $1 + 1 = 2$. With the modified offset we now have $2 + 0 = 2$. In fact the variation $+1$ in the dectab is correctly compensated for by the offset. Notice that 1 occurs in the first set of the guess.

Observe also that 1 does not occur in the other sets of the guess. Thus, if 1 originally occurred in the second place after decimalization it would still be turned into 2 by *dectab′* but the change in the offset this time would not compensate for this variation. In fact, with the original dectab and offset we would get $1 + 5 = 6$ (where 5 is the second value of the offset) while with the modified ones we would get $2 + 3 = 5$ (where 3 is the second value of the new offset). Thus, the modification of $+1$ in the dectab this time would not correspond to the variation -2 in the second offset digit.

By iterating this reasoning on all digits, we obtain that the API call gives *"PIN correct"* if and only if the extended guess in the Mastermind game gives 4 red markers, meaning that the four secret digits, correct after decimalization and before summing the offset, are in the four sets of the extended guess. This correspondence has been formally proved [31].

```
do_guess(S):
  if |S| != 1:
    # not a single solution yet

    min = |S|   # starting value, we want to decrease it

    # depending on the round and the number of surviving solutions,
    # generates the guesses (heuristics here ...)
    guesses = generate_guesses(S)

    for g in guesses: # for each guess g

      # intersect S and surviving solutions for g
      M_SOLS= S & surviving(g)

      # count matching and non-matching and take the max
      n_sol = max(|m_sols|,|S|-|m_sols|)

      # if we got a minimum, we store it
      if mas < min:
          min = n_sol
          MIN_SOLS = M_SOLS

    if min < |S|:
      # we are decreasing the surviving solution set size:
      # let us perform the guess

      # case1: guess was right, explore solutions in MIN_SOLS
      do_guess(MIN_SOLS)

      # case2: guess was wrong, explore solutions in S - MIN_SOLS
      do_guess(S - MIN_SOLS)

    else:
      FAIL         # we are looping
```

Table 4. An algorithm for finding PIN cracking strategies.

Improving attack strategies by playing Mastermind Now that we have shown a strict analogy between PIN cracking and an extension of the Mastermind game, we can find efficient attacks for recovering PINs by simply devising minimal strategies for winning the Mastermind game. The solution proposed by Knuth [39] is to build a balanced search tree by minimizing the maximum number of solutions surviving after a guess. A guess, in our case, gives a Boolean answer which partitions the set of possible solutions into two disjoint sets: matching or non-matching ones. The guess that is more balanced, i.e., that minimizes the maximum of the two sets, is picked and the procedure is recursively applied until it reaches a unique solution. This method does not guarantee an optimal solution as an unbalanced guess might produce a more balanced subtree later on, and vice versa, but it has been shown to work well in practice [31, 39].

In table 4, we report an algorithm inspired by the above idea, adapted to our setting. The recursive function `do_guess` is initially invoked with the whole $\mathcal{S}$ set of possible secrets. It then finds a guess that minimizes the maximum number of surviving solutions on the two possible answers and it descends recursively into the two sets in which the actual set is partitioned. A critical issue is how the set

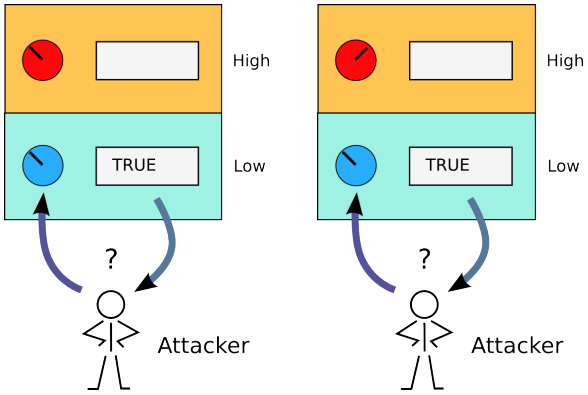

Fig. 10. Noninterference.

of candidate guesses are picked (function `generate_guesses(S)`). In our setting, considering all extended guesses consisting of all possible combinations of digit subsets for each position is computationally intractable. In [31] it is observed that the 'shape' of the guesses used in the strategies found is quite regular. This makes it possible to devise a few heuristics that incrementally extend the set of guesses when needed. As a result, the existing bounds on the average number of API calls required to recover a PIN has been reduced to 14.47 and a new bound of 19.3 has been given for 5-digits PINs. Both cases are close to the optimum, i.e., the depth of a perfectly balanced search tree.

4.3 Devising robust PIN management APIs

In the previous sections, we described one of the known attacks on PIN management APIs and its similarity to the Mastermind game. We now focus on remedies and we first try to understand what is the source of the attacks, i.e., what security property is missing. This will allow us to propose formal models and tools supporting a more 'robust' development of PIN processing APIs.

Preventing information flow Since the PIN recovery attacks involve the leakage of a secret value, a first attempt to understand the problem might be in terms of classical *noninterference* [34]. In this setting, data items are labelled with security levels taken from a lattice. Noninterference requires that data from a higher level, e.g. the secret level, should never affect the value of data with a lower level, e.g. the public level. The idea is depicted in Fig. 10. The system is (highly) simplified as a box containing two smaller boxes, with a switch for the input and a display for the output. The top box (high level input/output) contains the secret information while the bottom box (low level input/output) is public and possibly under the control of the attacker. Noninterference holds if a variation on the high input does not produce any change in the low level output.

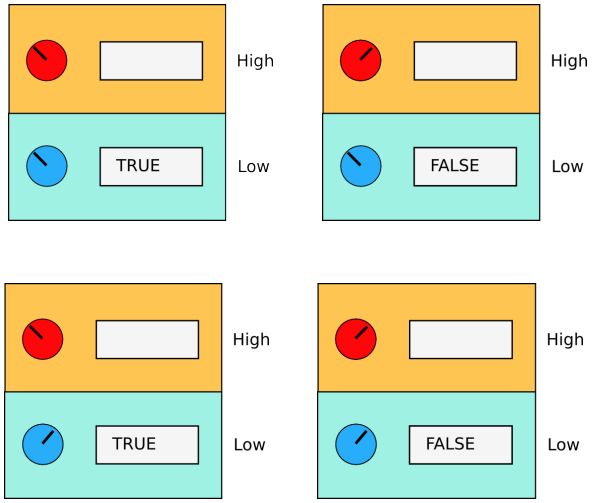

Fig. 11. Robust declassification.

The noninterference property is very strong and ensures that no secret information will ever be leaked, but this is too much for our setting. In fact, PIN verification needs to leak 1 bit of information about the correctness of the PIN. To see how the property is broken, it is enough to consider an API call with the correct encrypted PIN block (giving *"PIN correct"*), and another API call with a wrong encrypted PIN block (giving *"PIN wrong"*). This for example happens if the user types the wrong PIN at the ATM. By changing the secret data, i.e., the encrypted PIN, we have two different outcomes of the API, thus two different low level outputs. Clearly a leak of information about the correctness of the PIN is inevitable, but we would somehow like to limit this to prevent attacks.

Admitting controlled information leakage One approach to relaxing noninterference while still controlling information release is called *robustness*, and was proposed by Myers, Sabelfeld and Zdancewic (MSZ) [43]. This property admits some form of *declassification* (or downgrading) of confidential data, but requires that an attacker cannot influence the secret information which is declassified. In our case study of section 4.1, PIN_V returns the correctness of the typed PIN which is a one-bit leakage of information about a secret datum. Robustness requires that an attacker cannot abuse such a declassification in order to gain more information than intended.

The idea is illustrated in Fig. 11: at the top of the figure, we see that a change in the high, i.e., secret input is observable at the low, i.e., public level, meaning that noninterference does not hold and some information is leaked. However, at the bottom of the figure we notice that a change in the low level input does not affect the leaked information. This amounts to stating that what is leaked is never affected by untrusted, low level users. If we consider the decimalization

```
PIN_V_M(EPB, len, offset, vdata, dectab, MAC){
  if (mac_ak(EPB, len, offset, vdata, dectab) == MAC) then
    PIN_V(EPB, len, offset, vdata, dectab);
  else return("integrity violation");
}
```

Fig. 12. A fixed PIN verification API.

attack we notice that the attacker, by manipulating public data such as the dectab and the offset, can affect what is declassified, i.e., the outcome of the API call. By forbidding this, the attack should be prevented.

A robust PIN verification API In [20] we have formalized the above property in the setting of PIN management APIs, and we have given a patched version of PIN_V that prevents attacks by checking the integrity of the public data. The idea is to have a Message Authentication Code (MAC) of such data, i.e., an unforgeable cryptographic signature that once checked by the HSM, guarantees the integrity of data. If the attacker tries to tamper with the decimalization table or offset, the MAC check fails and the HSM aborts the call with an error message. Fig. 12 reports the code of the fixed PIN verification API, called PIN_V_M, which recomputes and checks the MAC before calling the original PIN_V.[5] In the paper we have also devised a type-system to statically prove the robustness of the new API.

Even if the proposed fix has been proved to be secure, the infrastructure changes needed to add full MAC calculation to ATMs and switches are seen as prohibitive by the banks [6]. We have proposed a way to implement a weaker version of our scheme whilst minimising changes to the existing infrastructure [32]. In the paper, we observe that cards used in the cash machine network already have an integrity checking value: the card verification code (CVC), or card verification value (CVV), is a 5 (decimal) digit number included in the magnetic stripe of most cards in use and contains a MAC of the customer's PAN, the expiry date of the card and some other data. The current purpose of the CVV is to make card cloning more difficult, since the CVV is not printed in the card and does not appear on printed POS receipts.[6]

Our idea is to use the CVV calculation method twice, in the manner of a hashed MAC or HMAC function. We calculate the CVV of a new set of data, containing the decimalisation table and offset. Then, we insert the result of the original CVV calculation to produce a final 5-digit MAC. The scheme is easily

[5] For technical reasons, the specification in [20] has that the Personal Account Number is in the list of parameters. Here we consider it as part of the validation data *vdata*.

[6] The CVV/CVC should not be confused with the CVC2 or CVV2, which is printed on the back of the card and is designed to counteract "customer not present" fraud when paying over the Internet or by phone.

implementable but we loose some security, since our MACs now have an entropy of only 5 decimal digits ($2^{16} < 10^5 < 2^{17}$).

4.4 Summary

We have seen how bank networks use special HSMs to secure ATM transactions. These sophisticated devices perform cryptographic operations such as PIN encryption/decryption and PIN verification via a security API. We have discussed attacks on this API and we have shown in detail the so called *decimalization table* attack. We have seen how these attacks progressively leak partial information about PIN digits similarly to what happens in the Mastermind game. We have shown that algorithms for the solution of Mastermind can be adapted to search for efficient attacking strategies on bank PINs. Finally, we have illustrated how the security of PIN management APIs can be modelled in terms of an information flow property called *robustness* and we have described a fixed API that can be proved to be *robust*. Intuitively, a robust API can leak partial information on a secret, e.g., whether the PIN typed at the ATM is correct or not, but this leakage should never be under the control of the attacker, as occurs in the decimalization attack.

5 Conclusions

There are many open research problems in security APIs, both theoretical and practical. In the domain of key management, we have seen that the current standard for APIs, PKCS#11, is very difficult to configure securely. This is evidenced by a number of attacks on models of the standard as well as attacks on real PKCS#11 security tokens: in our sample of 18 devices, we found 6 tokens that trivially gave up their sensitive keys in complete disregard of the standard, 3 that were vulnerable to a variety of 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 have seen that it is possible to propose secure configurations, [17, 33], but security proofs here are only in the symbolic model of cryptography: there is more work to be done to reconcile these proofs to the cryptographic details of real implementations, though work in this direction is underway [40].

Given the current situation it is perhaps not surprising that there have been articles proposing completely new designs for key management APIs [19, 24]. Indeed at least two new industrial standards which address key management are currently at the draft stage: IEEE 1619.3 [36] (for secure storage) and OASIS Key Management Interoperability Protocol (KMIP) [44]. It remains to be seen how these latter designs will address security concerns.

We have described attacks on bank HSMs for the ATM network. We have discussed possible fixes for the decimalization attack but the extent to which these fixes can be implemented in real devices is still unclear. There is a great deal of existing infrastructure in the ATM network that would be costly to

replace. Since the attacks involve obtaining PINs in order to withdraw money using closed magnetic stripe cards, one possibility would be to upgrade the whole network to the new chip-based cards [4]. This seems unlikely to happen in the short term, and even if a decision were taken to do so it might take years to cover the whole world-wide network. In the meanwhile the network will fall-back to the magnetic stripe protocol whenever the chip is not supported by the ATM, and well-organised attackers will continue to 'cash out' their stolen PINs in countries where magnetic stripe ATMs are still the norm.

We have seen that the problem of PIN processing API attacks, in a sense, an excessive parametrization of some of the functionalities. Since these parameters are public, and attacker can manipulate them in order to affect the information that is declassified by the PIN check. Some of these parameters could be 'locked-down': the dectab, for example, could be fixed on the HSMs of a particular bank, assuming that one bank only uses one dectab for all users. As a more general solution, we have proposed adding a MAC of (a subset of) the function parameters, but this of course requires to change HSMs to support this new operation. A cheap low-impact fix could be obtained using CVVs but this is not completely satisfactory from a security point of view, as it only mitigates the attacks. In our opinion, the direction that bank industry will take is still unclear.

The field of security API analysis is highly active with an annual international workshop (ASA - Analysis of Security APIs) and a chapter of Anderson's essential textbook [7] devoted to the theme. Recent articles at ASA and elsewhere have proposed security API techniques for solving problems in anything from social networks to browser Javascript engines to utility meters. Thanks to the ASA forum, the formal methods and security community has come closer to industrial users of security APIs, leading to research results that are applicable in the short term, and hopefully, in the longer term, influencing the way secure devices are developed, programmed and used. It is, in fact, a very exciting time to be researching security APIs.

References

1. Hackers crack cash machine PIN codes to steal millions. The Times online. `http://www.timesonline.co.uk/tol/money/consumer_affairs/article4259009.ece`.
2. Mastermind. `http://commons.wikimedia.org/wiki/File:Mastermind.jpg`.
3. PIN Crackers Nab Holy Grail of Bank Card Security. Wired Magazine Blog 'Threat Level'. `http://blog.wired.com/27bstroke6/2009/04/pins.html`.
4. The EMV Standard. `http://www.emvco.com/`.
5. R. Anderson. The correctness of crypto transaction sets. In *8th International Workshop on Security Protocols*, April 2000. `http://www.cl.cam.ac.uk/ftp/users/rja14/protocols00.pdf`.
6. R. Anderson. What we can learn from API security (transcript of discussion). In *Security Protocols*, pages 288–300. Springer, 2003.
7. R. Anderson. *Security Engineering*. Wiley, 2nd edition, 2007.
8. A. Armando, D.A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. Hankes Drielsma, P. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim,

D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

9. 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`.
10. O. Berkman and O. M. Ostrovsky. The unbearable lightness of PIN cracking. In Springer LNCS vol. 4886/2008, editor, *11th International Conference, Financial Cryptography and Data Security (FC 2007), Scarborough, Trinidad and Tobago*, pages 224–238, February 12-16 2007.
11. B. Blanchet. From secrecy to authenticity in security protocols. In M. Hermenegildo and G. Puebla, editors, *International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359, Madrid, Spain, September 2002.
12. 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.
13. M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, 34(10):67–75, October 2001.
14. M. Bond and J. Clulow. Extending security protocol analysis: New challenges. *Electronic Notes in Theoretical Computer Science*, 125(1):13–24, 2005.
15. M. Bond and P. Zielinski. Decimalization table attacks for pin cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, 2003. `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf`.
16. Mike Bond and Jolyon Clulow. Encrypted? randomised? compromised? (when cryptographically secured data is not secure). In *Cryptographic Algorithms and their Uses*, pages 140–151, 2004.
17. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269, Chicago, Illinois, USA, October 2010. ACM Press.
18. C. Cachin and J. Camenisch. Encrypting keys securely. *IEEE Security & Privacy*, 8(4):66–69, 2010.
19. 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.
20. M. Centenaro, R. Focardi, F.L. Luccio, and G. Steel. Type-based analysis of PIN processing APIs. In Springer LNCS vol. 5789/2009, editor, *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 09)*, pages 53–68, 2009.
21. 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.
22. J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.
23. J. Clulow. On the security of PKCS#11. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425. Springer, 2003.

24. V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In Michael Backes and Peng 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, September 2009. Springer.
25. 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.
26. S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.
27. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.
28. A. Durante, R. Focardi, and R. Gorrieri. A compiler for analyzing cryptographic protocols using noninterference. *ACM Transactions on Software Engineering and Methodology*, 9(4):488–528, 2000.
29. 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.
30. R. Focardi and F.L. Luccio. Secure upgrade of hardware security modules in bank networks. In *Proceedings of Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security - Joint Workshop, (ARSPA-WITS'10)*, pages 95–110, Paphos, Cyprus, March 2010. Springer LNCS vol. 6186.
31. R. Focardi and F.L. Luccio. Guessing bank pins by winning a mastermind game. *Theory of Computing Systems*, 2011. To appear.
32. R. Focardi, F.L. Luccio, and G. Steel. Blunting differential attacks on PIN processing APIs. In Springer LNCS vol. 5838/2009, editor, *Proceedings of the 14th Nordic Conference on Secure IT Systems (NORDSEC 09)*, pages 88–103, October 2009.
33. S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In Pierpaolo Degano and Luca Viganò, editors, *Revised Selected Papers 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, August 2009. Springer.
34. J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
35. J. Herzog. Applying protocol analysis to security device interfaces. *IEEE Security & Privacy Magazine*, 4(4):84–87, July-Aug 2006.
36. IEEE 1619.3 Technical Committee. IEEE storage standard 1619.3 (key management) (draft). available from `https://siswg.net/`.
37. International Organization for Standardization. ISO 9564-1: Banking personal identification number (PIN) management and security. 30 pages.
38. G. Keighren. Model checking security APIs. Master's thesis, University of Edinburgh, 2007.
39. D. Knuth. The Computer as a Master Mind. *Journal of Recreational Mathematics*, 9:1–6, 1976.
40. S. Kremer, G. Steel, and B. Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, Cernay-la-Ville, France, June 2011. IEEE Computer Society Press. To appear.
41. D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

42. G. Lowe. Breaking and fixing the Needham Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055, pages 147–166. Springer Verlag, 1996.
43. A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, May 2006.
44. OASIS Key Management Interoperability Protocol (KMIP) Technical Committee. KMIP – key management interoperability protocol. available from `http://xml.coverpages.org/KMIP/`, february 2009.
45. openCryptoki. `http://sourceforge.net/projects/opencryptoki/`.
46. C.A. Pickover. *The Math Book: From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics*. Sterling, 2009.
47. RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.
48. B. Schneier. *Applied Cryptography*. John Wiley and Sons, 2nd edition, 1996.
49. G. Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.
50. J. Stuckman and G. Zhang. Mastermind is NP-Complete. *INFOCOMP Journal of Computer Science*, 5:25–28, 2006.
51. E. Tsalapati. Analysis of PKCS#11 using AVISPA tools. Master's thesis, University of Edinburgh, 2007.
52. 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.