Formal Analysis of Security APIs

Graham Steel

Hubert COMON-LUNDH	examinateur
Claude KIRCHNER	examinateur
Ralf KÜSTERS	examinateur
Yassine LAKHNECH	rapporteur
Gavin LOWE	rapporteur
John MITCHELL	examinateur
Michaël RUSINOWITCH	rapporteur
Andre SCEDROV	examinateur

Habilitation à Diriger des Recherches
Laboratoire Spécification et Vérification
Departement d'Informatique
Ecole Normale Supérieure de Cachan
2011

Abstract

An Application Program Interface (API) is considered a security API when it is designed not only to offer access to functionality but also to enforce a security policy, i.e. no matter what commands are sent to the interface, some security properties continue to hold. They are used, for example, as interfaces to cryptographic hardware modules and smartcards. They are very difficult to design, and errors in security APIs have been shown to give rise to critical vulnerabilities in a variety of real world systems, from cash machine PIN processing modules to authentication tokens.

Formal analysis of security APIs aims to use techniques from program verification both to find attacks on faulty APIs, and prove security properties of correct ones. This thesis describes some of my work on developing this area from 2004-2010.

We focus on APIs for cryptographic key management. We start by defining a Dolev-Yao like model for security APIs, which leads in general to an undecidable security problem. We show decidability for a number of subclasses and soundness for some abstractions. We show how these results have been applied to real commercially available devices, resulting in the unearthing of a number of previously unknown vulnerabilities. We propose a new API for key management which we prove secure in our model. Finally we evaluate the work and give some perspectives.

Acknowledgements

Thanks to my wife Sam and my colleagues at the Universities of Karlsruhe, Genova and Edinburgh and at the École Normale Superiéure de Cachan.

Publications

In this habilitation thesis, citations of my own work will be given as numerical references (e.g. [2]), and references to the work of others in alpha style (e.g. [DLMS99]). Much of the work reported here was produced in collaboration with others, including:

Matteo Bortolozzo (Universita' Ca' Foscari, Venice)
Matteo Centenaro (Universita' Ca' Foscari, Venice)
Véronique Cortier (LORIA & CNRS, Nancy)
Stéphanie Delaune (ENS Cachan & CNRS)
Riccardo Focardi (Universita' Ca' Foscari, Venice)
Sibylle Fröschle (Universität Oldenburg, Germany)
Gavin Keighren (University of Edinburgh, UK)
Steve Kremer (ENS Cachan & INRIA)

Table of Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Existing Work . . . 1
1.3 Contributions and Structure of this Thesis . . . 2

2 Formal Model **5**
2.1 Basic Notions . . . 5
2.1.1 Description language . . . 6
2.1.2 Queries . . . 7
2.2 Variations . . . 8
2.2.1 Equational Theories . . . 8
2.2.2 Explicit Destructors . . . 8
2.2.3 Pairing and Projection . . . 9

3 Decidability Results and Abstractions for Security API Models **11**
3.1 Undecidability of the General API Problem . . . 11
3.2 APIs using XOR . . . 12
3.2.1 WFX APIs . . . 12
3.2.2 Proof of Decidability . . . 14
3.3 Key Conjuring . . . 15
3.3.1 Key Conjuring . . . 18
3.3.2 Transformation on the API rules . . . 18
3.3.3 A New Decidable Class . . . 20
3.4 Stateful API Models . . . 21
3.4.1 Model . . . 21
3.4.2 Decidability . . . 21
3.4.3 Preliminaries . . . 23
3.4.4 Existence of a well-moded derivation . . . 24
3.4.5 Decidability result . . . 25
3.5 Abstractions for Fresh Data . . . 26
3.5.1 Abstraction Definition . . . 26
3.5.2 Stateless APIs with No Equational Theory . . . 26
3.5.3 Stateless APIs with XOR . . . 27
3.5.4 Stateful APIs . . . 27
3.6 Discussion . . . 28
3.6.1 Related Work . . . 28
3.6.2 Future Work . . . 29

4 **Results on Commercial Devices** **31**
4.1 Visa Security Module . . . 31
4.2 IBM CCA . . . 32
4.2.1 Representation of XOR terms . . . 32
4.2.2 The Implemented Procedure . . . 33
4.2.3 Results . . . 34
4.3 PKCS#11 Devices . . . 34
4.3.1 An Introduction to PKCS#11 . . . 34
4.3.2 Vulnerabilities in the Standard . . . 36
4.3.3 Testing on Commercial Devices . . . 37
4.3.4 Limitations of Reverse Engineering . . . 43
4.3.5 Results . . . 43
4.3.6 Implemented functionality . . . 44
4.3.7 Attacks . . . 45
4.3.8 Model-checking results . . . 45
4.3.9 Finding Secure Configurations . . . 46
4.3.10 Conclusion . . . 47

5 **A New Generic API for Key Management** **51**
5.1 Model . . . 52
5.1.1 Syntax . . . 52
5.1.2 Model . . . 52
5.2 Presentation of the Generic API . . . 53
5.2.1 API rules . . . 53
5.2.2 Using the API . . . 54
5.2.3 Comparison with PKCS#11 . . . 56
5.3 Using the Generic API to Implement a Protocol . . . 56
5.3.1 Algorithm . . . 57
5.3.2 Example . . . 58
5.4 Security of the API . . . 59
5.5 Security of the API under compromised handles . . . 61
5.6 Security of the API with periodic erasure . . . 62
5.7 Comparison with the Cachin-Chandran API . . . 64
5.7.1 Scenario . . . 64
5.7.2 Objects and Operations . . . 64
5.7.3 Security Policy . . . 65
5.7.4 Comparison . . . 65
5.7.5 Summary . . . 66
5.8 Discussion . . . 66

6 **Future Work and Perspectives** **69**
6.1 Evaluation . . . 69
6.2 Security APIs vs Security Protocols . . . 70
6.3 Future Work . . . 70
6.3.1 Computational Soundness . . . 70
6.3.2 Decidability . . . 71
6.3.3 Non-monotonic State . . . 71
6.3.4 Composition . . . 72
6.3.5 Quantitative Models . . . 72
6.3.6 More General Security APIs . . . 72

6.4 Perspective 73

7 Conclusions **75**

My Publications
(since completing my PhD) **77**

Bibliography **81**

List of Figures

3.1 Rules of the IBM 4758 CCA API. 13
3.2 Rules of the IBM CCA API in an explicit decryption model 16
3.3 PKCS#11 key management subset. 22

4.1 Fragment of the VSM API 31
4.2 PKCS#11 Wrap/Decrypt Attack 35
4.3 PKCS#11 Encrypt/Unwrap Attack 36
4.4 PKCS#11 Re-import Attack 1 37
4.5 PKCS#11 Re-import Attack 2 37
4.6 Symmetric Key Management subset of the Eracom PKCS#11 API. 38
4.7 Eracom PKCS#11 Lost Session Key Attack 38
4.8 Revised Wrap/Unwrap Mechanism for the Eracom API 41
4.9 Tookan system diagram 41

5.1 Formal Description of API rules 55
5.2 Threat model for our API 59

List of Tables

4.1 Results using our decision procedure to verify IBM recommendations for the CCA API 34
4.2 Syntax of Meta-language for describing PKCS#11 configurations 39
4.3 PKCS#11 key management subset with side conditions from the meta-language of table 4.2 40
4.4 Summary of Results on Commercial PKCS#11 Devices 49
4.5 Key for table 4.4 50

Chapter 1

Introduction

1.1 Context

As society becomes more dependent on networked computer infrastructure for critical functionality, security becomes an increasing concern. Over the last 25 years or so, computer scientists have been increasingly using formal methods techniques to analyse the security of distributed systems. Security-critical systems often employ cryptography in order to achieve their goals. Analysis of cryptographic primitives such as encryption or hashing, and of cryptographic protocols for key exchange, such as Kerberos and TLS, is now a mature field of computer science. Since securing commodity personal computers from viruses and malware has proved extremely difficult, distributed systems increasingly employ tamper-resistant cryptographic hardware such as a smartcard or hardware security module (HSM) to secure the cryptographic keys used in the protocols. In the early 2000s, a series of attacks emerged on the HSMs used to store keys and implement secure processing of PINs in the cash machine network [Bon01, Clu03a]. These attacks, found by by pain-staking analysis of the specifications of the devices, were of a new type, since they attacked neither the cryptographic primitives nor the protocol used for communication in the network, but the way the operations required by the protocols were implemented by the device. They became known as API attacks, since they exploit the Application Program Interface (API) made available by the device.

Whilst completing my PhD on the formal analysis of protocols for group key management in 2004, I read about these attacks and proposed a research project to develop formal methods to detect and verify the absence of such attacks on an API design. I recruited the UK government IT infrastructure security agency (CESG) and a manufacturer of cryptographic hardware (nCipher) as collaborators. The proposal was accepted by the UK funding agency EPSRC, and I began my work at the University of Edinburgh in October 2004. In October 2007, at the end of the project, I moved to the Laboratoire Spécification et Vérfication at the ENS Cachan, where I continued to research the topic. This habilitation thesis presents a synthesis of a part of this research together with an evaluation of the work and a discussion of future projects.

1.2 Existing Work

We informally define a security API as a set of commands offered by one processs to other processes that not only provides access to functionality, like a normal API, but also enforces a security policy. A security API must be designed such that no matter what calls to the interface are made, some security properties should continue to hold. 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 [LR92]. Their article

showed how Prolog could be used to analyse a key management interface of a cryptographic device. Although the device was not identified at 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 [And00], and the term 'security API' was coined by Bond and Anderson [Bon01] in a subsequent paper 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 [CB02]. 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 [Clu03a]. Clulow also published attacks on the industry standard for cryptographic key management APIs, RSA PKCS#11 [Clu03b].

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 [YAB+05]). The researchers lacked experience with the tools. They 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 analyzing the device." [Her06].

Meanwhile, the formal analysis of protocols for e.g. cryptographic key exchange and authentication had become a mature field. A particularly successful approach had centered 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 [DY83]. 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 [ABB+05, Bla02, Low96]. 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, there are some important specificities to security APIs that make the adaptation of protocol analysis tools to the problem non-trivial. 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). Even more conventional API attacks, which directly reveal a secret value, often make use of the algebraic properties of cryptofunctions used, such as exclusive-OR (XOR). Protocol analysis tools were only beginning to support such properties. In addition, 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. We will discuss all these specificities in the chapters that follow.

1.3 Contributions and Structure of this Thesis

In this document, we will summarise work on developing verification techniques for security APIs. The aim of the research, broadly speaking, was to extend automated formal analysis techniques in order to detect attacks on or prove security properties of real security APIs de-

ployed in systems such as smartcards, authentication tokens, and the cash machine network. Attacks on such interfaces may occur at different levels: they may be due to a badly designed API ("logical" flaws), which can be detected by a formal analysis of a symbolic (DY) model. They may also occur at a "computational" level: the actual cryptographic security primitives used do not coincide with the assumptions required for a proof of security at the abstract level. Finally, they may occur in the implementations, which may not satisfy (explicit or implicit) assumptions of the higher level models. The work presented in this thesis will touch on all these aspects of security API analysis.

We will cover foundational issues such as decidability of security problems in various formal models, abstraction techniques, automation of analysis, and results of testing our tools on real APIs in use in commercially deployed systems. We will exhibit new interface designs with formally proven security properties. We will show the specificities of API analysis as compared to the more mature field of protocol analysis, and also highlight how developments in API analysis have fed back in to protocol analysis to improve the state of the art. In addition, the study of real systems with real vulnerabilities provides an opportunity to assess the adequacy of formal methods in general for the analysis of security problems, where unlike in conventional program verification or testing we must consider an unpredictable and malicious intruder. To keep the thesis down to a reasonable length, we will concentrate on security APIs for key management. Work specific APIs for processing PINs in the cash machine network [2, 9, 19] will not be discussed, and neither will recent work on the Trusted Plaform Module API [5] nor work on vehicular ad-hoc networks [4, 17].

The structure of this document is as follows: we will first introduce the term algebra model used for our analysis of APIs (§2). We then explain the formal verification techniques developed for key management APIs, starting first with results on decidability of formal security for increasingly expressive security API models, giving abstraction techniques for dealing with APIs that generate fresh data (§3). We give some practical results from the application of our techniques to real APIs in use in commercial devices (§4). We present a design for a new key management API, prove strong security properties for it, and compare it to a rival proposal (§5). We discuss future work and perspectives (§6), and finally summarise and conclude (§7).

Chapter 2

Formal Model

We will study the security of key management APIs in an abstract term algebra model following the classical approach of Dolev and Yao [DY83], enriching the model as necessary to model different aspects of API operations. Bitstrings such as keys, nonces, ciphertexts etc. are represented as terms, and operations on bitstrings such as encryption and signing are represented as functions on those terms.

2.1 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 countably infinite) set of *names* $\mathcal{N}$ and a (possibly countably infinite) set of *variables* $\mathcal{X}$. In our notation, symbols beginning with $\mathsf{x}, \mathsf{y}, \mathsf{z}$ are variables. Names represent keys, data values, nonces, etc. and function symbols model cryptographic primitives. In particular, we will write $\{\!|\mathsf{x}|\!\}_{\mathsf{y}}$ representing symmetric encryption of plaintext x under key y, and $\{\mathsf{x}\}_{\mathsf{y}}$ representing public key encryption of x under y. We will call terms like $\{\!|\mathsf{x}|\!\}_{\mathsf{y}}$, $\{\mathsf{x}\}_{\mathsf{y}}$ ciphertexts. We will use the symbol $\mathsf{pub}(\mathsf{x})$ to denote the public half of the public-private keypair $\mathsf{pub}(\mathsf{x}), \mathsf{x}$. Function symbols of arity 0 are called *constants*. This includes the Boolean constants true ($\top$) and false ($\bot$). The set of *plain terms* $\mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})$ is defined by the following grammar:

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

In addition to the signature Σ, assume a finite set $\mathcal{F}$ of predicate symbols, disjoint from Σ, from which we derive a set of *facts*, defined as

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

In this way, we can explicitly express the Boolean value b of a predicate p on a term t by writing $p(t, b)$.

We write $vars(t)$, resp. $names(t)$, for the set of variables, resp. names, that occur in the term t and extend *vars* and *names* to sets of terms in the expected way. If $vars(t) = \emptyset$ then t is called *ground*.

A *position* is a finite sequence of positive integers. The empty sequence is denoted ε. The set of positions $pos(t)$ of a term t is defined inductively as $pos(u) = \{\varepsilon\}$ for $u \in \mathcal{N} \cup \mathcal{X}$ and $pos(f(t_1, \ldots, t_n)) = \{\varepsilon\} \cup \bigcup_{1 \le i \le n} i \cdot pos(t_i)$ for $f \in \Sigma \cup \mathcal{A}$. If p is a position of t then the subterm of t at position p is written $t|_p$, i.e. $t|_\varepsilon = t$ and $f(t_1, \ldots, t_n)|_{i \cdot p} = t_i|_p$. The set of *subterms* of a term t, written $st(t)$, is defined as $\{t|_p \mid p \in pos(t)\}$. We denote by top the

function that associates to each term t its root symbol, i.e. $\mathsf{top}(u) = u$ for $u \in \mathcal{N} \cup \mathcal{X}$ and $\mathsf{top}(f(t_1, \ldots, t_n)) = f$. We say that a term t is headed with f if its root symbol is f.

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

2.1.1 Description language

To model APIs and attacker capabilities we define a rule-based description language. It is similar to a guarded command language *à la Dijkstra* ([Dij75]), and resembles security protocol analysis formalisms such as the multi-set rewriting framework (e.g. [Mit02]) or the 'intermediate format' set rewriting language used in the AVISPA toolsuite [Pro].

Syntax and informal semantics. The description of a system is given as a finite set of rules of the form

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

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

We suppose for the moment that L' is always satisfiable, i.e. it does not contain both $a(t, \bot)$ and $a(t, \top)$. Moreover, we require that $names(T \cup L) = \emptyset$ and $names(T' \cup L') \subseteq \tilde{n}$. We also suppose that any variable appearing in T' also appears in T, i.e. $vars(T') \subseteq vars(T)$, and any variable appearing in L' also appears in L, i.e. $vars(L') \subseteq vars(L)$.

Example 1 *The intruder can add to his knowledge set either by calling a command from the API under analysis, or by using one of the following rules for symmetric and asymmetric encryption and decryption.*

$$\begin{array}{rcl} \mathsf{x}, \mathsf{y} & \to & \{\!|\mathsf{x}|\!\}_{\mathsf{y}} \\ \{\!|\mathsf{x}|\!\}_{\mathsf{y}}, \mathsf{y} & \to & \mathsf{x} \\ \mathsf{x}, \mathsf{y} & \to & \{\mathsf{x}\}_{\mathsf{y}} \\ \{\mathsf{x}\}_{\mathsf{pub}(\mathsf{y})}, \mathsf{y} & \to & \mathsf{x} \end{array}$$

Note that these rules contain no state component L - they may be executed no matter what the state of the API is.

Semantics. The formal semantics of our description language is given in terms of a *transition system*. We assume a given signature Σ, a set of predicates $\mathcal{F}$, a set of names $\mathcal{N}$, a set of variables $\mathcal{X}$, and a set of rules $\mathcal{R}$ defined over $\mathcal{T}(\mathcal{F}, \Sigma, \mathcal{N}, \mathcal{X})$. A partial valuation V of ground fact terms is a partial function[1] $V : \mathcal{FT}(\mathcal{F}, \Sigma, \mathcal{N}) \to \{\top, \bot\}$. We extend valuations to facts as

[1]We temporarily overload $\bot, \top$ in this subsection.

$$V(\ell) \quad = \quad \begin{cases} V(f,t) & \text{if } \ell = p(t,\top) \\ \neg V(f,t) & \text{if } \ell = p(t,\bot) \end{cases}$$

We extend the V to sets of facts (interpreted as a conjunction) as $V(L) = \bigwedge_{\ell \in L} V(\ell)$ when $\{f \mid f \in L\} \subseteq dom(V)$. Moreover, we assume a given set of ground terms $S_0 \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ and a partial valuation of ground facts to represent the initial state. In the following we say that the rule

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

is a fresh renaming w.r.t. a set of names $N \subseteq \mathcal{N}$ of a rule

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

if $v_i = u_i[n_1 \rightarrow n'_1, \ldots, n_k \rightarrow n'_k]$ $(1 \leq i \leq p)$, $L'' = L'[n_1 \rightarrow n'_1, \ldots, n_k \rightarrow n'_k]$ and $n'_1, \ldots, n'_k$ are distinct names not in N. These fresh renamings allow us to get rid of the new in the rule. The transition system $(Q, q_0, \rightsquigarrow)$ is defined as follows:

- Q is the set of *states*: each state is a pair (S,V), such that $S \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ and V is any partial valuation of $\mathcal{FT}(\mathcal{F}, \Sigma, \mathcal{N})$.
- $q_0 = (S_0, V_0)$ is the *initial state*. S_0 is the initial attacker knowledge and V_0 defines the initial valuation of some facts.
- $\rightsquigarrow \subseteq Q \times Q$ is the transition relation defined as follows. We have that $(S,V) \rightsquigarrow (S',V')$ if

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

  is a fresh renaming w.r.t. $names(S \cup dom(V))$ of a rule in $\mathcal{R}$ and there exists a grounding substitution θ for R such that

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

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

  $$dom(V') = dom(V) \cup \{p,t \mid p(t,b) \in L'\}$$
  $$V'(p,t) = \begin{cases} b' & \text{if } p(t,b') \in L' \\ V(p(t,b)) & \text{otherwise} \end{cases}$$

We therefore dynamically extend the domain of the valuation whenever facts are created on fresh names. When a fact is not in the domain of the valuation it is considered to be undefined. An instance of a rule with such an undefined fact on the left-hand side cannot be fired.

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

2.1.2 Queries

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

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

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

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

2.2 Variations

In this thesis, we will consider several variations to the model to account for different aspects of the APIs under analysis. Here we mention some important examples.

2.2.1 Equational Theories

Sometimes we will consider an equational theory in order to more closely model the semantics of the API. For example, the following theory for bitwise exclusive-OR (XOR):

$$\begin{aligned} x \oplus (y \oplus z) &= (x \oplus y) \oplus z \\ x \oplus y &= y \oplus x \\ x \oplus x &= 0 \end{aligned}$$

When considering APIs that make use of XOR, we equip the intruder with a further rule for increasing his knowledge:

$$x, y \quad \rightarrow \quad x \oplus y$$

2.2.2 Explicit Destructors

Decryption of ciphertexts in Dolev-Yao style models is typically modelled in one of two ways. The first, known as 'implicit' encryption, involves a model with no symbol for decryption. We simply model decryption using rules like the one shown above

$$\{\!|x|\!\}_y, y \quad \rightarrow \quad x$$

Thus decryption only occurs when the patterns match correctly, i.e. y was really used to encrypt x. Furthermore, a rule requiring a ciphertext encrypted under a particular key on the left hand side can only be fired if the intruder knowledge contains a ciphertext encrypted under that key.

The other style of modelling encryption is to add an explicit destructor $\mathsf{dec}(x, y)$ signifying an attempt to decrypt the ciphertext x using key y, and an equation

$$\mathsf{dec}(\{\!|x|\!\}_y, y) = x$$

We may also add the equation

$$\{\!|\mathsf{dec}(x, y)|\!\}_y = x$$

Since unification is now modulo this equational theory, the attacker is able to fire rules with a ciphertext on the left hand side even if he does not posses (and cannot construct) a ciphertext with the correct key. For example, if there is a rule

$$\{\!|x|\!\}_k \rightarrow \mathsf{secret}$$

and the intruder knowledge does not contain k or any ciphertexts encrypted under k, the intruder can still fire the rule using some arbitrary term t, since $t = \{\!|\mathsf{dec}(t, k)|\!\}_k$.

Which of these is the 'correct model' depends on the API we are modelling. If a device uses an *authenticated* encryption mode, then with high probability the device will reject a forged ciphertext (i.e. one constructed without knowledge of the key). However, many of the devices whose APIs we analysed allow plain unauthenticated encryption, where forging a ciphertext is trivial. For certain classes of security protocol, Millen [Mil03] and then Lynch and Meadows [LM05] have shown that the two models are equivalent. However most of the APIs we studied are outside this class. We will discuss this further in sections 3.6.2 and 6.3.1.

2.2.3 Pairing and Projection

Sometimes we will add to the model concatenation of bitstrings using for example a pairing function symbol ., and destructor symbols l, r for left and right projection

$$\mathsf{l}(\mathsf{x}.\mathsf{y}) = \mathsf{y}$$

$$\mathsf{r}(\mathsf{x}.\mathsf{y}) = \mathsf{y}$$

We will allow the intruder is to use both the pairing constructor and the projection destructors.

Chapter 3

Decidability Results and Abstractions for Security API Models

Having defined our model for security APIs, and the security problem for such APIs in terms of reachability queries in the model, it is natural that we should ask about decidability of this problem. In this chapter we present work on that question. We start by showing undecidability of the API security problem in the full generality of our language. We then give positive results for several language fragments, giving examples of real-world APIs that correspond to these fragments. The proofs follow a similar pattern: roughly, we will show that in order to reach a state in the particular fragment, it suffices to consider intermediate states that are also within the fragment, and then if necessary we introduce some bounds on the amount of fresh material generated in order to achieve a finite number of possible states. At the end of the chapter we will show how abstractions may be used to remove this bound in some cases.

3.1 Undecidability of the General API Problem

To show that the security API problem is undecidable in the language of section 2.1.1, we can borrow from results in the field of security protocol analysis. There are several undecidability results for protocols of various flavours (bounded/unbounded term size, bounded/unbounded fresh data,'compound keys' etc.) [CLC03, TEB05, DLM04]. Most of these results involve constructing a protocol which preserves secrecy of some term if and only if it encodes a particular instance of Post's correspondence problem (PCP). In some cases these constructions would lead to rather unrealistic looking APIs, but following the construction of Durgin et al. for the case with unbounded fresh nonces but bounded message size, we obtain quite a reasonable looking API that fits our language and also encodes an instance of PCP, thereby showing undecidability of the general problem. We require only symmetric key encryption and tripling, i.e. tuples of fixed length three, to reproduce an API corresponding to the protocol in table 12, page 298 [DLM04].

As a side remark, notice that the example of Durgin et al. looks arguably even more natural when considered as an API than when viewed as a protocol: the protocol has Alice contacting a number of 'tile' principles who add the top and bottom of their tile to the current top and bottom chain. Viewed as an API with a shared global state, the tile principles are merely commands that extend an encryption, albeit in an unusual way, somewhat like commands used by real APIs for extending encryptions in, say, CBC mode. The tile commands naturally share state so can immediately share the session key. As a further remark, note that in reality an unbounded number of fresh nonces is not realistic: nonces in an API typically have some fixed length of, say 160 bits, so that oly 2^{160} different nonces are available. But this undecidability result does

at least give us an idea as to why analysing APIs (and protocols) is difficult.

Having seen that the general problem is undecidable, we now show how to recover decidability in various pertinent fragments of the language.

3.2 APIs using XOR

Let us start with a simple but non-trivial subset of our description language. We will consider APIs with just symmetric key encryption, no pairing, no state, bounded amounts of fresh data, and the equational theory of XOR. Since we have no state to consider, intruder capabilities and the protocol behaviour can be described using rules of the form $t_1,\dots,t_n \to t_{n+1}$ where the t_i are terms. Having a single term on the right hand side is not a restriction in a stateless model, since any rule adding several terms $t_1,\dots,t_m$ to the intruder's knowledge can be written as m separate rules with an identical left hand side and one term from $t_1,\dots,t_m$ on the right. Since there are no state changes, these rules are equivalent (for reachability) to the single rule. The set of *deducible terms* is the reflexive and transitive closure of the rewrite rules.

The following definition will be useful for our proof, which appears in a paper with Cortier and Keighren [14].

Definition 3 *Let $\mathcal{R}$ be a set of rules. Let S be a set of ground terms. The term u is* one-step deducible *from S if there exists a rule $t_1,\dots,t_n \to t \in \mathcal{R}$ and a ground substitution θ such that $t_i\theta \in S$ and $u = t\theta$.*

A term u is deducible *from S, denoted by $S \vdash_{\mathcal{R}} u$, if $u \in S$ or there exist ground terms $u_1,\dots,u_n$ such that $u_n = u$ and u_i is one-step deducible from $S \cup \{u_1,\dots,u_{i-1}\}$ for every $1 \leq i \leq n$. The sequence $u_1,\dots,u_n$ is a* proof *that $S \vdash_{\mathcal{R}} u$.*

We write $\vdash$ instead of $\vdash_{\mathcal{R}}$ when $\mathcal{R}$ is clear from the context.

Example 2 *Let $\mathcal{R}$ be the set of rules described in Example 4. Let $S = \{\{\!|\mathsf{n}|\!\}_{\mathsf{a}}, a \oplus b, b\}$. Then n is deducible from S and $\{\!|\mathsf{n}|\!\}_{\mathsf{a}}, a \oplus b, b, a$ is a proof of $S \vdash n$. Indeed a is one-step deducible from $\{a \oplus b, b\}$ using the rule $x, y \to x \oplus y$ and the fact that $(a \oplus b) \oplus b = a$ and n is one-step deducible from $\{\{\!|\mathsf{n}|\!\}_{\mathsf{a}}, a\}$ using the rule $\{\!|\mathsf{x}|\!\}_{\mathsf{y}}, y \to x$.*

3.2.1 WFX APIs

We now define our restricted class of APIs. Below we will give a real world example of an important API fitting in to this class.

Definition 4 *A term t is an* XOR *term if $t = \bigoplus_{i=1}^{n} u_i$, $n \geq 1$ where each u_i is a variable or a constant.*

A term t is an encryption *term if $t = \{\!|\mathsf{u}|\!\}_{\mathsf{v}}$ where u and v are XOR terms.*

A term t is a well-formed term *if it is either an encryption term or an XOR term. In particular, a well formed term contains no nested encryption.*

A rule $t_1,\dots,t_n \to t_{n+1}$ is well formed *if*

- *each t_i is a well-formed term.*
- *$Var(t_{n+1}) \subseteq \bigcup_{i=1}^{n} Var(t_i)$ (no variable is introduced in the right-hand-side of a rule).*

A proof is well-formed *if it only uses well-formed terms.*

Definition 5 *The* WFX-class *of APIs consists of a pair $(\mathcal{R}, S)$, where $\mathcal{R}$ is a finite set of well-formed rules, and S is a finite set of ground, well-formed terms.*

Key Part Import 1:
$xkpNew\ ,\ xtype \rightarrow \{\!|xkpNew|\!\}_{\mathtt{KM}\oplus xtype\oplus \mathtt{KPART}}$
Key Part Import 2:
$xkpNew\ ,\ xtype\ ,\ \{\!|xkpOld|\!\}_{\mathtt{KM}\oplus xtype\oplus \mathtt{KPART}} \rightarrow \{\!|xkpNew\oplus xkpOld|\!\}_{\mathtt{KM}\oplus xtype\oplus \mathtt{KPART}}$
Key Part Import 3:
$xkpNew\ ,\ xtype\ ,\ \{\!|xkpOld|\!\}_{\mathtt{KM}\oplus xtype\oplus \mathtt{KPART}} \rightarrow \{\!|xkpNew\oplus xkpOld|\!\}_{\mathtt{KM}\oplus xtype}$
Key Import:
$\{\!|xkey|\!\}_{xkek\oplus xtype}\ ,\ xtype\ ,\ \{\!|xkek|\!\}_{\mathtt{KM}\oplus \mathtt{IMP}} \rightarrow \{\!|xkey|\!\}_{\mathtt{KM}\oplus xtype}$
Key Export:
$\{\!|xkey|\!\}_{\mathtt{KM}\oplus xtype}\ ,\ xtype\ ,\ \{\!|xkek|\!\}_{\mathtt{KM}\oplus \mathtt{EXP}} \rightarrow \{\!|xkey|\!\}_{xkek\oplus xtype}$
Encipher:
$x\ ,\ \{\!|xkey|\!\}_{\mathtt{KM}\oplus \mathtt{DATA}} \rightarrow \{\!|x|\!\}_{xkey}$
Decipher:
$\{\!|x|\!\}_{xkey}\ ,\ \{\!|xkey|\!\}_{\mathtt{KM}\oplus \mathtt{DATA}} \rightarrow x$
Key Translate:
$\{\!|xkey|\!\}_{xkek1\oplus xtype}\ ,\ xtype\ ,\ \{\!|xkek1|\!\}_{\mathtt{KM}\oplus \mathtt{IMP}}\ ,\ \{\!|xkek2|\!\}_{\mathtt{KM}\oplus \mathtt{EXP}} \rightarrow \{\!|xkey|\!\}_{xkek2\oplus xtype}$

Figure 3.1: Rules of the IBM 4758 CCA API.

We call our class WFX since these are well-formed APIs using the XOR operator.

Example 3 *In Figure 3.1 we give the key management subset of the IBM CCA API, used to manage keys in a variety of cryptographic Hardware Security Modules such as the 4758, which is widely deployed in the cash machine network.*

In a typical ATM network application, HSMs are used, for example, to decrypt, encrypt and verify PINs. Many different keys may are used for these operations. IBM's Common Cryptographic Architecture (CCA) API [CCA06] supports various key types, such as data keys, key encryption keys, import keys and export keys. Each type is represented by a public 'control vector' which is XOR-ed with the security module's master key (which is stored inside the HSM), before being used to encrypt the particular key. For example, a data key would be encrypted under $\mathtt{KM}\oplus\mathtt{DATA}$.[1] *Keys encrypted in this manner are known as working keys and are stored outside of the security module. They can then only be used by sending them back into the HSM under the desired API command. Only particular types of keys will be accepted by the HSM for particular operations. For example, data keys can be used to encrypt arbitrary messages, but so-called 'PIN Derivation Keys' (PDKs, with control vector* $\mathtt{PIN}$*) cannot, which is critical for security: a customer's PIN is just his account number encrypted under a PIN derivation key.*

In 2001, Bond exhibited an attack whereby the intruder uses API commands to change the type of a key, exploiting the algebraic properties of XOR [Bon01]. This allows a PIN derivation key to be converted into a data key, which can then be used to encrypt data. Hence the attack allows a criminal to generate a PIN for any account number. For more details of Bond's 'Chosen Key Difference' attack, see [Bon01, §5.1].

[1] $\oplus$ represents bitwise XOR.

3.2.2 Proof of Decidability

The key idea of our decidability result is to show that only well-formed terms need to be considered when checking for the deducibility of a (well-formed) term. In particular, there is no need to consider nested encryption. This allows us to consider only a finite number of terms: we have a finite number of atoms in the initial set of rules which can only be combined by encryption and XORing, and XORing identical atoms results in cancellation. At the end of the proof, we comment on the complexity of the resulting decision procedure.

We first prove that whenever an encryption occurs in a deducible term, the encryption is itself deducible.

Proposition 1 *Let $\mathcal{R}$ be a set of well-formed rules. Let S be a set of ground well-formed terms (intuitively the initial knowledge). Let u be a term such that $S \vdash u$ and let $\{\!|\mathsf{u_1}|\!\}_{\mathsf{u_2}}$ be a subterm of u. Then $S \vdash \{\!|\mathsf{u_1}|\!\}_{\mathsf{u_2}}$.*

The proof is by induction on the number of steps needed to obtain u. The full proof is in [26].

Our main result states that only well-formed terms need to be considered when checking for deducibility.

Proposition 2 *Let $\mathcal{R}$ be a set of well-formed rules and S be a set of ground well-formed terms such that*

- *$\mathcal{R}$ contains the rule $x, y \rightarrow x \oplus y$;*
- *S contains 0 (the null element for XOR should always be known to an intruder).*

Let u be a ground well formed term deducible from S. Then there exists a well-formed proof of $S \vdash u$.

We briefly sketch the proof here (a full proof appears in [26]). Taking advantage of the form of the rules, the main idea is to show that, if we consider a proof of a well-formed term u and removing all inside encrypted terms, we obtain a (well-formed) proof of u. We define a function $t \mapsto \bar{t}$ that removes interior encryptions. For example, we have $\overline{\{\!|\mathsf{a} \oplus \{\!|\mathsf{a}|\!\}_\mathsf{b}|\!\}_\mathsf{c} \oplus \{\!|\mathsf{c}|\!\}_\mathsf{b}} = \{\!|\mathsf{a}|\!\}_\mathsf{c} \oplus \{\!|\mathsf{c}|\!\}_\mathsf{b}$. Roughly, we show by induction on the length of the proof that whenever $u_1, \dots, u_n$ is a proof then $\overline{u_1}, \dots, \overline{u_n}$ is a proof. Assume $u_1, \dots, u_n, u_{n+1}$ is a proof and $t_1, \dots, t_k \rightarrow t$ is the last rule that has been applied. There is a substitution θ such that $t\theta = u_{n+1}$ and $t_i\theta = u_{j_i}$. Since t is a well-formed term, any interior encryption e of u_{n+1} must appear under a variable x in t thus e also appears in some u_{j_i}. Intuitively, there is a case analysis depending on whether x also appears under an encryption in t_i. If x does not appear under an encryption, that is $t = x \oplus t'$, we use the fact that e is deducible (Proposition 1) thus $u_{j_i} \oplus e$ is also deducible and we could have chosen $x\theta' = x\theta \oplus e$, removing the encryption from u_{n+1}.

Using Proposition 2, we can now easily conclude the decidability of deducibility.

Theorem 1 *The following problem*

- *Given a finite set of well-formed rules $\mathcal{R}$ containing the rule $x, y \rightarrow x \oplus y$, a finite set S of ground well-formed terms containing 0 and a ground well-formed term u,*
- *Does $S \vdash_{\mathcal{R}} u$?*

is decidable in exponential time in the size of $\mathcal{R}$, S and u.

Let $a_1,\dots,a_n$ be the constants that occur in $\mathcal{R}$, S or u. Let k be the maximal number of terms in the left-hand side of a rule in $\mathcal{R}$. For any $t_1,\dots,t_l \to t \in \mathcal{R}$, we have $l \le k$. We show that $S \vdash_{\mathcal{R}} u$ can be decided in $O(2^{(k+1)(2n+1)})$.

The decision procedure is as follows: we saturate S by adding any well-formed deducible terms. We obtain a set S^*. By Proposition 2, $S \vdash_{\mathcal{R}} u$ if and only if $u \in S^*$. In S^* there are at most

- 2^n XOR terms
- and $2^n \times 2^n = 2^{2n}$ encryption terms

thus $|S^*| \le 2^{2n+1}$. Note that we consider here terms modulo AC which means that we only consider one concrete representation for each class of terms equal modulo AC. This can be done for example by fixing an arbitrary order on the constants and using it to normalise terms.

Now, at each iteration, for each rule $t_1,\dots,t_l \to t \in \mathcal{R}$ we consider any tuple of terms $(u_1,\dots,u_l)$ with u_i in the set that is being saturated and compute the set $\mathcal{M}$ of most general unifiers of $(u_1,\dots,u_l) = (t_1,\dots,t_l)$ (which can be done in polynomial time for well-formed terms, see [26]). Then we add any well-formed instance of $t\sigma$ for any $\sigma \in \mathcal{M}$. We consider at most $|S^*|^k \le 2^{k(2n+1)}$ tuples at each iteration. All together, we need at most $O(2^{(k+1)(2n+1)})$ operations to compute S^*.

We postpone a discussion of our decidability result to the end of the chapter. In chapter 4.3.5, we will show how we implemented our procedure to analyse the IBM CCA API.

3.3 Key Conjuring

Key conjuring is the process by which an attacker obtains an unknown, encrypted key by repeatedly calling a cryptographic API function with random values in place of keys [Bon01]. Learning the encrypted value of a key might not seem useful, but several attacks have been presented that leverage this trick in order to compromise the security of an API [Bon01, Clu03b, CB02].

The aim of the work in this section, carried out in collaboration with Cortier and Delaune [13] is to enrich our API model so that it allows computationally feasible key conjuring operations to take place. We give a model for security APIs that uses explicit decryption, and show how to apply a transform to it to identify feasible key conjuring operations. We then show decidability in this richer model, provided that the key conjuring trick is used a bounded number of times.

In this variant of the model, we require on a sort system for terms. Terms which respect this sort-system are said to be *well-typed*. It includes a set of base type Base and a set of ciphertext type Cipher. We have variables and constants of both types. Moreover we assume that our function symbols have the following type:

$$\begin{array}{rcccccc} \oplus & : & \mathsf{Base} & \times & \mathsf{Base} & \to & \mathsf{Base} \\ \{_\}__ & : & \mathsf{Base} & \times & \mathsf{Base} & \to & \mathsf{Cipher} \\ \mathsf{dec} & : & \mathsf{Cipher} & \times & \mathsf{Base} & \to & \mathsf{Base} \end{array}$$

A pure term t is a well-typed term whose only encryption symbol (when such a symbol exists) is at its root position.

In the CCA API (see above), as in many others, symmetric keys are subject to *parity checking*. The 4758 uses the DES (and 3DES) algorithm for symmetric key encryption. A (single length) DES key consists of 64 bits in total, which is divided into eight groups, each consisting of seven key bits and one associated parity bit. For an odd parity key, each parity bit must be

Key Part Imp. 1 :

$$\mathsf{xk1},\ \mathsf{xtype} \quad \rightarrow \quad \{\!|\mathsf{xk1}|\!\}_{\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype}}$$
$$\mathsf{chkOdd}(\mathsf{xk1}),\ \mathsf{chkEven}(\mathsf{xtype})$$

Key Part Imp. 2:

$$\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{y},\ \mathsf{xk2},\ \mathsf{xtype} \quad \rightarrow \quad \{\!|\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})\oplus\mathsf{xk2}|\!\}_{\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype}}$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype}))$$
$$\mathsf{chkEven}(\mathsf{xk2})$$

Key Part Imp. 3:

$$\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{y},\ \mathsf{xk3},\ \mathsf{xtype} \quad \rightarrow \quad \{\!|\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})\oplus\mathsf{xk3}|\!\}_{\mathsf{km}\oplus\mathsf{xtype}}$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype}))$$
$$\mathsf{chkEven}(\mathsf{xk3})$$

Key Import:

$$\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{y},\ \mathsf{xtype},\ \mathsf{z} \quad \rightarrow \quad \{\!|\mathsf{dec}(\mathsf{y},\mathsf{dec}(\mathsf{z},\mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype})|\!\}_{\mathsf{km}\oplus\mathsf{xtype}}$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{z},\mathsf{km}\oplus\mathsf{imp}))$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{dec}(\mathsf{z},\mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype}))$$

Key Export:

$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{z},\mathsf{km}\oplus\mathsf{exp})),\ \mathsf{y},\ \mathsf{xtype},\ \mathsf{z} \quad \rightarrow \quad \{\!|\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{xtype})|\!\}_{\mathsf{dec}(\mathsf{z},\mathsf{km}\oplus\mathsf{exp})\oplus\mathsf{xtype}}$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{xtype}))$$
$$\mathsf{chkEven}(\mathsf{xtype})$$

Encrypt Data:

$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{data})),\ \mathsf{x},\ \mathsf{y} \quad \rightarrow \quad \{\!|\mathsf{x}|\!\}_{\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{data})}$$

Decrypt Data:

$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{data})),\ \mathsf{x},\ \mathsf{y} \quad \rightarrow \quad \mathsf{dec}(\mathsf{x},\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{data}))$$

Translate Key:

$$\mathsf{chkEven}(\mathsf{xtype}),\ \mathsf{x},\ \mathsf{xtype},\ \mathsf{y1},\ \mathsf{y2} \quad \rightarrow \quad \{\!|\mathsf{dec}(\mathsf{x},\mathsf{dec}(\mathsf{y1},\mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype})|\!\}_{\mathsf{dec}(\mathsf{y2},\mathsf{km}\oplus\mathsf{exp})\oplus\mathsf{xtype}}$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y1},\mathsf{km}\oplus\mathsf{imp}))$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{y2},\mathsf{km}\oplus\mathsf{exp}))$$
$$\mathsf{chkOdd}(\mathsf{dec}(\mathsf{x},\mathsf{dec}(\mathsf{y1},\mathsf{km}\oplus\mathsf{imp})\oplus\mathsf{xtype}))$$

Figure 3.2: Rules of the IBM CCA API in an explicit decryption model

set so that the overall parity of its group is odd. For an even parity key, the parity bits must be set so that all groups are of even parity. If the groups have mixed parities, then the key is of undefined parity and considered invalid. The CCA API checks that all DES keys are of odd parity, and all control vectors are even, so that a key XORed against a control vector will give another odd parity key. These parity considerations are important for our analysis of key conjuring, and are represented in our formalism by occurrences of the predicate symbols chkEven and chkOdd, each having a term as argument. Intuitively, $\mathsf{chkOdd}(t)$ means that t has an odd parity[2]. Among the constants in Σ, some have a parity. By default (no explicit parity given to a constant), we will assume that such a constant has no parity. Moreover, we have some rules to infer parity from known facts, which are:

$$\begin{array}{rcl} \mathsf{chkEven}(x_1),\ \mathsf{chkEven}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkOdd}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2) \\ \mathsf{chkEven}(x_1),\ \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkOdd}(x_1 \oplus x_2) \end{array}$$

Intruder capabilities and the protocol behaviour are described as usual using *rules*, but in this explicit decryption setting they will have a restricted form, as defined below.

Definition 6 (API rule) *An API rule is a rule of the form* $\mathsf{chk}_1(u_1),\ldots,\mathsf{chk}_k(u_k),x_1,\ldots,x_n \rightarrow t$*, where*

- $x_1,\ldots,x_n$ *are variables,*
- t *is a term such that* $vars(t) \subseteq \{x_1,\ldots,x_n\}$*,*
- $u_1,\ldots,u_k$ *are terms of* Base *type not headed with* $\oplus$*,*
- $\mathsf{chk}_i \in \{\mathsf{chkOdd},\mathsf{chkEven}\}$, $1 \leq i \leq k$.

We also assume that the rule only involves pure terms.

The third condition might seem restrictive. However, it merely requires that when parity checking, we check each component of a sum rather than the entire sum. For example, if the sum $v_1 \oplus \cdots \oplus v_k$ has some expected parity, each v_i should also have some expected parity, and we ask that their parity is checked separately.

Example 4 *In the explicit decryption model, the intruder capabilities are represented by the following set of three API rules:*

$$\begin{array}{rcll} x,\ y & \rightarrow & \{\!|\mathsf{x}|\!\}_{\mathsf{y}} & \textit{encryption} \\ x,\ y & \rightarrow & \mathsf{dec}(x,y) & \textit{decryption} \\ x,\ y & \rightarrow & x \oplus y & \textit{xoring} \end{array}$$

Example 5 *Commands may include several parity checks. In Figure 3.2, we give the symmetric key management subset of the IBM 4758 API, written in our notation. The terms* km*,* imp*,* exp*,* kp*,* data *and* pin *denote constant of* Base *type whereas* xtype*,* xk1*, … denote variables. Note that all the rules satisfies conditions stated in Definition 6. For instance, Key Import is used to make a new working key for an HSM. The new key is sent to the target HSM encrypted under a transport key. The command decrypts the imported package, and returns the key encrypted under the local master key XOR the appropriate control vector.*

[2] In the language of our model in section 2, the argument of a predicate is a pair of a term and a Boolean constant $\top$, but since in this section we will only use $\top$ and not $\perp$ in all facts, we will simply write $\mathsf{chkOdd}(t)$ instead of $\mathsf{chkOdd}((t,\top))$

3.3.1 Key Conjuring

As we have seen, key management APIs like the CCA keep working keys outside the HSM, safely encrypted, so that they can only be used by sending them back into the HSM under the terms of the API. What happens when an intruder wants to use a particular command in an attack, but does not have access to an appropriate key? For example, suppose he has no data keys (terms of the form $\{d1\}_{km \oplus data}$), but wants to use the *Encipher* command. In an implicit decryption formalism, the command is defined like this

$$x, \{\!|xkey|\!\}_{km \oplus data} \to \{\!|x|\!\}_{xkey}$$

This suggests that the command cannot be used if the intruder does not have a data key. However, in reality, an intruder could just guess a 64 bit value and use that in place of the data key. The HSM will decrypt the guessed value under $km \oplus data$, and check the parity of the resulting 64 bit term to see if it is a valid key before, enciphering the data. Usually, the check will fail and the HSM will refuse to process the command, but if the intruder guesses randomly, he can expect that 1 in every 256 guessed values will result in a valid key. This notion is captured by our formalism, in which we write the *Encipher* command like this:

$$chkOdd(dec(y, km \oplus data)),\ x,\ y \to \{\!|x|\!\}_{dec(y, km \oplus data)}$$

It may seem useless for the intruder to simply guess values, since the result is a term he knows enciphered under an unknown key, but used cleverly, this technique can result in serious attacks. For example, Bond's so called import-export attack [Bon01], uses key conjuring to convert a PIN derivation key into an encryption key, allowing an intruder to generate the PIN for any given account number. This attack is effective even when the attacker does not posses a key ready for import, as is required by the 'chosen difference' attack discussed previously (3). The attack is naturally represented in our model ([13]). In 2003, it came to light that this attack was impossible in practice, as an undocumented check in the CCA's implementation prevents key parts being passed to *Key Import*. However, further attacks using key conjuring had been discovered by then, [CB02, Clu03b], on both the CCA API and other APIs. Clulow notes in [Clu03b] that key conjuring can be prevented by using a hash or MAC to test the authenticity of keys, but many designs do not include such measures, which increase the key management overhead.

A straightforward 'explicit decryption' model is not sufficient for a key conjuring analysis, since though this allows an attack like Bond's be discovered, it doesn't take into account parity checks. This means that the model cannot distinguish between feasible and non-feasible key conjuring steps, leading to false attacks. For example, for a command like *Key Import* (see Example 5), an explicit decryption model without parity checking would allow an intruder to conjure values for both y and z, which in practice is highly unlikely: only 1 in every 2^{16} pairs of values will pass. Our transform ensures that the intruder has to guess values for at most one parity check. However, we think the transform could be easily parametrized to allow more conjuring steps, or to allow different numbers of conjuring steps for different predicates (to accommodate double length DES keys for examples). This will be discussed at the end of the chapter.

3.3.2 Transformation on the API rules

We propose a transformation allowing us to model key conjuring. This transformation is generic enough to deal with any API made up of rules satisfying the conditions given in Definition 6.

First we assume that our set $\mathcal{N}$ contains an infinite number of fresh names of both types. Rules obtained after transformation are called key conjuring rules and have the following form:

$$\begin{array}{ccc} x_1,\ldots,x_n & \overset{\mathsf{new\ n}}{\rightarrow} & t,\mathsf{n} \\ \mathsf{chk}_1(u_1),\ldots,\mathsf{chk}_k(u_k) & & \mathsf{chk}'_1(v_1),[\mathsf{chk}'_2(\mathsf{n})] \end{array}$$

The notation $[\mathsf{chk}'_2(\mathsf{n})]$ is used to express the fact that $\mathsf{chk}'_2(\mathsf{n})$ is optional.

Let $\mathcal{R}_l \to \mathcal{R}_r = \mathsf{chk}_1(u_1),\ldots,\mathsf{chk}_k(u_k),x_1,\ldots,x_n \to t$ be an API rule. For each i such that $1 \leq i \leq k$, since u_i is a term of Base type not headed with $\oplus$ and which contains no encryption symbol, we have that u_i is either a constant, a variable or a term of the form $\mathsf{dec}(z,t)$. In this last case, we compute the key conjuring rules associated to $\mathcal{R}_l \to \mathcal{R}_r$ as follows:

1. Let $\sigma = \{z \mapsto n\}$, we consider the new rule

$$(\mathcal{R}_l \smallsetminus \{z,\, \mathsf{chk}_j(u_j)\} \overset{\mathsf{new\ n}}{\rightarrow} \mathcal{R}_r \cup \{z,\, \mathsf{chk}_j(u_j)\})\sigma$$

2. Moreover, we have that

$$t = \bigoplus_{i=1}^{p} y_i \oplus \bigoplus_{i=1}^{\ell} c_i \oplus \bigoplus_{i=1}^{q} \mathsf{dec}(z_i,t_i).$$

   for some variables y_i, z_i, some constants c_i and some terms t_i. For each j such that $1 \leq j \leq p$, we let $\sigma = \{y_j \mapsto n\}$ and we consider the new rule

$$(\mathcal{R}_l \smallsetminus \{y_j,\, \mathsf{chk}_j(u_j)\} \overset{\mathsf{new\ n}}{\rightarrow} \mathcal{R}_r \cup \{y_j,\, \mathsf{chk}_j(u_j)\})\sigma$$

   Moreover, we push also on the right hand-side the check performed on y_j if such a check exists.

Given an API rule $\mathcal{R}$, we denote by $\mathsf{KeyCj}(\mathcal{R})$ the set of rules obtained after applying the transformation described above. This notation is extended as expected to sets of API rules.

Example 6 *Consider the rule $\mathcal{R}$, namely Key Part Import 3 described below.*

$$\begin{array}{l} \mathsf{y},\mathsf{xk3},\mathsf{xtype} \rightarrow \{\!|\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})\oplus\mathsf{xk3}|\!\}_{\mathsf{km}\oplus\mathsf{xtype}} \\ \mathsf{chkEven}(\mathsf{xtype}) \\ \mathsf{chkEven}(\mathsf{xk3}) \\ \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})) \end{array}$$

The purpose of this rule is to allow a user to add a final key part xk3 *to a partial key* y *with control vector* xtype*. After applying our transformation, the set* $\mathsf{KeyCj}(\mathcal{R})$ *contains the two rules described below:*

$$\begin{array}{ll} \mathsf{xk3},\mathsf{xtype} \overset{\mathsf{new\,n}}{\rightarrow} & \{\!|\mathsf{dec}(\mathsf{n},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})\oplus\mathsf{xk3}|\!\}_{\mathsf{km}\oplus\mathsf{xtype}} \\ \mathsf{chkEven}(\mathsf{xtype}) & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{n},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{xtype})) \\ \mathsf{chkEven}(\mathsf{xk3}) & \end{array}$$

$$\begin{array}{lcr} \mathsf{y},\ \mathsf{xk3} & \overset{\mathsf{new\,n}}{\rightarrow} & \{\!|\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{n})\oplus\mathsf{xk3}|\!\}_{\mathsf{km}\oplus\mathsf{n}} \\ \mathsf{chkEven}(\mathsf{xk3}) & & \mathsf{chkOdd}(\mathsf{dec}(\mathsf{y},\mathsf{km}\oplus\mathsf{kp}\oplus\mathsf{n})) \\ & & \mathsf{chkEven}(\mathsf{n}) \end{array}$$

This represents the two ways the intruder can use the rule for key conjuring. In the first, he conjures a partially completed key (this is the rule used in step 1 of Bond's attack mentioned

above). In the second, for a fixed constant y*, he conjures a control vector that will allow* y *to be decrypted to form a valid partial key. Note that the conjured control vector is of even parity, so the intruder learns two parity facts in this case. Our transform allows this kind of conjuring because it is assumed the intruder can set the parity of the terms he uses as guesses. The value that is checked for even parity is under his control. Hence the probability of success is the same as for the first conjuring variant.*

Note that our transformation will sometimes produce rules which the intruder cannot use. This happens when the fresh nonce appears in a parity check on the left. The intruder cannot use this rule, since he does not know any parity information about the new nonce before the command is used. This corresponds to a case where the intruder would have to guess a value that decrypts to give a valid key, k, such that k also decrypts some other value to give a valid key. For single length DES keys, this gives the intruder a 1 in 2^{16} chance of success, which we consider unrealistic. However, if the intruder has extended access to a live HSM running the API, we believe our transformation could be quite naturally extended to these more costly operations (see Section 3.6).

3.3.3 A New Decidable Class

Before we define our new decidable class, we have to take into account the properties of Xor and how this relates to parity checking. Essentially, we want our parity checks to be consistent.

Definition 7 (consistent) *Let* $S = \{\mathsf{chk}_1(u_1), \ldots, \mathsf{chk}_i(u_i)\} \cup T$ *where* $u_1, \ldots, u_i$ *are ground terms of* Base *type and* T *is a set of terms. We denote by* $\mathsf{SatChk}(S)$ *the smallest set which contains* S *and that is closed by application of the following rules modulo* Xor.

$$\begin{array}{rcl}\mathsf{chkEven}(x_1),\, \mathsf{chkEven}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2)\\ \mathsf{chkOdd}(x_1),\, \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkEven}(x_1 \oplus x_2)\\ \mathsf{chkEven}(x_1),\, \mathsf{chkOdd}(x_2) & \rightarrow & \mathsf{chkOdd}(x_1 \oplus x_2)\end{array}$$

We say that S *is* consistent *if for any term* t, $\mathsf{chkOdd}(t)$ *and* $\mathsf{chkEven}(t)$ *are not both in* $\mathsf{SatChk}(S)$.

A fact $\mathsf{chkX}(t)$ *ground* if the term t is ground and it is said to be pure if the term t is pure and of Base type inside a parity check.

Example 7 *Let* S *be the following set:*

$$S = \{\mathsf{chkEven}(a \oplus b), \mathsf{chkEven}(b \oplus c), \mathsf{chkOdd}(a \oplus c)\}$$

S is not consistent. Indeed, since $(a \oplus b) \oplus (b \oplus c) =_{\mathsf{Xor}} a \oplus c$*, we have that* $\mathsf{chkEven}(a \oplus c) \in \mathsf{SatChk}(S)$ *and also that* $\mathsf{chkOdd}(a \oplus c) \in \mathsf{SatChk}(S)$.

We define the security problem as usual following the notion of queries in section 2. Of course, at each step of the proof the set of ground facts obtained has to be consistent with respect to the parity checking predicates. However, this will be the case by construction, since the only rules which add parity facts are the key conjuring ones, which always introduce something fresh in the parity facts.

Each time the adversary wants to conjure a key, it requires a significant amount of access to the API. We assume in what follows that the use of these rules by the adversary is limited. This is modelled by introducing a parameter k that bounds the maximum number of applications of the key conjuring rules induced by the protocol. The value of k could be set based on

the amount of time an attacker may have access to a live HSM, based on physical security measures, auditing procedures in place, etc. Note however that we do not bound the number of offline key conjuring since it is much easier for an adversary to try numerous values offline. We extend the usual Dolev-Yao intruder rules I (see section 1) with appropriate conjuring rules to obtain the set I^+ (see [13] for details) and then define the security problem with key conjuring as:

Security Problem

Entries: A finite set $\mathcal{A}$ of API rules, a pure ground initial state (S_0, V_0) that is consistent a pure ground term s (the secret) and a bound $k \in \mathbb{N}$ (number of key conjuring steps).

Question: Is the query (s, V) deducible from (S_0, V_0) by using the rules in $\mathcal{A} \cup I^+$ and at most k instances of rules in $\mathsf{KeyCj}(\mathcal{A})$ (modulo $\mathsf{E_{API}}$)?

In a technical report [25] we prove the decidability of this question with a few extra restrictions on the form of the rules (we call this class 'well formed' APIs in the paper). the restrictions are quite light, requiring e.g. that the same term is not decrypted under two different keys. We also show that our decision algorithm is non-deterministic 2-EXPTIME. Again we will postpone our discussion of the result to the end of the chapter, and move on to a class of APIs that do not make use of XOR, but do include non-monotonic global state.

3.4 Stateful API Models

In this section, we will show decidability for a class of APIs with mutable non-monotonic global state. This is in contrast to the previous section where the state facts (the parity checks) accumulate monotonically and persistently during a derivation. This class of APIs contains a model of RSA PKCS#11, an industry standard for designing APIs that has been very widely adopted. The work in this section was carried out with Delaune and Kremer [12, 1].

3.4.1 Model

In this section, our model includes symmetric and asymmetric cryptography, but no equational theory. As running example we will consider the rules given in Figure 3.3, which model a part of PKCS#11. We detail the first rule which allows wrapping of a symmetric key with a symmetric key. Intuitively the rule can be read as follows: if the attacker knows the handle $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$ (which can be thought of as a pointer or reference to a symmetric key y_1 stored on the device), and a second handle $\mathsf{h}(\mathsf{x}_2, \mathsf{y}_2)$, a reference to a symmetric key y_2, and if the attribute wrap is set for the handle $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$ (note that the handle is uniquely identified by the nonce x_1) and the attribute $\mathsf{extract}$ is set for the handle $\mathsf{h}(\mathsf{x}_2, \mathsf{y}_2)$ then the attacker may learn the wrapping $\mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1)$, i.e. the encryption of y_2 with y_1.

PKCS#11 is an important example that will be discussed in much more detail in chapter 4.3.5.

3.4.2 Decidability

In this section, we first define the class of *well-moded* rules, using the notation introduced in Section 2.1.1. As in previous sections, we will show again that for this class, when checking the satisfiability of a query, it is sufficient to consider only well-moded terms (Theorem 2). The idea of a well moded term in a way generalizes the ideas of well-formedness we introduces in the previous two sections, though here there is no equational theory. The notion of mode

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

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

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

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

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

$$
\begin{array}{lrcl}
\mathsf{Set_Wrap}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{wrap}(\mathsf{x}_1,\bot) & \rightarrow & \mathsf{wrap}(\mathsf{x}_1,\top) \\
\mathsf{Set_Encrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{encrypt}(\mathsf{x}_1,\bot) & \rightarrow & \mathsf{encrypt}(\mathsf{x}_1,\top) \\
\vdots & & \vdots & \\
\mathsf{UnSet_Wrap}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{wrap}(\mathsf{x}_1,\top) & \rightarrow & \mathsf{wrap}(\mathsf{x}_1,\bot) \\
\mathsf{UnSet_Encrypt}: & \mathsf{h}(\mathsf{x}_1,\mathsf{y}_1);\ \mathsf{encrypt}(\mathsf{x}_1,\top) & \rightarrow & \mathsf{encrypt}(\mathsf{x}_1,\bot) \\
\vdots & & \vdots &
\end{array}
$$

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

Figure 3.3: PKCS#11 key management subset.

is inspired from a paper by Chevalier and Rusinowitch [CR06]. It is similar to the idea of having well-typed rules, but we prefer to call them well-moded to emphasize that we do not have a typing assumption. Informally, we do not assume that a device is able discriminate between bitstrings that were generated e.g. as keys and random data, we simply show that if there is an attack, then there is an attack where bitstrings are used for the purpose they were originally created. This is similar to our approach in the previous sections. For the rules given in Figure 3.3 that model a part of PKCS#11, the notion of mode we consider allows us to bound the size of terms involved in an attack. Unfortunately, the secrecy problem is still undecidable in this setting. Therefore as in previous sections, we introduce a bound, by restricting ourselves to a bounded number of fresh nonces (see Section 3.4.5).

3.4.3 Preliminaries

In the following we consider a set of modes Mode and we assume that there exists a *mode* function $\mathsf{M} : \Sigma \cup \mathcal{A} \times \mathbb{N} \to \mathsf{Mode}$ such that $\mathsf{M}(f,i)$ is defined for every symbol $f \in \Sigma \cup \mathcal{A}$ and every integer i such that $1 \leq i \leq ar(f)$. We also assume that a function $\mathsf{sig} : \Sigma \cup \mathcal{A} \cup \mathcal{X} \cup \mathcal{N} \to \mathsf{Mode}$ returns the mode to which a symbol f belongs. Note that variables and nonces are also uniquely moded. As usual, we extend the function sig to terms: $\mathsf{sig}(t) = \mathsf{sig}(\mathsf{top}(t))$. We will use a rule-based notation $f : \mathsf{m}_1 \times \ldots \mathsf{m}_n \to \mathsf{m}$ for each $f \in \Sigma \cup \mathcal{A}$ and $u : \mathsf{m}$ for $u \in \mathcal{N} \cup \mathcal{X}$ to define the functions M and sig such that $\mathsf{M}(f,i) = \mathsf{m}_i$ for $1 \leq i \leq n$, $\mathsf{sig}(f) = \mathsf{m}$, and $\mathsf{sig}(u) = \mathsf{m}$.

We say that a position $p \neq \varepsilon$ of a term t is *well-moded* if $p = p'.i$ and $\mathsf{sig}(t|_p) = \mathsf{M}(\mathsf{top}(t|_{p'}), i)$. In other words the position in a term is well-moded if the subterm at that position is of the expected mode w.r.t. to the function symbol immediately above it. If a position is not well-moded, it is *ill-moded*. By convention, the root position of a term is ill-moded. A term is well-moded if all its non-root positions are well-moded. A fact ℓ such that $\ell = p(b,t)$ is well-moded if $b \in \{\bot, \top\}$ and t is well-moded. This notion is extended as expected to sets of terms, rules, queries and states.

Note that any term can be seen as a well-moded term if there is a unique mode, e.g. Msg, and any symbol $f \in \Sigma \cup \mathcal{A} \cup \mathcal{X} \cup \mathcal{N}$ is such that $f : \mathsf{Msg} \times \ldots \times \mathsf{Msg} \to \mathsf{Msg}$. However, we will use modes which imply that the message length of well-moded terms is bounded which will allow us to reduce the search space.

Example 8 *We consider the following set of modes:*

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

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

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

The rules described in Figure 3.3 are well-moded w.r.t. the mode and signature function described above. This is also the case of the following rules which represent the deduction

capabilities of the attacker:

$$\begin{array}{rcl} \mathsf{y_1},\ \mathsf{y_2} & \to & \mathsf{senc}(\mathsf{y_1},\mathsf{y_2}) \\ \mathsf{senc}(\mathsf{y_1},\mathsf{y_2}),\ \mathsf{y_2} & \to & \mathsf{y_1} \\ \mathsf{y_1},\ \mathsf{y_2} & \to & \mathsf{aenc}(\mathsf{y_1},\mathsf{y_2}) \\ \mathsf{aenc}(\mathsf{y_1},\mathsf{pub}(\mathsf{z})),\ \mathsf{priv}(\mathsf{z}) & \to & \mathsf{y_1} \\ \mathsf{aenc}(\mathsf{y_1},\mathsf{priv}(\mathsf{z})),\ \mathsf{pub}(\mathsf{z}) & \to & \mathsf{y_1} \\ \mathsf{z} & \to & \mathsf{pub}(\mathsf{z}) \end{array}$$

3.4.4 Existence of a well-moded derivation

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

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

- The set of facts V is necessarily well-moded (Lemma 1). Indeed any instance of a well-moded rule that can be triggered from a state (S,V) where the set of facts V is well-moded, leads to a state (S',V') in which V' is also well-moded.
- Any ill-moded term v' occurring in a term in S is itself deducible (Lemma 2). Note that an instance of a well-moded rule can be triggered from a well-moded state (S,V) to reach an ill-moded state (S',V') (see Example 8). However, in such a situation, the subterm that occurs at an ill-moded position also occurs in S. This is due to the fact that the rules we consider are well-moded.

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

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

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

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

Definition 8 ($\overline{\cdot}^{\mathsf{m}}$, $\overline{\cdot}$) *For each mode* $\mathsf{m} \in \mathsf{M}$ *we define inductively a function* $\overline{\cdot}^{\mathsf{m}}$ *as follows:*

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

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

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

In Proposition 3 we show that this transformation allows us to map any derivation to a well-moded derivation. This well-moded derivation is obtained by applying at each step the same rule, say R. However, while the original derivation may use an instance $\mathsf{R}\theta$ of this rule the transformed derivation will use the instance $\mathsf{R}\theta'$, where θ' is obtained from θ as described in the following lemma.

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

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

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

Proposition 3 *Let* $\mathcal{R}$ *be a set of well-moded rules. Let* (S_0, V_0) *be a well-moded state and consider the derivation:* $(S_0, V_0) \rightsquigarrow (S_1, V_1) \rightsquigarrow \ldots \rightsquigarrow (S_k, V_k)$.

Moreover, for each mode $\mathsf{m} \in \{\mathsf{sig}(t) \mid t \in \mathcal{PT}(\Sigma, \mathcal{N})\}$, *we assume that there exists a well-moded term* t_{m} *of mode* m *such that* $t_{\mathsf{m}} \in S_0$ *(i.e.* t_{m} *is known by the attacker) and* $\overline{\cdot}$ *is defined w.r.t. to these* t_{m}*'s. We have that* $(\overline{S_0}, V_0) \rightsquigarrow (\overline{S_1}, V_1) \rightsquigarrow \ldots \rightsquigarrow (\overline{S_k}, V_k)$ *by using the same rules (but different instances).*

By relying on Proposition 3, it is easy to prove the following result.

Theorem 2 *Let* $\mathcal{R}$ *be a set of well-moded rules. Let* $q_0 = (S_0, V_0)$ *be a well-moded state such that for each mode* $\mathsf{m} \in \{\mathsf{sig}(t) \mid t \in \mathcal{PT}(\Sigma, \mathcal{N})\}$, *there exists a well-moded term* $t_{\mathsf{m}} \in S_0$ *of mode* m*. Let* Q *be a well-moded query that is satisfiable. Then there exists a well-moded derivation witnessing this fact.*

Note that we do not assume that an implementation enforces well-modedness. We allow an attacker to construct derivations that are not well-moded. Our result however states that whenever there exists an attack that uses a derivation which is not well-moded there exists another attack that is. Our result is arguably even more useful than an implementation that would enforce typing: it seems unreasonable that an implementation could detect whether a block has been encrypted once or multiple times, while our result avoids consideration of multiple encryptions, as such a term is ill-moded with the modes given in Example 8.

3.4.5 Decidability result

Theorem 2 on its own is not very informative. As already noted, it is possible to have a single mode Msg which implies that all derivations are well-moded. However, the modes used in our modelling of PKCS# 11 (see Example 8) imply that all well-moded terms have bounded message length. It is easy to see that well-moded terms have bounded message length whenever the graph on modes that is defined by the functions M and sig is acyclic (the graph whose set of vertices is Mode with edges between modes m_i $(1 \leq i \leq n)$ and m whenever there exists a rule

$f : \mathsf{m}_1 \times \ldots \times \mathsf{m}_k \to \mathsf{m}$). Note that for instance a rule which contains nested encryption does not yield a bound on the message size.

Bounded message length is not sufficient for decidability, as shown in section 3.1. Therefore we bound the number of atomic data of each mode, and obtain the following corollary of Theorem 2:

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

Our main application is the fragment of PKCS#11 described in Figure 3.3. We will show how this result is put to use in an implementation is section 4.3.5.

3.5 Abstractions for Fresh Data

In the previous sections we gave a number of decidability results for API languages where the amount of fresh material is bounded, i.e. we have some combination of rules $T;L \to T';L'$ we may use an arbitrary, unbounded number of times in a derivation, and rules $T;L \xrightarrow{new\, n} T';L'$ that my only be fired some bounded number of times. In this section, we show how safe abstractions may be made to these languages to give security proofs for an unbounded number of invocations of both sets of rules, i.e. for unbounded fresh material. By 'safe', we mean that while our abstractions may lead to the discovery of false attacks, i.e. attacks which cannot be executed in the concrete model, if we are able to prove security (i.e. that a query is not derivable), then we can be sure that the the query is not derivable in the concrete model for an unbounded amount of fresh material. In other words, we over-approximate the set of derivations. These abstractions are similar to those used e.g. in the ProVerif protocol analysis tool [Bla02]. The work in this section appeared in a paper with Fröschle [10] (with a slightly different presentation).

3.5.1 Abstraction Definition

We define our abstraction A of an API $\mathcal{R}$, $A(\mathcal{R})$, as follows. For each rule $T;L \xrightarrow{n_1,\ldots,n_k} T';L' \in \mathcal{R}$, define a substitution $\sigma(n_1/m_1, \ldots, n_k/m_k)$ where the m_is are names from $\mathcal{N}$ that do not occur in any rules in $\mathcal{R}$ (including previously calculated abstract rules). The corresponding abstract rule is $T;L \to T;\sigma;L'\sigma$.

In other words, we rename the freshly generated terms on the right hand side so that names are unique across rules, and then we remove the generation of fresh terms from the rule, so that every time the rule is executed, the same names will result on the right hand side.

3.5.2 Stateless APIs with No Equational Theory

We start with a simple case. Assume an API $\mathcal{R}$ with no state, i.e. all rules $\in \mathcal{R}$ have empty l, l', and as observed in chapter 3.2 we can write the API $\mathcal{R}$ so that each rule has a single term on the right hand side.

Theorem 3 *For a stateless API $\mathcal{R}$, set of ground terms (initial knowledge) S and secret term $u \in \mathcal{PT}$, if u is not derivable in $A(\mathcal{R})$, then u is not derivable in $\mathcal{R}$*

Proof. We prove the safety of the abstraction by showing that if u is derivable in $\mathcal{R}$, then it is also derivable in $A(\mathcal{R})$. Take a derivation $u_1,\ldots,u_n$ of u (see definition 3.2), where each u_i corresponds to the right hand side of an application of a single rule r_i in $\mathcal{R}$. There is an obvious possible derivation in $A(\mathcal{R})$ using the corresponding abstract rules r_i', giving a new derivation $u_1',\ldots,u_n'$. Note that u is defined before the execution of any rules, so it cannot contain any names generated by a 'new' during the derivation, so it is not affected by the abstraction transform. To see that this is a legitimate derivation of u, assume for contradiction that this is not the case. Then take the first step j such that the abstract version of the rule used to derive u_j from $S \cup u_1,\ldots,u_{j-1}$ does not derive u_j' from $S \cup u_1',\ldots,u_{j-1}'$. Observe that we have only renamed names from $\mathcal{N}$ that following our semantics, will be chosen freshly during the derivation. Therefore they cannot occur on the left hand side of a rule, concrete or abstract, so pattern matching with the left hand side of the rule cannot fail. Observe further that we do not have any inequality constraints for our rules, so renaming several fresh names in the concrete derivation to the same name in the abstract derivation will not prevent the rule firing. □

Example 9 *The key management API of the Visa Security Module, used in the cash machine network, and shown to be vulnerable to an attack by Bond [Bon01], fits into this class. In the next chapter, we describe a mechanised proof of security of a patched version of the API.*

3.5.3 Stateless APIs with XOR

Consider an extension of the result above to the case with XOR, with an eye on the CCA example discussed in section 3.2. When considering whether the extension is immediate, we may be worried that the effect of our abstraction is to make more pairs of terms equal, which will lead to more cancellations thanks to the nilpotency of XOR. However, observe that, as above, cancellations amongst fresh names cannot break a derivation, since fresh names cannot appear in the left hand side of rules.

Theorem 4 *For an API of the WFX class (see section 3.2) $\mathcal{R}$ and secret term $s \in \mathcal{PT}$, if s is not derivable in $A(\mathcal{R})$, then s is not derivable in $\mathcal{R}$*

In the next chapter, we apply this extension to various configurations of the IBM CCA API.

3.5.4 Stateful APIs

Take an API $\mathcal{R}$ with no equational theory, but with non-empty state, i.e. some non-trivial subset R of rules $T;L \xrightarrow{n_1,\ldots,n_k} T';L'$ in r that have a non-empty L, L'. We say the API $\mathcal{R}$ has *static state* if $\forall r \in R$:

- For every L' on the right hand side of a rule $T;L \xrightarrow{n_1,\ldots,n_k} T';L'$, $names(L') \subseteq n_1,\ldots,n_k$ and $vars(L') = \emptyset$.

Effectively, this means that for APIs like PKCS#11 that use the state facts to manage attributes of objects, newly created objects must be assigned a state when they are created, and after that the state cannot be changed.

We have shown that the same abstraction is sound for this class of APIs [10] (though the presentation is slightly different there, using specific details of PKCS#11 to make a tighter abstraction). In the next chapter we show how we used this abstraction to verify a secure configuration of PKCS#11.

3.6 Discussion

We have shown decidability for three fragments of our API description language. In each case the result is for an unbounded number of command invocations, but a bounded amount of fresh material. In the final section, we showed how abstractions can be used to obtain proofs for the unbounded case. In the next chapter we will show that this is indeed practical, i.e. we are able to obtain proofs without finding false attacks in many cases.

3.6.1 Related Work

Looking at our first decidability result, we note that the a previous attempt at formal analysis of the IBM CCA API used (without proof) a heuristic that splits intruder knowledge into an encrypted and unencrypted part [YAB+05]. Our results for the WFX class proposed in section 3.2 show that their heuristic preserves attack-completeness.

The class of well-formed API rules defined in section 3.3 is closely related to the WFX class. The main difference is that in section 3.3 a model with explicit decryption. This is necessary for modelling key-conjuring, but introduces a more complex equational theory. Note that the two classes are formally incomparable: while well-formed API rules enable explicit decryption, thus potentially more attacks, there are no equality checks between components of the received messages. For example, the following rule

$$\{\!|\mathsf{x}|\!\}_{\mathsf{k}_1}, \{\!|\mathsf{x}|\!\}_{\mathsf{k}_2} \to k_3$$

belongs to the WFX-class but cannot be expressed as a well-formed API rule. We would need to extend API rules with explicit equality checks.

To the best of our knowledge, there exist only two other decidable classes [CLC03, VSS05] for secrecy preservation for protocols with XOR, for an unbounded number of sessions. In both cases, the main difference with our class is that we make restrictions on the combination of functional symbols rather than on the occurrences of variables. As a consequence, our class is incomparable to the two existing ones. In particular, the IBM CCA API cannot be modelled in either of these two other classes.

Turning to the class of rules with mutable state for modelling PKCS#11 introduced in section 3.4, we note that previous attempts at formal models for the PKCS#11 API have not included non-monotonic state [Tsa07, You04]. In addition, there do not seem to be any previous decidability results for protocol models with non-monotonic mutable state. We recover decidability because we show that only a finite number of states can be reached thatnks to our well modedness result. Some APIs, such as the API of the TPM, store a 160 bit value in a state which may be constructed by repeated application of a hash function. Clealry our well modedness result is not immediately helpful here. This is an area of ongoing work (see section 6.3.3).

For our abstractions, the main limitation at the moment is that our abstractions do not directly deal with mutable state. They instead require that one abstracts the configuration to one with static state [10]. We will see in the next chapter that this abstraction is quite useful in practice, but even so, there are configurations that we would not be able to verify using this abstraction. During the editing of this habilitation thesis, a new approach to abstracting state in protocols was published by Mödersheim [MÏ0], designed specifically to handle mutable state that enforces security, e.g. by keeping keys in mutually exclusive 'revoked' and 'usable' states, and giving examples from simple key management APIs. It requires that the state consists of a finite set of sets of which the objects to be abstracted (keys, etc.) may or may not be members. An attempt to use this approach 'as is' on the PKCS#11 example failed because too many false

attacks were produced. Inspection revealed that the problem was that although the handles were correctly abstracted to membership or non-membership of the sets corresponding to the attributes, sensitive, extractable etc., the freshly generated keys were all abstracted to the same value, causing the false attacks. A promising line of work would be to combine ideas from this abstraction with the abstraction presented in section 3.5.4, where we identify keys by their original set memberships.

3.6.2 Future Work

There are a number of open problems. First there is the question of the modelling of encryption and decryption. In the work in section 3.3 we have seen a concrete example of how extending the model with explicit destructors for decryption brings attacks into the model that were not covered by the model in section 3.2. This is perhaps not surprising, since we are outside the 'EV class' of protocols shown by Millen to be relatively complete (compared to the explicit decryption model) when using implicit decryption [Mil03]. Explicit decryption models are more complex both from the point of view of showing decidability (we had to make more restrictions on the form of rules) and in terms of implementation of the procedure (in chapter 4.3.5 we will give results of our implementations, all of which are for implicit decryption). An attractive idea would be to provide the intruder with some extra terms in an implicit encryption model, for example some unknown fresh constants encrypted under keys (as if they have been conjured) and then show that this model is equivalent to the explicit decryption model for some reasonable class.

Second, we have introduced here decidability results motivated by case studies of real APIs. It's hard to extract any general rules about the decidability of APIs as described in our language, beyond the observation that the well-moded, WFX and well-formed classes all follow the same basic idea: they don't assume that the device can distinguish between types, but they show that the API exhibits a sort of 'unambiguity of intention' for each rule - an aspect of good security protocol design that has been commented on before [AN96]. It would be interesting to try to generalize this idea and perhaps use it to give more general decidability results.

Chapter 4

Results on Commercial Devices

In this chapter, we show how we applied the foundational results of the previous chapter to real APIs.

4.1 Visa Security Module

We start with a very simple example. The ‘VISA Security Module’ (VSM) was one of the earliest financial HSM designs, commissioned by VISA to improve PIN processing security in the ATM (cash machine) network, so that member banks might be encouraged to permit processing of each other’s customer PINs [Bon04]. The VSM had very little internal storage but was required to handle a large number of keys, and so used internal master keys (KMs) to encrypt the ‘working keys’ so that they could be securely stored on disk. These working keys were divided into types, depending on their purpose. There were terminal master keys (TMKs), to be used as master keys by the ATMs themselves. There wer terminal communications keys (TCs) to be used for encrypting communications between the VSM and the ATMs. There were also PIN derivation keys (PDKs), which as we mentioned in section 3, are used to derive a customer’s PIN from non-secret data such as his account number (PAN) by encryption.

The designers of the VSM realised that the PDKs should be kept separate from the TCs, because they needed a command that would import a clear TC. So, they used two master keys, km and km2, to encrypt the working keys. The TMKs and PDKs are encrypted under km, and only the TCs are encrypted under km2.

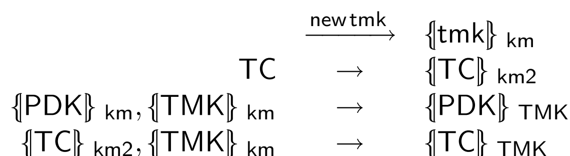

Figure 4.1: Fragment of the VSM API

In figure 4.1 we give a subset of the commands from the VSM API, with uppercase for variables and lowercase for ground values. Taking an initial set of intruder knowledge $I = \{\{\!|\mathsf{pdk}|\!\}_{\mathsf{km}}, \mathsf{pan}\}$, we see there is an easy attack which allows the intruder to obtain the PAN encrypted uner the PDK (and hence the customer’s PIN). He first uses the second command but instead of inputting a new TC, he inputs the customer’s PAN, i.e.

$$\mathsf{PAN} \rightarrow \{\!|\mathsf{PAN}|\!\}_{\mathsf{km2}}$$

He then uses the fourth command, designed for sending a TC to a terminal, to obtain the PIN:

$$\{\!|\mathsf{pan}|\!\}_{\mathsf{km2}}, \{\!|\mathsf{pdk}|\!\}_{\mathsf{km}} \rightarrow \{\!|\mathsf{pan}|\!\}_{\mathsf{pdk}}$$

The true VSM API contains another 8 rules, but like the 4 given, they all fit the WFX class of APIs (note that XOR is not used), hence we may conclude that it is sufficient when searching for an attack to consider only well-formed terms. This enables us to construct a Horn-clause model which can be given to a first-order theorem prover. Indeed, the problem of finding this attack in the VSM API figures as a problem in the leading corpus of problems for such tools, TPTP[1], as problem SWV237. Moreover, we can apply the first abstraction from the previous chapter (§3.5.2) to the generation commands like the first command in figure 4.1. This enables us to pose a verification problem: with the command for inputting a clear TC disabled (the fix proposed by VISA), is the new API secure? This appears as problem SWV238. At the CASC theorem proving competition at FLoC 2010, both problems were used: 9/17 provers entered could find the attack, but only one, E[2] was able to find a model for the verification problem and show that the patched API is indeed secure in this model[3].

4.2 IBM CCA

We described the CCA API in figures 3.1 and 3.2, in an implicit and explicit decryption model respectively. We have only implemented experiments with the implicit encryption model.

As described in section 3, the original design of the API had a vulnerability that allowed an attacker to change the 'type' of a key from PIN derivation to data encryption, allowing any customer's PIN to be obtained from his PAN. IBM proposed a series of fixes for this vulnerability, but it was not clear whether the resulting API was secure. We implemented the decision procedure described in section 3.2 to investigate this. Our efforts to implement the decision procedure using existing tools such as theorem provers (Vampire and E) and model finders (Paradox and Darwin) were unsuccessful. The combinatorial complexity caused by the XOR operation prevents any of the tools from finding a saturation. Since our models have a finite Herbrand universe, and hence are effectively propositional, we considered a manual encoding as a Boolean satisfiability problem, for use with a SAT solver. Unfortunately, for n atoms (our models typically have $n = 12$), we will need 2^n (possible XOR terms) + $2^n \times 2^n$ (possible encryption terms) propositional variables to represent the intruder's knowledge. Additionally, writing out ground versions of the 8 well formed rules in the API will result is an enormous problem, far too large for any SAT solver. In the end, we solved the problem by making a change of representation, and writing an ad-hoc decision procedure for that representation.

4.2.1 Representation of XOR terms

The representation consists of encoding an XOR term as a binary string, accomplished by assigning an (arbitrary) order to the set of finite atoms (or 'base terms'). For example, if we have the ordered set of base terms `KM, KP, KEK, IMP, EXP, DATA, PIN`, we would represent `KEK⊕PIN⊕DATA` as

[1] `www.tptp.org`

[2] `eprover.org`

[3] `http://www.cs.miami.edu/~tptp/CASC/J5/`

		KM	KP	KEK	IMP	EXP	DATA	PIN
KEK ⊕PIN ⊕DATA	→	0	0	1	0	0	1	1

$$\downarrow$$
$$19$$

Hence KEK⊕PIN⊕DATA is represented by the decimal integer 19. Notice the order of the atoms in the term does not matter - we still get the same integer - so our representation effectively normalises the term with respect to the properties of XOR. Notice further that if we have two terms $x1$ and $x2$, that are represented by integers l and m, then the integer representing $x1 \oplus x2$ is just $l \oplus m$. So, we represent XOR using XOR, which is an attractive feature of the representation. For example, we can write the intruder rule

$$x1, x2 \rightarrow x1 \oplus x2$$

by

$$l, m \rightarrow l \oplus m$$

For encryption terms, which consist of one XOR term encrypted by another, we simply shift the bits of the integer representing the message term n places to the left (where n is the number of base terms), and add the integer representing the key. We obtain a unique number in the range $0 \ldots 2^{2n}$ for each encryption term. For example, the term $\{\!|\mathtt{KEK} \oplus \mathtt{PIN} \oplus \mathtt{DATA}|\!\}_{\mathtt{KM} \oplus \mathtt{DATA}}$ is represented by

KM	KP	KEK	IMP	EXP	DATA	PIN	KM	KP	KEK	IMP	EXP	DATA	PIN
0	0	1	0	0	1	1	1	0	0	0	0	1	0

$$\downarrow$$
$$2498$$

4.2.2 The Implemented Procedure

Our decision procedure starts by allocating enough space in memory for $2^{2n} + 2^n$ integers, and setting all these memory locations to 0. Then, set all locations corresponding to the intruder's initial knowledge $\mathcal{S}$ to 1, indicating that the intruder can obtain these terms. For each rule r_i in $\mathcal{R}$, with k terms on the left hand side, encode the operation as a partial function $f_i : \mathbb{N}^k \rightarrow \mathbb{N}$. As a simple example, for the 'Encipher' rule:

Encipher:
$x \ , \ \{\!|xkey|\!\}_{\mathtt{KM} \oplus \mathtt{DATA}} \quad \rightarrow \quad \{\!|x|\!\}_{xkey}$
Assuming KM⊕DATA is represented by the integer value p, we write

$$f : x, [xkey|p], \rightarrow [x|xkey]$$

where the braces [|] denote decomposition of the 2^{2n}-long bitstring into two 2^n length bitstrings. A more complicated example is the **Key Import** command:

Key Import:
$\{\!|xkey|\!\}_{xkek \oplus xtype} \ , \ xtype \ , \ \{\!|xkek|\!\}_{\mathtt{KM} \oplus \mathtt{IMP}} \quad \rightarrow \quad \{\!|xkey|\!\}_{\mathtt{KM} \oplus xtype}$
Assuming KM⊕IMP is represented by the integer q, and KM is represented by r, we write

$$f : [xkey|x], xtype, [xkek|q] \rightarrow [xkey|q \oplus xtype] \quad \text{IF} \quad x = xkek \oplus xtype$$

It will always be possible to write WFX class API rules as integer functions in this way provided the rules are executable, that is provided the HSM itself can work out the values

of the bitstrings it needs to carry out the XORing or encryption/decryption required by the command. If the rule is executable, this creates the integer formula needed to represent the command. It is reasonable to assume that we will only be interested in verifying executable APIs.

To obtain the fixpoint of the intruder's knowledge, we apply each rule exhaustively, looking for combinations of k suitable integers that the intruder already knows, and setting to 1 any location that we can now reach using these rules. We do this for all the rules in an iterative manner until no more rules apply. We check to see if any of the secret terms are now set to 1. If so, we have found an attack. If not, we have verified the API secure.

4.2.3 Results

IBM's first recommendation was to use public key cryptography to import keys. We found an attack on this proposal based on the fact that the imported key may be known to the intruder. We reported this vulnerability to IBM. They pointed out that this vulnerability also exists in PKCS#11 (as indeed it does, see below), but promised to add a security note to the command in the manual. We suggested our own patch for the command based on access control, and were able to verify the security of this new version in our model. We found that recommendations 2 (more separation of duty) and recommendation 3 (check values for keys) were also secure in the model. The paper gives full details [14]. In table 4.1 we summarise the performance of the procedure. We note that these verification problems have now become a standard benchmark for protocol verification tools designed to handle XOR [KT08, CDP09].

Model	Iterations	Terms Derived	Run-Time
Recommendation1_KeyImp	3	17015	0.23
Recommendation1_SymKeyImp	3	13045	3.04
Recommendation2_PersonB	2	4473	8.09
Recommendation2_PersonC	3	4413	12.10
Recommendation2_PersonE	2	1089	2.02
Recommendation3	3	4281	0.19

Table 4.1: Results using our decision procedure to verify the recommendations.

4.3 PKCS#11 Devices

The most commonly used standard for designing token interfaces is RSA PKCS#11 [RSA04]. In this section, we first describe the operation of APIs following the standard and some attacks that were already known before our formal work. We then describe attacks found on the standard following on from our work on well-modedness and decidability 3.4. Finally we describe attacks found on real devices available on the open market.

4.3.1 An Introduction to PKCS#11

RSA PKCS#11 describes its 'Cryptoki' API in just under 400 pages [RSA04]. We give here only a brief description, and we will concentrate on the details that give rise to the category of vulnerabilities found by our symbolic models. 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

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:	$\mathsf{h}(\mathsf{n}_2, \mathsf{k}_2), \{\|\mathsf{k}_1\|\}_{\mathsf{k}_2}$	$\rightarrow$	k_1

Figure 4.2: Wrap/Decrypt attack

conceded in the security discussion in the standard [RSA04, 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, cannot "compromise keys marked 'sensitive', since a key that is sensitive will always remain sensitive", [RSA04, 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.

Clulow first published attacks on PKCS#11 based APIs in 2003 [Clu03b]. One 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 [Clu03b]. To determine the value of a sensitive key, the attacker simply wraps it and then decrypts it, as shown in Figure 4.2. 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.

4.3.2 Vulnerabilities in the Standard

Inspired by Clulow's results, and using our results from section 3.4, we implemented a simple procedure for constructing models for the model checker NuSMV representing various configurations of PKCS#11. We then tried to find ways to prevent Clulow's attacks in the hope of finding a secure configuration. We give here just a few examples of attacks found to give a flavour of the results described in our paper [1]. All these experiments concern the fragment of PKCS#11 given in figure 3.3.

First we follow Clulow's first suggestion for preventing the attack in figure 4.2: attribute changing operations are prevented from allowing a stored key to have both wrap and decrypt set. Note that in order to do this, it is not sufficient merely to check that decrypt is unset before setting wrap, and to check wrap is unset before setting decrypt. One must also add wrap and decrypt to the list of 'sticky' attributes which once set, may not be unset, or the attack is not prevented, [Tsa07]. This means the unset rules will be omitted from the model for these attributes. Having applied these measures, we discovered a previously unknown attack, given in Figure 4.3. The intruder imports his own key k_3 by first encrypting it under k_2, and then unwrapping it. He can then export the sensitive key k_1 under k_3 to discover its value.

Initial state: The intruder knows the handles $h(n_1, k_1)$, $h(n_2, k_2)$ and the key k_3; n_1 has the attributes sensitive and extract set whereas n_2 has the attributes unwrap and encrypt set.

Trace:

SEncrypt:	$h(n_2, k_2), k_3$	$\rightarrow$	$\{\!|k_3|\!\}_{k_2}$
Unwrap:	$h(n_2, k_2), \{\!|k_3|\!\}_{k_2}$	$\xrightarrow{\text{new } n_3}$	$h(n_3, k_3)$
Set_wrap:	$h(n_3, k_3)$	$\rightarrow$	$\text{wrap}(n_3)$
Wrap:	$h(n_3, k_3), h(n_1, k_1)$	$\rightarrow$	$\{\!|k_1|\!\}_{k_3}$
Intruder:	$\{\!|k_1|\!\}_{k_3}, k_3$	$\rightarrow$	k_1

Figure 4.3: Attack using encrypt and unwrap

To prevent the attack shown in Figure 4.3, we add encrypt and unwrap to the list of conflicting attribute pairs. Another new attack is discovered (see Figure 4.4) of a type discussed by Clulow, [Clu03b, Section 2.3]. Here the key k_2 is first wrapped under k_2 itself, and then unwrapped, gaining a new handle $h(n_4, k_2)$. The intruder then wraps k_1 under k_2, and sets the decrypt attribute on handle $h(n_4, k_2)$, allowing him to obtain k_1.

We attempt to prevent the attack in Figure 4.4 by adding wrap and unwrap to our list of conflicting attribute pairs. In addition to the initial knowledge from the first three experiments, we give the intruder an unknown k3 encrypted under k2. Again he is able to affect an attack similar to the one above, this time by unwrapping $\{\!|k_3|\!\}_{k_2}$ twice (see Figure 4.5).

This sample of the attacks found show how difficult PKCS#11 is to configure in a safe way. In a subsequent paper [10], we investigated a proposal for a secure configuration using keyed hashes (HMACs) to bind key attributes to keys when wrapping. This proposal comes from the Eracom Protectserver series of HSMs. The model we used for the API is given in figure4.6. Note that we have assumed that all keys are sensitive and extractable, and further that keys either have the wrap and unwrap attributes set (wu) or the encrypt and decrypt attributes set (ed) (we give some justification for this in the article). Using our model checker, we found a subtle attack: if the intruder is able to compromise a session key (i.e. and 'ed' key), he can compromise all the keys on the device, by turning it into a wrap/unwrap key using a faked

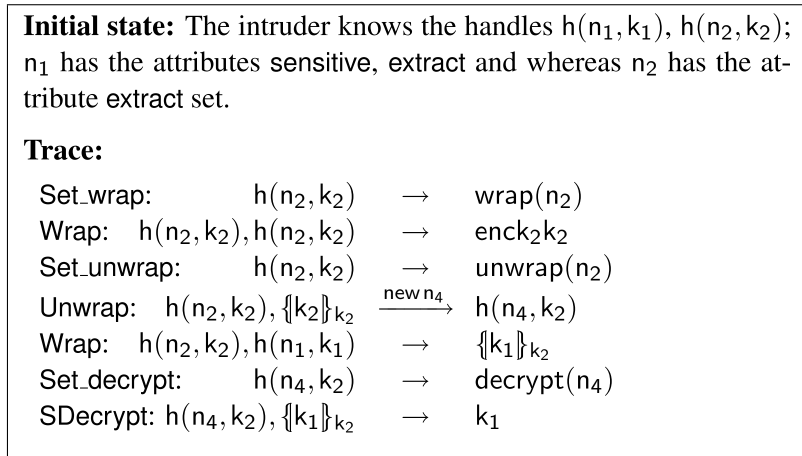

Figure 4.4: Re-import attack 1

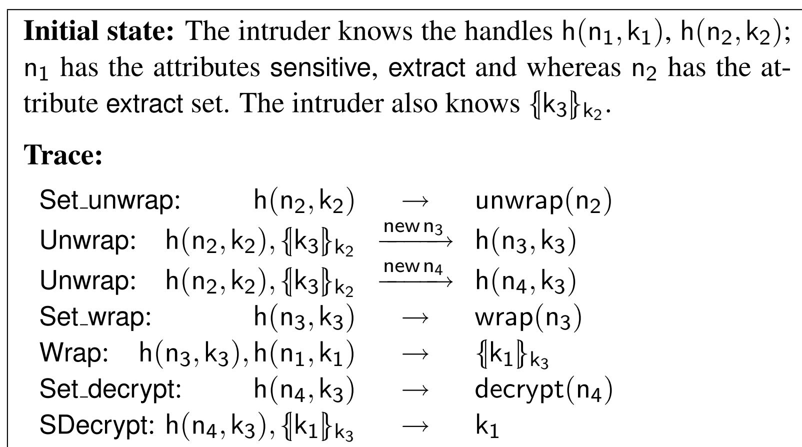

Figure 4.5: Re-import attack 2

HMAC (see figure 4.7). We proposed a fixed version of the API (figure4.8, we give just the altered commands), which we are able to prove secure, using the abstraction of section 3.5.4, for unbounded numbers of fresh keys and handles. This was the first such result for a key management API. Note however that this API exhibits behaviour liable to make cryptographers pf the provable cryptography school nervous, such as allowing key cycles ($\{\!|\mathsf{k}|\!\}_{\mathsf{k}}$) and giving hashes of keys in the clear. We will discuss configurations with cryptographic security proofs in chapters 5 and 6.

4.3.3 Testing on Commercial Devices

In this section we describe Tookan[4], an automated tool that reverse engineers the particular functionality offered by a PKCS#11 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 [6]. We used this tool to investigate the extent to which the attacks form the previous section affect real commercial devices. We enrich the model from section 3.4 to

[4] Tool for cryptoki analysis

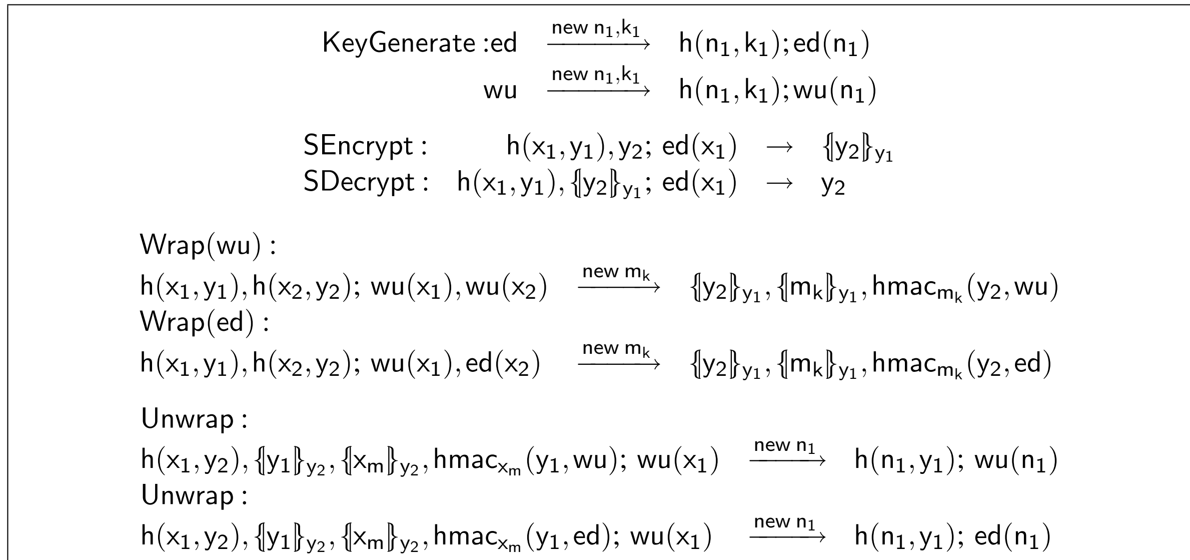

Figure 4.6: Symmetric Key Management subset of the Eracom PKCS#11 API.

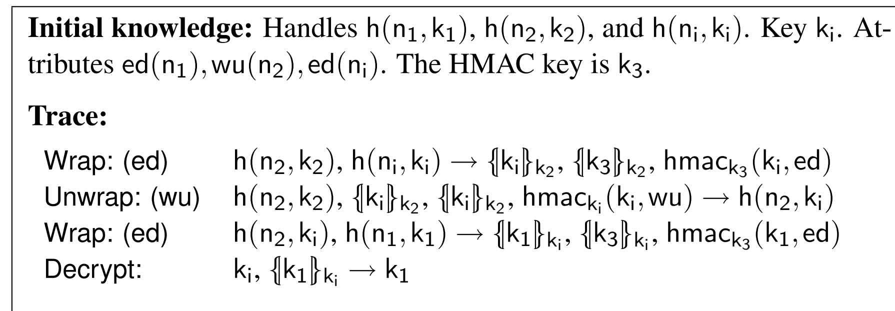

Figure 4.7: Lost session key attack

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).

Our experiments on the tokens proceed following the system diagram in figure 4.9. 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 [AC08]. Model checker output (3) is sent to Tookan for testing on the token (4).

In table 4.2 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 4.3 we give our model for PKCS#11 showing how

```
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 4.2: Syntax of Meta-language for describing PKCS#11 configurations

$$\begin{array}{llll}
\mathsf{KeyGenerate}: & \xrightarrow{\text{new } n,k} & h(n,k); \mathcal{A}(n,B) & (\text{with } B \in \mathcal{G}) \\
\mathsf{KeyPairGenerate}: & \xrightarrow{\text{new } n,s} & h(n,s), pub(s); \mathcal{A}(n,B) & (\text{with } B \in \mathcal{G})
\end{array}$$

$$\begin{array}{llll}
\mathsf{Wrap\,(sym/sym)}: & h(x_1,y_1), h(x_2,y_2);\ wrap(x_1,\top), extract(x_2,\top) & \rightarrow & \{\!|y_2|\!\}_{y_1} \\
\mathsf{Wrap\,(sym/asym)}: & h(x_1,priv(z)), h(x_2,y_2);\ wrap(x_1,\top), extract(x_2,\top) & \rightarrow & \{y_2\}_{pub(z)} \\
\mathsf{Wrap\,(asym/sym)}: & h(x_1,y_1), h(x_2,priv(z));\ wrap(x_1,\top), extract(x_2,\top) & \rightarrow & \{\!|priv(z)|\!\}_{y_1}
\end{array}$$

$$\begin{array}{llll}
\mathsf{Unwrap\,(sym/sym)}: & h(x,y_2), \{\!|y_1|\!\}_{y_2};\ unwrap(x,\top), & \xrightarrow{\text{new } n_1} & h(n_1,y_1);\ \mathcal{A}(n_1,B) \\
 & & & (\text{with } B \in \mathcal{U}) \\
\mathsf{Unwrap\,(sym/asym)}: & h(x,priv(z)), \{y_1\}_{pub(z)};\ unwrap(x,\top), & \xrightarrow{\text{new } n_1} & h(n_1,y_1);\ \mathcal{A}(n_1,B) \\
 & & & (\text{with } B \in \mathcal{U}) \\
\mathsf{Unwrap\,(asym/sym)}: & h(x,y_2), \{\!|priv(z)|\!\}_{y_2};\ unwrap(x,\top), & \xrightarrow{\text{new } n_1} & h(n_1,priv(z));\ \mathcal{A}(n_1,B) \\
 & & & (\text{with } B \in \mathcal{U})
\end{array}$$

$$\begin{array}{llll}
\mathsf{SEncrypt}: & h(x_1,y_1), y_2;\ encrypt(x_1,\top) & \rightarrow & \{\!|y_2|\!\}_{y_1} \\
\mathsf{SDecrypt}: & h(x_1,y_1), \{\!|y_2|\!\}_{y_1};\ decrypt(x_1,\top) & \rightarrow & y_2
\end{array}$$

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

$$\begin{array}{llll}
\mathsf{SetAttribute}: & h(x_1,y_1);\ a(x_1,\bot), \mathcal{A}^{conf(a)}(x_1,\bot) & \rightarrow & ;\ a(x_1,\top), \mathcal{A}^{tied(a)}(x_1,\top) \\
 & & & (\text{with } a \in \mathcal{A} \setminus \mathtt{sticky_off_attributes}) \\
\mathsf{UnsetAttribute}: & h(x_1,y_1);\ a(x_1,\top) & \rightarrow & ;\ a(x_1,\bot), \mathcal{A}^{tied(a)}(x_1,\bot) \\
 & & & (\text{with } a \in \mathcal{A} \setminus \mathtt{sticky_on_attributes})
\end{array}$$

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

$$\mathsf{GetAttribute}:\quad h(n,x); extract(n,b_e), sensitive(n,b_s) \quad \rightarrow \quad x$$
$$\left(\begin{array}{l}
\text{with } b_e, b_s \in \{\bot,\top\} \text{ and} \\
\mathtt{sensitive_prevents_read}(\top) \Rightarrow b_s = \bot \text{ and} \\
\mathtt{unextractable_prevents_read}(\top) \Rightarrow b_e = \top
\end{array}\right)$$

Notation:
- $\mathcal{A} = \{a_1, \ldots, a_m\}$ denotes the (ordered) set of `attributes`
- $B = \{b_1, \ldots, b_m\}$ denotes a *template*, i.e. a set of Boolean values for attributes $\mathcal{A}$
- $\mathcal{A}(n,B)$ stands for $a_1(n,b_1), \ldots, a_m(n,b_m)$ while $\mathcal{A}(n,b)$ stands for $a_1(n,b), \ldots, a_m(n,b)$
- $\mathcal{B}(n,B)$, with $\mathcal{B} = \{a_{j_1}, \ldots, a_{j_k}\} \subseteq \mathcal{A}$ denotes $a_{j_1}(n,b_{j_1}), \ldots, a_{j_k}(n,b_{j_k})$, i.e., the projection of $\mathcal{A}(n,B)$ on $\mathcal{B}$
- $\mathcal{A}^{conf(a)}$ is the subset of attributes $a' \in \mathcal{A}$ conflicting with a, i.e., such that $\mathtt{conflict}(a',a)$
- $\mathcal{A}^{tied(a)}$ is the subset of attributes $a' \in \mathcal{A}$ tied to a, i.e., such that $\mathtt{tied}(a',a)$

Table 4.3: PKCS#11 key management subset with side conditions from the meta-language of table 4.2

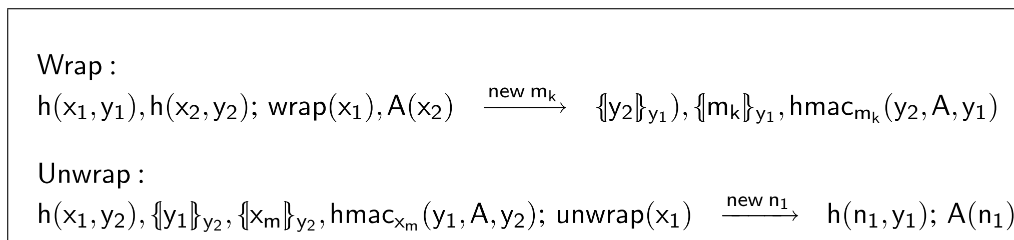

Figure 4.8: Revised Wrap/Unwrap Mechanism for the Eracom API

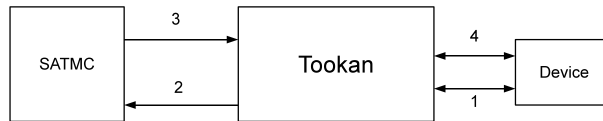

Figure 4.9: Tookan system diagram

it is parametrised by the meta-model. We describe this relationship in more detail below. Note that for brevity, 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[5].

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 4.3.

Functions

Tookan returns a list of `functions` supported, including one important function not modelled in the work described in section 3.4 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.

Key Generation Templates

A major difference between Tookan's model and the model of section 3.4 is that here we take into account *key templates*. In section 3.4, the key generation commands create a key with all its attributes unset (see Figure 3.3). 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

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

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 [RSA04, Table 15]: sensitive for secret keys, for example. As shown in table 4.3, 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 4.3, 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 4.3, 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 4.3, 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 the previous chapter we defined abstractions that can be used for proving security with unbounded numbers of fresh handles and keys. Tookan has an option that builds a model following these abstractions. 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.

4.3.4 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.

4.3.5 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[6]. We also tested our tool on two software simulators, intended for development purposes. Table 4.4 summarises the outcome of the analysis. For each token, we give a summary of the configuration information obtained

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

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[7].

4.3.6 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 [12].

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

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

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.

4.3.7 Attacks

Attack a1 is a wrap/decrypt attack as discussed in section 4.3.1. 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 2:

$$\begin{array}{lrl} \text{Wrap:} & h(n_2,k_2), h(n_1,k_1) \rightarrow & \{\!|k_1|\!\}_{k_2} \\ \text{SDecrypt:} & h(n_2,k_2), \{\!|k_1|\!\}_{k_2} \rightarrow & k_1 \end{array}$$

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:

$$\begin{array}{lrl} \text{CreateObject:} & k_2 \xrightarrow{\text{new } n_2} & h(n_2,k_2) \\ \text{Wrap:} & h(n_2,k_2), h(n_1,k_1) \rightarrow & \{\!|k_1|\!\}_{k_2} \end{array}$$

The attacker can then decrypt $\{\!|k_1|\!\}_{k_2}$ since he knows key k_2. SATMC discovered this variant of the attack on several vulnerable tokens. We note that despite its apparent simplicity, this attack had not appeared before in the PKCS#11 security literature [Clu03b], [12].

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

$$\begin{array}{lrl} \text{Wrap:} & h(n_2,\mathsf{pub}(z)), h(n_1,k_1) \rightarrow & \{k_1\}_{\mathsf{pub}(z)} \\ \text{ADecrypt:} & h(n_2,\mathsf{priv}(z)), \{k_1\}_{k_2} \rightarrow & k_1 \end{array}$$

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.

4.3.8 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. We evaluate the performance of the model checker further in section 4.3.10.

4.3.9 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 [1]. 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 [BV10].

Our starting point is openCryptoki [ope], an open-source PKCS#11 implementation for Linux including a software token for testing. As shown in Table 4.4, 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 [Clu03b], [12]. 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 [1].

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 [10]. 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 4.3.3). 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[8].

4.3.10 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 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 sent vulnerability reports to the manufacturers concerned at least 5 months before publication. Their responses can be viewed at the project website[9].

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 4.3.9, 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

[8] `http://secgroup.ext.dsi.unive.it/cryptokix`
[9] `http://secgroup.ext.dsi.unive.it/pkcs11-security`

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 [Com09a] (for secure storage) and OASIS Key Management Interoperability Protocol (KMIP) [Com09b]. 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 [CC09], [8].
- 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 [CC09], [8].
- Make provision for compliance testing, to weed out poorly implemented tokens, and make testing results publicly available.

	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	✓	✓		✓								
	RSA	SecurID 800	✓	✓	✓	✓					✓	✓	✓	a3
	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

Table 4.4: Summary of results on devices

	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 4.5: Key for table 4.4

Chapter 5

A New Generic API for Key Management

We have seen in the proceeding chapters that the state of the art in key management APIs leaves something to be desired. The most widely used standard, PKCS#11, is very difficult to configure in a secure way and lacks support for modern cryptographic primitives at a native level. In this chapter, we propose an API of our own design and describe its formally proven security properties.

We set out to tackle the problem from a novel direction. We suggest a way to infer functional properties of a security API for a tamper resistant device (TRD) from the security protocols the device is supposed to support. Our API is generic in the sense that it can implement a wide class of symmetric key protocols. The key idea is that confidential data should be stored inside a secure component together with the set of agents that are granted access to it. Then our API will encrypt data only if the agents that are granted access to the encryption key are all also granted access to the encrypted data. To illustrate the generality of our API, we show how to instantiate the API commands for a given protocol using a simple algorithm that has been implemented in Prolog. In particular, we show that our API supports a suite of well-known key establishment protocols.

We propose a formal model for a threat scenario where TRDs may sometimes be connected to a clean host machine, and sometimes to a corrupted one where the attacker can execute arbitrary code. Additionally, the attacker is assumed to have defeated the tamper resistance on some devices, obtaining the long term keys of some users. We show in particular that our API guarantees the confidentiality of any (non public) data that is meant to be shared between honest agents only (honest agents are those whose TRDs are intact). The property holds even when honest agents APIs are controlled by an attacker (in case e.g. an honest user's machine has been infected by a worm). Considering an even stronger attack scenario, where the attacker is also given old confidential keys, we show that our API still provides security provided it is switched to a restricted mode where the API decrypts a cyphertext only when it is able to perform some freshness test. This restricted mode allows us to implement fewer protocols. In particular, of course it does not allow us to implement protocols subject to replay attacks. It does not cover all notions of freshness, but in fact, we discovered that any symmetric key establishment protocol of the Clark and Jacob library [CJ97] can be implemented within the restricted mode, except for protocols that are known to suffer from replay attacks.

The work in the chapter was carried out in Collaboration with Véronique Cortier [8]. A full version containing all proofs is available as a technical report [18].

5.1 Model

The model we will use in this chapter is similar but slightly different from what we have seen before. In particular, we include in the mode predicates modelling the knowledge of various different agents.

5.1.1 Syntax

As usual, messages are represented using a term algebra. We assume a finite set of agents Agent and infinite sets of nonces Nonce and keys Key. We also assume an infinite set of variables Var, among which we distinguish a set VarKey of variables of sort key and a set VarNonce of sort nonce.

$$\begin{aligned} \mathsf{Keyv} &::= \mathsf{Key} \mid \mathsf{VarKey} \\ \mathsf{Noncev} &::= \mathsf{Nonce} \mid \mathsf{VarNonce} \\ \mathsf{Msg} &::= \mathsf{Agent} \mid \mathsf{Keyv} \mid \mathsf{Noncev} \mid \mathsf{Var}\{\mathsf{Msg}\}_{\mathsf{Keyv}} \mid \langle \mathsf{Msg}, \mathsf{Msg} \rangle \\ \mathsf{Handle} &::= h_a^\alpha(\mathsf{Nonce}, \mathsf{Msg}, i, S) \end{aligned}$$

where $i \in \{0,1,2,3\}, S \subseteq \mathsf{Agent}, a \in \mathsf{Agent}, \alpha \in \{r,g\}$. In what follows, we only consider well-sorted substitution. We may write $t_1, t_2, \ldots, t_n$ instead of $\langle t_1, \langle t_2, \langle \ldots, t_n \rangle \ldots \rangle \rangle$.

The API does not give direct access to secret messages but provides the user with a handle that can be used later to indicate to the API to use a specific message. A handle $h_a^\alpha(n,m,i,S)$ represents a reference stored on the API belonging to a for a message m of security level i. The set S represents the set of users that are allowed to access to m. By convention, the special constant All will indicate public data. The nonce n is used to avoid confusion between handles that refer to the same data. The label α distinguishes the handles corresponding to values m generated by the API ($\alpha = g$) from values m received by the API ($\alpha = r$). This distinction allows the API to check for freshness. The values stored inside the TRD will typically be nonces or keys. However, in order to reflect the inability of an TRD to check whether an arbitrary bitstring is a key or not, we *a priori* allow any message to be stored inside the TRD. We consider four levels of security:

- 0: public data
- 1: secret data that are not used for encryption (typically nonces)
- 2: short term keys
- 3: long term keys

We consider the set $\mathcal{P} = \{P_a \mid a \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$ of predicates. P_a with $a \in \mathsf{Agent}$ to represent the knowledge of an agent a. The predicate P_{int} is a special predicate that represents the knowledge of the intruder.

5.1.2 Model

A before our model is a state-based transition system. A rule is an expression of the form

$$P_1(u_1), \ldots, P_k(u_k) \overset{N_1,\ldots,N_p}{\Longrightarrow} Q_1(v_1), \ldots, Q_l(v_l)$$

where the u_i, v_i are messages or handles possibly with variables, the N_i are variables and P_i, Q_i are predicates.

Example 10 *The following set* INTRUDER *of rules represents the ability of an attacker to pair and project and to encrypt and decrypt when he knows the key.*

$$\begin{array}{rcl} P_{\mathsf{int}}(x), P_{\mathsf{int}}(y) & \Rightarrow & P_{\mathsf{int}}(\langle x,y \rangle) \\ P_{\mathsf{int}}(\langle x,y \rangle) & \Rightarrow & P_{\mathsf{int}}(x) \\ P_{\mathsf{int}}(\langle x,y \rangle) & \Rightarrow & P_{\mathsf{int}}(y) \\ P_{\mathsf{int}}(x), P_{\mathsf{int}}(y) & \Rightarrow & P_{\mathsf{int}}(\{x\}_y) \\ P_{\mathsf{int}}(\{x\}_y), P_{\mathsf{int}}(y) & \Rightarrow & P_{\mathsf{int}}(x) \end{array}$$

A state of our execution model is the current knowledge of the intruder and the users. It is formally represented by a family $\{S_b \mid b \in \mathsf{Agent} \cup \{\mathsf{int}\}\}$ where int is a special index representing the intruder. The S_b are sets of messages and handles. Given a family $\mathcal{S}$ of sets and an index $b \in \mathsf{Agent} \cup \{\mathsf{int}\}$, we denote by $\mathcal{S}_b$ the set of $\mathcal{S}$ indexed by b.

The knowledge of the agents evolves following the rules. Given a set of rules $\mathcal{R}$, we say that a state $\mathcal{S}$ is accessible in one step from a state $\mathcal{S}'$, denoted by $\mathcal{S} \Rightarrow_{\mathcal{R}} \mathcal{S}'$ if there exists a rule $P_{a_1}(u_1), \ldots, P_{a_k}(u_k) \stackrel{N_1,\ldots,N_p}{\Longrightarrow} P_{b_1}(v_1), \ldots, P_{b_l}(v_l)$ of $\mathcal{R}$ and a substitution θ such that

- $u_i\theta \in \mathcal{S}_{a_i}$ for any $1 \leq i \leq k$;
- $N_j\theta$ are fresh nonces (that do not appear in $\mathcal{S}$);
- $\mathcal{S}'$ is the smallest family such that $\mathcal{S}_b \subseteq \mathcal{S}'_b$ for any $b \in \mathsf{Agent} \cup \{\mathsf{int}\}$ and $v_i\theta \in \mathcal{S}'_{b_i}$ for any $1 \leq i \leq l$.

$\Rightarrow^*_{\mathcal{R}}$ denotes the reflexive and transitive closure of $\Rightarrow_{\mathcal{R}}$. We may omit $\mathcal{R}$ when the set of rules is clear from the context.

Note that we retrieve the usual deducibility notion by saying that a term m is deducible from a set of terms S, which is denoted by $S \vdash m$, whenever there exists $\mathcal{S}'$ such that $\mathcal{S} \Rightarrow^*_{\mathsf{INTRUDER}} \mathcal{S}'$ and $m \in \mathcal{S}'_{\mathsf{int}}$ where $\mathcal{S}$ is defined by $\mathcal{S}_a = \emptyset$ for any $a \in \mathsf{Agent}$ and $\mathcal{S}_{\mathsf{int}} = S$.

5.2 Presentation of the Generic API

We assume a tamper resistant device with a limited (but for the moment unspecified) amount of memory, capable of symmetric key cryptography. The device is to be deployed to facilitate the execution of symmetric key distribution protocols, and the subsequent use of the session keys established by these protocols. To that end, we design an API that allows users to manage secret data inside the tamper-resistant device (TRD). A user should never have direct access to the stored secret values but should use the API commands to require the TRD to encrypt and decrypt for him, referring to the secrets by their handles.

Our API has simply three commands: generation of new data, encryption and decryption. We give the commands in the language of our formal model below, along with an informal explanation of their operation and of the design rationale. We give an example of the use of the API to implement a protocol, and briefly compare our API to the industry standard RSA PKCS#11

5.2.1 API rules

Our generic API is fully specified by the family of rules Figure 5.1 using the language of our formal model defined in section 2. The set of all the rules is denoted by API. We will explain each rule.

5.2.1.0.1 Generation The API allows a user to generate a new secret nonce or key of security level $i \in \{1,2\}$ for a group $S \subseteq \mathsf{Agent}$ of agents. The result is a handle h binding the name N to the newly generated object K and the key metadata i and S.

Additionally, the API allows the user to generate a public value of security level 0, returning both a handle to the object and the value itself. The idea of storing a public value securely on the device is that this will allow the API to ensure it is only used once to guarantee freshness of sessions keys. We will see how this is done in section 5.6.

5.2.1.0.2 Encryption The encryption command allows the user to encrypt a mixture of public data, level 1 data (i.e. nonces or other secrets), and level 2 data (session keys). This he does by first supplying the handle for the encryption key, and then for all the objects to be encrypted $m_1, \ldots m_n$, he supplies either the handle or (in the case of level 0 data) the value itself. All encrypted data must be of a strictly lower security level than the encryption key used, meaning that session keys (level 2) can only be encrypted under long-term keys (level 3). Additionally, for every object stored on the device that will be included in the encrypted payload, the set S of agents associated with that object must be a superset of the set of agents associated with the encryption key. The rationale is that by encrypting the data under a key associated with a list of agents S', we are effectively making that data available to those agents, so we should only make data available to agents who are supposed to have access to it. In the encrypted packet created by the device, each piece of data to be encrypted will be tagged with both its security level, and with the list of agents allowed access to it.

5.2.1.0.3 Decryption On receiving a ciphertext, a user can call the decrypt function to have it deciphered. He supplies the handle for the decryption key and the encrypted packet. The device works through the packet. For each encrypted object $m_1, \ldots, m_p$, the device examines the tags first and makes the same checks as are made for encryption: the level i must be strictly lower than the security level of the key, and the set S must be a superset of the set S' associated with the decryption key. Then for each object the API returns either a handle to the object which is now stored on the device (when the object is level 1 or 2), or the value of the object itself (when it is of level 0).

There is one further detail to the decryption rule. The user can give some handles to objects of security level 1 or 0 to be used as test values ($m'_j, j \in L$). The decryption command will then only succeed if all of these tests are passed, i.e. values identical to those pointed to by the test handles are found in the ciphertext. This testing behaviour is vital for avoiding replay attacks, as we will show in section 5.6.

5.2.2 Using the API

We now show how the APIs commands can be used to implement a simple protocol from the literature.

Example 11 *Carlsen's Secret Key Initiator Protocol [Car94, Figure 2]*

$$
\begin{array}{lllll}
1. & A & \rightarrow & B & : \mathrm{A, Na} \\
2. & B & \rightarrow & S & : \mathrm{A, Na, B, Nb} \\
3. & S & \rightarrow & B & : \{\mathrm{K_{ab}, N_b, A}\}\mathrm{K_{bs}}, \{\mathrm{N_a, B, K_{ab}}\}\mathrm{K_{as}} \\
4. & B & \rightarrow & A & : \{\mathrm{N_a, B, K_{ab}}\}\mathrm{K_{as}}, \{\mathrm{N_a}\}\mathrm{K_{ab}}, \mathrm{N'_b} \\
5. & A & \rightarrow & B & : \{\mathrm{N'_b}\}_{\mathrm{Kab}}
\end{array}
$$

The aim of the protocol is to establish a fresh session key K_{ab} for participants a *and* b *using a key server* s. *In the first message,* a *sends her name and a fresh nonce to* b. *In message 2,* b

$$\overset{N,K}{\Rightarrow} P_a(h_a^g(N,K,i,S)) \quad i \in \{1,2\}$$ **(Secure Generate)**

$$\overset{N,K}{\Rightarrow} P_a(K), P_a(h_a^g(N,K,0,\mathsf{All}))$$ **(Public Generate)**

$$P_a(h_a^\alpha(X_n,X_k,i_0,S_0)), P_a(m_1),\ldots,P_a(m_n) \Rightarrow P_a(\{m'_1,\ldots,m'_n\}_{X_k})$$ **(Encrypt)**

where

- $\alpha \in \{r,g\}$, $k \in \mathcal{N}$, $a \in S_0 \subseteq \mathsf{Agent}$, $i_0 \in \{2,3\}$, $X_k \in \mathsf{VarKey}$;
- $m'_j = m_j, 0$ if $m_j \in \mathsf{Var}$ is a variable.
- $m'_j = X_{k_j}, i_j, S_j$ with $i_j < i_0$ and $S_0 \subseteq S_j$ if $m_j \in \mathsf{Handle}$ is a handle of the form $h_a^{\alpha_j}(X_{n_j}, X_{k_j}, i_j, S_j)$.

$$P_a(h_a^\alpha(X_n,X_k,i_0,S_0)), P_a(\{m_1,\ldots,m_p\}_{X_k}), \bigcup_{j\in L} P_a(m'_j) \overset{N_1,\ldots,N_p}{\Rightarrow} \cup_{j\notin L} P_a(m'_j)$$

(Decrypt)

where

- $L \subseteq \{1,\ldots,p\}$, $\alpha \in \{r,g\}$, $k \in \mathcal{N}$, $a \in S_0 \subseteq \mathsf{Agent}$, $i_0 \in \{2,3\}$, $N_1,\ldots,N_k \in \mathsf{VarNonce}$;
- for any $j \in L$, $m'_j = h_a^g(X_{n_j}, X_j, 0, \mathsf{All})$ if m_j is of the form $X_j, 0$ and $m'_j = h_a^g(X_{n_j}, X_j, i_j, S_j)$ if m_j is of the form i_j, S_j, X_j with $i_j \geq 1$.
- for any $j \notin L$, $m'_j = x_j$ if m_j is of the form $x_j, 0$ (data of security level 0 are given to the user) and $m'_j = h_a^r(N_j, y_{k_j}, i_j, S_j)$ if m_j is of the form y_{k_j}, i_j, S_j with $i_j \geq 1$, $i_j < i_0$ and $S_0 \subseteq S_j$.

Figure 5.1: Formal Description of API rules

forwards these values together with his own fresh nonce to the server s. *The server generates* K_{ab} *and encrypts it first for* b, *under* b*'s long term key* K_{bs}*, in a package together with his nonce and* a*'s name, and then for* a, *under her long term key* K_{as}*, together with her nonce and* b*'s name. The server sends both packets to* b. *In message 4,* b *forwards to* a *her encrypted package,* a*'s nonce* N_a *encrypted under the session key* K_{ab}*, and a further fresh nonce* N'_b*. In message 5,* a *returns this nonce encrypted under* K_{ab}*. Now both* a *and* b *should accept* K_{ab} *as the session key.*

To implement this protocol using our API, a *should have a handle* $h_a^r(n'_{KAS}, k_{as}, 3, \{a,s\})$ *to the key* k_{as} *of level 3. The agent* a *can execute its first protocol's rule by using the following API command:*

$$\overset{N,N_A}{\Rightarrow} P_a(N_A), P_a(h_a^g(N,N_A,0,\mathsf{All}))$$

where N, N_A *are nonce variables.* a *obtains both a fresh (public) nonce* N_A *and a handle* $h_a^g(N,N_A,0,\mathsf{All})$ *for it.*

a*'s second step in the protocol (rule 5) can also be performed using the API's commands. Upon receiving a message of the form* $\{\mathrm{N_a}, \mathrm{a}, \mathrm{K_{ab}}\}\mathrm{K_{as}}, \{\mathrm{N_a}\}\mathrm{K_{ab}}, \mathrm{N'_b}$*,* a *can split it into two parts* x_1*,* x_2 *and* x_3*. Intuitively,* x_1 *should correspond to* $\{\mathrm{N_a}, \mathrm{b}, \mathrm{K_{ab}}\}\mathrm{K_{as}}$*, the part* x_2 *should*

correspond to $\{\mathrm{N_a}\}\mathrm{K_{ab}}$ *and* x_3 *should correspond to* N'_{b}. *Then* a *can decrypt* x_1 *using the following decryption command (with* $L = \{1\}$*, that is the first component should be checked):*

$$P_a(h^r_a(N'_{KAS}, K_{as}, 3, \{a,s\})), P_a(\{N_A, 0, y, 0, x, 2, \{a,b,s\}\}_{K_{as}}),$$
$$P_a(h^g_a(N, N_A, 0, \mathsf{All}))$$
$$\stackrel{N'}{\Rightarrow} P_a(y), P_a(h^r_a(N', x, 2, \{a,b,s\}))$$

where $N, N_A, N'_{kas}, K_{as}, x, y$ *are variables.* a *can check that* y *is equal to* b *and receives a handle* $P_A(h^r_A(N', x, 2, \{a,b,s\}))$ *that refers to* x *and should correspond to the inside key* K_{ab}. *Then* a *can decrypt* x_2 *using the following decryption command (with again* $L = \{1\}$*, that is the first component should be checked):*

$$P_a(h^r_a(N', K_{ab}, 2, \{a,b,s\})), P_a(\{N_A, 0\}_{K_{ab}}), P_a(h^g_a(N, N_A, 0, \mathsf{All})) \Rightarrow$$

where N, N_A, N', K_{ab} *are variables. If the command succeeds, the agent* a *knows that the second component* x_2 *indeed corresponds to* $\{\mathrm{N_a}\}\mathrm{K_{ab}}$. *Then* a *can build her message for* b *by using the following encryption command.*

$$P_a(h^r_a(N', K_{ab}, 2, \{a,b,s\})), P_a(x_3) \Rightarrow P_a(\{x_3, 0\}_{K_{ab}})$$

where N', K_{ab} *are variables.*

5.2.3 Comparison with PKCS#11

We now compare our design with the RSA standards PKCS#11 (see section 3.4). PKCS#11-based APIs have been shown to be vulnerable to a variety of attacks whereby sensitive keys are compromised [Clu03b],[12]. Our API has several features designed specifically to counter these kinds of threats. Firstly, we insist on an encryption scheme whereby data from the host machine and secret data from inside the TRD are tagged differently when encrypted to avoid confusion. PKCS#11 does not do this, and this confusion is exploited by many of the known attacks. Secondly we insist that keys are stored with specific roles, either as session keys or long term keys, and these roles cannot be changed. Allowing the roles of keys to change (signified by their *attributes* in PKCS#11) is another major source of vulnerabilities in the Cryptoki API. Finally, we store the identities of agents for whom a key is intended to be used inside the TRD, and include these identities as tags in our encryption scheme. PKCS#11 makes no such provision, but it seems necessary in order to obtain security properties which are preserved when some TRDs are compromised.

5.3 Using the Generic API to Implement a Protocol

In this section we show how the generic API can be used to implement symmetric key protocols, including in particular symmetric key distribution protocols from the venerable Clark-Jacob survey [CJ97].

To deduce the API commands, we first require the protocol to be specified in a manner following e.g. [RT01], that is each protocol step is given as a rule

$$A : u \xrightarrow{new\ \mathcal{N}} v$$

A is the agent who plays the role. The u, v are terms in our algebra from section 2, where agent names, keys and nonces are given as variables. The set $\mathcal{N}$ of nonce and key variables

represents freshly generated data. In addition we require the terms in the protocol to be tagged with their type (agent, nonce, key or message), and nonces and session keys must be tagged with the name of the agent which generated them, their level (0 for a nonce is sent in the clear, 1 for a nonce only ever sent encrypted, 2 for a session key) and the set of participants expected to share secrets. Everything generated by the participants during the protocol (i.e. keys and nonces) will be assumed to be shared between all participants. We will not attempt to deduce whether a nonce is kept secret from the server, or secret from Bob, etc. Tagged nonces in a protocol will be written $n(A,N_A,L,Set)$, where A is the agent, N_A the name for the nonce, L the level and Set the set. Similarly, we have tagged keys $k(S,K_A,L,Set)$, agent names $a(A)$ and message variables $m(X)$. This tagging can be easily guessed by a user reading the protocol but could also be found automatically (for example, by trying several possible taggings).

Given a tagged term t, $\mathsf{un}(t)$ denotes its untagged version obtained from t by removing all the tags. For example, $\mathsf{un}(n(A,N_A,L,Set)) = N_A$. Moreover, given a term t, we denotes by $\bar{t}$ the term obtained from t by replacing each subterm $\{u\}_v$ of t by the variable $X_{\{u\}_v}$. The function $\bar{\cdot}$ is a one-to-one mapping.

5.3.1 Algorithm

We give a simple algorithm for constructing API commands for a given protocol below in informal pseudocode. The algorithm relies on a global store H of handles that each participant in the protocol will expect to have when a protocol step is executed. This store has an initial state. For example, for the three-party key exchange protocols, the initial state is

$h^r_a(N_{Kas}, kas, 3, \{a,s\})$ `% A handle for kas`
$h^r_b(N_{Kbs}, kbs, 3, \{b,s\})$ `% B handle for kbs`
$h^g_s(N'_{Kas}, kas, 3, \{a,s\})$ `% S handle for kas`
$h^g_s(N'_{Kbs}, kbs, 3, \{b,s\})$ `% S handle for kbs`

Note that where we give agent names a, b, and s as ground terms these should be interpreted as parameters - it is up to the implementer to equip the TRD with the handles and API for the roles of a, b or s as appropriate.

Implementing a single protocol step requires:

1. zero or more Decryption Commands, followed by
2. zero or more Generate commands, followed by
3. zero or more Encryption Commands

To construct the commands for rule $u \xrightarrow{new\ \mathcal{N}} v$ played by agent A:

Decryption

For each encryption $\{m_1,\ldots,m_p\}_{Xk}$ occurring in u:

Retrieve $h^\alpha_A(N,X_k,j,Set)$ from store H. If none exists then the algorithm fails. The protocol is actually not executable since the agent does not have the decryption key (and encrypted packets for forwarding must be marked as message variables).

Select the first m_i such that $m_i = n(A,X,I,Set)$ and $h^g_A(N',X,I,Set)$ is in the handle store and set $L = [P_A(h^g_A(N',X,I,Set))]$. If no such m_i exists, and $j = 3$ then output the warning "missing freshness test" and set $L = []$. We will see later that tests ensure a higher level of security.

Add decryption command of the form

$$P_A(h_A^\alpha(N, X_k, j, Set)), P_A(\{\overline{\mathsf{un}(m_1)}, \ldots, \overline{\mathsf{un}(m_p)}\}_{X_k}), L \overset{N_1,\ldots,N_p}{\Rightarrow} \bigcup_{j \neq i} P_A(m'_i)$$

where the m'_i are defined from the $\overline{\mathsf{un}(m_i)}$ as in section 5.2.1.

Generate

For each $n(A, X, 0, Set) \in \mathcal{N}$, add generate command

$$\overset{N,X}{\Rightarrow} P_A(X), P_A(h_A^g(N, X, L, Set))$$

Add $h_A^g(N, X, 0, Set)$ to the handle store H.

For each $n(A, X, i, Set) \in \mathcal{N}$, $i \in \{1, 2\}$, add generate command

$$\overset{N,X}{\Rightarrow} P_A(h_A^g(N, X, i, Set))$$

Add $h_A^g(N, X, i, Set)$ to the handle store H.

Encryption

For each encryption $\{m_1, \ldots, m_p\}_{Xk}$ occurring in v:

Retrieve $h_A^\alpha(N, X_k, i, Set)$ from the handle store H.
Add encryption command of the form

$$P_A(h_A^\alpha(N, X_k, i, S)), P_A(m'_1), \ldots, P_A(m'_k) \Rightarrow P_A(\{\overline{\mathsf{un}(m_1)}, \ldots, \overline{\mathsf{un}(m_k)}\}_{X_k})$$

where m'_i is

- h if $m_i = n(A, Y, 1, S)$ is a level 1 nonce with a handle $h = h_A^\alpha(N', Y, 1, S) \in H$
- h if $m_i = k(A, X, 2, S)$ is a key with a handle $h = h_A^\alpha(n', Y, 2, S) \in H$
- $\overline{\mathsf{un}(m_i)}$ if m_i is an agent name, a nonce of level 0, a message variable or a cyphertext.
- The algorithm fails otherwise, that is, in case m_i is of level security 1 or 2 with no corresponding handle in the store (or if m_i is of higher security level). This corresponds to a case where the agents is enable to build the message thus the protocol is not executable.

We consider encrypted terms to be terms of level 0. In this way we can treat nested encryptions by recursively generating encryption commands, treating the innermost encryption first.

5.3.2 Example

Consider the role of A in the Carlsen's Secret Key Initiator Protocol. Using our algorithm, we retrieve the API commands presented in example 11.

A Prolog implementation has been tested on all the protocols in section 6.3 of the Clark-Jacob survey, excepting those where freshness is assured by timestamps. The Prolog source and the results are available via http [1]. We give the results in section 4.3.5, after we discuss the security properties of our API.

[1] http://www.lsv.ens-cachan.fr/GenericAPI/

5.4 Security of the API

Recall that our API is designed to be used on a device which may sometimes be connected to a corrupted host machine, and sometimes to a ‘clean’ machine. When all machines involved in a run of a protocol are ‘clean’ the formal threat model reduces to the so-called Dolev-Yao model: all network traffic goes through the intruder, but computations on honest users’ machines remain secure. In this case, our API merely implements the protocol, and does not provide extra security. We are interested in what guarantees our API offers when one or more of the machines involved in a protocol run are corrupted, but the TRDs are still intact. If a host machine is corrupted, then all the public data on the machine (level 0 terms in our model) is assumed to be lost. We want to show that secret terms stored on the device (level ≥ 1) remain secret. Further, we want to show that session keys established while the device was connected to the corrupted machine can still be trusted, even if some (other) session keys have been lost. These properties give (we claim) an intuitively easy to understand security policy for the API. Furthermore, they are precisely the properties that are violated by previously discovered attacks on existing APIs [12, 10]. We will prove that our API preserves these properties.

We first give a precise formal model of the threat scenario. The aim of the API is to protect the confidentiality of secret data for a certain group of users, called *honest agents*. Let H be such a set. H is assumed to be a fixed set in the remaining of the paper. Agents that are not in H are said to be *compromised*.

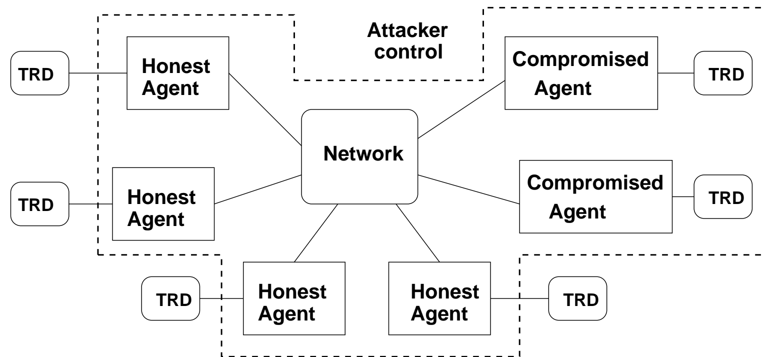

Figure 5.2: Threat model. The attacker controls the network, all machines, and has obtained access to the memory of some compromised agents’ TRDs

We assume the intruder to have complete control not only of the network, but also of the machines of the honest users (using viruses or worms for example). We also assume that he has access to the long-term secret values of some compromised users (by defeating the tamper resistance of their devices or some other means). The only trusted secure parts are the secure storage components (TRDs) of the honest users, managed by the API (see Figure 5.2). This can be easily modeled by adding the following set CONTROL of rules

$$P_a(x) \;\Rightarrow\; I(x) \qquad (5.1)$$

$$I(x) \;\Rightarrow\; P_a(x) \qquad (5.2)$$

$$P_b(h_b^{\alpha}(x,y,i,S)) \;\Rightarrow\; I(y) \qquad (5.3)$$

for any $a,b \in \mathsf{Agent}$ such that $b \notin H$, $i \in \{1,2,3\}$, $\alpha \in \{r,g\}$ and $S \subseteq \mathsf{Agent}$. This models the fact that the intruder can access any value known by the user (including handles) and can also store messages on users machines in order to then communicate with the API. The last rule indicates the fact that the intruder is given any value that may be stored in a TRD of a

compromised agent. Given a state $\mathcal{S}$ of our execution model and by abuse of notation, we write $t \in \mathcal{S}$ (resp. $\mathcal{S} \vdash t$) instead of $t \in \bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b$ (resp. $\bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \vdash t$).

When the API is initialized, keys of level 3 are generated and distributed between the secure components managed by APIs and users are given handles to these keys. These keys are initially unknown to the intruder. Thus we say that a state $\mathcal{S}$ is *initial* if $\mathcal{S}_{\mathsf{int}} \subseteq \mathsf{Agent} \cup \mathsf{Nonce} \cup \mathsf{Key}$ is a set of atomic messages and if for any $a \in \mathsf{Agent}$, the set $\mathcal{S}_a$ only contains handles of the form $h_a^\alpha(n,k,i,S)$ with $n \in \mathsf{Nonce}$, $k \in \mathsf{Nonce} \cup \mathsf{Key}$ and such that

- n, k do not appear in $\mathcal{S}_{\mathsf{int}}$,
- $h_a^\alpha(n,k,i,S) \in \mathcal{S}_a$, $h_b^{\alpha'}(n',k,i',S') \in \mathcal{S}_b$, $a, b \in H$ imply $i = i'$ and $S = S'$: honest tokens are consistently set up.

The security of the API can be expressed as follows: given a state $\mathcal{S}$ of the system, secret data of honest users should not be known to the intruder. Secret data of honest users are values k for which there are handles of the form $h_a^\alpha(n,k,i,S)$ where S is a subset of honest users. This is reflected by the following formula:

$$\forall a \in H, \forall x,y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r,g\}, \forall S \subseteq H \quad \mathcal{S} \vdash h_a^\alpha(x,y,i,S) \Rightarrow \mathcal{S} \not\vdash y \quad (\mathbf{Sec})$$

This also ensures that whenever a value k is stored for a set S of honest users, then k is indeed a key or a nonce.

We can show that our generic API satisfies the security property **Sec** as the API is correctly initialized. This is an important feature since it guarantees confidentiality of sensitive data for our API even if the intruder has control of all honest users machines.

Theorem 5 *Let $\mathcal{S}_0$ be an initial state. Then for any state $\mathcal{S}$, accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API} \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}} \mathcal{S}$, we have that $\mathcal{S}$ satisfies property* **Sec**.

Proof: (sketch) We first start by adding more power to the intruder, providing him access to any value m stored on a dishonest device and providing him access to any value m for which there exists a handle $h_a^\alpha(n,m,i,S)$ where some participant of S is dishonest, even if a is honest, meaning that the value m is stored on a non-compromised TRD. Formally, we write $\mathcal{S} \vdash^* t$ when $\bigcup_{b \in \mathsf{Agent} \cup \{\mathsf{int}\}} \mathcal{S}_b \cup \{m \mid h_a^\alpha(n,m,i,S) \in \mathcal{S}, S \not\subseteq H, a \in \mathsf{Agent}\} \cup \{m \mid h_a^\alpha(n,m,i,S) \in \mathcal{S}, a \notin H\} \vdash t$.

We then consider a stronger version of property **Sec**.

$$\forall a \in H, \forall x,y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r,g\}, \forall S \subseteq H$$
$$\mathcal{S} \vdash^* h_a^\alpha(x,y,i,S) \Rightarrow \mathcal{S} \not\vdash^* y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \quad (\mathbf{Sec*})$$

Intuitively, the property **Sec*** ensures in addition to **Sec** that values stored for honest agents are always of type Nonce or Key, and or non-deducible, even for the stronger version of deducibility $\vdash^*$.

We show in the full paper [18] that **Sec*** together with the two following properties are invariant by application of rules of $\mathsf{API} \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$:

$$\begin{gathered}\forall u,k \in \mathsf{Msg}, \mathcal{S} \vdash^* \{u\}_k \Rightarrow \mathcal{S} \vdash^* k \text{ or} \\ \exists n \in \mathsf{Msg}, \exists S \subseteq H, \exists i \in \{0,1,2,3\}, \exists a \in H, \exists \alpha \in \{r,g\}, \text{ s.t. } \mathcal{S} \vdash^* h_a^\alpha(n,k,i,S) \\ \exists m_1,\ldots,m_p \in \mathsf{Msg}, \exists i_1,\ldots,i_p \in \{0,1,2,3\}, \\ \exists S_1 \ldots, S_p, S \subseteq S_j, \forall j, \text{ s.t. } u = i_1, S_1, m_1, \ldots, i_p, S_p, m_p, \\ \text{and } \forall i_j \geq 1, m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and } \exists n_j \in \mathsf{Nonce}, \exists b \in H, \exists \alpha' \in \{r,g\}, \mathcal{S} \vdash^* h_b^{\alpha'}(n_j,m_j,i_j,S_j)\end{gathered} \quad (\mathbf{Enc})$$

Intuitively, property **Enc** ensures that any encrypted message sent over the network can either be build by the intruder, or corresponds to the output of an API, with the corresponding handles for each value.

$$\forall k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i_1, \ldots, i_p \in \{0,1,2,3\}, \forall j \text{ s.t. } i_j = 0$$
$$\mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \Rightarrow \mathcal{S} \vdash^* m_j \quad (\textbf{Enc0})$$

Intuitively, property **Enc0** ensures that for any encrypted message, any underlying plain-text value that is indicated as security level 0 is indeed a public value.

$$\forall n, n', m \in \mathsf{Msg}, \forall i, i' \in \{0,1,2,3\}, \forall \alpha, \alpha' \in \{r, g\}, \forall a, b \in H$$
$$\mathcal{S} \vdash^* h_a^{\alpha}(n,k,i,S) \text{ and } \mathcal{S} \vdash^* h_b^{\alpha'}(n',k,i',S') \Rightarrow i = i' \text{ and } S = S' \quad (\textbf{SecDegree})$$

Intuitively, property **SecDegree** ensures that honest tokens agree on security levels and sets of agents being granted access to common values. This last property is not needed for proving **Sec** but will be needed in the next section.

Theorem 5 then easily follows since any initial state satisfies the four properties **Sec***, **Enc**, **Enc0**, and **SecDegree** and property **Sec** is an immediate consequence of property **Sec***.

5.5 Security of the API under compromised handles

We have seen in the previous section that our API protects any data for which there is an honest handle $h_a^{\alpha}(n,k,i,S)$ with $S \subseteq H$. Imagine that some secret data is leaked to the attacker by some out of model means, possibly using a brute force attack, a physical side-channel attack or some other means. So, the attacker knows both $h_a^{\alpha}(n,k,i,S)$ and k. Then the attacker can learn any data of security level strictly smaller than the security level i of k, stored by the API of a, for which he has a handle $h_a^{\alpha'}(n',k',j,S')$ with $j < i$, $S \subseteq S'$. Indeed, the attacker can use the encryption command of the API

$$\mathsf{Encrypt} \quad h_a^{\alpha}(n,k,i,S) \quad h_a^{\alpha'}(n',k',j,S')$$

and obtain the cyphertext $\{j, S', k'\}_k$ thus k'. Note that this attack requires the attacker to control the API of a and only allows handles of strictly lower security level to be compromised, for values that are granted to sets of participants that include all agents having a granted access to the lost key k.

We show in this section that, even when some keys are lost, all other keys are still protected, except those that are of strictly lower level for larger sets of participants.

Let K be a set of keys (or nonces) that represent the values learned by the attacker (e.g. by brute force attacks) and let $\mathcal{S}$ be a state of the system. We define the security degree of K in $\mathcal{S}$ to be the set $\mathsf{SecDegree}(K,\mathcal{S})$ of security degrees of keys in K as they appear in handles of honest tokens.

$$\mathsf{SecDegree}(K,\mathcal{S}) = \{(i,S) \mid \exists n \in \mathsf{Msg}, \exists a \in H, \exists k \in K, \exists \alpha \in \{r,g\} \quad h_a^{\alpha}(n,k,i,S) \in \mathcal{S}\}$$

We say that a security degree (i_1,S_1) is strictly smaller than (i_2,S_2), denoted by $(i_1,S_1) < (i_2,S_2)$, if $i_1 < i_2$ and $S_2 \subseteq S_1$. By abuse of notation, we write $(i_1,S_1) < \mathsf{SecDegree}(K,\mathcal{S})$ if there exists $(i_2,S_2) \in \mathsf{SecDegree}(K,\mathcal{S})$ such that $(i_1,S_1) < (i_2,S_2)$. We have seen that knowing the keys of K, the attacker can easily learn any key of strictly smaller degree than keys in K. We can show that the confidentiality of any other key is still preserved by our generic API.

Theorem 6 *Let $\mathcal{S}_0$ be an initial state. Let $\mathcal{S}_1$ be any state accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API}\cup\mathsf{INTRUDER}\cup\mathsf{CONTROL}} \mathcal{S}_1$. Let K be any set of keys and nonces and let $\mathcal{S}'_1$ be the state $\mathcal{S}_1$ in which the intruder learns all keys in K, that is $\mathcal{S}'_{1,a} = \mathcal{S}_{1,a}$ for any $a \in \mathsf{Agent}$ and $\mathcal{S}'_{1,\mathsf{int}} = \mathcal{S}_{1,\mathsf{int}} \cup K$. Then, for any state $\mathcal{S}$ accessible from $\mathcal{S}'_1$, that is $\mathcal{S}'_1 \Rightarrow^*_{\mathsf{API}\cup\mathsf{INTRUDER}\cup\mathsf{CONTROL}} \mathcal{S}$, we have that the following property* **SecW** *is satisfied by $\mathcal{S}$:*

$$\forall a \in H, \forall x,y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r,g\}, \forall S \subseteq H$$
$$\mathcal{S} \vdash h^{\alpha}_a(x,y,i,S) \text{ and } (i,S) \not< \mathsf{SecDegree}(K,\mathcal{S}_1) \Rightarrow \mathcal{S} \not\vdash y \text{ or } y \in K \quad (\mathbf{SecW})$$

Proof: (sketch) As for the proof of Theorem 5, we first give more power to the intruder, providing him access to any value m stored on a dishonest device and to any value m for which there exists a handle $h^{\alpha}_a(n,m,i,S)$ where some participant of S is dishonest or where $(i,S) < \mathsf{SecDegree}(K,\mathcal{S}_1)$, even if a is honest, meaning that the value m is stored on a non-compromised TRD. Formally, we write $\mathcal{S} \vdash^W t$ when $\bigcup_{b\in\mathsf{Agent}\cup\{\mathsf{int}\}} \mathcal{S}_b \cup \{m \mid h^{\alpha}_a(n,m,i,S) \in \mathcal{S}, S \not\subseteq H, a \in \mathsf{Agent}\} \cup \{m \mid h^{\alpha}_a(n,m,i,S) \in \mathcal{S}, a \notin H\} \cup \{m \mid h^{\alpha}_a(n,m,i,S) \in \mathcal{S}, (i,S) < \mathsf{SecDegree}(K,\mathcal{S}_1)\} \vdash t$.

We prove that a stronger invariant is preserved by our API. We consider the three following properties:

$$\forall a \in H, \forall x,y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall \alpha \in \{r,g\}, \forall S \subseteq H$$
$$\mathcal{S} \vdash^W h^{\alpha}_a(x,y,i,S) \text{ and } (i,S) \not< \mathsf{SecDegree}(K,\mathcal{S}_1) \Rightarrow y \in K \text{ or}$$
$$\mathcal{S} \not\vdash^W y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \quad (\mathbf{SecW^*})$$

$$\forall u,k \in \mathsf{Msg}, \mathcal{S} \vdash^W \{u\}_k \Rightarrow \mathcal{S} \vdash^W k \text{ or}$$
$$\exists n \in \mathsf{Msg}, \exists S \subseteq H, \exists i \in \{0,1,2,3\}, \exists a \in H, \exists \alpha \in \{r,g\}, \text{ s.t. } \mathcal{S} \vdash^W h^{\alpha}_a(n,k,i,S)$$
$$\exists m_1,\dots,m_p \in \mathsf{Msg}, \exists i_1,\dots,i_p \in \{0,1,2,3\},$$
$$\exists S_1\dots,S_p, S \subseteq S_j, \forall j, \text{ s.t. } u = i_1,S_1,m_1,\dots,i_p,S_p,m_p,$$
$$\text{and } \forall i_j \geq 1, m_j \in \mathsf{Key}\cup\mathsf{Nonce} \text{ and } \exists n_j \in \mathsf{Nonce}, \exists b \in H, \exists \alpha' \in \{r,g\}, \mathcal{S} \vdash^W h^{\alpha'}_b(n_j,m_j,i_j,S_j) \quad (\mathbf{EncW})$$

$$\forall k,m_1,\dots,m_p \in \mathsf{Msg}, \forall i_1,\dots,i_p \in \{0,1,2,3\}, \forall j \text{ s.t. } i_j = 0$$
$$\mathcal{S} \vdash^W \{i_1,S_1,m_1,\dots,i_p,S_p,m_p\}_k \Rightarrow \mathcal{S} \vdash^W m_j \quad (\mathbf{Enc0W})$$

We show in the full paper (by a careful inspection of the rules) that properties **SecW***, **EncW** and **Enc0W** are invariant under application of the rules of $\mathsf{API}^r \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$ [18]. We also show that $\mathcal{S}'_1$ satisfies the three properties **SecW***, **EncW** and **Enc0W**.

Theorem 6 then easily follows since property **SecW** is an immediate consequence of property **Sec***.

5.6 Security of the API with periodic erasure

We have seen in the previous section that our API still preserves some security when keys are lost. This situation is not completely satisfactory since some groups of agents are definitively not protected.

Thus we assume that (honest) agents periodically erase from the API any handle that corresponds to a data of a security level strictly lower than 3. Since data of security level 2 are

typically short-term session key and data of security level 1 are typically nonces, it makes sense to refresh them periodically. Formally, we say that a state $\mathcal{S}$ is *refreshed* if $\mathcal{S}_{\mathsf{int}} \subseteq \mathsf{Msg}$ is any set of messages and if for any $a \in H$, the set $\mathcal{S}_a$ only contains handles of the form $h_a^\alpha(n,k,3,S)$ with $n \in \mathsf{Nonce}$, $k \in \mathsf{Nonce} \cup \mathsf{Key}$ and such that k only (possibly) appears in $\mathcal{S}$ in key position[2] whenever $S \subseteq H$. Note that we do not make any assumption on the states of compromised agents (besides that honest keys of level 3 only appear in key position).

This is however still not sufficient to guarantee the security of the API in case the attacker is able to learn old keys. Indeed, assume that an attacker knows a cyphertext $\{j,S',k'\}_k$ where k is a long-term (honest) key (of security level 3) such that he also knows k' (possibly using brute force attacks) of security level 2. For every (honest) agent a that has access to k using some handle of the form $h_a^r(n,k,3,S)$, the attacker can register k' using the decryption command of the API of a.

$$\mathsf{Decrypt} \quad h_a^r(n,k,3,S) \quad \{j,S',k'\}_k$$

The attacker then learns $h_a^\alpha(n',k',2,S')$, a fresh handle that refers to k', which allows him to mount the previous attack, again allowing the attacker to learn any data of security level 1 stored by the TRD of a. This corresponds a classical replay attack. Intuitively, since our API can be used to implement a protocol subject to replay, it suffers from replay attack as well.

To prevent such replay attacks, we reinforce the security of the API by restricting the use of decryption rules: the API should allow decryption with keys of level 3 only if at least one component is checked for freshness. In particular, our restricted API will not allow the implementation of protocols subject to this form of replay attack. Formally this corresponds to considering only decryption rules of the form

$$P_a(h_a^\alpha(X_n,X_k,i_0,S_0)),P_a(\{m_1,\dots,m_p\}_{X_k}),\bigcup_{j\in L} P_a(m_j') \quad \overset{N_1,\dots,N_p}{\Rightarrow} \quad \bigcup_{j\notin L} P_a(m_j')$$

where J must not be the emptyset whenever $i_0 = 3$ (and all the other conditions of the decryption rule of Figure 5.1 are fulfilled). Let API^r be the set of rules obtained from API by removing the decryption rules where J is empty when $i_0 = 3$.

Our restricted API preserves secrecy of its confidential values, even when the attacker is able to learn old keys and to control honest APIs, provided honest agents have refreshed the data in their TRDs.

Theorem 7 *Let $\mathcal{S}_0$ be a refreshed state. Then for any state $\mathcal{S}$, accessible from $\mathcal{S}_0$, that is $\mathcal{S}_0 \Rightarrow^*_{\mathsf{API}^r\cup\mathsf{INTRUDER}\cup\mathsf{CONTROL}} \mathcal{S}$, we have that $\mathcal{S}$ satisfies property* **Sec**.

Proof (sketch): Let $\mathcal{S}_0$ be a refreshed state. We define Fresh to be the set of *fresh* values, that is the set of nonces and keys that do not occur in $\mathcal{S}_0$. As for the proof of Theorem 5, we first re-enforce the properties that are invariant under $\mathsf{API}^r \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$. We consider the three following properties.

$$\forall a \in H, \forall x,y \in \mathsf{Msg}, \forall i \in \{1,2,3\}, \forall S \subseteq H, \forall \alpha \in \{r,g\},\ \mathcal{S} \vdash^* h_a^\alpha(x,y,i,S) \ \Rightarrow$$
$$\mathcal{S} \not\vdash^* y \text{ and } y \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and in case } i \neq 3 \text{ then } y \in \mathsf{Fresh} \quad \textbf{(SecFresh*)}$$

$$\forall n,k,m_1,\dots,m_p \in \mathsf{Msg}, \forall i,i_1,\dots,i_p \in \{0,1,2,3\}, \forall \alpha \in \{r,g\}, \forall a \in H, \forall j$$
$$i_j \geq 1, S_j \subseteq H, \forall S \subseteq H, \mathcal{S} \vdash^* \{i_1,S_1,m_1,\dots,i_p,S_p,m_p\}_k, \mathcal{S} \vdash^* h_a^\alpha(n,k,i,S) \Rightarrow$$
$$(m_j \in \mathsf{Key} \cup \mathsf{Nonce} \text{ and } \exists n_j \in \mathsf{Nonce}, b \in H, \exists \alpha' \in \{r,g\}, \mathcal{S} \vdash^* h_b^{\alpha'}(n_j,m_j,i_j,S_j))$$
$$\text{or } \{i_1,S_1,m_1,\dots,i_p,S_p,m_p\}_k \in \mathcal{S}_0 \quad \textbf{(Enc')}$$

[2] That is, whenever k occurs at position p in a message t of $\mathcal{S}$, then $p = p'.2$ and $t|_{p'} = \{t'\}_k$.

$$\forall k, m_1, \ldots, m_p \in \mathsf{Msg}, \forall i_1, \ldots, i_p \in \{0,1,2,3\}, \forall j \text{ s.t. } i_j = 0$$
$$\mathcal{S} \vdash^* \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \Rightarrow$$
$$\mathcal{S} \vdash^* m_j \text{ or } \{i_1, S_1, m_1, \ldots, i_p, S_p, m_p\}_k \in \mathcal{S}_0 \quad \textbf{(Enc0')}$$

We show in the full paper (by a careful inspection of the rules) that these three properties are invariant under application of the rules of $\mathsf{API}^r \cup \mathsf{INTRUDER} \cup \mathsf{CONTROL}$ [18]. Theorem 7 then easily follows since any refreshed state satisfies the three properties **SecFresh***, **Enc'** and **Enc0'** and property **Sec** is an immediate consequence of property **SecFresh***.

Note that our freshness condition does not require agents to erase their data after each session. Intuitively, refreshment should occur only when a leak from an honest user is suspected or when keys have been stored and used for a sufficient time to allowing brute force attacks. Thus refreshment could occur every hour, day, week, month or year depending on application-specific factors.

5.7 Comparison with the Cachin-Chandran API

After the work in this chapter was submitted for publication, a paper by Cachin and Chandran appeared describing independent work in developing a secure key management API with a proof of security [CC09]. Since these are the only two such proposals in the literature at present at a time when standards bodies are considering questions of key management [Com09b, Com09a], a comparison seems appropriate.

The first major difference between our API and the Cachin-Chandran API is that the latter supports asymmetric cryptography for encryption, decryption, signing and verifying signatures, but not for key wrapping and unwrapping (i.e. encryption of keys for transport). Since it is the key management aspects that provide the most interesting security problems, and our own API is only for symmetric keys we will consider just the symmetric key operations of the Cachin-Chandran API in this comparison.

5.7.1 Scenario

The Cachin-Chandran API is designed to operate on a single central key server in an enterprise, rather than on distributed tokens. The server supports multiple users, all of whom are assumed to be able to log in via some secure authentication protocol. For each object on the server there is an access control list for each user, describing the operations the user may perform with respect to the object. The server maintains a log of all operations on the server, and uses this log of previous operations to make decisions about access to keys. The existence of this single global log makes a great deal of difference to the security policy, as we shall see.

5.7.2 Objects and Operations

In addition to key creation, encryption, decryption, wrapping and unwrapping operations, the Cachin-Chandran API supports the following operations: **derive**, for producing a new key based on an existing key, **read**, **delete**, **getattr** for reading an object's attributes and **setattr** for setting them. Note that the delete operation does not erase the history of the keys usage from the log.

Symmetric key objects in the Cachin-Chandran API have a Boolean attribute **unextractable**, which once set to true remains true. Other attributes are defined by user access control lists, and include **admin** (defines whether a user may modify the attributes of a key), **read**, **encrypt**, **decrypt**, **wrap** and **unwrap**. There are also three attributes for each key whose

values are derived automatically from the log: **dependent**, a list of keys whose secrecy depends on the secrecy of this key (i.e. they have been wrapped by or derived from this key or a key dependent on it), **readers**, a list of users who have executed a successful **read** operation on the key or a key on which this key depends, and **usage**, the set of all cryptographic operations that this key has been used for (by any user). There is one attribute derived from the access control lists, which is **owners**, the list of users who have **admin** permissions on the key. Note that a key is created with all attributes set to false except the **admin** attribute which is set to true for the user which created the key.

Cryptographic operations are specified in some detail: encryption must be a randomised scheme resistant against adaptive chosen ciphertext attacks, whilst key wrapping is specified as a randomized version of deterministic key wrap [RS06].

5.7.3 Security Policy

There are two layers to the security policy implemented by the API. First, only a user with **admin** permissions may change the attributes on a key for herself or for other users. Secondly, certain operations will fail if the derived attributes indicate that they would be unsafe. For example, even if a user has **read** permissions for a key, a read operation will fail if she does not also have **read** permissions for all the keys dependent on this key. Similarly, a wrap command will fail if there exists a reader of the wrapping key who does not have **read** permission for the key about to be wrapped.

Proof that the interface is secure is given in a ‘provable cryptography’ style, where the attacker is an arbitrary probabilistic polynomial tine Turing machine that can call the functions of the API as oracles. Security of the interface is posed in the usual fashion as a game where at the start, with equal probability, the attacker is put up against the real API or an API that returns random bitstrings instead of encryptions. To prove security, one must prove that the attacker’s advantage in guessing which one he is facing is negligible.

A curious feature of Cachin and Chanrdan’s proof is that in order to disallow ‘trivial’ wins by the attacker, he is not allowed to wrap a key and then subsequently unwrap it. This is because the authenticated encryption scheme used for wrapping would not accept a random string of bits as a wrapped key, and hence the intruder could immediately tell if he was dealing with the random API simply by testing the output of a wrap command with unwrap. A side effect of this is that a version of the API which in fact failed to wrap the key along with its attributes, or wrapped a key along with a set of incorrect attributes, would still be proved secure in their framework. This is because the security properties required for the proof all come from the derived attributes extracted from the log, which are not affected by wrapping and unwrapping.

5.7.4 Comparison

There are some striking commonalities between the two APIs, especially when compared to previous proposals such as PKCS#11. Both use the identification of distinct users as a way to implement a strong and comprehensible security policy. Both suggest the binding of key usage information to wrapped keys, our API by its tagging scheme, and the Cachin-Chandran API by its inclusion of the access control lists.

The major differences seem to come from the difference in scenario: Cachin and Chandran are able to rely on a centralised log, whereas we cannot. Naïvely extending the Cahin-Chandran API to more than one server immediately introduces attacks. A user could create a key with no attributes on one server, wrap it, unwrap the wrapped key on the second server, set the read permission on copy on the original server and the wrap permission on the copy on the second, and proceed to wrap all the secret keys on the second server. Having read the value of the keys

on the first server, he could now decrypt the unwrapped packets and obtain the value of all the keys. Our API avoids these kinds of attacks by insisting that when keys are created, the user creating them sets the capabilities of the key (in our API this is the key type, 2 or 3) and the list of users who are allowed to access secrets encrypted under the key. In addition, we never allow users to directly read the value of keys. It would be interesting to consider how to add this functionality without breaking our security properties for the distributed tokens scenario.

Our proofs of security are 'symbolic', i.e. we use the usual Dolev-Yao abstractions of bit-strings to logical terms, and prove secrecy in the sense of unreachability of a state where the intruder derives a secret term. Cachin and Chandran's proofs are 'cryptographic' in the sense we explained in section 5.7.3. At first this would seem to indicate that the Cachin-Chandran proofs are stronger, in that their attacker can carry out any polynomial time computation. We have already pointed out that the Cachin-Chandran proof sidesteps the wrapping and unwrapping of keys. We believe this part of the proof could be easily patched. However, our proofs also consider security when the intruder has obtained some keys by out-of-model means. We show that if an intruder obtains an old session key, periodic erasing of old keys prevents him from re-using it (in the restricted version of our API), and if he obtains a long term key, he can only obtain with it session keys which are intended for a subset of the agents who had access to the long term key. By contrast, if an attacker obtains access to the value of a wrapping key in the Cachin-Chandran API without recording himself (or any other user) as a reader in the log, he could potentially wrap all the other keys on the server and read them provided they are not marked as **unextractable**. If some other users are marked as **readers** of the corrupted wrapping key, this would restrict the loss to keys also readable by these users.

5.7.5 Summary

The Cachin-Chandran API is similar to ours in that it identifies tagging objects with respect to particular users as central to robust security, but differs in that since it assumes a global log of operations, it can allow a lot more functionality, in particular allowing the permissions for each key to evolve over time rather than being fixed at the moment of creation. The security breaks down immediately if the API is ported naïvely to distributed servers, or if keys are corrupted.

5.8 Discussion

In this chapter, we have presented a generic API for a tamper-resistant device. We have shown how it could be used to implement many symmetric-key protocols. We have proposed a formal model and threat scenario for a system of such devices where the adversary is assumed to control the network and the host machines connected to the devices, and to have obtained some long-term secrets from inside a subset of the devices. We have proved vital security properties of the API which are independent of the protocol that has been implemented. If an attacker can learn old secret values, our API should be switched to a restricted mode, in which case fewer protocols can be implemented, but protection against replay attacks is enforced. Although our API is limited to symmetric key cryptography and a particular notion of freshness checking which may not accommodate all correct protocols, we believe we have established that it is possible to construct a secure API with a satisfactory level of generality by examining the protocols it is supposed to implement. Extensions to asymmetric cryptography, signatures, PKI certificates, etc. remain as future work.

The Cachin-Chandran API [CC09], as we saw in section 5.7, provides a solution when all operations take place on a single server, but does not immediately extend to the distributed case. At the end of their article, Cachin and Chandran note that a more definitive solution

will require distributed devices with perhaps different APIs to take account of different devices with different roles. Designing such an API, or suite of APIs, combining the advantages of both designs, is an exciting future project. The first step will be to provide a model that captures security of a key management API in both a single server (i.e. with a history log) and distributed setting, work on this model has already started, in collaboration with Kremer and Warinschi.

Chapter 6

Future Work and Perspectives

In this chapter, we first summarise and evaluate the work described in this thesis, before describing some possible future projects, and then stepping back to look at the development of the theory of security API analysis in the field of formal computer security as a whole. In particular, we will discuss the relationship between API analysis and protocol analysis.

6.1 Evaluation

We have presented here decidability results, abstraction techniques, and practical results for formal models of key management APIs. In doing so, we have seen examples of vulnerabilities in APIs deployed in real-world systems of three different types: pure logical or symbolic 'Dolev-Yao' level attacks (for example the VSM attack in section 4.1), vulnerabilities arising in APIs which have symbolic proofs of security, but where cryptographic primitives used do not meet the assumptions required by these proofs (e.g. the key conjuring attacks discussed in section 3.3), and vulnerabilities arising from implementation errors (e.g. the results on PKCS#11 devices presented in section 4.3).

How good are our tools and techniques at discovering these attacks on APIs? We have seen that this depends on the cryptographic primitives used, and on the use of global state. For example in section 3.3 we were able to give decidability results for a model using XOR and an unauthenticated encryption primitive that implements parity checking on certain plaintexts. However, we have been unable to produce an efficient implementation of such a model so far. If we use an implicit encryption model (noting that this may miss some attacks), we have decidability results (§3.2), sound abstractions (§3.5.2), and an efficient decision procedure (§4.2). Using these models has proved very effective at finding attacks on flawed designs, as evidenced by the attack found on the IBM CCA fixes (§4.2), the attacks on the PKCS#11 standard and the attacks found on real devices implementing the standard (§4.3).

Given that we know that our model is not complete even relative to the explicit decryption symbolic model, how meaningful are proofs of security as guarantees of absence of an attack? There is clearly room for improvement here. One approach is to start again with a new API design using modern cryptographic schemes with strong proven security properties. We discuss this further below. The other would be to work on enriching the symbolic model with more and more properties of the weak cryptographic schemes used in the old designs. The difficulty with this latter approach is that it creates models that are typically much harder to reason with, and it is hard to prove that one has really captured all the properties that matter.

Our automated tool Tookan presented in section 4.3 was able to detect a number of implementation errors in real devices that lead to attacks. Again here we know that our reverse engineering process is necessarily incomplete. To give stronger assurances of the absence of

implementation errors we need to be able to look at the source code. Work addressing this using typing has already started with a PhD student [11].

6.2 Security APIs vs Security Protocols

Having seen the development of theory and practical tools for analysis of key management APIs, we can ask: is there a real difference between security API analysis and security protocol analysis? The similarities seem clear: we are typically interested in showing reachability properties in a transition system with cryptography modelled in a highly abstract way. We could quite easily formalise protocols using the language of chapter 2, either using state predicates to model the internal state of the agents as in the AVISPA or multiset rewriting approach approach [ABB$^+$05, DLMS99], or by hard-coding the abstractions used in ProVerif [Bla02]. The difference is therefore not at the level of the semantics of the model. The picture becomes clearer when we attempt to analyse the models. As we saw in section 3.2, early experiments using tools optimised for security protocols to analyse APIs were not very successful: heuristics for precalculating interleaving between protocol runs, for example, are expensive and ineffective when used on APIs which have a 'flat' structure of many commands but few interleavings. Moreover the restriction to a bounded number of sessions that can often be easily justified for security protocols is less easy to justify for APIs and less effective in reducing the search space to a manageable level [Kei07]. Tools that model an unbounded number of sessions using abstractions such as ProVerif produce false attacks when applied to APIs with state [YAB$^+$05, Tsa07]. Our theoretical work in chapter 3 tries to address these problems by establishing decidable classes that can be practically treated by automated tools, as shown in chapter 4.3.5.

Other differences are more subtle. For example, security APIs are typically rather general-purpose, being designed to manage keys and provide secure primitives for a variety of protocols. Specifying their security properties is not easy. As we saw in chapter 5, inferring the specification from the protocols to be implemented is one possible approach. Another difference arises from the fact that cryptographic operations are typically specified at a lower (i.e. more fine grained) level than for protocols, and there are attacks arising from these low level details. These are most clearly seen in the error oracle attacks on PIN processing commands that we have not discussed here [2, 9, 7], but also in the key conjuring attacks we described in section 3.3.

In summary, security APIs and security protocol analysis are very closely related, but the subtle differences are enough to require non-trivial changes to the analysis techniques.

6.3 Future Work

We discuss now a number of future projects in the area of security API analysis.

6.3.1 Computational Soundness

As we have discussed, there is a clear need for work on reconciling the symbolic models we have used for analysing APIs with the real cryptographic primitives sued and their associated security notions. This line of work has been vigorously pursued over the last 10 years for security protocols, and a number of positive results have been achieved, essentially showing that given strong but reasonable assumptions on the cryptographic schemes (typically IND-CCA2), a proof in the Dolev-Yao model is sound in the standard 'computational' model of cryptography. However, these results make assumptions which do not generally apply to key management APIs. For example, APIs encrypt keys under other keys, pass them outside the

API, then re-import them and use them to encrypt known plaintexts. This is not allowed by most models. However, we feel that the major problems here are 'merely' technical, and in ongoing work with Kremer and Warinschi, we are optimistic about producing 'computational' security definitions and soundness results suitable for key management APIs. Note that these results typically require us to make stronger requirements on the cryptography in use that most deployed APIs are actually using. For example, our experiments with commercial devices show that they typically allow all manner of cryptographic sins such as repeated use of a zero initialisation vector in CBC mode, and lack support for modern cryptographic schemes. However, there are a number of initiatives pursuing the next generation of standards for APIs that we might hope to influence [Com09b, Com09a].

6.3.2 Decidability

Returning to the purely symbolic model, our work on decidability of API formalisms leaves several questions open. For example, we have seen that a well known encoding of PCP into a security protocol formalism with fixed message size requires messages to contain three elements in order to form chains of tiles of arbitrary length (§3.1). Looking at a very simple API like the VSM (§4.1) it seems we are dealing with a much less expressive class. It should be possible to define the class and show that the abstraction given in chapter 3.5 is exact for this class, and perhaps build up larger decidable fragments. This remains as an interesting topic perhaps for a student project.

6.3.3 Non-monotonic State

As we noted at the end of section 3.5, our abstractions are not able to deal with non-monotonic state in the sense that if the non-monotonicity of the state is necessary for enforcing security (e.g. by making certain operations mutually exclusive based on state changes), our abstractions will lead to false attacks.

Non-monotonic state is used extensively in the API of the Trusted Platform Module (TPM), a security chip that is supplied on the motherboard of many high end PCs. The idea of this chip is that it contains some secure storage and a cryptographic capabilities that allow a high degree of security to be achieved with a consumer PC without the need for external smart cards or USB tokens. One feature of the TPM is its Platform Control Registers, which are 160-bit values stored inside the chip. These values can only be changed by the operation 'extend', which essentially hashes the current value of the register along with some given bitstring to obtain the new value. Thus if one can prove that one has extended ones PCR value away from a certain value, it should be very hard to return it to that value. If access e.g. to particular key is contingent on the value of the PCR, then a variety of new protocols become possible.

We would like to be able to verify properties of the TPM API that rely on this state. The fact that an attacker may extend the state at any time with any arbitrary value that he hold introduces a great deal of combinatorial complexity into a concrete symbolic model of the API, and it is clear that some kind of abstractions are needed. Note that the abstractions recently proposed by Mödersheim [MÏ0] and discussed in section 3.6.1 would not be suitable here, essentially because the PCR possible values are (in general) not known before the protocol executes, so we cannot create the finite set of sets required to construct the abstraction. Finding suitable abstractions for this kind of API is a project we are actively pursuing [5].

6.3.4 Composition

Designing a security API such that the various functions interact safely without creating vulnerabilities seems similar to the problem of composing cryptographic security protocols. One major difference is shared mutable state. In order to compose protocols securely, one typically requires (amongst other things) that they share no state (i.e. they use different cryptographic keys). Safely composing protocols that share state is a current research topic in both formal security analysis and cryptography. Indeed Yao et al. recently described the goal of composability with arbitrary protocols and shared state as "one of the most ambitious, as well as the hardest, in cryptography" [YYZ09]. Our models have to account for the fact that this shared state can also be modified during an interaction with the API: cryptographic keys can be revoked, updated, backed up, and generated afresh. Capabilities and access permissions may be modified or delegated.

A challenging project suitable perhaps for a PhD student would be to provide a 'computational' semantics and notion of security that can accommodate such operations while still giving strong guarantees. In our favour is the fact that we will typically have a model for the whole API, so we know what functions we are composing before we have to consider the composition with the environment. To produce automated composition based security proofs for APIs would be also be an important step in making the theory usable by designers.

6.3.5 Quantitative Models

When dealing with 'real world' APIs, one would like to reason quantitatively about the level of security given certain parameters for 'soft secrets' such as PINs, passwords or a short challenges, widely used in security APIs. Techniques already exist for checking a protocol to see if a soft secret is susceptible to an offline guessing attack [Bau05]. However, these techniques do not attempt to distinguish the amount of resources required to guess a particular term - a type is considered guessable, or unguessable. In real systems, the quantitative difference between a secret requiring seconds to guess and one requiring hours may lead to a qualitative difference in outcome.

A project on modelling and analysing quantitative leaks from APIs would make a good PhD project. A major challenge of modelling quantitative information leaks for security APIs is that we do not yet have a good model. Other frameworks for information leakage use Shannon entropy or probability of a single correct guess to evaluate leaks. In both of these semantics, one can give examples of systems which are intuitively insecure but which the measure describes as secure. We expect that our work on such an analysis for security APIs where such 'soft secrets' are common, will help to clarify the situation.

6.3.6 More General Security APIs

With the increasing tendency towards creating applications on the fly via services or web 2.0 'mash-ups', security APIs provide one possible solution to the inherent problem of composing an interface with unknown, unpredictable calling code Various authors have proposed the use of security APIs for a variety of security goals such as privacy enforcement in metering applications [Gun07] and social networks [ABS09]. At the moment our modelling is specifically tailored towards cryptographic APIs, and it is not clear for the moment how one would even specify privacy properties for a security API, but a more 'blue skies' future project could investigate this possibility. Analysis of protocols designed to provide privacy properties using notions of equivalence is a research topic we are already pursuing [4]. A particular area of interest might be APIs for mobile devices that regulate how e.g. location and contact data is

revealed to untrusted 'apps' on the device, a subject of much current interest e.g. in W3C[1].

6.4 Perspective

Finally, we take step back from security APIs and ask whether continuing to pursue this line of research is worthwhile. On the positive side (from the motivational point of view), we have seen that vulnerabilities in security APIs can lead to real exploits and large scale criminal fraud[2]. Building more secure APIs is clearly an important problem. Our work on APIs so far has unearthed flaws in real systems, as well as teaching us about the elements required in a secure key management API, and leading to the proposition of a new design. However, formal analysis does have its limitations. All formal approaches are subject to the criticism that no matter how detailed our models are, they are only models with a finite scope of attacker and system behaviour. The attacker may always find some out-of-model way to break the security of the device, e.g. by physically resetting it or exploiting some side channel. Notwithstanding this limitation, there is still a strong motivation to get the API design right as evidence suggests this is one of the current weakpoints. Finally, one might argue that designing and analysis individual APIs cannot lead to the construction of a secure distributed system, since components cannot be considered in isolation. Two secure components may compose to give an insecure system. This is certainly the case, but we feel that if one can specify and formally verify security properties for a single API in isolation, one can hope to use these properties to build a secure system using a suitable calculus (like that of Datta et al. [DFGK09]). So in conclusion we argue that researching key management APIs and security APIs in general merits further effort. The projects proposed in this chapter suggest concrete next steps.

[1] http://www.w3.org/2009/dap/

[2] See http://www.wired.com/threatlevel/2009/04/pins/

Chapter 7

Conclusions

We have presented work on the formal analysis of secure APIs for cryptographic key management devices. In the course of the presentation, we have seen how differences in the formal models compared to the well known models for security protocols require new heuristics, new decidability results and new algorithms for analysis. We have seen how these results have fed back into protocol analysis motivating new techniques for e.g. handling XOR and abstracting state. We have seen our techniques applied to a number of real world APIs leading to the discovery of a number of previously unknown flaws. Finally he have presented a design for a new API with strong formal proofs of security. Future projects include results linking our abstract models to more detailed models of cryptography and better abstractions for dealing with the variety of state manipulations performed by APIs. Formal analysis of security APIs would appear to have a bright future.

My Publications

(since completing my PhD)

International Journal Papers (Refereed)

[1] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.

[2] Graham Steel. Formal analysis of PIN block attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.

International Conference Papers (Refereed)

[3] Morten Dahl, Stéphanie Delaune and Graham Steel. Formal Analysis of Privacy for Anonymous Location Based Services. In *Proceedings of the Workshop on Theory of Security and Applications (TOSCA'11)*, Saarbrcken, Germany, March-April 2011. To appear.

[4] Morten Dahl, Stéphanie Delaune, and Graham Steel. Formal analysis of privacy for vehicular mix-zones. In Dimitris Gritzalis and Bart Preneel, editors, *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS'10)*, pages 55–70, Lecture Notes in Computer Science, Athens, Greece, September 2010. Springer.

[5] Stéphanie Delaune, Steve Kremer, Mark Ryan, and Graham Steel. A formal analysis of authentication in the TPM. In Sandro Etalle and Joshua Guttman, editors, *Proceedings of the 7th International Workshop on Formal Aspects in Security and Trust (FAST'10)*, Pisa, Italy, September 2010. To appear.

[6] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham 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.

[7] Matteo Centenaro, Riccardo Focardi, Flamina L. Luccio, and Graham Steel. Type-based analysis of PIN processing APIs. 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 53–68, Saint Malo, France, September 2009. Springer.

[8] Véronique Cortier and Graham 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.

[9] Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Blunting differential attacks on PIN processing APIs. In Svein Knapskog and Torleiv Maseng, editors, *Proceedings of the 14th Nordic Workshop on Secure IT Systems (NordSec'09)*, pages 88–103, Lecture Notes in Computer Science, Oslo, Norway, October 2009. Springer.

[10] Sibylle Fröschle and Graham Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In P. Degano and L. Viganò, editors, *Preliminary Proceedings 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, 2009. Springer.

[11] Gavin Keighren, David Aspinall, and Graham Steel. Towards a type system for security APIs. In Pierpaolo Degano and Luca 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 173–192, York, UK, August 2009. Springer.

[12] Stéphanie Delaune, Steve Kremer, and Graham 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.

[13] Véronique Cortier, Stéphanie Delaune, and Graham Steel. A formal theory of key conjuring. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF20)*, pages 79–93, Venice, Italy, 2007.

[14] Véronique Cortier, Gavin Keighren, and Graham Steel. Automatic analysis of the security of XOR-based key management schemes. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 4424 in LNCS, pages 538–552, 2007.

[15] Graham Steel. Deduction with XOR constraints in security API modelling. In R. Nieuwenhuis, editor, *Proceedings of the 20th Conference on Automated Deduction (CADE 20)*, number 3632 in Lecture Notes in Artificial Intelligence, pages 322–336, Tallinn, Estonia, July 2005. Springer-Verlag Heidelberg.

Invited contributions, workshop papers, technical reports

[16] Morten Dahl, Stéphanie Delaune, and Graham Steel. Formal analysis of privacy for vehicular mix-zones. In *Proceedings of the FCS-PRIVACY Workshop, FLoC'10*, 2010

[17] Graham Steel. Towards a Formal Analysis of the SeVeCoM API In *Proceedings of the ESCAR Workshop*, 2009.

[18] Véronique Cortier and Graham Steel. Synthesising secure APIs. Technical Report RR-6882, INRIA, 2009. 27 pages.

[19] Matteo Centenaro, Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Type-based analysis of financial APIs (work in progress). In *Presented at ARSPA-WITS'09*, 2009.

[20] Graham Steel. Computational Soundness for APIs In *Proceedings of the 5th workshop on Formal and Computational Cryptography (FCC '09)*, July 2009

[21] Graham Steel and Judicaël Courant. A formal model for detecting parallel key search attacks. In *Proceedings of the 3rd workshop on Formal and Computational Cryptography (FCC '07)*, July 2007.

[22] Véronique Cortier and Graham Steel. On the decidability of a class of XOR-based key-management APIs. In *Proceedings of the FCS-ARSPA workshop*, August 2006.

[23] Graham Steel. Visualising first-order proof search. In *Workshop on User Interfaces for Theorem Provers (UITP '05)*, pages 179–189, Edinburgh, Scotland, April 2005.

[24] Graham Steel. The importance of non-theorems and counterexamples in program verification. In *Position Papers of the ETH-Zürich VSTTE conference – Verified Software: Theories, Tools, Experiments*, 491–495. Springer, 2006.

[25] Véronique Cortier, Séphanie Delaune, and Graham Steel. A formal theory of key conjuring. Technical Report 6234, Unité de recherche INRIA Lorraine, February 2007.

[26] Véronique Cortier, Gavin Keighren, and Graham Steel. Automatic analysis of the security of XOR-based key management schemes. Inf. Research Report EDI-INF-RR-0863, U. of Edinburgh, 2006.

Bibliography

[ABB$^+$05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent 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.

[ABS09] Jonathan Anderson, Joseph Bonneau, and Frank Stajano. Security APIs for online applications. In *Third International Workshop on Analysis of Security APIs (ASA-3)*, Long Island, NY, USA, 2009.

[AC08] 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`.

[AN96] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[And00] 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`.

[Bau05] M. Baudet. Deciding security of protocols against off-line guessing attacks. In *Computer and Communications Security*, pages 16–25. ACM, 2005.

[Bla02] 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.

[Bon01] 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.

[Bon04] M. Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge, 2004.

[BV10] L. Baloci and A. Vianello. Un sistema per lo studio della sicurezza. Baccalaureate Thesis, University of Venice, Italy, April 2010.

[Car94] U. Carlsen. Optimal privacy and authentication on a portable communications system. *SIGOPS Oper. Syst. Rev.*, 28(3):16–23, 1994.

[CB02] 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.

[CC09] 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.

[CCA06] *CCA Basic Services Reference and Guide*, October 2006. Available online at `http://www-03.ibm.com/security/cryptocards/pdfs/bs327.pdf`.

[CDP09] Xihui Chen, Ton Deursen, and Jun Pang. Improving automatic verification of security protocols with xor. In *ICFEM '09: Proceedings of the 11th International Conference on Formal Engineering Methods*, pages 107–126, Berlin, Heidelberg, 2009. Springer-Verlag.

[CJ97] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Available via `http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz`, 1997.

[CLC03] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *LNCS*, pages 148–164, Valencia, Spain, 2003. Springer.

[Clu03a] J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[Clu03b] J. Clulow. On the security of PKCS#11. In *Proceedings of CHES 2003*, pages 411–425, 2003.

[Com09a] IEEE 1619.3 Technical Committee. Ieee storage standard 1619.3 (key management) (draft). available from `https://siswg.net/`, 2009.

[Com09b] OASIS Key Management Interoperability Protocol (KMIP) Technical Committee. Kmip – key management interoperability protocol. available from `http://xml.coverpages.org/KMIP/`, february 2009.

[CR06] Yannick Chevalier and Michaël Rusinowitch. Hierarchical combination of intruder theories. In *Proceedings of the 17th International Conference on Term Rewriting and Applications (RTA'06)*, volume 4098 of *LNCS*, pages 108–122, Seattle, USA, 2006. Springer.

[DFGK09] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 221–236, 2009.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DLM04] Nancy A. Durgin, Patrick Lincoln, and John C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[DLMS99] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy*, July 1999. Electronic proceedings available at `http://www.cs.bell-labs.com/who/nch/fmsp99/program.html`.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[Gun07] Carl A. Gunter. Using APIs to assure conformance to privacy regulations. In *First International Workshop on Analysis of Security APIs*, Venice, Italy, 2007.

[Her06] J. Herzog. Applying protocol analysis to security device interfaces. *IEEE Security & Privacy Magazine*, 4(4):84–87, July-Aug 2006.

[Kei07] G. Keighren. Model checking security APIs. Master's thesis, University of Edinburgh, 2007.

[KT08] R. Küsters and T. Truderung. Reducing Protocol Analysis with XOR to the XOR-free Case in the Horn Theory Based Approach. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 129–138. ACM Press, 2008.

[LM05] C. Lynch and C. Meadows. On the relative soundness of the free algebra model for public key encryption. *Electr. Notes Theor. Comput. Sci.*, 125(1):43–54, 2005.

[Low96] 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.

[LR92] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

[MÏ0] S. Mödersheim. Abstraction by set-membership: Verifying security protocols and wes services with databases. In *Proceedings of ACM Computer and Communication Security Conference (CCS)*, page To appear, 2010.

[Mil03] J. K. Millen. On the freedom of decryption. *Inf. Process. Lett.*, 86(6):329–333, 2003.

[Mit02] John C. Mitchell. Multiset rewriting and security protocol analysis. In *Proceeding of the 13th International Conference on Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *LNCS*, pages 19–22, Copenhagen, Denmark, 2002. Springer.

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

[Pro] AVISPA Project. Deliverable 2.3: The intermediate format. Available from `http://www.avispa-project.org/delivs/2.3`.

[RS06] P. Rogaway and T. Shrimpton. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. In *Advances in Cryptology (EUROCRYPT '06)*, volume 4004 of *LNCS*, pages 373–390, 2006.

[RSA04] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.

[RT01] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proc. of the 14th Computer Security Foundations Workshop (CSFW'01)*, pages 174–190, Cape Breton, Nova Scotia, Canada, 2001. IEEE Computer Society Press.

[TEB05] Ferucio Laurentiu Tiplea, Constantin Enea, and Catalin V. Birjoveanu. Decidability and complexity results for security protocols. In *Proceedings of the Verification of Infinite-State Systems with Applications to Security (VISSAS'05)*, volume 1 of *NATO Security through Science Series D: Information and Communication Security*, pages 185–211. IOS Press, 2005.

[Tsa07] E. Tsalapati. Analysis of PKCS#11 using AVISPA tools. Master's thesis, University of Edinburgh, 2007.

[VSS05] K. N. Verma, H. Seidl, and T. Schwentick. On the complexity of equational Horn clauses. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *LNCS*, pages 337–352, Tallinn, Estonia, 2005. Springer.

[YAB$^+$05] 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.

[You04] Paul Youn. The analysis of cryptographic APIs using the theorem prover Otter. Master's thesis, Massachusetts Institute of Technology, 2004.

[YYZ09] Andrew C. C. Yao, Frances F. Yao, and Yunlei Zhao. A note on the feasibility of generalised universal composability. *Math. Struct. in Comp. Science*, 19:193–205, 2009.