Analysis of the PKCS#11 API Using the Maude-NPA Tool

Antonio González-Burgueño[1], Sonia Santiago[2], Santiago Escobar[3], Catherine Meadows[4](✉), and José Meseguer[2](✉)

[1] University of Oslo, Oslo, Norway
antonigo@ifi.uio.no
[2] University of Illinois at Urbana-Champaign, Champaign, USA
{soniasp,meseguer}@illinois.edu
[3] DSIC-ELP, Universitat Politècnica de València, Valencia, Spain
sescobar@dsic.upv.es
[4] Naval Research Laboratory, Washington DC, USA
meadows@itd.nrl.navy.mil

Abstract. Cryptographic Application Programmer Interfaces (Crypto APIs) are designed to allow a secure interoperation between applications and cryptographic devices such as smartcards and Hardware Security Modules (HSMs). However, several Crypto APIs have been shown to be subject to attacks in which sensitive information is disclosed to an attacker, such as the RSA Laboratories Public Key Standards PKCS#11, an API widely adopted in industry. Recently, there has been a growing interest on applying automated crypto protocol analysis methods to formally analyze APIs. However, the PKCS#11 has been proven difficult to analyze using such methods since it involves non-monotonic mutable global state. In this paper we specify and analyze the PKCS#11 in Maude-NPA, a general purpose crypto protocol analysis tool.

Keywords: PKCS#11 · Cryptographic application programming interfaces (cryptographic APIs) · Symbolic cryptographic protocol analysis · Maude-NPA

1 Introduction

Standards for cryptographic protocols have long been attractive candidates for formal verification. Cryptographic protocols are tricky to design and subject to non-intuitive attacks even when the underlying cryptosystems are secure. Furthermore, when protocols that are known to be secure are implemented as

The rights of this work are transferred to the extent transferable according to title 17 U.S.C. 105.

This work has been partially supported by NSF grant CNS 13-19109, by the EU (FEDER) and the Spanish MINECO under grant TIN 2013-45732-C4-1-P, and by Spanish Generalitat Valenciana under grant PROMETEOII/2015/013.

© Springer International Publishing Switzerland 2015
L. Chen and S. Matsuo (Eds.): SSR 2015, LNCS 9497, pp. 86–106, 2015.
DOI: 10.1007/978-3-319-27152-1_5

standards, the modifications that are made during the standardization process may introduce new security flaws. Thus a considerable amount of work has been done in the application of formal methods to cryptographic protocol standards [1,7,27,29]. In this work the protocols are treated *symbolically*, with the cryptosystems treated as black-box function symbols. A formal methods tool attempts to show that there is no way an attacker, by interacting with the protocol and applying the cryptographic functions symbols in any order, can break the security of the protocol. Such tools can be used both to search for attacks and to prove security with respect to the symbolic model.

Such symbolic formal analyses can be of great benefit to standards development in two ways. First of all, they offer means of verifying claims for security made by the standard, so that people can use it with more confidence. Furthermore, they can be useful in improving the standard, by discovering vulnerabilities in the protocols and attacks that can be mounted through exploiting the vulnerabilities. The explicit attacks discovered by the tools are particularly useful in that they provide the protocol designers information that can be used to help assess and repair the vulnerability.

For most analyses it has been the case that the same tool and general approach has been used to both verify the security of a protocol and to find attacks. Many of the tools employ methods that allow one to conclude that the protocol is secure if the tool terminates without finding an attack, such as heuristics that allow one to rule out redundant or useless paths (e.g. OFMC [3], Maude-NPA [14], Tamarin [30]) or abstractions (e.g. ProVerif [4]). Thus, one can use the tool first to find vulnerabilities, and then to verify security of the protocol once the vulnerabilities have been fixed. In a number of cases this has facilitated collaboration between standards developers and formal methods experts, e.g. as in [28]. This approach has worked particularly well for standards for key generation and secure communication, and these are the types of protocols that are most widely standardized, and the most well understood from the point of view of symbolic formal analysis.

However, recently another type of application has begun to attract interest: *Cryptographic Application Programming Interfaces*, or cryptographic APIs for short. A cryptographic API is a set of instructions by which a developer of an application may allow it to take advantage of the cryptographic functionality of a secure module. These APIs allow an application to perform such functions as creating keys, using keys to encrypt and decrypt data, and exporting and importing keys to and from other devices. Cryptographic APIs should also enforce security policies. In particular, no application should be able to retrieve a key in the clear.

Developers of cryptographic APIs have traditionally concentrated on providing functionality, not on guaranteeing security properties. This has resulted in a number of attacks on cryptographic APIs in which researchers have shown how many popular APIs can be led into an unsafe state (e.g. one in which a key is revealed to an untrusted application) via a series of legal steps. Indeed, much of the earliest work on formal analysis of cryptographic protocols focused on cryptographic APIs, e.g. [21,25,26]. However, the analysis of cryptographic

APIs, did not become an area of research on its own until the early 2000's in particular after the attack found by Bond [5] in 2001 on IBM's CCA API. Many more attacks on CCA and other systems, as well as new techniques for verifying the security of APIs, have followed; see for example the attacks described in Chap. 18 of Ross Anderson's *Security Engineering* [2].

One API that has attracted particularly wide attention is PKCS#11 [24]. This is a standard that provides both a set of commands that could be used by a cryptographic API and mechanisms for setting and enforcing security policies. These security policies are specified in terms of attributes on keys and other data that declare which operations using these terms are legal or illegal. However, no guidance is provided on what sort of restrictions should be put on setting attributes on keys and other data so that undesirable states are avoided. Indeed, if no restrictions at all are put on the way attributes are set, it is possible to wind up with an application learning a key in the clear in just a very few steps, as Clulow points out in [9].

Since PKCS#11 is a widely used standard, much attention has been focused on correcting these deficiencies, in particular on developing means for formally verifying that a policy rules out undesirable states. But because PKCS#11 is intended to be applied to a wide variety of platforms, the problem is harder than verifying the security of an API such as IBM's CCA [20], which was only intended to be used for applications running on certain IBM systems. For one thing, the set of attributes forms a *mutable global state* which must be accounted for. Secondly, any formal verification system must be capable of verifying not just one or two policies specified by the developers of the API, as was the case of CCA, but any of a large class of policies that could be specified by a user.

Because of the complexity of the problem, researchers have tended to narrow their focus when applying cryptographic protocol analysis tools to PKCS#11. Most use of tools for the analysis of PKCS#11 makes some restriction on the policies analyzed, usually with an appeal to practicality or common use cases. They may also develop tools specifically designed for PKCS#11 analysis, or at the very least prove additional results specific to PKCS#11 that allow them to limit the size of the search space. Finally, they may concentrate primarily on either proving security or finding attacks, but not both.

In this paper we investigate the applicability of cryptographic protocol verification tools, in particular the Maude-NRL Protocol Analyzer (Maude-NPA) [14], to the analysis of cryptographic APIs such as PKCS#11, primarily concentrating on assessing its ability to find attacks. In order to perform the verification of PKCS#11, we make use of the results in [18] which show that, for a large class of reasonable policies, it is sufficient to assume that attributes never change; that is, that policies are *static*. We then show how, assuming static policies, it is possible to specify policies that put restrictions on what combination of attributes can be set to true for a particular key using Maude-NPA *never patterns*, a feature that allows the user to specify what events Maude-NPA should avoid in generating an attack. Since most policies proposed to date for PKCS#11 are expressed in this form, this leaves us in a good position to express both the API and policies in Maude-NPA. We then use Maude-NPA to reproduce the attacks found by Delaune et al. in [12]. Finally, we discuss the performance of Maude-NPA and compare it with other applications of cryptographic protocol analysis tools to the analysis of PKCS#11, in particular the use of the AVISPA tool in [32] and the use of Tamarin in [23].

The contributions of this paper are twofold. First, we advance investigation of the verification of PKCS#11 by performing the analysis of this API in a model more general than those in other works in the literature, namely a fully-unbounded session model with no abstraction nor approximation of fresh values, and making no other restrictions on policies other than that they are static. Second, we provide a new example of applicability of Maude-NPA to the analysis of cryptographic APIs. This paper extends the work presented in [19] on the analysis of IBM CCA by showing that Maude-NPA does not only support the specification of these APIs, but also the specification of policies restricting their behavior.

The rest of the paper is organized as follows. In Sect. 2 we give a high-level summary of Maude-NPA and its use of never patterns. In Sect. 3 we give an overview of PKCS#11 along with previous attacks and formal analyses. In Sect. 4 we describe how we specify the PKCS#11 API and policies in Maude-NPA. In Sect. 5 we describe the experiments we conducted and explain the results obtained. In Sect. 6 we discuss related work. Finally, in Sect. 7 we conclude the paper and discuss future work.

2 Maude-NPA

In this section we give a high-level summary of Maude-NPA. For further information, please see [14].

2.1 Preliminaries on Unification and Narrowing

We assume an order-sorted signature $\mathbf{\Sigma} = (\mathsf{S}, \leq, \Sigma)$ with a poset of sorts $(\mathsf{S}, \leq)$ and an S-sorted family $\mathcal{X} = \{\mathcal{X}_\mathsf{s}\}_{\mathsf{s} \in \mathsf{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathsf{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ is the set of terms of sort s, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ is the set of ground terms of sort s. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding order-sorted term algebras. For a term t, $\mathcal{V}ar(t)$ denotes the set of variables in t.

Positions are represented by sequences of natural numbers denoting an access path in the term when viewed as a tree. The top or root position is denoted by the empty sequence ϵ. The subterm of t at position p is $t|_p$ and $t[u]_p$ is the term t where $t|_p$ is replaced by u.

A *substitution* $\sigma \in \mathcal{S}ubst(\Sigma, \mathcal{X})$ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of σ is $Dom(\sigma) = \{X_1, \ldots, X_n\}$ and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $Ran(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$.

A *Σ-equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$ for some sort $\mathsf{s} \in \mathsf{S}$. Σ and a set E of Σ-equations, The E-equivalence class of a term t is denoted by $[t]_E$ and $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ and $\mathcal{T}_{\Sigma/E}$ denote the corresponding order-sorted term algebras modulo E. An *equational theory* (Σ, E) is a pair with Σ an order-sorted signature and E a set of Σ-equations.

An *E-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_E t'\sigma$. For $\mathcal{V}ar(t) \cup \mathcal{V}ar(t') \subseteq W$, a set of substitutions $CSU_E^W(t = t')$ is said to be a *complete* set of unifiers for the equality $t = t'$ modulo E away from W iff: (i) each $\sigma \in CSU_E^W(t = t')$ is an E-unifier of $t = t'$; (ii) for any E-unifier ρ of $t = t'$ there is a $\sigma \in CSU_E^W(t = t')$ such that $\sigma|_W \sqsupseteq_E \rho|_W$ (i.e., there is a substitution η such that $(\sigma \circ \eta)|_W =_E \rho|_W$); and (iii) for all $\sigma \in CSU_E^W(t = t')$, $Dom(\sigma) \subseteq (\mathcal{V}ar(t) \cup \mathcal{V}ar(t'))$ and $Ran(\sigma) \cap W = \emptyset$.

A *rewrite rule* is an oriented pair $l \to r$, where $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$ for some sort $\mathsf{s} \in \mathsf{S}$. An *(unconditional) order-sorted rewrite theory* is a triple (Σ, E, R) with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. The (R, E) rewriting relation $\to_{R,E}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ is defined as: $t \to_{p,R,E} t'$ iff there exist $p \in Pos_{\Sigma}(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p =_E l\sigma$ and $t' = t[r\sigma]_p$.

Let t be a term and W be a set of variables such that $\mathcal{V}ar(t) \subseteq W$, the *$R, E$-narrowing* relation on $\mathcal{T}_{\Sigma}(\mathcal{X})$ is defined as $t \rightsquigarrow_{p,\sigma,R,E} t'$ if there is a non-variable position $p \in Pos_{\Sigma}(t)$, a rule $l \to r \in R$ properly renamed s.t. $(\mathcal{V}ar(l) \cup \mathcal{V}ar(r)) \cap W = \emptyset$, and a unifier $\sigma \in CSU_E^{W'}(t|_p = l)$ for $W' = W \cup \mathcal{V}ar(l)$, such that $t' = (t[r]_p)\sigma$.

2.2 Maude-NPA Syntax and Semantics

Given a protocol $\mathcal{P}$, states are modeled as elements of an initial algebra $\mathcal{T}_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$, where $\Sigma_{\mathcal{P}}$ is the signature defining the sorts and function symbols (for the cryptographic functions and for all the state constructor symbols) and $E_{\mathcal{P}}$ is a set of equations specifying the *algebraic properties* of the cryptographic functions and the state constructors. Therefore, a state is an $E_{\mathcal{P}}$-equivalence class $[t]_E \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$ with t a ground $\Sigma_{\mathcal{P}}$-term.

In Maude-NPA a *state pattern* for a protocol P is a term t of sort State (i.e., $t \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}(\mathcal{X})_{\mathsf{State}}$) which has the form $\{S_1 \,\&\, \cdots \,\&\, S_n \,\&\, \{IK\}\}$ where &is an associative-commutative union operator with identity symbol $\emptyset$. Each element in the set is either a *strand* S_i or the *intruder knowledge* $\{IK\}$ at that state.

The *intruder knowledge* $\{IK\}$ belongs to the state and is represented as a set of facts using the comma as an associative-commutative union operator with identity element *empty*. There are two kinds of intruder facts: *positive* knowledge facts (the intruder knows m, i.e., $m{\in}\mathcal{I}$), and *negative* knowledge facts (the intruder *does not yet know* m but *will know it in a future state*, i.e., $m{\notin}\mathcal{I}$), where m is a message expression.

A *strand* [16] specifies the sequence of messages sent and received by a principal executing the protocol and is represented as a sequence of messages $[msg_1^-, msg_2^+, msg_3^-, \ldots, msg_{k-1}^-, msg_k^+]$ such that msg_i^- (also written $-msg_i$)

represents an input message, msg_i^+ (also written $+msg_i$) represents an output message, and each msg_i is a term of sort Msg (i.e., $msg_i \in T_{\Sigma_\mathcal{P}/E_\mathcal{P}}(\mathcal{X})_{\mathsf{Msg}}$).

Strands are used to represent both the actions of honest principals (with a strand specified for each protocol role) and the actions of an intruder (with a strand for each action an intruder is able to perform on messages). In Maude-NPA strands evolve over time; the symbol $|$ is used to divide past and future. That is, given a strand $[\ m_1{}^\pm,\ \ldots,\ m_i{}^\pm \mid m_{i+1}{}^\pm,\ \ldots,\ m_k{}^\pm\ ]$, messages $m_1^\pm$, $\ldots, m_i^\pm$ are the *past messages*, and messages $m_{i+1}^\pm, \ldots, m_k^\pm$ are the *future messages* ($m_{i+1}^\pm$ is the immediate future message). A strand $[msg_1^\pm, \ldots, msg_k^\pm]$ is shorthand for $[nil \mid msg_1^\pm, \ldots, msg_k^\pm, nil]$. An *initial state* is a state where the bar is at the beginning for all strands in the state, and the intruder knowledge has no fact of the form $m{\in}\mathcal{I}$. A *final state* is a state where the bar is at the end for all strands in the state and there is no intruder fact of the form $m{\notin}\mathcal{I}$.

Since Fresh variables must be treated differently from other variables by Maude-NPA, we make them explicit by writing $:: r_1, \ldots, r_k ::\ [m_1^\pm, \ldots, m_n^\pm]$, where each r_i first appears in an output message $m_{j_i}^+$ and can later appear in any input and output message of $m_{j_i+1}^\pm, \ldots, m_n^\pm$. If there are no Fresh variables, we write $:: nil ::\ [m_1^\pm, \ldots, m_n^\pm]$.

Example 1. Let us consider a subset of the PKCS#11 API. A symmetric key generated by principal A is denoted by $skey(A, r)$, where r is a unique Fresh variable and A denotes who generated the key. The symmetric encryption of a message M with a key $skey(A, r)$ is denoted by $senc(M, skey(A, r))$. The intruder's ability to generate its own symmetric keys is specified in Maude-NPA by the strand:

$$:: r ::\ \ [skey(i, r)^+]$$

where i is a constant denoting the intruder's name. Note that we have made explicit that the fresh variable r is generated in this strand. In this protocol the intruder is allowed to perform the symmetric encryption of a message M with a key K, assuming that it has received both M and K. This ability is specified by the following strand:

$$:: nil ::\ \ [M^-,\ K^-,\ (senc(M, K)^+)]$$

where M is a variable of the sort for messages and K is a variable of the sort for symmetric keys. Note that no fresh variables are generated in this strand.

Since the number of states $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$ is in general infinite, rather than exploring concrete protocol states $[t]_{E_\mathcal{P}} \in T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$ Maude-NPA explores *symbolic state patterns* $[t(x_1, \ldots, x_n)]_{E_\mathcal{P}} \in T_{\Sigma_\mathcal{P}/E_\mathcal{P}}(\mathcal{X})$ on the free $(\Sigma_\mathcal{P}, E_\mathcal{P})$-algebra over a set of variables $\mathcal{X}$. In this way, a state pattern $[t(x_1, \ldots, x_n)]_{E_\mathcal{P}}$ represents not a single concrete state but a possibly infinite set of such states, namely all the *instances* of the pattern $[t(x_1, \ldots, x_n)]_{E_\mathcal{P}}$ where the variables $x_1, \ldots, x_n$ have been instantiated by concrete ground terms.

The semantics of Maude-NPA is expressed in terms of *rewrite rules* that describe how a protocol moves from one state to another via the intruder's interaction with it. One uses Maude-NPA to find an attack by specifying an insecure state pattern called an *attack pattern*. Maude-NPA attempts to find a path from an initial state to the attack pattern via backwards narrowing (narrowing using the rewrite rules with the orientation reversed).

Example 2. Let us continue Example 1. In order to analyze whether the intruder can learn an honest key we specify the following attack pattern below:

$$\{SS \,\&\, \{skey(a,r) {\in} \mathcal{I},\ IK\}\}$$

where SS and IK are variables of the sort for sets of strands and for the intruder's knowledge, respectively. This attack pattern represents an insecure situation in which the intruder has learnt the honest key $skey(a,r)$. Note that a is a constant of the sort for names, which is different to the intruder's name i.

The backwards narrowing sequence from an initial state to an attack state is called a *backwards path* from the attack state to the initial state. Maude-NPA attempts to find paths until it can no longer form any backwards narrowing steps, at which point it terminates. If at that point it has not found an initial state, the attack pattern is judged *unreachable*. Note that Maude-NPA puts *no bound on the number of sessions*, so reachability is undecidable in general. Note also that Maude-NPA does not perform any data abstraction such as bound number of nonces. However, the tool makes use of a number of sound and complete state space reduction techniques that help to identify unreachable and redundant states [15], and thus make termination more likely.

2.3 Never Patterns in Maude-NPA

It is often desirable to exclude certain patterns from transition paths leading to an attack state. For example, one may want to determine whether or not authentication properties have been violated, e.g., whether it is possible for a responder strand to appear without the corresponding initiator strand. For this there is an optional additional field in the attack state containing the *never patterns*. Each never pattern is itself a state pattern. When we provide an attack pattern A and some never patterns $NP_1, \ldots, NP_k$ to Maude-NPA, every time the tool produces a state S via backwards narrowing from A, it checks whether there is a substitution θ such that $NP_i\theta =_{E_\mathcal{P}} S$. If that is the case, the state is discarded[1]. We will write an attack pattern A with the never patterns $NP_1, \ldots, NP_k$ as $A \mid\mid \text{never}(NP_1) \ldots \mid\mid \text{never}(NP_k)$.

Although never patterns were introduced as a means for specifying authentication properties, they can also be used to reduce the search space in a not necessarily complete way (an attack could be missed). In this work we only found it necessary to use such never patterns once in analysis (see Sect. 5).

Example 3. Let us continue Example 2. In order to exclude from the backwards path from the attack pattern of Example 2 the case in which the intruder uses

[1] Maude-NPA also checks whether $NP_i\theta$ satisfies *irreducibility constraints*, as described in [13].

the key $skey(a, r)$ to perform the symmetric encryption of any message M, we extend the attack pattern above with a never pattern as shown below:

$$\{SS \ \& \ \{skey(a,r) {\in} \mathcal{I},\ IK\}$$
$$||\ never(\{ :: nil :: [(M)^-,\ (skey(a,r))^-,\ (senc(M, skey(a,r)))^+] \,\&\, SS' \ \& \ \{IK'\}\})$$

where SS' and IK' are variables of the sort for sets of strands and for the intruder's knowledge, respectively.

3 PKCS#11

RSA Laboratories originally developed the Public Key Standards (PKCS) #11 in order to define a platform-independent API "Cryptoki"for the management of cryptographic tokens. Recently (in 2012) the responsibility of the maintenance of the standard was transitioned to the OASIS standards committee [31], but the standard is still referred to as PKCS#11.

PKCS#11 is intended to protect sensitive cryptographic keys as follows [17]. Once a session is initiated, the application may access the objects stored in the token, such as keys and certificates. However, access to the objects is controlled in the API via handles (which can be thought of as pointers to, or names for, the objects). These objects have attributes, e.g. boolean flags signalling properties of the object, namely `wrap`, `unwrap`, `encrypt`, `decrypt`, `sensitive` and `extract`. These flags can be either in positive form (l) or in negative form ($\neg l$), denoting that an attribute l is set or unset, respectively. Depending on whether these attributes are set or unset, certain API commands may be enabled or disabled.

New handles can be created by calling a key generation command, or by "unwrapping" an encrypted key packet. For example, if the *encrypt* function is called with the handle for a particular key, that key must have its `encrypt` attribute set. Also, a key may be exported outside the device if it is encrypted by another key, but only if it has the attributes `sensitive` and `extract` set. It is important to know that protection of the keys essentially relies on these two attributes, `sensitive` and `extract`.

Table 1 provides an informal description of a subset of the PKCS#11 key management commands. There are two kinds of commands. First, there are commands that correspond to PKCS#11 actions: the ones for wrapping and unwrapping keys, namely "Wrap" and "Unwrap", respectively; and for symmetric and asymmetric encryption and decryption, e.g. "SEncrypt" is the command for symmetric encryption, whereas "ADecrypt" corresponds to the command for asymmetric decryption. Note that there are several possibilities for the "Wrap" and "Unwrap" commands, depending on whether they use symmetric or asymmetric keys. Second, there are commands to modify attribute values, namely the "Set" and "Unset" commands. For example, "Set-Wrap" sets to true the `wrap` attribute of a key, whereas "Unset-Wrap" sets it to false.

The behavior of each command is described in Table 1 by rules of the form $T; L \overset{new\ \tilde{n}}{\rightarrow} T'; L'$. T is the set of messages that need to be received, whereas T' denotes the set of messages that are sent as a result of the messages in T

Table 1. Subset of PKCS#11 v2.01 key management commands

Name	API Command Description
Wrap (sym-sym)	$h(n_1, k_1), h(n_2, k_2)\,;\, wrap(n_1), extract(n_2) \rightarrow senc(k_2, k_1)$
Wrap (sym-asym)	$h(n_1, priv(z)), h(n_2, k_2)\,;\, wrap(n_1), extract(n_2) \rightarrow aenc(k_2, pub(z))$
Unwrap (sym-sym)	$h(n_1, k_2), senc(k_1, k_2)\,;\, unwrap(n_1) \xrightarrow{new\ r} h(r, k_1)\,;\, extract(r), L$
Unwrap (sym-asym)	$h(n_1, priv(z)), aenc(k_1, pub(z))\,;\, unwrap(n_1) \xrightarrow{new\ r} h(r, k_1)\,;\, extract(r), L$
SEncrypt	$h(n, k), m\,;\, encrypt(n) \rightarrow senc(m, k)$
SDecrypt	$h(n, k), senc(m, k)\,;\, decrypt(n) \rightarrow m$
AEncrypt	$h(n, priv(z)), m\,;\, encrypt(n) \rightarrow aenc(m, pub(z))$
ADecrypt	$h(n, priv(z)), aenc(m, pub(z))\,;\, decrypt(n) \rightarrow m$
Set-Wrap	$h(n, k)\,;\, \neg wrap(n) \rightarrow wrap(n)$
Set-Encrypt	$h(n, k)\,;\, \neg encrypt(n) \rightarrow encrypt(n)$
Unset-Wrap	$h(n, k)\,;\, wrap(n) \rightarrow \neg wrap(n)$
Unset-Encrypt	$h(n, k)\,;\, encrypt(n) \rightarrow \neg encrypt(n)$

$L = \neg wrap(r), \neg unwrap(r), \neg encrypt(r), \neg decrypt(r), \neg sensitive(r)$

being received. L and L' are sets of attributes. More specifically, L denotes the attributes that must be set in order to execute the command, whereas L' denotes the value of the attributes after the command is executed. L' can include attributes in negative form and attributes related to freshly generated handles not appearing in L. The expression $new\, \tilde{n}$ represents the generation of fresh data that will appear in T' and L'.

The set L' is assumed to be satisfiable, i.e., it cannot contain two literals l and $\neg l$. Any variable appearing in T' must appear in T, i.e. $\mathcal{V}ar(T') \subseteq \mathcal{V}ar(T)$ and any variable appearing in L' also appears in L, i.e. $\mathcal{V}ar(L') \subseteq \mathcal{V}ar(L)$. The only new variables that can appear in T' and L' are those indicated in $new\, \tilde{n}$.

In Table 1 public and private keys are represented by terms of the form $pub(A)$ and $priv(A)$, respectively. Handles for keys are specified as terms of the form $h(N, K)$, where N is a nonce uniquely identifying the handle and K is the key. Symmetric and asymmetric encryption of a message M with a key K is denoted by terms $senc(M, K)$ and $aenc(M, K)$ respectively. Finally the attributes are specified as terms of the form $l(N)$, where l is the attribute's name, and N is a nonce that refers to a unique handle $h(N, K)$.

Example 4. Let us consider the rule corresponding to the "Wrap (sym-sym)" command. This rule allows wrapping a symmetric key using another symmetric key. If the attacker knows the handles $h(n_1, k_1)$ and $h(n_2, k_2)$, i.e., references to symmetric keys k_1 and k_2, the handle $h(n_1, k_1)$ has the attribute `wrap` set, and the handle $h(n_2, k_2)$ has the attribute `extract` set, then the attacker can learn the symmetric encryption of k_2 with k_1, represented by the term $senc(k_2, k_1)$.

In a cryptographic API threat model we assume that the application is malicious and in league with the adversary. Thus, in particular applications should not learn keys in the clear. In [9] Clulow presented a number of attacks in which sensitive keys are compromised. One of the best-known attacks is the so-called "key separation attack". The name refers to the fact that the attributes of a key can be set and unset in such a way as to give a key conflicting roles, allowing the attacker to learn sensitive keys. Figure 1 shows an example of a key separation attack presented in [9]. In this attack the attacker learns the value of a sensitive

Initial knowledge: The intruder knows $h(n_1, k_1)$ and $h(n_2, k_2)$; n_2 has the attributes `wrap` and `decrypt` set, whereas n_1 has the attributes `sensitive` and `extract` set.

Trace:

Wrap:	$h(n_2, k_2), h(n_1, k_1)$	$\rightarrow$	$senc(k_1, k_2)$
SDecrypt:	$h(n_2, k_2), senc(k_1, k_2)$	$\rightarrow$	k_1

Fig. 1. Decrypt/Wrap attack in PKCS#11 v2.01

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:	$h(n_2, k_2), k_3$	$\rightarrow$	$senc(k_3, k_2)$
Unwrap:	$h(n_2, k_2), senc(k_3, k_2)$	$\xrightarrow{new\ n_3}$	$h(n_3, k_3)$
Set_wrap:	$h(n_3, k_3)$	$\rightarrow$	$wrap(n_3)$
Wrap:	$h(n_3, k_3), h(n_1, k_1)$	$\rightarrow$	$senc(k_1, k_3)$
Intruder:	$senc(k_1, k_3), k_3$	$\rightarrow$	k_1

Fig. 2. Encrypt/Unwrap attack in PKCS#11 v2.01

key by wrapping it and then decrypting the resulting cyphertext with a key that has the attributes for decryption (`decrypt`) and wrapping (`wrap`) set.

Clulow suggested that this attack could be avoided by restricting key attribute changing operations so that a stored key could not have both the `decrypt` and `wrap` attributes set. However, as described in [12], this restriction does not prevent other key separation attacks not discovered by Clulow. For example, as shown in Fig. 2, if the attacker imports its own key by first encrypting it under a key k_2 whose handle has the attributes `unwrap` and `encrypt` set, and then unwrapping it, then it can export a sensitive key k_1 under k_3 to discover its value.

Again, in [12] the authors showed that restricting key attribute changing operations to avoid the attacks shown in Figs. 1 and 2 is not enough to make the API secure. The attacker can learn a sensitive key k_1 performing the sequence of commands shown in Fig. 3 if it knows a handle $h(n_2, k_2)$ which has both the `wrap` and `unwrap` attributes set.

Furthermore, when asymmetric encryption is considered, PKCS#11 is subject to the "Trojan Wrapped Key" attack, first discovered in [9], and found later on in [12], too. In this attack the attacker has the ability to smuggle a key of his own choice onto the token. If there is a key pair $pub(s_1), priv(s_1)$ on the token such that $pub(s_1)$ can be used for encrypting data and $priv(s_1)$ for unwrapping keys then the attacker can first encrypt a known key k_3 under $pub(s_1)$ and then plant its trojan key by unwrapping k_3 into a new handle. The attacker can then use this Trojan key to export other keys from the device, which it can then decrypt and recover. Figure 4 shows the exchange of messages of this attack.

Initial state: The intruder knows the handles $h(n_1, k_1)$ and $h(n_2, k_2)$, and the key k_3; n_1 has the attributes `sensitive` and `extract` set, n_2 has the attribute `extract` set. The intruder also knows the public key pub(s1) and its associated handle $h(n_3, priv(s_1))$.

Trace:

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

Fig. 3. Wrap/Unwrap attack in PKCS#11 v2.01

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, n_2 has the attribute `extract` set. The intruder also knows the public key pub(s1) and its associated handle $h(n_3, priv(s_1))$.

Trace:

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

Fig. 4. Trojan Wrapped Key attack in PKCS#11 v2.01

Another experiment performed in [12] shows that more recent versions of the API that include new mechanisms to improve security are still subject to the same type of attacks. For example, version 2.20 of the PKCS#11[2] standard uses two more attributes: `wrap_with_trusted` and `trusted`. Whenever the "Wrap" command is executed, it tests whether if the key to be wrapped has `wrap_with_trusted` set then the wrapping key has `trusted` set. However, this does not prevent the attack shown in Fig. 5, where the attacker first attacks the trusted wrapping key, and then obtains the sensitive key k_1.

4 Specification of PKCS#11 in Maude-NPA

In this section we explain how to specify and analyze the core key management commands of PKCS#11 in Maude-NPA. First, in Sect. 4.1 we provide a

[2] Attacks shown in Figs. 1-4 actually correspond to attacks of PKCS#11 version 2.01, whereas the attack shown in Fig. 5 is an attack discovered for PKCS#11 version 2.20.

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

Trace:

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

Fig. 5. Wrap with trusted key attack in PKCS#11 v2.20

high-level description of how we model PKCS#11 API version 2.20 in Maude-NPA. Then, in Sect. 4.2 we show in detail the specification of the PKCS#11 commands in Maude-NPA's syntax.

4.1 Formal Model of PKCS#11 in Maude-NPA

It is well-known that specifying and verifying API protocols in protocol analysis tools is challenging. Although this is not the first time an API protocol is analyzed in Maude-NPA (see [19]), there are specific features of PKCS#11 API that make its specification and analysis in Maude-NPA a non-trivial task.

First, the number of keys and handles is infinite a priori, e.g., the "Unwrap" command allows creating new handles for an existing key. The approach taken in [12] bounds the number of keys and handles for each key that can be created. For example, to find the attack shown in Fig. 1 the authors allow a maximum of 3 symmetric keys and 2 handles for each symmetric key (see [12] Sect. 5). In [18] the authors assume a bound on the number of fresh values that are generated in the course of an attack, and prove that it is sound and complete for the static policies of the specific systems they consider. However, these bounds may not apply to other systems. By contrast, in this paper we do not impose these restrictions and perform the analyses in an *unbounded session model*, making no abstractions of fresh data, e.g., an infinite number of keys an nonces can be generated. There was one exception, the analysis where attack shown in Fig. 5, where ultimately we had to restrict the number of keys the attacker generates to prevent state space explosion. However, in this case the attacker could still generate an unbounded number of *fresh* handles by executing the "Unwrap" command. The work presented in [23] also considers an unbounded session model using no abstraction, but its goal is different from ours (see Sect. 6 for further details).

Another feature of PKCS#11 is use of a global mutable state consisting of attributes. We avoid this issue by taking advantage of previous work in this area by Fröschle and Steel [18] to simplify the types of policies that need to be analyzed. In [18] Fröschle and Steel define a construct they call an *attribute policy*, and show that, for a large class of "reasonable" attribute policies called *complete* policies, it is enough to prove security in the case of *static* policies, in which the attributes of a key are never changed after it is created. In [18] the attribute state is a function assigning attribute valuations (specifications of which attributes are set and which are not) to keys. An attribute policy is a finite directed graph whose nodes are the set of allowable states, and whose edges are allowed transitions between states. A complete policy is one in which the transition policy consists of a collection of disjoint, disconnected cliques, and for each clique C, and each pair of states $c_0, c1 \in C$, we have $c_0 \cup c_1 \in C$. This allows for certain natural behaviors for which Fröschle and Steel point out that any well-designed policy should take into account. They then show that each clique has a unique *end point* in which the attacker has the greatest power. Thus any attacker behavior allowed by the policy in which each clique is replaced by the end point is allowed by any node in the clique. Thus analysis of the end point policy will tell us whether or not the original policy was safe. Moreover, since the end point policy is static, it is enough to be able to analyze static policies.

The main difference between the Fröschle-Steel paper and the model used in the Maude-NPA analysis is in the definition of attribute state. The Maude-NPA analysis has no such notion of attribute state. However, it does keep a history that may include the setting and unsetting of attribute values, and these can be mapped to allowable states. For example, if the attribute `wrap` was set for a key, and it was not subsequently unset, we may conclude that that key currently has its attribute `wrap` set. It is thus also possible to define policies for PKCS#11 keys as they are represented in Maude-NPA by mapping the states both before and after a transition T to attribute states, and then determining whether the two states and the transition edge connecting them belong to the policy. This in particular allows us to define both complete and static policies in terms of Maude-NPA histories and thus apply the results of Fröschle-Steel to analyze only static policies with the assurance that this analysis applies to the associated complete policies as well.

The use of static policies means that we do not need to represent attribute values explicitly. Instead, we can express policies defined in terms of what attributes can be set for a given key in terms of what combinations of actions enabled by those attributes are allowed using specific keys, without compromising completeness. Thus, instead of saying that the attributes $decrypt(n)$ and $wrap(n)$ cannot both we set, we say that the attacker cannot perform both "decrypt" and "wrap" actions using the same key. This allows us, as in [18], to dispense with global state entirely.

Another issue is to what extent we can use Maude-NPA to search for attacks assuming a certain policy is being enforced. Fortunately, this is easy for static policies that require that certain pairs of conflicting attributes not be both set for the same key. This policy is enforced if and only if there is no history in which the functions associated with each of the two conflicting attributes are executed on the same key. This allows us to specify policies using *never patterns* as described in Sect. 2.3. For example, in order to enforce a policy requiring, say, that no key can have the attributes `wrap` and `decrypt` set we specify a never pattern describing a generic state in which both `wrap` and `decrypt` strands have executed using the same key. Such a policy is shown in Example 6. Note that the never pattern does not specify which strand completed first, just that each should have executed at some time in the past.

4.2 Specification of PKCS#11 in Maude-NPA's Syntax

In this section we describe the specification of the PKCS#11 v2.20 key management commands in Maude-NPA's syntax. The API's signature is specified as follows. A nonce generated by principal A is denoted by $n(A, r)$, where r is a unique variable of sort Fresh and A denotes the principal that generated the nonce. This representation makes it easier to specify and keep track of the origin of nonces. E.g., one can use the notation to specify a state in which a principal accepts a nonce as coming from A when it actually comes from some $B \neq A$. Handles are represented by terms of the form $h(n(A, r), K)$, where K can be either a symmetric or an asymmetric key. Symmetric keys are represented by terms of the form $skey(A, r')$, where A denotes a name and r' is a fresh variable. Public and private keys, and symmetric and asymmetric encryption are specified similarly as explained in Sect. 3.

Each command of the PKCS#11 API is specified in Maude-NPA as a strand. Table 2 shows the specification of the commands of Table 1 representing PKCS#11 actions in Maude-NPA's syntax as an example. More specifically, for each rule $T; L \overset{new\ \tilde{n}}{\rightarrow} T'; L'$, messages in T are represented as received messages, i.e., terms of the form $-(M)$, whereas messages in T' are represented as sent messages, i.e., terms of the form $+(M)$. The generation of fresh data denoted by *new* $\tilde{n}$ is represented by variables $r_1, \ldots, r_i$ of sort Fresh made explicit at the beginning of the strand.

Example 5. The "Unwrap (sym-sym)" command of Table 1, which generates a new fresh data r, is specified in Maude-NPA as the following strand:

$$:: r :: \ [-(h(N_2, K_2)), -(senc(K_1, K_2)), +(h(n(A, r), K_1))]$$

where N_2 is a variable of the sort for nonces, K_1 and K_2 are variables of the sort for symmetric keys, and r is a variable of sort Fresh used to create a new handle for K_1.

Additionally, in PKCS#11 the attacker can perform symmetric encryption and decryption and create any number of symmetric keys. We assume it knows any public key and can generate only its private key. The attacker can perform asymmetric encryption with any public or private key it knows. As explained in Sect. 2, we specify each one of these capabilities as a strand in Maude-NPA.

Table 2. PKCS#11 key management commands in Maude-NPA

Command	Specification in Maude-NPA
Wrap (sym/sym)	$:: nil :: [-(h(N_1, K_1)), -(h(N_2, K_2)), +(senc(K_2, K_1))]$
Wrap (sym/asym)	$:: nil :: [-(h(N_1, priv(A))), -(h(N_2, K_2)), +(aenc(K_2, pub(A)))]$
Unwrap (sym/sym)	$:: r :: [-(h(N_2, K_2)), -(senc(K_1, K_2)), +(h(n(A, r), K_1))]$
Unwrap (sym/asym)	$:: r :: [-(h(N_1, priv(B))), -(aenc(K_1, pub(B))), +(h(n(A, r), K_1))]$
SDecrypt	$:: nil :: [-(h(N, K)), -(senc(M, K)), +(M)]$
SEncrypt	$:: nil :: [-(h(N, K)), -(M), +(senc(M, K))]$
ADecrypt	$:: nil :: [-(h(N, priv(A))), -(aenc(M, pub(A))), +(M)]$
AEncrypt	$:: nil :: [-(h(N, priv(A))), -(M), +(aenc(M, priv(A)))]$

We specify constraints on conflicting attributes as follows. Since in our model we do not explicitly represent attributes, we express these conditions in terms of the commands enabled when the conflicting attributes are set by adding *never patterns* (see Sect. 2.3) to the attack states we use to perform the analysis in Maude-NPA. More specifically, these never patterns are specified in such a way that they discard states where these commands are executed using the same handle. Let us illustrate this idea with the example below.

Example 6. Let us consider the attack shown in Fig. 2 where `wrap` and `decrypt` are considered as conflicting attributes, i.e., a given handle cannot have both the `wrap` and `decrypt` attributes set. In order to search for this attack in Maude-NPA one can specify the attack pattern shown below, which has a never pattern that discards any state that has, at least, two strands using the same handle $(h(N, K))$: one for wrapping, and the other one for decryption.

$$\begin{aligned}
&\{SS\ \&\ \{skey(A, r1') \in \mathcal{I},\ IK\} \\
&\|\ never(\{ :: nil :: [-(h(N, K)), -(h(N_1, K_1)), +(senc(K_1, K))]\ \& \\
&\qquad\qquad :: nil :: [-(h(N, K)), -(senc(M, K)), +(M)] \\
&\qquad\qquad SS'\ \&\ \{IK'\}\})
\end{aligned}$$

Note that SS and SS' are variables of the sort for sets of strands, IK and IK' are variables of the sort for the intruder knowledge, A is a variable of the sort for names, and K_1 and K are variables of the sort for keys, and M is a variable of the sort for messages. The term $skey(a, r1')$ represents a sensitive key that should not be revealed to the attacker.

5 Experiments

We have specified the PKCS#11 API and analyzed several configurations in Maude-NPA following the methodology explained in Sect. 4. More specifically, we have rediscovered the attacks shown in Figs. 1, 2, 3, 4, and 5 (see Sect. 3).

Note that, unlike the experiments performed in [12] and in [18], we have not bounded the number of keys and handles that can be generated. The protocol specifications to reproduce our experiments are available on-line at http://www.dsic.upv.es/~sescobar/Maude-NPA/pkcs.html.

Table 3. Experimental Results

Attack	Length	dec / wrap	enc/ unwrap	wrap/ unwrap
Fig. 1	4	-	-	-
Fig. 2	7	✓	-	-
Fig. 3	6	✓	✓	-
Fig. 4	7	✓	✓	✓
Fig. 5	9	✓	✓	✓

Table 3 gathers the results of our experiments. For each one of the attacks explained in Sect. 3 we specify in the second column the length of the backwards reachability analysis performed by Maude-NPA until each attack was found. In the third, fourth, and fifth columns we show the different constraints on conflicting attributes that have been considered in each experiment. For example, the attack of Fig. 1 was found by Maude-NPA after 4 reachability steps, and no restriction on conflicting attributes was considered. The attack shown in Fig. 3 was found by Maude-NPA after 6 reachability steps and we considered two restrictions, namely that `decrypt` and `wrap`, and `encrypt` and `unwrap` are conflicting attributes.

In the experiments to find the attacks shown in Figs. 2 to 5 we used never patterns to specify policies on conflicting attributes. Additionally, in the case of the attack shown in Fig. 5 in order to control the size of the state search space we added an *attack preserving* never pattern (see [19]) that forces Maude-NPA to only search for states in which the attacker generated the minimal number of keys.

we added a never pattern to discard states containing more than one instance of the initial knowledge strand to reduce the state search space. Note that this never pattern preserves the completeness of the analysis because this PKCS#11 configuration is subject to an attack. That is, if Maude-NPA finds the attack when only one instance of the initial knowledge strand is allowed, it will still find the same attack (or an equivalent one) if several instances are allowed.

6 Related Work

There is a vast amount of research on the formal analysis of cryptographic APIs, so in this related work section we will concentrate on the work that is closest to ours, namely the formal analysis of PKCS#11 and PKCS#11-like systems and the use of cryptographic protocol analysis tools to analyze APIs.

Besides the work on formalizing and verifying PKCS#11 that we have already discussed, there has been further work focused on building tools for analyzing policies for PKCS#11 and PKCS#11-like systems. In [8] Centenaro et al. design a typed-base system for reasoning about the security of PKCS#11 policies, and use it to verify the security of new classes of PKCS#11 security policies they propose. This work is even able to verify implementations of PKCS#11. In [10]

Cortier and Steel develop a generic model for PKCS#11-like systems, and an algorithm and tool for verifying policies in this model. Interestingly, they show that a number of cryptographic protocols can also be modeled using their system, and they demonstrate their tool on them as well, thus showing that the relationship between cryptographic APIs and cryptographic protocols runs both ways.

The Tookan tool [6] can be used on PKCS#11 implementations. It reverse engineers security tokens, builds a formal model similar to that of [12], whose security can be checked by a model checker, and then runs any attack trace found directly on the token to validate it. The methods used by Tookan were further developed and commercialized in the commercial tool Cryptosense [11], which includes a component Cryptosense Analyzer that analyzes PKCS#11 configurations for insecurities and then tests attack traces on the system. Thus, analysis of PKCS#11 and systems like it, although challenging, has proved achievable enough to have potential commercial application.

An approach closer to ours is the work of Künnemann presented in [23], in that it relies on a protocol analysis tool, Tamarin [30] with some features in common with Maude-NPA (in particular, it performs backwards search over an unbounded number of sessions using no abstraction). In this paper Künnemann models PKCS#11 v2.20 in the Sapic calculus presented in [22], a variant of the applied pi calculus augmented with operators for state manipulation. This high-level protocol specification is then translated to a multiset rewrite system that can be verified using Tamarin. Using this tool-chain the author provides a configuration of the API and proves that it preserves the secrecy of sensitive keys.

However, the goal in [23] is different from ours. Instead of searching for attacks, Künnemann uses Tamarin to prove security of a particular configuration of PKCS#11 v2.20. This allows him to define a more restrictive model that includes their secure configuration but rules out others, in particular many of the configurations we analyzed with Maude-NPA. It also allows him to leave out certain features, such as asymmetric encryption, which, given the restrictions of his model does not give the intruder any more capabilities than symmetric encryption, and so can be omitted as redundant.

For the case of applying cryptographic protocol analysis tools one of the biggest issues facing the analysis of cryptographic APIs, is the problem of keeping the search space of a manageable size. Solutions that work for key distribution protocols, such as bounding the number of sessions, do not apply as well to cryptographic APIs, where the number and kind of "sessions" executed is under complete control of the adversary. The earliest work on formal analysis of APIs [21,25,26] dealt with the problem by relying to an extent on user input. For example, the analysis in [25] allowed users to tell the tool when they thought a state was reachable, and the analysis in [26] relied on lemmas that were conceived of and proved by the user, with machine assistance. More recently, the AVISPA analysis of PKCS#11 reported by Tsalapati in [32] uses simplifications such as a monotonic state and allowing only one handle per key.[3] According to [12],

[3] The thesis in which this work is contained is not publicly available, so we are relying on the account given in [12].

Steel and Carbone were able to enrich Tsalapati's AVISPA model to include non-monotonic state; however the number of sessions still needed to be bounded for analysis to be tractable, and the bounds needed to be relatively small. The Maude-NPA analysis of IBM's CCA API described in [19] relies on the use of *never patterns*, which can be used to tell Maude-NPA ignore classes of states specified by the user.

In [23], termination of the analysis is achieved, not only by restricting the kinds of configurations considered, but also by specifying model-specific heuristics that allow a more efficient evaluation of the operations that manipulate the protocol's state. In order to reduce the state search space and speed up the analysis the author defined a number of model-specific helping lemmas to rule out some states describing impossible situations or actions that do not allow the attacker to learn more knowledge in their model.

Another issue is the type of policies that need to be considered. In IBM CCA policies are represented in terms of separation of duties, which are straightforward to model in a tool such as Maude-NPA. For PKCS#11 the problem is apparently harder, since policies are expressed in terms of a global mutable state, Our first attempt to analyze PKCS#11 included a faithful model of PKCS#11 state, and analysis with respect to this model proved to be intractable. Steel and Carbone seemed to have encountered similar problems in their AVISPA analysis.

However, closer study reveals that there are natural restrictions on policies one can enforce. Fröschle and Steel [18] in particular show that it is possible to safely assume that policies are static, if one imposes some very natural restrictions on them. This means that one can specify policies in terms of which combinations of attributes are allowable. Künnemann also argues for static policies in [23], on the grounds of practicality and safety: the ability to turn attributes on and off is not needed, and can lead to security problems. The Tookan tool [6] restricts itself to a combination of static, sticky-on (once an attribute is turned on, it can't be turned off), and sticky-off (once an attribute is turned off, it can't be turned back on) attributes, on the grounds that this is what is normally seen in real implementations. These restrictions do much to make analyses more tractable and to limit the complexity of state representation. Indeed, in our Maude-NPA analysis we found we did not need to model state explicitly at all.

7 Conclusions

In this paper we have described the analysis of some PKCS#11 configurations in Maude-NPA, a cryptographic protocol analyzer tool that operates in the unbounded session model. This allowed us to perform the analysis of this API in a fully-unbounded session model making no abstraction nor approximation of fresh values, and with no assumptions about the policies other than that they were static. This in particular allowed us to reproduce attacks on PKCS#11 configurations found by Delaune et al. in [12].

We consider our work as complementary to that of [23]. In [23] Künnemann uses the protocol analysis tool Tamarin to prove security of a configuration in a restricted model. We use the protocol analysis tool Maude-NPA to reproduce attacks in a less restrictive model. This provides evidence that these tools can be of assistance in both proving security and in finding attacks, as they are for key generation and secure communication protocols.

What remains to be seen is how generally applicable these tools are to PKCS#11 and similar APIs. In particular, we note that our PKCS#11 analysis, although it was successful at reproducing attacks, did not achieve termination, so it is likely that Maude-NPA would not be helpful in proving security within the rather general model we use without some further improvements. However, we plan to keep on investigating this to determine to what degree performance can be improved, for example via the use of state space reduction techniques specific to these types of models. For example, it would be interesting to investigate the model-specific lemmas in [23] to see if they could be used in Maude-NPA. We also plan to investigate whether lemmas appropriate to other classes of models could be formulated and proved.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just fast keying in the pi calculus. ACM Trans. Inf. Syst. Secur. 10(3) (2007). doi:10.1145/1266977.1266978. http://dblp.uni-trier.de/rec/bib/journals/tissec/AbadiBF07
2. Anderson, R.J.: Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd edn. Wiley Publishing (2008). http://dblp.uni-trier.de/rec/bib/books/daglib/0020262
3. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. Int. J. Inf. Secur. **4**(3), 181–208 (2005)
4. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14), Cape Breton, Nova Scotia, Canada, pp. 82–96. IEEE Computer Society, June 2001
5. Bond, M.: Attacks on cryptoprocessor transaction sets. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 220–234. Springer, Heidelberg (2001)
6. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and fixing pkcs# 11 security tokens. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 260–269. ACM (2010)
7. Butler, F., Cervesato, I., Jaggard, A.D., Scedrov, A.: A formal analysis of some properties of kerberos 5 using msr. In: CSFW, p. 175. IEEE Computer Society (2002)
8. Centenaro, M., Focardi, R., Luccio, F.L.: Type-based analysis of PKCS#11 key management. In: Degano, P., Guttman, J.D. (eds.) Principles of Security and Trust. LNCS, vol. 7215, pp. 349–368. Springer, Heidelberg (2012)
9. Clulow, J.: On the security of PKCS #11. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 411–425. Springer, Heidelberg (2003)
10. Cortier, V., Steel, G.: A generic security API for symmetric key management on cryptographic devices. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 605–620. Springer, Heidelberg (2009)
11. Cryptosense. Cryptosense Web Page. https://cryptosense.com/

12. Delaune, S., Kremer, S., Steel, G.: Formal analysis of pkcs#11. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, 23–25 June 2008, Pittsburgh, Pennsylvania, pp. 331–344. IEEE Computer Society (2008)
13. Erbatur, S., Escobar, S., Kapur, D., Liu, Z., Lynch, C., Meadows, C., Meseguer, J., Narendran, P., Santiago, S., Sasse, R.: Effective symbolic protocol analysis via equational irreducibility conditions. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 73–90. Springer, Heidelberg (2012)
14. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: Cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, J., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009)
15. Escobar, S., Meadows, C., Meseguer, J., Santiago, S.: State space reduction in the Maude-NRL Protocol Analyzer. Inf. Comput. **238**, 157–186 (2014)
16. Thayer Fabrega, F.J., Herzog, J., Guttman, J.: Strand spaces: what makes a security protocol correct? J. Comput. Secur. **7**, 191–230 (1999)
17. Focardi, R., Luccio, F.L., Steel, G.: An introduction to security API analysis. In: Aldini, A., Gorrieri, R. (eds.) FOSAD 2011. LNCS, vol. 6858, pp. 35–65. Springer, Heidelberg (2011)
18. Fröschle, S., Steel, G.: Analysing PKCS#11 key management APIs with unbounded fresh data. In: Degano, P., Viganò, L. (eds.) ARSPA-WITS 2009. LNCS, vol. 5511, pp. 92–106. Springer, Heidelberg (2009)
19. González-Burgueño, A., Santiago, S., Escobar, S., Meadows, C., Meseguer, J.: Analysis of the IBM CCA security API protocols in Maude-NPA. In: Chen, L., Mitchell, C. (eds.) SSR 2014. LNCS, vol. 8893, pp. 111–130. Springer, Heidelberg (2014)
20. IBM. CCA basic services reference and guide: CCA basic services reference and guide for the IBM 4758 PCI and IBM 4764 (2008). http://www-03.ibm.com/security/cryptocards/pdfs/bs327.pdf.
21. Kemmerer, R.A.: Using formal verification techniques to analyze encryption protocols. In: IEEE Symposium on Security and Privacy, pp. 134–139. IEEE Computer Society (1987)
22. Kremer, S., Künnemann, R.: Automated analysis of security protocols with global state. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, 18–21 May, 2014, Berkeley, CA, USA, pp. 163–178 (2014)
23. Künnemann, R.: Automated backward analysis of PKCS#11 v2.20. In: Focardi, R., Myers, A. (eds.) POST 2015. LNCS, vol. 9036, pp. 219–238. Springer, Heidelberg (2015)
24. RSA Laboratories. PKCS#11: Cryptographic token interface standard. https://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm
25. Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Comput. Secur. **11**(1), 75–89 (1992)
26. Meadows, C.: Applying formal methods to the analysis of a key management protocol. J. Comput. Secur. 1(1) (1992)
27. Meadows, C., Cervesato, I., Syverson, P.: Specification and analysis of the group domain of interpretation protocol using NPATRL and the NRL protocol analyzer. J. Comput. Secur. **12**(6), 893–932 (2004)
28. Meadows, C., Syverson, P.F., Cervesato, I.: Formal specification and analysis of the group domain of interpretation protocol using NPATRL and the NRL protocol analyzer. J. Comput. Secur. **12**(6), 893–931 (2004)

29. Meadows, C.: Analysis of the internet key exchange protocol using the NRL protocol analyzer. In: IEEE Symposium on Security and Privacy, pp 216–231. IEEE Computer Society (1999)
30. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013)
31. OASIS. OASIS PKCS 11 TC. OASIS PKCS 11 TC Home Page. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11
32. Tsalapati, E.: Analysis of PKCS#11 using AVISPA tools. Master's thesis, University of Edinburgh (2007)