M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

# Design and Analysis of PKCS#11 key management With AIF

**Supervisor**: Sebastian Alexander Mödersheim

**Author**: Ming Ye
**Student No**: s114671

DTU

Kongens Lyngby 2014

**DTU Compute**
**Department of Applied Mathematics and Computer Science**
**Technical University of Denmark**

Matematiktorvet, building 303B,
DK - 2800 Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk

# Abstract

Nowadays, the security of cryptographic tokens is a big concern in the era of digital technology and internet. The Public-Key Cryptography Standards PKCS#11 defines API for security tokens such as hardware security modules (HSMs) and smartcards. This standard is widely adopted by most industries to enhance the security of their products. In this thesis we will analyze PKCS#11 Key Management API using the AIF framework which is based on an abstraction technique called set-abstraction. AIF allow us to model and automatically analyze distributed systems where each participant can have a database, e.g. HSM storing keys. This abstraction allows for the analysis without bounding the number of steps. During the AIF modeling and verification, we will experiment some intruder models and assumptions, and design rules and policies for PKCS#11 API in the AIF language.

# Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a Master Degree of Science in Computer Science and Engineering.

This thesis deals with the API security of key management of PKCS#11.

Kongens Lyngby, August 24, 2014

Ming Ye (s114671)

# Acknowledgements

My sincerest thanks go to my supervisor Sebastian Alexander Mödersheim who helped me to get a good start and lead me to the right theory and perspective while I was facing obstacles. I'm very grateful for his patience, enthusiasm and immense knowledge, without his guidance and support I may never reach a positive result of this thesis.

# Content

# Glossary:

| | |
|---|---|
| **PROVERIF** | Automated tool for protocol verification |
| **API** | Application Programming Interface |
| **Attribute** | A characteristic of a key |
| **Cryptoki** | API by PKCS#11 |
| **Token** | Security token, physical electronica device |
| **Application** | Any computer program that calls the Cryptoki interface. |
| **HSMs** | Hardware Security Module, a physical computing device |
| **Object** | The object class in Cryptoki |
| **State** | transition state of AIF, composed by a number of facts |
| **System** | That system running key management API of PKCS#11 |
| **SATMC** | Security protocol model checker. |
| **HC** | Horn Clauses |
| **ERACOM** | Eracom Technologies, a leading developer and global supplier of cryptographic technologies and solutions. |

# CHAPTER 1

# Introduction

## 1.1 Background

PKCS#11, a standard by RSA Laboratories, specifies an application programming interface (API) which allows application to interact with cryptographic tokens such as hardware security modules (HSMs) and smartcards. These devices hold cryptographic information and perform cryptographic functions. The token, like HSMs, can be used to store some high-value keys (e.g., TLS server keys, certificate signing keys, authentication keys) and like CA HSMs could be critical part of public key infrastructure or the HSMs used for ATM transaction and online banking application. Apparently, the concern about security issue of these tokens is growing. Many cryptographic functions, like encryption and decryption, are embedded into the HSMs token. The sensitive keys/data must be strictly prevented from leaking to an untrusted party/machine. This motivates the analysis of PKCS#11 key management and design rules/policies for it, since PKCS#11 is widely adopted on most of this type devices.

There have been some people's works of analyzing PKCS#11, such as G.Steel [8] analyzing PKCS#11 Key Management APIs using model checker SATMC [7], as well as a master thesis [9] which using AVISPA tools [11] for the analysis of PKCS#11. Tools like AVISPA are restricted by number of steps that participant can make, other tools like Scyther [10] are even restricted by "simple" protocol that only consists of message exchange. To overcome such limitations, we use AIF framework [1] for the modeling and the verification tool ProVerif [3] for model checking. The AIF is based on a novel set-abstraction technique that enables the modeling the databases of messages that do not necessarily monotonically grow and the modeling and verification are not restricted by the bounding number of steps that participants can make. We got good inspirations from the work of G.Steel [8], and we extend it from bounded to unbounded model by using AIF framework. We show that this framework is well-suited for analyzing the PKCS#11 key management system, since each security tokens may have a key databases to maintain and the keys are associated with "attribute" values, e.g. sensitive or decrypt…, which could be set/unset/.. and those actions happen frequently.

We conduct our work by using the AIF (Abstract Intermediate Format) framework which consists of the AIF specification language and a translator from AIF to first-order Horn clauses. The workflow is illustrated as below:

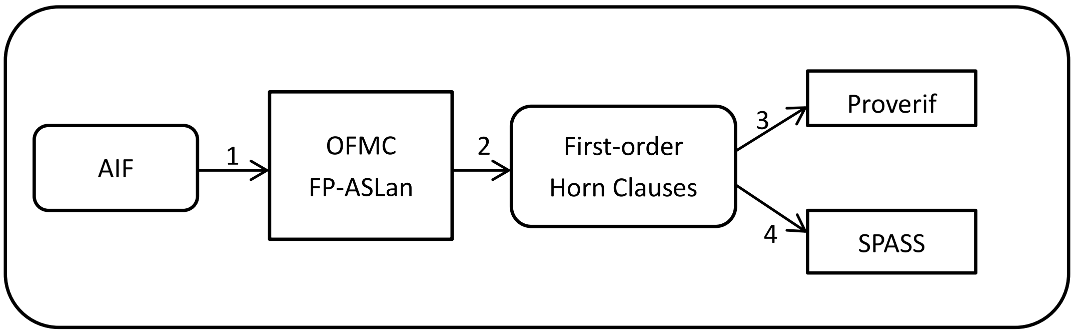


Figure 1 protocol verification workflow

First, we construct AIF file reflects the system modeling and input (step 1) it to the translator called OFMC-FP-ASLan [2] which is for translating (step 2) the AIF language to first-order Horn clauses. For checking the generated Horn Clauses that incorporates the set-abstraction, we input (step 3/4) the HC to protocol verification tool ProVerif (used in further works) or SPASS. When the tool stops, it either gives us an attack or proof of security. The attack could be caused by the tool's over-approximation like ProVerif, the methods of over-approximation do not consider a state transition system, but just a set of derivable facts like intruder knowledge.

## 1.2 PCKS#11 key management

The defined API by PKCS#11 is called Cryptoki, and there are many diverse cryptographic function sets inside Cryptoki, some examples are: Encryption Functions, Decrypt Functions, Object Management Functions, Key Management Functions, Singing and Macing Functions. See the full list from document [4].

Our design and analysis would base on the Key Management API of Eracom PKCS#11, this subset API include functions such as WrapKey, i.e. encryption of keys for secure transport, and UnwrapKey, generateKey…etc., that would be the crucial part of the tokens' security. Of course, we may also need to use some other essential functions, like encryption and decryption, as well as some functions of Object Management which allow us to set/unset attribute to object. In PKCS#11, an object could have many types, e.g. key object, certificate object, mechanism object, or domain parameter object….etc. Since we focus on the key management of PKCS#11, therefore objects mentioned in the following sections are referred to key objects.

## 1.3 Scope of Contribution

This work is inspired by a similar work of G.Steel [8], who modeled the Key Management APIs of PKCS#11 using SATMC [7]. His modeling was conducted by a SAT-based abstraction approach such an approach is bounded to a number of steps that honest or dishonest agent can perform. The AIF framework allows for modeling and verification with an unbounded number of steps that participants can make, unlike the standard model-checking approaches, AIF is based on a new way of abstraction, called set-abstraction that can handle a database in which the data/key can be added/removed and the states/facts do not monotonically growth. With those advantages, we show that AIF is a suitable language for studying HSM security, and begin with the APIs of key management. We show how we model the system and conduct the analysis in AIF.

Our work start with using AIF to formalize the modeling of the system which has basic API functions, some parts of that system are inspired by a typical known attack, so called “key-separation”, which was presented by Jolyon Clulow’s 2003 CHES paper [5]. We translate the AIF modeling to first-order Horn Clauses and solve it in ProVerif, after analyzing the output attack trace, we introduce some rules/policies as counter measures and apply them in AIF form, again this modified version shall be verified through ProVerif. From this start point, we successfully expand the modeling of the API system, and experience several intruder models, e.g. re-import attack which allow the intruder to obtain multiple handles (i.e. a pointer to a key) for the same key and reveal sensitive key in clear, and the loss-key attack where a certain type key loss to intruder. For preventing attacks, we do the analysis and present the solutions as AIF, then we verify the solutions through ProVerif, and from the verification result we can tell what rules must be applied to fix breaches. In the end, we successfully verified the system security properties.

# CHAPTER 2

# AIF - Abstract Intermediate Format

AIF is the language that we used for specifying security protocols in which allows each participant has a database, and enable us to analyze the protocol without bounding the number of step that participant can make. In this section, we briefly go through some basic features of AIF, the formal definition can be found in [1].

**State**: A state is a set of some true facts in that state, e.g.,
$\{ iknows(enc(M, PK)). iknows(PK). iknows(sign(inv(PK), M))\}$, in such a state intruder know a public-key encrypted message $enc(M, PK)$, the public key $PK$ and a signature signed by its private key $(sign(inv(PK), M)$. All the symbols/predicates above, include $iknows()$, do not have predefined meaning; their meaning is defined through transition rules which manage the transitions among different states. Some rules are intruder deduction rules that reflect intruder's capability. For example, if intruder gains message M and a public key PK, then he may able to encrypt such message; this can be reflected by transition rule:

$$iknows(M). iknows(PK) => iknows(enc(M, PK));$$

Likewise, if intruder has knowledge of the encrypted message and the private key, he can decrypt the cipher:

$$iknows(enc(M, PK)). iknows(inv(PK)) => iknows(M);$$

In the AIF language, variable names start with uppercase letters and constant names start with lowercase letters. The transition rule can be applied to the state that contains the facts matching the left hand side of arrow and new facts are generated on the right.
Transition rule that are not required to have any facts on the left hand side, it can be taken in any state, for instance:

$$= [PK] => iknows(inv(PK)). iknows(PK)$$

The term $= [PK] =>$ means that the value $PK$ is freshly created when the transition is taken, and the right hand side contains the facts relate to the new value. Interpret this rule as intruder can generate a key pair for himself at any time.

**Sets**: In the example of an HSM need to maintain a database of keys that has different status, e.g. valid, revoked or outdated. There are several classical abstraction approaches can model the protocol in unbounded steps that participant can make, but these approaches may cause the states/true facts monotonically growth, because the techniques however have a kind of monotonicity built-in: what is true at some point cannot become false later. To extend the classical approaches, AIF has a way to express transitions in which the state does not monotonically grow, namely using **Sets**. And AIF has fixed number of sets.

To get good insight of **Sets**, we use a running example that is similar to the SeVeCom case [6]: Assume that there are two security token namely$\{token1, token2\}$, and the key types are $\{root, ltsig, stsig, ltdec, stdec\}$, i.e. root key, long/short term signing key and long/short term decryption key and the status of key $\{valid,\ revoked,\ outdated\}$. For instance the all valid short term decrypt key in $token1$ can be denoted by the set $db(token1, stdec, valid)$, and the all revoked long term signing key in $token2$, $db(token2, ltsig, revoked)$. Totally, we have a family of 30 sets $db(TOKEN, KeyType, Status)$.

**Set-Memberhsip Transition**: Different from the normal fact, i.e. $iknows()$, the set-fact has the form $m \in S$ where $m$ is an element and $S$ is a set. For example to describe a key that is a valid long term signing key stored in $token1$ can be expressed as $k \in db(token1, ltsig, valid)$, and the transition between set-facts can be modeled by the rules like:

$$k \in db(token1, ltsig, valid) \Rightarrow k \in db(token1, ltsig, revoked);$$

Such rule may be applied when security token like HSM receives an API command to revoke the long term signing key $k$ of $token1$. Over the set-facts transitions, the state does not monotonically grow because the set-facts on left-hand side will be removed from system if they do not appear on right-hand side. For instance:

$$k \in db(token1, ltdec, valid). k \in db(token1, stdec, valid) => k \in db(token1, ltdec, valid)$$

A key $k$ is both a valid long term and short term decrypt key of $token1$, after applied this transition rule the set-fact that k is the short term decrypt key for token1 get removed, because the fact $k \in db(token1, stdec, valid)$ does not appear on the right-hand side.

Of course the facts that appear on the right-hand side but not on the left-hand side can be seen as the new generated facts, for instance:

$$k \in db(token1, ltdec, valid) => k \in db(token1, ltdec, valid). k \in db(token1, stdec, valid)$$

Key k is a valid long term decrypt key for token1, after this transition it becomes both the long term and short term decrypt key, because a new facts, $k \in db(token1, stdec, valid)$, appear on the right hand side and still the left-hand side fact, $k \in db(token1, ltdec, valid)$, is kept on the right.

**Goals**: There is one built-in fact symbol in AIF, which is **attack**. We can put the symbol at right-hand side of a rule to specify the attack state. E.g. if any valid root key revealed to intruder, it is an attack:

$$k \in db(TOKEN, root, valid). iknows(k) \Rightarrow attack;$$

After all, AIF allows protocol abstraction by set-membership that participants can have databases of keys where keys revocation is possible. For checking the AIF modeling, first we use a translator, called OFMC-FP-ASLan [2], for translating the AIF language to standard Horn clauses, then use the verifier ProVerif to solve them. But the tool doesn't not always terminate, once it stops, it give either an attack trace or the security proof. And we can tell whether our AIF specification is secure.

CHAPTER 3

# Attributes and Functions

In key management of PKCS#11, through API commands intruder/honest user is able to change states of keys and apply those key to do cryptographic operations. The states of key are determined by its attributes. Some API calls like set-attribute and unset-attribute can add/remove attribute to/from a key, we denoted such procedure as p1 in the graph below. The states of key decide the way of its application, the procedure of user apply keys through API is denoted as p2. The workflow of key management could be simply illustrated as following:

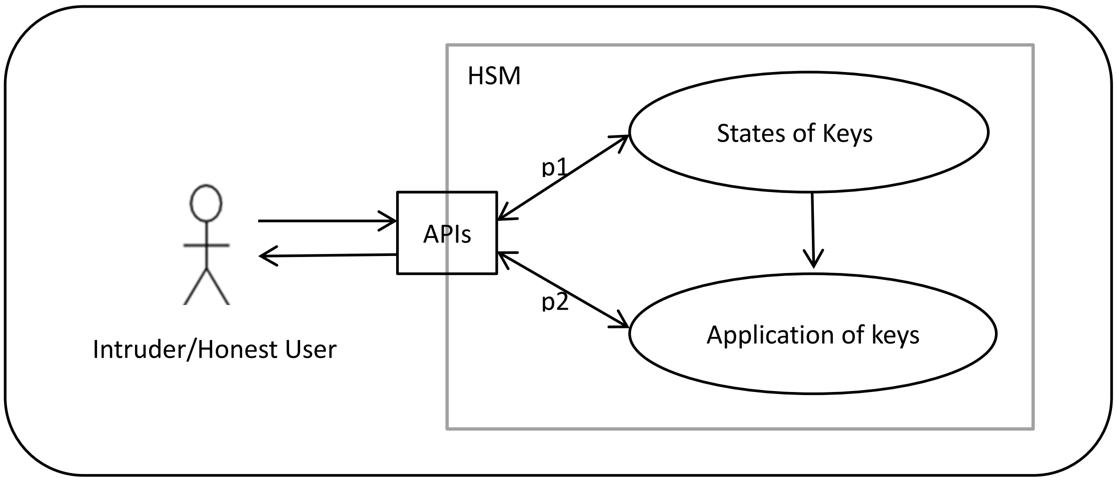


Figure 2 key Management workflow illustration

The key's attributes determine the states and through the API commands one can set/unset key's attributes and manipulate the key to perform some functions (listed below) such as wrap, unwrap…etc. Some functions may require the target key obtain the relevant attributes (listed below), for an instance a key can only be wrapped if it has the $extract$ attribute, likewise the $wrap$ attribute is the precondition for a key to wrap other keys.

Our modeling would cover the following functions of API calls and key attributes:

**Functions –API calls**

| | |
|---|---|
| Wrap: | Wrapping a key for transport purpose, generate the cipher text. |
| Unwrap: | unwrap a cipher that may contain the wrapped key and assign a handle to the key. (The handle, i.e. pointer to a key, see later section for more details). |
| Decrypt: | Decrypt cipher to get plain text of key. |
| Set Attribute: | Set an attribute to a key. |
| Unset Attribute: | Remove an attribute from a key. |

## Attributes

| | |
|---|---|
| Sensitive: | Sensitive key is high value key, which must not be revealed off token. It supports core functions of a token, such as PIN derivation. |
| Wrap: | Support the wrap function, a key that has the wrap attribute is used to wrap other keys. |
| Unwrap: | Support the unwrap function, a key that has the unwrap attribute is used to unwrap a cipher that may contain the wrapped key. |
| Decrypt: | Support the decrypt function, a key has decrypt attribute can be used to decrypt a cipher |
| Extract: | A key with extract attribute can be wrapped for transport purpose. |

## Attribute Modeling in AIF

We use the **Sets** in AIF to model key's attributes, because all the keys that have one certain type attribute can be treated as a key database. And the changes/shifting on key's attributes are exactly can be modeled by the transitions between Sets-Membership. For each type of attributes, we create the relevant set such as: $wrap(TOKEN)$, $unwrap(TOKEN)$, $decrypt(TOKEN)$, $sensitive(TOKEN)$, $extract(TOKEN)$. A wrapping key set of token can be expressed as: $wrap(token)$ and as well as the sensitive key: $sensitive(token)$. Of course a key may have several attributes, for modeling this we just use a combination of facts. For example a key has $extract$ and $wrap$ attribute can be denoted by:

$$K \in sensitive(token). K \in extract(token)$$

The transitions of set-facts can be used to express the changes of key's attributes, and for instance a HSM receives an API command to set key $k$ attribute to wrap, it can be denoted as:

$$k \in decrypt(token) \Rightarrow k \in wrap(token);$$

And to add wrap attribute to this key:

$$k \in decrypt(token) \Rightarrow k \in decrypt(token). k \in wrap(token);$$

The key k of token1 has decrypt attribute, apply this rule to give it wrap attribute as well.
To delete decrypt attribute from this key:

$$k \in decrypt(token) \Rightarrow iknow(i);$$

If left-hand side membership fact doesn't appear on the right-hand side, it means that fact does no longer exist in the modeling state space anymore. We put fact $iknow(i)$ on the right in case no new fact generated, because the right-hand part of transition rule cannot be empty. The fact $iknow(i)$ can be described as intruder knows his own name as a dummy fact.

# CHAPTER 4

# Analysis and Modeling Phase One

The security tokens, such as HSMs or Smartcards, can be viewed as a device that store objects and perform cryptographic functions with those objects. In PKCS#11, an object could be referred to a key, a certificate, a mechanism, or domain parameter .etc. And the aim of this thesis is to model and analyze the key management API of PKCS#11, that is the subset of Cryptoki functions, therefore the objects will always be referred to cryptographic keys. Each object is associated with an identifier, which is also called a handle that can be thought as a pointer to the object. One object is allowed to have several handles. In Cryptoki, the objects are manipulated through its handle, for example if the user wants to make an API call to use a key for decrypting a cipher text, and the API function need to know the handle of the decryption key to initialize the operation. Of course, to accomplish that operation, the decrypt attribute have to be set to the key. Attributes are the characteristics possessed by keys, so which functions can be applied to the keys is determined by attributes.

In this phase modeling and analysis, we aim to model the system that has the basic functions of key management APIs, which could help us to conduct the analysis of some trivial intruder models. And the ASCII syntax for the notation "∈" and "∉" in AIF is simply "in" and "notin", which are used in our modeling files (appendix).

## 4.1 Key Separation Attack

In AIF specification, we use a function to express the handle of a key by $h(K)$. And for representing the function of symmetric encryption, we use $senc(M,K)$ to denote the cipher of message $M$ encrypted by $K$.

In order to get our work started, the some parts of the first system modeling are inspired by an known attack presented by Jolyon Clulow [5] called key-separation, where a key may have conflicting roles. Conflicting means that a key both has wrap and decrypt attribute which allow intruder to manipulate this key to reveal other sensitive key in clear.

To model that system, we create 4 key sets and 5 rules in AIF.

The sets are: $sensitive(token1), extract(token1), wrap(token1), decrypt(token1)$

Each set reflect the one kind attribute in the system, and we use the following two rules to define the initial knowledge of intruder:

$$= [K1] => K1 \in sensitive(token1). K1 \in extract(token1). iknows(h(K1));$$

$$= [K2] => K2 \in wrap(token1). K2 \in decrypt(token1). iknows(h(K2));$$

The above transition rules can be taken by intruder at any time, that means intruder can create any value of key that has $sensitive$ and $extract$ attribute (first rule) or $wrap$ and $decrypt$ attribute (second rule). The key is stored in HSM token and intruder only knows the key's handle.

Above, we use normal fact to express the fact of intruder knowledge: $iknows(h(K1))$, and use set fact to denote the attribute characteristics of keys:
$K1 \in sensitive(token1).K1 \in extract(token1)$, likewise $K2$ has $wrap$ and $decrypt$ attributes and its handle known by intruder. The term $=[K1]$ means that the value $K1$ can be freshly created at any time in the system modeling, and the right hand side of transition decides the states of the new value.

To model function of wrap in AIF, we apply this transition rule:

$$\begin{aligned}&iknows\big(h(K1)\big).K1 \in extract(token1).iknows\big(h(K2)\big).K2 \in wrap(token1)\\&=> iknows\big(senc(K1,K2)\big).K2 \in wrap(token1).K1 \in extract(token1);\end{aligned}$$

This rule means that if intruder know two keys' handles, then he can wrap the key that has $extract$ attribute by using the other key which has $wrap$ attribute. After the operation, intruder gains the cipher text of the wrapped key, denoted by a normal fact:
$iknows\big(senc(K1,K2)\big)$. Note that the set facts on the left hand side will be removed by the transition rule if they do not appear on the right hand side. We keep the same set-facts on the right hand side, $K1 \in extract(token1).K2 \in wrap(token1)$, because the wrap operation doesn't change any keys' attributes.

The decrypt function:

$$\begin{aligned}&K2 \in decrypt(token1).iknows\big(h(K2)\big).iknows\big(senc(M,K2)\big)\\&=> iknows(M).K2 \in decrypt(token1);\end{aligned}$$

This rule expresses that if the intruder knows a handle of a key that has decrypt attribute and the cipher of a message which encrypted by this key, then he is able to conduct a decrypt operation on this message. Afterwards, the intruder gains the plaintext of the message, denoted as $iknows(M)$ in right hand side of the decrypt rule. Note that the message $M$ could be a key. Again, we keep the set-fact on right hand side to maintain the states of key, $K2 \in decrypt(token1)$.

The basic decrypt ability of intruder is reflected by the rule:

$$iknows(senc(M,K2)).iknows(K2) => iknows(M);$$

Intruder can decrypt a cipher if he gains the key that's used to encrypt the message.

The last rule describes the security goal, i.e. the attack fact:

$$K1 \in sensitive(token1).iknows(K1) => attack;$$

We say it is an attack when a sensitive key is known by intruder. The whole AIF file, **key_separation.aif**, of this modeling can be found in **appendix 1**.

We verified the system modeling through ProVerif and it gives us exactly the same "key-separation" attack trace presented by G.Steel in [8]. To describe the attack trace, here we use $M => M'$ denote that the intruder send command with message M to the HSM and receives M' as an answer, for instance this "key-separation" attack is expressed as following:

| Initial Knowledge of Intruder: $h(k1), k1 \in sensitive(token1), k1 \in extract(token1)$ $h(k2), k2 \in wrap(token1), k2 \in decrypt(token1)$ | | |
|---|---|---|
| 1 | WRAP | $h(k1).h(k2).k2 \in wrap(token1).k1 \in extract(token1)$ $=> senc(k1,k2);$ |
| 2 | DECRYPT | $h(k2).senc(k1,k2).k2 \in decrypt(token1) => k1;$ |

Figure 3 attack trace of key separation

The key $k1$ has attribute $sensitive$ and $extract$, and key $k2$ has attribute $wrap$ and $decrypt$. Both of their handles $h(k1)$ and $h(k2)$ are known by intruder and after a sequence of valid PKCS#11 commands, the sensitive key being revealed in clear. The extract attribute of $k1$ allow the key itself to be wrapped, and because intruder gains the knowledge of the handles of two keys, the wrap attribute of $k2$ allow intruder to apply k2 for wrapping (step 1) the sensitive key $k1$ and gain the cipher text $senc(k1, k2)$. Due to the conflicting attributes of $k2$, intruder can also exploit its decrypt attribute that enable him to apply the same key $k2$ to decrypt (step 2) the cipher and reveal the sensitive key $k1$ in clear.

## 4.2 Attack on the unset of attribute

In the APIs, intruder is able to set/unset an attribute to the key if he gain the knowledge of the key's handle. To model that, we could apply the 4 transition rules:

Set and unset wrap attribute:

$$iknows(h(K2)) => K2 \in wrap(token1);$$
$$K2 \in wrap(token1).iknows(h(K2)) => iknows(i);$$

Set and unset decrypt attribute:

$$iknows(h(K2)) => K2 \in decrypt(token1);$$
$$K2 \in decrypt(token1).iknows(h(K2)) => iknows(i);$$

But for preventing the attack in Figure 3, we might need a policy that no key should have both wrap and decrypt attribute. To that end, we add a precondition/fact for the "set" transition rule, for example, one can only set the wrap attribute to a key if the key doesn't have decrypt attribute, likewise, give the decrypt attribute to a key if the key doesn't obtain the wrap attribute. Therefore, we change the rules above into the following:

Set and unset wrap attribute:

$$K2 \notin decrypt(token1).iknows(h(K2)) => K2 \in wrap(token1);$$
$$K2 \in wrap(token1).iknows(h(K2)) => iknows(i);$$

Set and unset decrypt attribute:

$$K2 \notin wrap(token1).iknows(h(K2)) => K2 \in decrypt(token1);$$
$$K2 \in decrypt(token1).iknows(h(K2)) => iknows(i);$$

The whole AIF, **attack_unset.aif**, file of this modeling can be found in **appendix 2**.
And by the verification result, another trivial attack is revealed as following:

| Initial Knowledge of Intruder: $h(k1), k1 \in sensitive(token1), k1 \in extract(token1)$ $h(k2), k2 \in wrap(token1)$ | | |
|---|---|---|
| 1 | WRAP | $h(k1).h(k2).k1 \in extract(token1).k2 \in wrap(token1)$ $=> senc(k1, k2);$ |
| 2 | Unset Wrap | $h(k2).k2 \in wrap(token1) => k2 \notin wrap(token1);$ |
| 3 | Set Decrypt | $k2 \notin wrap(token1).h(k2) => k2 \in decrypt(token1);$ |
| 4 | Decrypt | $k2 \in decrypt(token1).h(k2).senc(k1, k2) => k1;$ |

Figure 4 attack trace of attribute unset attack

Note that the above set-facts such as $k2 \in decrypt(token1)$ or $k2 \notin wrap(token1)$ are not messages but shows the current state of key's attributes. Intruder uses key $k2$ to wrap the sensitive key $k1$, since he know both of their handles. After he got cipher $senc(k1, k2)$, unset wrap attribute from $k2$, and set its attribute as decrypt, and finally he can apply $k2$ to decrypt $senc(k1, k2)$ and obtains sensitive key $k1$ without breaking the policy of no key should have both wrap and decrypt attribute.

The attack in Figure 4 inspires us that we should declare the wrap and decrypt as "sticky" which cannot be unset. To model that, we delete those two transition rules of unset from our modeling:

Remove: $K2 \in wrap(token1).iknows(h(K2)) => iknows(i);$
$K2 \in decrypt(token1).iknows(h(K2)) => iknows(i);$

After those changes, we composed the new AIF, **attack_unset_revised.aif**, file of the modeling can be found in **appendix 3**. The verification result shows that no attack found on this modeling. But we can't say that it's the final proof of the system, because for the simplicity reason we didn't model the unwrap function of APIs at beginning of the work. The unwrap function will be included and analyzed in next sections. However, this verification result does support the policies that:

- ✓ A key is not allow to have both wrap and decrypt attribute
- ✓ wrap and decrypt attribute should not be unset

# CHAPTER 5

# Analysis and Modeling Phase Two

This phase focuses on modeling a more complicated system which includes the unwrap function. The verification result gives us a new intruder model called “Re-Import”, our analysis goes step by step until the proof of security of the system is found.

## 5.1 System Modeling

**The Handle of a key**

The wrap function is to wrap a key for transport purpose, so there must be a function to “unpack” the wrapped key, that is called unwrap. During the unwrap process, the wrapped key will be assigned with a fresh generated handle. As before, the handle of a key is modeled in AIF specification $h(k)$, that specification is difficult for us to model the wrap and unwrap function, especially on the modeling of generating new handle for a key. Therefore we change modeling of handle to the term $h(n, k)$. Interpret that $h$ is the function to bind the nonce $n$ with key $k$. Compare to $h(k)$ , it’s much easier for us to model the generation of a new handle. For instance, two different handles of the same key can be denoted as $h(n1, k)$ and $h(n2, k)$.

**Set-Membership of Handle**

In previous sections, the attribute of a key is modeled as set membership. E.g. A key has decrypt attribute can be expressed as the key is member of decrypt set, denoted as $k \in decrypt(token1)$. But now we model the set-membership on key handles, for example, a key’s handle expressed is as $h(n, k)$ and the handle has decrypt attribute is reflected on the nonce $n \in decrypt(token1)$. We made this change because in PKCS#11 API a key is associated with handles and handles are associated with attributes, of course a key may have several handles. We didn’t apply this modeling in early sections because modeling the set-membership on the key is already sufficient to reflect the early system version. And modeling the set-membership on the nonce would be easier for us to model a more complex system, for instance, the unwrap function gives a new handle to the wrapped key, and the new handle must inherit some attributes for restraining the key’s usage (see detail in later sections).

**Modeling of rules and functions**

The analysis in section 4 confirms that those two policies below is the foundation for our new development:

- ✓ A key is not allowed to have both $wrap$ and $decrypt$ attribute.
- ✓ $wrap$ and $decrypt$ attribute should be sticky/cannot be unset.

We update our AIF rules for the new modeling of handle of a key and implement those two policy into our new system modeling, i.e. one handle cannot obtain both $wrap$ and $decrypt$ attribute and we should not have rule for unset $wrap$ and $decrypt$ attribute. Based on all the changes we discussed above, rules are updated and some crucial rules are listed as below:

AIF rules of setting $wrap$ attribute and $unwrap$ attribute:

$$N \notin sensitive(token1).N \notin decrypt(token1).iknows\big(h(N,K)\big)$$
$$=> N \in wrap(token1);$$

New Rule: $N \notin sensitive(token1).iknows(h(N,K)) => N \in unwrap(token1);$

The second rule is new that allows for setting the unwrap attribute to a handle. The condition for setting the $wrap$ attribute to a key's handle is that this handle must not possess either the $sensitive$ attribute or $decrypt$ attribute. We add the new condition that the handle does not has $sensitive$ attributes, because sensitive keys are used for encryption, decryption/digital signing or signature verification….etc. And people use other type keys to wrap/unwrap the sensitive key for transport purpose. For example, a private key in HSM has the $sensitive$ attribute, and it's not allowed to have the $wrap$ attribute for wrapping other keys, but it may have the $extract$ attribute for being wrapped.

AIF rules of setting $decrypt$ attribute:

$$N \notin wrap(token1).iknows(h(N,K)) => N \in decrypt(token1);$$

Before set $decrypt$ attribute to a handle, check that the handle does not has $wrap$ attribute.

The WRAP rule:

$$iknows\big(h(N1,K1)\big).N1 \in extract(token1).iknows\big(h(N2,K2)\big).N2 \in wrap(token1)$$
$$=> iknows(senc(K1,K2)).N1 \in extract(token1).N2 \in wrap(token1);$$

To wrap one key with another key, the wrapping key must has the $wrap$ attribute and the other should has the $extract$ attribute to allow itself to be wrapped. Here, we use key $K2$ to wrap the key $K1$.

The UNWRAP rule:

$$iknows\big(senc(M,K)\big).iknows\big(h(N,K)\big).N \in unwrap(token1).$$
$$= [Nnew] => iknows(h(Nnew,M)).N \in unwrap(token1);$$

To be unwrapped, the data format must be $senc(M, K)$, here $M$ is an "untyped" value in AIF which could be replaced by any other values. We can say that any value wrapped by key $K$ can also be unwrapped by $K$ if $K$ has $unwrap$ attribute on its handle. During unwrapping, a fresh handle is generated and assigned to the unwrapped value $M$ which could be any message but we can treat it as key in this rule of unwrap.

The initial knowledge:

$= [K1, N1] => N1 \in sensitive(token1). N1 \in extract(token1).$
$\quad iknows(h(N1, K1)). K1 \in sensitive(token1);$
$= [K2, N2] => iknows(h(N2, K2)). N2 \in wrap(token1). N2 \in extract(token1);$

We apply above two rules to define initial knowledge of intruder; those two transitions can be taken by intruder at any time. He gains the knowledge of two handles, which can be expressed by normal facts: $iknows(h(N1, K1))$ and $iknows(h(N2, K2))$. And the handles' attributes are reflected on the nonce:

$$N1 \in sensitive(token1). N1 \in extract(token1)$$
$$N2 \in wrap(token1). N2 \in extract(token1)$$

Obviously, $h(N1, K1)$ could be the handle of sensitive key and $h(N2, K2)$ could be the handle of wrapping key. In the first rule of creating the sensitive key value, we give an attribute to the key value: $K1 \in sensitive(token1)$. We do this because it makes thing easier when analyzing the output of Proverif, i.e. to distinguish the sensitive key and understand the attack trace. We don't use the fact, $K1 \in sensitive(token1)$, as precondition for any transition rules except the rule of attack: $K1 \in sensitive(token1). iknows(K1) => attack$. Therefore, we can say that this fact doesn't have any impacts on the system modeling but it help on finding attack correctly.

The AIF file, **system_phase2.aif**, of this system modeling could be found in **Appendix 4**.

## 5.2 Re-Import Attack Version 1

We verify the system modeling, **system_phase2.aif**, through the automated tool and the crucial part of output from Proverif can be found in **appendix 5**. Here we used it as an illustration to see how the output from the tool looks like, all the verification result of modeling files will be attached with report. In the output of Proverif, the fact $iknows(h(N1, K1))$ is expressed as

$$iknows: h(val(Num1[], Num1[], Num0[], Num0[], Num0[]),$$
$$val(Num1[], Num0[], Num0[], Num0[], Num0[]))$$

Set membership of $N1$ is denoted as $val(Num1[], Num1[], Num0[], Num0[], Num0[])$ and Set membership of $K1$ is denoted as $val(Num1[], Num0[], Num0[], Num0[], Num0[]))$.
We are modeling 5 sets (attributes), respectively:

$$val(Sensitive, Extract, Wrap, Decrypt, Unwrap)$$

Therefore, we can see that N1 is member of set $sensitive(token1)$ and $extract(token1)$. The key value shall not have any set membership except it is a sensitive key (explained in previous section).

Note that all the verification result from Proverif are attached with the report but are not putted in the appendix. After the analysis on the verification result, we conclude the attack as following:

| Initial Knowledge of Intruder: $h(n1,k1), n1 \in sensitive(token1), n1 \in extract(token1), k1 \in sensitive(token1);$ $h(n2,k2), n2 \in wrap(token1), n2 \in extract(token1)$ | | |
|---|---|---|
| 1 | WRAP | $h(n1,k1).h(n2,k2).n1 \in extract(token1).n2 \in wrap(token1)$ $=> senc(k1,k2);$ |
| 2 | Set Unwrap | $h(n2,k2) => n2 \in unwrap(token1);$ |
| 3 | Unwrap | $senc(k1,k2).h(n2,k2).n2 \in unwrap(token1) => h(nnew,k1);$ |
| 4 | Set Wrap | $nnew \notin decrypt(token1).h(nnew,k1)$ $=> nnew \in wrap(token1);$ |
| 5 | WRAP: (self wrap) | $nnew \in wrap(token1).h(nnew,k1)$ $=> senc(k1,k1);$ |
| 6 | Set Decrypt | $h(n1,k1) => n1 \in decrypt(token1);$ |
| 7 | Decrypt | $senc(k1,k1).h(n1,k1).n1 \in decrypt(token1) => k1;$ |

Figure 5 attack trace of Re-Import Version 1

The central point of this attack is that the intruder can obtain multiple handles for the same key by calling the unwrap function, and use different attributes on those handles to reveal the sensitive key off the token in plain text. The trace above shows that intruder first wrap key $k1$ with $k2$ for getting $senc(k1,k2)$, then he gives the $unwrap$ attribute to the handle of $k2$, and unwrap the data $senc(k1,k2)$ with $k2$ (step3), this is where the new handle being generated and bound to $k1$. The new generated handle $h(Nnew,k1)$ is not assigned with any attributes, therefore intruder can set any attributes to it.

What happens next is that the intruder set $wrap$ attribute to the new handle (step 4) and by the knowledge of $h(Nnew,k1)$, he can do a key self-wrapping (step 5) which wrap $k1$ by $k1$ and gain $senc(k1,k1)$. At this time, the intruder may still keep knowledge of the original handle $h(n1,k1)$ of the same key $k1$, and set $decrypt$ attribute to it. Finally (step 7) intruder is able to decrypt message $senc(k1,k1)$ by applying the original handle of $k1$, $h(n1,k1)$.

## 5.3 Re-Import Attack Version 2

The analysis of the previous attack shows that the fresh generated handle in unwrap function breaks the security, because it's not bound with any attributes while being generated, intruder take advantage of this and could assign "conflicting" attributes to different handles of the same key. To counter this flaw, a good solution could be letting the fresh handle inherits the attributes from the other handle of the same key. E.g., in above case $h(n1,k1)$ and $h(Nnew,k1)$, both are the handle of the key $k1$, while the generation of $Nnew$, this new handle should inherit the same attributes from $n1$. Therefore, what the intruder cannot do with $n1$ (i.e. set the wrap attribute to $n1$), he also cannot do with $Nnew$. Those two handles must have same level of being manipulation.

To that end, we modify the wrap and unwrap rules in AIF:

The WRAP rule:

$$iknows(h(N1,K1)).N1 \in extract(token1).iknows(h(N2,K2)).N2 \in wrap(token1)$$
$$=> iknows(senc(K1,K2)).N1 \in extract(token1).$$
$$N2 \in wrap(token1).iknows(bind(N1,K1));$$

Compare to the previous wrap rule, what difference is that we introduced a new function, called $bind$, to bind the handle's attributes with the wrapped key, here expressed by term $bind(N1,K1)$. This is inspired by the work of G.Steel [8] who used an HMAC to bind the attributes to the wrapped key for preventing the "re-import" attack. They use a fresh key to generate the HMAC, and the fresh key is encrypted by the wrapped key. For simplicity reason, we modeled this process by the binding function: $bind(N1,K1)$. The fresh key is not modeled because it doesn't have any impacts on the system modeling. Because that if the fresh key is known by intruder, he cannot fake the HMAC message without knowing the wrapping key, and if the intruder knows the wrapping key, obviously he can gain the knowledge of the fresh key.

In AIF definition, we define $bind$ as a function which is used as precondition for unwrap rule:

The UNWRAP rule part 1, $sensitive$ attribute inheritance:

$$iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 \in sensitive(token1).$$
$$iknows(h(N2,K2)).N2 \in unwrap(token1).=[Nnew] =>$$
$$iknows(h(Nnew,M2)).Nnew \in sensitive(token1).$$
$$N2 \in unwrap(token1).M1 \in sensitive(token1);$$

In this rule, the attributes binding $bind(M1,M2)$ is one of the preconditions to unwrap the data $senc(M2,K2)$. Here, $M1$ and $M2$ is "untyped" value and could be replaced by any other values. In another word, we could say that to unwrap the key $M2$, its handle's attribute $M1$ must be accompany. If $M1$ has the $sensitive$ attribute, then this attribute must be inherited by the new generated handle. Why inherit $sensitive$ attribute to the new handle? Because it obviously counter the attack shown in Figure 5, i.e. when the new handle inherit the sensitive attribute, the intruder cannot set the wrap attribute to it (see the rule of set-wrap). That motive us to start experiment with this attribute first.

The UNWRAP rule part 2, case of other attributes:

$$iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 \notin sensitive(token1).$$
$$iknows(h(N2,K2)).N2 \in unwrap(token1).=[Nnew] =>$$
$$iknows(h(Nnew,M2)).N2 \in unwrap(token1);$$

Except the $sensitive$ attribute on $M1$, the other attributes will not be inherited to the new handle. The AIF file, **re_import_att_bind_attributes.aif**, of this system modeling could be found in **Appendix 6.**

Through verification, the attack trace is shown below:

| Initial Knowledge of Intruder: $h(n1,k1), n1 \in sensitive(token1), n1 \in extract(token1), k1 \in sensitive(token1)$; $h(n2,k2), n2 \in wrap(token1), n2 \in extract(token1)$ | | |
|---|---|---|
| 1 | Wrap: (self wrap) | $n2 \in wrap(token1).n2 \in extract(token1).h(n2,k2)$ $=> senc(k2,k2).bind(n2,k2)$; |
| 2 | Set-Unwrap | $h(n2,k2) => n2 \in unwrap(token1)$; |
| 3 | Unwrap | $senc(k2,k2).h(n2,k2).bind(n2,k2).n2 \in unwrap(token1)$ $=> h(nnew,k2)$; |
| 4 | Set Decrypt | $h(Nnew,k2) => nnew \in decrypt(token1)$; |
| 5 | WRAP | $h(n1,k1).h(n2,k2).n2 \in wrap(token1).n1 \in extract(token1)$ $=> senc(k1,k2)$; |
| 6 | Decrypt | $senc(k1,k2).h(nnew,k2).nnew \in decrypt(token1) => k1$; |

Figure 6 attack trace of Re-Import Version 2

In the attack of Figure 5, the fresh handle of sensitive key is without any constraints. After we updated the wrap and unwrap rules, the new generated handle will inherit the sensitive attribute while being unwrapped, that prevent the attack in Figure 5. But the intruder model shown in Figure 6 is based on the new generated handle of the wrapping key $k2$, this key $k2$ is manipulated to wrap itself, and because the wrapped key's handle doesn't possess sensitive attributes but has the $wrap$ and $extract$ attributes. Still, the intruder is able to give attributes to the new handle and that allows re-importing several copies of keys with different handles.

First, due to the intruder's knowledge $h(n2,k2)$, he can wrap key k2 by itself and gain $senc(k2,k2)$ and $bind\ (n2,k2)$. He sets (step 2) $unwrap$ attribute to the handle $n2$, and performs (step 3) the unwrap action on the wrapped key $senc(k2,k2)$, and the new handle is generated without inheriting any attributes from $n2$. Afterwards, the intruder set the decrypt attribute to the new handle and do a regular wrap action (step5) to get cipher data $senc(k1,k2)$, finally (step 6) decrypt $senc(k1,k2)$ by applying $h(Nnew,k2)$ and gain sensitive key $k1$ in plain text.

## 5.4 System Verified

Observe the attack in Figure 6, we can see that only copying the $sensitive$ attribute to the new handle is not enough, obviously the $wrap$ attribute should also be copied to prevent intruder to set any conflicting attributes to the new handle. E.g. in the attack model shown in Figure 5 the intruder set the $wrap$ attribute to the new handle which bound with a key and the key's original handle has the $sensitive$ attribute. As well as the attack in Figure 6 intruder set the $decrypt$ attribute to the new handle which belong to the key and the key's original handle has the $wrap$ attribute. Therefore, we decide to copy all the "sticky" attributes to the new generated handle during the UNWRAP operation, the "sticky" attributes which are not allowed to be unset, that includes the $sensitive$, $wrap$, and $decrypt$ attribute.

To do that, we modify the unwrap rule and it consists of four parts:

The UNWRAP rule part 1, $sensitive$ attribute inheritance:

$$\begin{gathered} iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 \in sensitive(token1). \\ iknows(h(N2,K2)).N2 \in unwrap(token1).=[Nnew] => \\ iknows(h(Nnew,M2)).Nnew \in sensitive(token1). \\ N2 \in unwrap(token1).M1 \in sensitive(token1); \end{gathered}$$

Part1 is same as earlier used for copying the $sensitive$ attribute.

The UNWRAP rule part 2, $wrap$ attribute inheritance:

$$\begin{gathered} iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 \in wrap(token1). \\ iknows(h(N2,K2)).N2 \in unwrap(token1).=[Nnew] => \\ iknows(h(Nnew,M2)).Nnew \in wrap(token1). \\ N2 \in unwrap(token1).M1 \in wrap(token1); \end{gathered}$$

Unlike the rule part 1, this part is for copying the $wrap$ attribute to the new handle. Similarly, intruder knows the attributes binding $bind(M1,M2)$, this is one of the preconditions to unwrap the data $senc(M2,K2)$ and if $M1$ has the $wrap$ attribute, then this attribute must be inherited by the new generated handle on the right hand side of the rule.

The UNWRAP rule part 3, $Decrypt$ attribute inheritance:

$$\begin{gathered} iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 \in decrypt(token1). \\ iknows(h(N2,K2)).N2 \in unwrap(token1).=[Nnew] => \\ iknows(h(Nnew,M2)).Nnew \in decrypt(token1). \\ N2 \in unwrap(token1).M1 \in decrypt(token1); \end{gathered}$$

This rule is for copying the $decrypt$ attribute to the new handle. As the rule above, the intruder knows attributes binding $bind(M1,M2)$ is one of the preconditions to unwrap the data cipher $senc(M2,K2)$ and if $M1$ has the $decrypt$ attribute, then it will be copied to the new generated handle.

The UNWRAP rule part 4, case of other attributes:

$$\begin{gathered} iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 \notin sensitive(token1). \\ M1 \notin wrap(token1).M1 \notin decrypt(token1).iknows(h(N2,K2)). \\ N2 \in unwrap(token1).=[Nnew] => \\ iknows(h(Nnew,M2)).N2 \in unwrap(token1); \end{gathered}$$

Part 4 is to handle the case that $M1$ doesn't contain any one of the attributes $sensitive$, $wrap$, and decrypt. Then the new generated handles will be bound without any attributes.
After applied those changes, the modeling file is expanded, **system_verified.aif**, could be found in **appendix 7**. We verify this system modeling in Proverif, and finally we have a positive result that no attack is found, the tool output the verification result in less than 2 sec.

## 5.5 Modeling of a lost key scenario

After the previous positive verification result, we now consider a scenario that a key lost to intruder by some means that beyond the scope of model, let it be a key that has the decrypt attribute since $decrypt$ is more “sensitive” compare to others like the $wrap$ and $unwrap$ attribute. Therefore, compare to the initial knowledge before, additionally we give intruder the lost key that its handle has $decrypt$ and $extract$ attribute, denoted in AIF:

$$= [K3, N3] => iknows\big(h(N3, K3)\big). N3\ in\ decrypt(token1).$$
$$N3\ in\ extract(token1). K3\ in\ decrypt(token1). iknows(K3);$$

We also give a decrypt-set membership on the key value that allows us to easier analyze the attack trace output by Proverif. The $extract$ attribute allow this key to be wrapped like other keys. With this lost key $K3$, intruder shall be able to decrypt any cipher that’s encrypted by this key (this AIF rule included in previous modeling), and he is also able to generate his own “$\mathrm{bind}$” message:

$$iknows(K3). iknows(h(N2, K2)) => iknows(bind(N2, K3));$$

If intruder know a key and a handle of any keys, he can bind the key with the attributes the handle possessed. We start using “bind” for binding the key’s attributes with the key in section 5.3, it’s introduced as a counter measure for “re-import attack”. This rule shows that if intruder has the lost key ($K3$), he can bind attributes that belong to a known handle $h(N2, K2)$ with this key. After applied those additional rules into the previous modeling, the AIF file, **lost_key_att.aif**, could be found in **appendix 8**. We checked this modeling in ProVerif, and it throws an attack:

| Initial Knowledge of Intruder: $h(n1, k1), n1 \in sensitive(token1), n1 \in extract(token1), k1 \in sensitive(token1);$ $h(n2, k2), n2 \in wrap(token1), n2 \in extract(token1);$ $h(n3, k3). k3, n3 \in extract(token1), n3 \in decrypt(token1), k3 \in decrypt(token1);;$ | | |
|---|---|---|
| 1 | Set-Unwrap | $h(n2, k2) => n2 \in unwrap(token1);$ |
| 2 | Bind | $h(n2, k2). k3 => bind(n2, k3);$ |
| 3 | WRAP | $h(n3, k3). h(n2, k2). n2 \in wrap(token1). n3 \in extract(token1) => senc(k3, k2);$ |
| 4 | Unwrap | $senc(k3, k2). h(n2, k2). n2 \in unwrap(token1). bind(n2, k3) => h(n2, k3);$ |
| 5 | WRAP | $h(n1, k1). h(n2, k3). n2 \in wrap(token1). n1 \in extract(token1) => senc(k1, k3);$ |
| 6 | Decrypt | $senc(k1, k3). h(n3, k3). n3 \in decrypt(token1) => k1;$ |

Figure 7 attack trace of lost key scenario

Because of the lost key $k3$, the intruder can fake a “Bind” message and using the fake message during unwrap operation, intruder can assign the desired handle to $k3$, e.g. intruder can assign the handle ($n2$) that has the $wrap$ attribute to the key $k3$ whose original handle ($n3$) possess $decrypt$ attribute. Therefore, using $k3$ intruder may encrypt the sensitive key by a wrap command and decrypt the sensitive key by $k3$ as well. (More details on attack trace shown below).

First, due to the intruder's knowledge $h(n2, k2)$ and $k3$, he set (step 1) the unwrap attribute to the handle of key $k2$ and fake (step 2) the attribute-bind message $bind(n2, k3)$. After a normal wrapping operation (step 3), intruder gains the cipher $senc(k3, k2)$. Now the intruder is ready for crucial step, the unwrapping (step4), he knows the handle $h(n2, k2)$ used for unwrapping and he inserts the faked message $bind(n2, k3)$ with $senc(k3, k2)$. Afterwards, the intruder the assign desired handle's attributes ($n2$) to $k3$, and gains $h(n2, k3)$. Now the key $k3$ have two handles $h(n2, k3)$ and $h(n3, k3)$, one ($n2$) has wrap attribute and the other ($n3$) has decrypt attributes. Eventually, the intruder can wrap (step5) key $k1$ by $h(n2, k3)$ and gains cipher $senc(k1, k3)$, afterwards apply $h(n3, k3)$ to decrypt (step6) cipher and get sensitive key $k1$ in plain text.

A good way to prevent such an attack is to add the wrapping key inside the "Bind" message, for example, when using wrapping $k2$ to wrap $k3$, it generates $bind(n1, k3, k2)$ which contains the handle's attributes of the wrapped key ($k3$) and both keys ($k2, k3$) of wrapping and being wrapped. Therefore, if the intruder only knows one of the keys, he's not able to fake the "Bind" message, but he can generate message like $bind(n1, k3, k3)$ which is not accepted by the unwrap operation. To apply this counter measure, we modified some AIF rule:

The WRAP rule:

$iknows(h(N1, K1)).N1 \in extract(token1).N2 \in wrap(token1).$
$iknows(h(N2, K2)) => iknows(senc(K1, K2)).N1 \in extract(token1).$
$N2 \in wrap(token1).iknows(bind(N1, K1, K2));$

Using one key to wrap other keys, now the wrapping operation generates a "Bind" message that include the handle's attributes of wrapped key and both participated keys.

For the unwrap operation, since it consists of 4 rules, and the changes made to each rule are similar, therefore we only illustrate one of its rules here:
The UNWRAP rule part 1:

$$iknows(senc(M2, K2)).iknows(bind(M1, M2, K2)).M1 \in sensitive(token1).$$
$$iknows(h(N2, K2)).N2 \in unwrap(token1).=[Nnew] =>$$
$$iknows(h(Nnew, M2)).Nnew \in sensitive(token1).$$
$$N2 \in unwrap(token1).M1 \in sensitive(token1);$$

Now the precondition for the unwrapping is changed, the $bind$ must contain keys of both wrapping and being wrapped, for instance, $senc(M2, K2)$ which stands for a cipher that encrypted by $K2$, and both $M2$ and $K2$ shall be inside the "Bind" message. Here, $M2$ can be treated as a key encrypted by $K2$.

Of course, intruder may have following abilities:

$$iknows(K3).iknows(K1).iknows(h(N2,K2)) => bind(N2,K1,K3).bind(N2,K3,K1);$$

This rule reflects that if intruder gains two keys ($K1$ and $K3$ are different keys), he is able to generate the "Bind" message, and if he only gains one key ($K1$ and $K3$ is same), then he still can generate similar "Bind" message like $\mathrm{bind}(n2,k3,k3)$. Finally, applied all the modification, we have our modeling file, **lost_key_att_countered.aif**, could be found in **appendix 9**. And the verification by ProVerif shows the absence of attacks.

CHAPTER 6

# Further improvement and Conclusion

We have presented the AIF framework for analyzing the key management APIs which is the subset of a fixed version API of PKCS#11, i.e. the version used by Eracom. This work was inspired by the similar work of G.Steel [8] who modeled the API system in bounded number of steps that participants can perform. Our work shows that this API system can be modeled and verified with unbounded number of steps. The other advantage we have is that the AIF modeling of security protocol does not have monotonically growth of states. To discover attacks, we translate the AIF modeling to first-order Horn clauses, and input it to verification tool Proverif. After the analysis of the intruder model, we design/suggest rules as a counter-measure and apply the rules in AIF. By this work flow, we experienced some intruder models and expanded our modeling step by step, and finally we verified the security properties of the API system.

For further improvement, first thing we can do is to extend our modeling to asymmetric cryptography, e.g. using a pair of keys to encrypt and decrypt, some basic rules could be:

$$= [K] => iknows\big(h(N1,K)\big).iknows\big(h(N2,inv(K))\big);$$
$$iknows\big(h(N1,K)\big).iknows(M) => iknows\big(asenc(M,K)\big);$$
$$iknows\big(h(N2,inv(K))\big).iknows\big(asenc(M,K)\big) => iknows(M);$$

Generating a key pair: public key $K$ and private key $inv(K)$, the asymmetric encryption can be expressed as $asenc(M,K)$ that can only be decrypted by the private key. Since our analysis only covered one subset of PKCS#11, i.e. the key management API, which means we could expand our modeling to more crucial subset, such as session management API and object management API. Moreover, except the PKCS#11 API system, our analysis method could be adapted to many complex API versions, such as the Secure Vehicle Communication system SeVeCom [6] which already being proved by using AIF, or the IBM Common Cryptographic Architecture that has proofs of security in bounded model and we may try to turn it to unbounded.

# References

1. S. Mödersheim, “Abstraction by Set-Membership – Verifying Security Protocols and Web Services with Databases”, in CSS 2010. Available at http://www2.imm.dtu.dk/~samo/
2. David Basin, Sebastian Mödersheim, Luca Viganò, “OFMC: A symbolic model checker for security protocols”, Department of Computer Science, ETH Zurich, 2004. Available at http://www.avispa-project.org/papers/ofmc-jis05.pdf, get latest module of OFMC from http://www2.imm.dtu.dk/~samo/
3. Bruno Blanchet. “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules”. In 14th IEEE Computer Security Foundations Workshop (CSFW-14), pages 82-96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society, available at http://prosecco.gforge.inria.fr/personal/bblanche/proverif/
4. RSA Laboratories, PKCS #11 Base Functionality v2.30: Cryptoki – Draft 4 available at *ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-30/pkcs-11v2-30b-d6.pdf*
5. J. Clulow. On the security of PKCS#11. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.3442
6. S. Mödersheim, “Verifying SeVeCom Using Set-based Abstraction”, in 7th wireless Communications and Mobile Computing Conference (IWCMC).
   Available at http://www2.imm.dtu.dk/~samo/
7. Alessandro Armando, Roberto Carbone, Luca Compagna. SATMC: A SAT-Based Model Checker for Security-Critical Systems. In the Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014), April 5-13, 2014, Grenoble, France, Springer, pp. 31-45.
8. G.Steel, “Analysing PKCS#11 Key management APIs with Unbounded Fresh Data”, University of Oldenburg, 2009.
9. E. Tsalapati. Analysis of PKCS#11 using AVISPA tools. Master thesis, University of Edinburgh, 2007.
10. C. Cremers, “The Scythe Tool: Verification, falsification, and analysis of security protocol,” in Computer Aided Verification. Springer, 2008, pp 414-418.
11. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Drielsma, P. Heam, O. Kouchnarenko, J. Mantovani et al., “The AVISPA tool for the automated validation of internet security protocols and applications," in Computer Aided Verification. Springer, 2005, pp. 281-285.

# Appendix

## Appendix 1

**key_separation.aif**

```
Problem: KEY_SEPARATION;

Types:
TOKEN :{token1};
K1,K2: value;
M: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/1;
private inv/1;

Facts:
iknows/1, attack/0;

Rules:
=[K1]=>K1 in sensitive(token1).K1 in extract(token1).iknows(h(K1));

=[K2]=>K2 in wrap(token1).K2 in decrypt(token1).iknows(h(K2));

% =====================wrap================
iknows(h(K1)).K1 in extract(token1).K2 in wrap(token1).iknows(h(K2)) =>
iknows(senc(K1,K2)).K2 in wrap(token1).K1 in extract(token1);

% =====================decrypt================
K2 in decrypt(token1).iknows(h(K2)).iknows(senc(M,K2)) =>
iknows(M).K2 in decrypt(token1);
iknows(senc(M,K2)).iknows(K2)=>iknows(M);

% =====================attacks================
K1 in sensitive(token1).iknows(K1)=>attack;
```

# Appendix 2

**attack_unset.aif**

```
Problem: ATTACK_UNSET;

Types:
D : {i};                              % Dishonest Agents
TOKEN :{token1};
K1,K2: value;
M: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/1;
private inv/1;

Facts:
iknows/1, attack/0;

Rules:
=[K1]=>K1 in sensitive(token1).K1 in extract(token1).iknows(h(K1));

=[K2]=>K2 in wrap(token1).iknows(h(K2));

% ======================wrap================
iknows(h(K1)).K1 in extract(token1).K2 in wrap(token1).iknows(h(K2))
=> iknows(senc(K1,K2)).K2 in wrap(token1).K1 in extract(token1);

% ======================set wrap================
K2 notin decrypt(token1).iknows(h(K2))=>K2 in wrap(token1);

% ======================unset wrap================
K2 in wrap(token1).iknows(h(K2))=> iknows(i);

% ======================set decrypt================
K2 notin wrap(token1).iknows(h(K2))=>K2 in decrypt(token1);
```

```
% =====================unset decrypt================
K2 in decrypt(token1).iknows(h(K2))=> iknows(i);

% =====================decrypt================
K2 in decrypt(token1).iknows(h(K2)).iknows(senc(M,K2)) =>
iknows(M).K2 in decrypt(token1);
iknows(senc(M,K2)).iknows(K2)=>iknows(M);

% =====================attacks================
K1 in sensitive(token1).iknows(K1)=>attack;
```

## Appendix 3

**attack_unset_revised.aif**

```
Problem: ATTACK_UNSET;

Types:
D : {i};                                    % Dishonest Agents
TOKEN :{token1};
K1,K2: value;
M: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/1;
private inv/1;

Facts:
iknows/1, attack/0;

Rules:
=[K1]=>K1 in sensitive(token1).K1 in extract(token1).iknows(h(K1));

=[K2]=>K2 in wrap(token1).iknows(h(K2));
```

```
% ======================wrap================
iknows(h(K1)).K1 in extract(token1).K2 in wrap(token1).iknows(h(K2))
=> iknows(senc(K1,K2)).K2 in wrap(token1).K1 in extract(token1);

% ======================set wrap================
K2 notin decrypt(token1).iknows(h(K2))=>K2 in wrap(token1);

% ======================set decrypt================
K2 notin wrap(token1).iknows(h(K2))=>K2 in decrypt(token1);

% ======================decrypt================
K2 in decrypt(token1).iknows(h(K2)).iknows(senc(M,K2)) =>
iknows(M).K2 in decrypt(token1);
iknows(senc(M,K2)).iknows(K2)=>iknows(M);

% ======================attacks================
K1 in sensitive(token1).iknows(K1)=>attack;
```

## Appendix 4

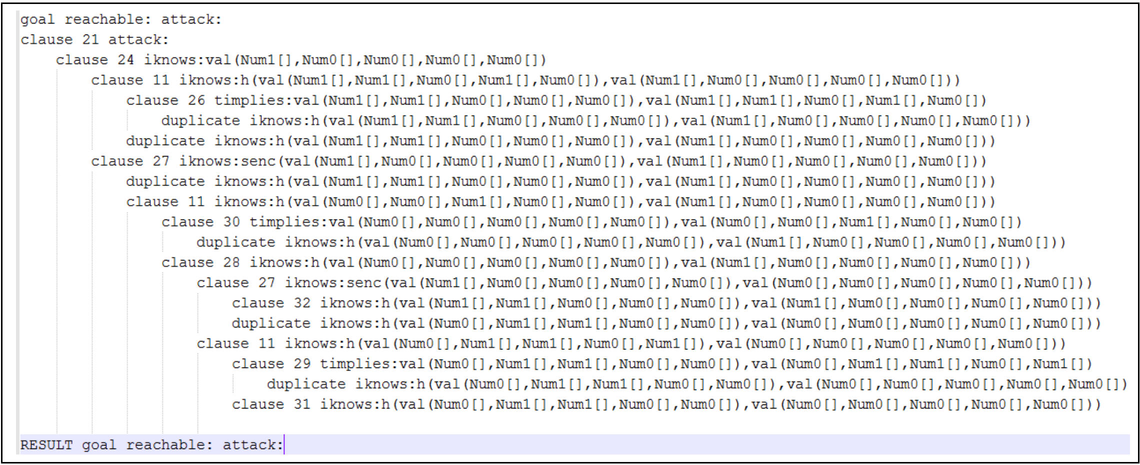
**system_phase2.aif**

```
Problem: SYSTEM_PHASE2;

Types:
D : {i};                                % Dishonest Agents
TOKEN :{token1};

K1,K2,N1,N2,Nnew: value;
M: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
unwrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/2;
private inv/1;
```

```
Facts:
iknows/1, attack/0;

Rules:
=[K1,N1]=>N1 in sensitive(token1).N1 in extract(token1).iknows(h(N1,K1)).K1 in
sensitive(token1);

=[K2,N2]=>iknows(h(N2,K2)).N2 in wrap(token1).N2 in extract(token1);

% =====set wrap=====
N2 notin sensitive(token1).N2 notin decrypt(token1).iknows(h(N2,K2)) =>
N2 in wrap(token1);

% =====set unwrap===
N2 notin sensitive(token1).iknows(h(N2,K2)) => N2 in unwrap(token1);

% =====unwrap, generate new handler======
iknows(senc(M,K2)).N2 in unwrap(token1).iknows(h(N2,K2)).=[Nnew]=>
iknows(h(Nnew,M)).N2 in unwrap(token1);

% =====================wrap================
iknows(h(N1,K1)).N1 in extract(token1).N2 in wrap(token1).iknows(h(N2,K2))
=> iknows(senc(K1,K2)).N1 in extract(token1).N2 in wrap(token1);

% =====set decrypt===
Nnew notin wrap(token1).iknows(h(Nnew,K2)) => Nnew in decrypt(token1);

% =====================decrypt================
Nnew in decrypt(token1).iknows(h(Nnew,K2)).iknows(senc(M,K2)) =>iknows(M);
iknows(senc(M,K2)).iknows(K2)=>iknows(M);

% =====================attacks================
K1 in sensitive(token1).iknows(K1)=>attack;
```

# Appendix 5

Partial output of verifying **system_phase2.aif**



```
goal reachable: attack:
clause 21 attack:
    clause 24 iknows:val(Num1[],Num0[],Num0[],Num0[],Num0[])
        clause 11 iknows:h(val(Num1[],Num1[],Num0[],Num1[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
            clause 26 timplies:val(Num1[],Num1[],Num0[],Num0[],Num0[]),val(Num1[],Num1[],Num0[],Num1[],Num0[])
                duplicate iknows:h(val(Num1[],Num1[],Num0[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
            duplicate iknows:h(val(Num1[],Num1[],Num0[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
        clause 27 iknows:senc(val(Num1[],Num0[],Num0[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
            duplicate iknows:h(val(Num1[],Num1[],Num0[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
            clause 11 iknows:h(val(Num0[],Num0[],Num1[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
                clause 30 timplies:val(Num0[],Num0[],Num0[],Num0[],Num0[]),val(Num0[],Num0[],Num1[],Num0[],Num0[])
                    duplicate iknows:h(val(Num0[],Num0[],Num0[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
                clause 28 iknows:h(val(Num0[],Num0[],Num0[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
                    clause 27 iknows:senc(val(Num1[],Num0[],Num0[],Num0[],Num0[]),val(Num0[],Num0[],Num0[],Num0[],Num0[]))
                        clause 32 iknows:h(val(Num1[],Num1[],Num0[],Num0[],Num0[]),val(Num1[],Num0[],Num0[],Num0[],Num0[]))
                        duplicate iknows:h(val(Num0[],Num1[],Num1[],Num0[],Num0[]),val(Num0[],Num0[],Num0[],Num0[],Num0[]))
                    clause 11 iknows:h(val(Num0[],Num1[],Num1[],Num0[],Num1[]),val(Num0[],Num0[],Num0[],Num0[],Num0[]))
                        clause 29 timplies:val(Num0[],Num1[],Num1[],Num0[],Num0[]),val(Num0[],Num1[],Num1[],Num0[],Num1[])
                            duplicate iknows:h(val(Num0[],Num1[],Num1[],Num0[],Num0[]),val(Num0[],Num0[],Num0[],Num0[],Num0[])
                        clause 31 iknows:h(val(Num0[],Num1[],Num1[],Num0[],Num0[]),val(Num0[],Num0[],Num0[],Num0[],Num0[]))

RESULT goal reachable: attack:
```

# Appendix 6

**re_import_att_bind_attributes.aif**

```
Problem: RE_IMPORT_ATT;

Types:
D : {i};                              % Dishonest Agents
TOKEN :{token1};

K1,K2,N1,N2,Nnew: value;
M1,M2: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
unwrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/2, bind/2;
private inv/1;

Facts:
iknows/1, attack/0;
```

```
Rules:
=[K1,N1]=>N1 in sensitive(token1).N1 in extract(token1).iknows(h(N1,K1)).K1 in
sensitive(token1);

=[K2,N2]=>iknows(h(N2,K2)).N2 in wrap(token1).N2 in extract(token1);

% =====set wrap=====
N2 notin sensitive(token1).N2 notin decrypt(token1).iknows(h(N2,K2)) => N2 in
wrap(token1);

% =====set unwrap===
N2 notin sensitive(token1).iknows(h(N2,K2)) => N2 in unwrap(token1);

% =====unwrap, generate new handler======
iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 in sensitive(token1).iknows(h(N2,K2)).N2
in unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in sensitive(token1).N2 in
unwrap(token1).M1 in sensitive(token1);

iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 notin
sensitive(token1).iknows(h(N2,K2)).N2 in unwrap(token1).=[Nnew]=>
iknows(h(Nnew,M2)).N2 in unwrap(token1);

% ====================wrap===============
iknows(h(N1,K1)).N1 in extract(token1).N2 in wrap(token1).iknows(h(N2,K2))
=> iknows(senc(K1,K2)).N1 in extract(token1).N2 in wrap(token1).iknows(bind(N1,K1));

% =====set decrypt===
Nnew notin wrap(token1).iknows(h(Nnew,K2)) => Nnew in decrypt(token1);

% ====================decrypt===============
Nnew in decrypt(token1).iknows(h(Nnew,K2)).iknows(senc(M1,K2)) =>iknows(M1);
iknows(senc(M1,K2)).iknows(K2)=>iknows(M1);

% ====================attacks===============
K1 in sensitive(token1).iknows(K1)=>attack;
```

# Appendix 7

**system_verified.aif**

```
Problem: RE_IMPORT_ATT;

Types:
D : {i};                                    % Dishonest Agents
TOKEN :{token1};

K1,K2,N1,N2,Nnew: value;
M1,M2: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
unwrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/2, bind/2;
private inv/1;

Facts:
iknows/1, attack/0;

Rules:
=[K1,N1]=>N1 in sensitive(token1).N1 in extract(token1).iknows(h(N1,K1)).K1 in
sensitive(token1);

=[K2,N2]=>iknows(h(N2,K2)).N2 in wrap(token1).N2 in extract(token1);

% =====set wrap=====
N2 notin sensitive(token1).N2 notin decrypt(token1).iknows(h(N2,K2)) => N2 in
wrap(token1);

% =====set unwrap===
N2 notin sensitive(token1).iknows(h(N2,K2)) => N2 in unwrap(token1);
```

```
% =====unwrap, generate new handler======
%----------the senstive attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 in sensitive(token1).iknows(h(N2,K2)).N2
in unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in sensitive(token1).N2 in
unwrap(token1).M1 in sensitive(token1);

%----------the wrap attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 in wrap(token1).iknows(h(N2,K2)).N2 in
unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in wrap(token1).N2 in
unwrap(token1).M1 in wrap(token1);

%----------the decrypt attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 in decrypt(token1).iknows(h(N2,K2)).N2 in
unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in decrypt(token1).N2 in
unwrap(token1).M1 in decrypt(token1);

iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 notin sensitive(token1).M1 notin
wrap(token1).M1 notin decrypt(token1).iknows(h(N2,K2)).N2 in
unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).N2 in unwrap(token1);

% ====================wrap===============
iknows(h(N1,K1)).N1 in extract(token1).N2 in wrap(token1).iknows(h(N2,K2))
=> iknows(senc(K1,K2)).N1 in extract(token1).N2 in wrap(token1).iknows(bind(N1,K1));

% =====set decrypt===
Nnew notin wrap(token1).iknows(h(Nnew,K2)) => Nnew in decrypt(token1);

% ====================decrypt===============
Nnew in decrypt(token1).iknows(h(Nnew,K2)).iknows(senc(M1,K2)) =>iknows(M1);
iknows(senc(M1,K2)).iknows(K2)=>iknows(M1);

% ====================attacks===============
K1 in sensitive(token1).iknows(K1)=>attack;
```

# Appendix 8

**lost_key_att.aif**

```
Problem: LOSS_KEY_ATT;

Types:
D : {i};                                  % Dishonest Agents
TOKEN :{token1};

K1,K2,K3,N1,N2,N3,Nnew: value;
M1,M2: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
unwrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/2, bind/2;
private inv/1;

Facts:
iknows/1, attack/0;

Rules:
=[K1,N1]=>N1 in sensitive(token1).N1 in extract(token1).iknows(h(N1,K1)).K1 in
sensitive(token1);

=[K2,N2]=>iknows(h(N2,K2)).N2 in wrap(token1).N2 in extract(token1);

=[K3,N3]=>iknows(h(N3,K3)).N3 in extract(token1).N3 in decrypt(token1).K3 in
decrypt(token1).iknows(K3);

% =====set wrap=====
N2 notin sensitive(token1).N2 notin decrypt(token1).iknows(h(N2,K2)) => N2 in
wrap(token1);

% =====set unwrap===
N2 notin sensitive(token1).iknows(h(N2,K2)) => N2 in unwrap(token1);
```

```
% =====unwrap, generate new handler======
%----------add the wrap attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 in wrap(token1).iknows(h(N2,K2)).N2 in
unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in wrap(token1).N2 in
unwrap(token1).M1 in wrap(token1);

%----------add the senstive attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 in sensitive(token1).iknows(h(N2,K2)).N2
in unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in sensitive(token1).N2 in
unwrap(token1).M1 in sensitive(token1);

%----------add the decrypt attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 in decrypt(token1).iknows(h(N2,K2)).N2 in
unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in decrypt(token1).N2 in
unwrap(token1).M1 in decrypt(token1);

iknows(senc(M2,K2)).iknows(bind(M1,M2)).M1 notin wrap(token1).M1 notin
sensitive(token1).M1 notin decrypt(token1).iknows(h(N2,K2)).N2 in
unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).N2 in unwrap(token1);

% =====================wrap================
iknows(h(N1,K1)).N1 in extract(token1).N2 in wrap(token1).iknows(h(N2,K2))
=> iknows(senc(K1,K2)).N1 in extract(token1).N2 in wrap(token1).iknows(bind(N1,K1));

% =====================bind generation================
iknows(K3).iknows(h(N2,K2))=>iknows(bind(N2,K3));

% =====set decrypt===
Nnew notin wrap(token1).iknows(h(Nnew,K2)) => Nnew in decrypt(token1);

% =====================decrypt================
Nnew in decrypt(token1).iknows(h(Nnew,K2)).iknows(senc(M1,K2)) =>iknows(M1);
iknows(senc(M1,K2)).iknows(K2)=>iknows(M1);

% =====================attacks================
K1 in sensitive(token1).iknows(K1)=>attack;
```

# Appendix 9

**lost_key_att_countered.aif**

```
Problem: LOSS_KEY_ATT;

Types:
D : {i};                                    % Dishonest Agents
TOKEN :{token1};

K1,K2,K3,N1,N2,N3,Nnew: value;
M1,M2: untyped;

Sets:
extract(TOKEN),
wrap(TOKEN),
unwrap(TOKEN),
decrypt(TOKEN),
sensitive(TOKEN);

Functions:
public senc/2, h/2,bind/3;
private inv/1;

Facts:
iknows/1, attack/0;

Rules:
=[K1,N1]=>N1 in sensitive(token1).N1 in extract(token1).iknows(h(N1,K1)).K1 in
sensitive(token1);

=[K2,N2]=>iknows(h(N2,K2)).N2 in wrap(token1).N2 in extract(token1);

=[K3,N3]=>iknows(h(N3,K3)).N3 in extract(token1).N3 in decrypt(token1).K3 in
decrypt(token1).iknows(K3);

% =====set wrap=====
N2 notin sensitive(token1).N2 notin decrypt(token1).iknows(h(N2,K2)) => N2 in
wrap(token1);

% =====set unwrap===
N2 notin sensitive(token1).iknows(h(N2,K2)) => N2 in unwrap(token1);
```

```
% =====unwrap, generate new handler======
%----------add the wrap attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2,K2)).M1 in wrap(token1).iknows(h(N2,K2)).N2
in unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).Nnew in wrap(token1).N2 in
unwrap(token1).M1 in wrap(token1);

%----------add the senstive attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2,K2)).M1 in
sensitive(token1).iknows(h(N2,K2)).N2 in unwrap(token1).=[Nnew]=>
iknows(h(Nnew,M2)).Nnew in sensitive(token1).N2 in unwrap(token1).M1 in
sensitive(token1);

%----------add the decrypt attr copy------------
iknows(senc(M2,K2)).iknows(bind(M1,M2,K2)).M1 in
decrypt(token1).iknows(h(N2,K2)).N2 in unwrap(token1).=[Nnew]=>
iknows(h(Nnew,M2)).Nnew in decrypt(token1).N2 in unwrap(token1).M1 in
decrypt(token1);

iknows(senc(M2,K2)).iknows(bind(M1,M2,K2)).M1 notin wrap(token1).M1 notin
sensitive(token1).M1 notin decrypt(token1).iknows(h(N2,K2)).N2 in
unwrap(token1).=[Nnew]=> iknows(h(Nnew,M2)).N2 in unwrap(token1);

% =====================wrap================
iknows(h(N1,K1)).N1 in extract(token1).N2 in wrap(token1).iknows(h(N2,K2))
=> iknows(senc(K1,K2)).N1 in extract(token1).N2 in wrap(token1).iknows(bind(N1,K1,K2));

% =====================bind generation================
iknows(K3).iknows(h(N2,K2))=>iknows(bind(N2,K3,K3));
iknows(K3).iknows(K1).iknows(h(N2,K2))=>iknows(bind(N2,K1,K3)).iknows(bind(N2,K3,K
1));

% =====set decrypt===
Nnew notin wrap(token1).iknows(h(Nnew,K2)) => Nnew in decrypt(token1);

% =====================decrypt================
Nnew in decrypt(token1).iknows(h(Nnew,K2)).iknows(senc(M1,K2)) =>iknows(M1);
iknows(senc(M1,K2)).iknows(K2)=>iknows(M1);

% =====================attacks================
K1 in sensitive(token1).iknows(K1)=>attack;
```