Security API analysis with the spi-calculus

Levente Buttyán *Ta Vinh Thong*
buttyan@crysys.hu *thong@crysys.hu*

Laboratory of Cryptography and Systems Security
Department of Telecommunications
Budapest University of Technology and Economics

Abstract: *API level vulnerabilities of hardware security modules represent a serious threat, thus, discovering and patching security holes in APIs are important. In this paper, we argue and illustrate that the application of formal verification methods is a promising approach for API analysis. In particular, we propose an API verification method based on process algebra. The proposed method seems to be extremely well-suited for API analysis as it allows for the straightforward modelling of the API, the precise definition of the security requirements, and the rigorous verification of the security properties offered by the API.*

1. Introduction

Hardware Security Modules (HSM) are indispensable in many applications, such as ATM[1] networks, public key infrastructures, electronic ticketing in public transportation, electronic payment systems, and electronic commerce, in general. A HSM is a hardware device (including the firmware and software components) which has some tamper resistance properties, and it is used to store cryptographic keys and to perform various security-critical cryptographic operations (e.g., generation of digital signatures and PIN[2] codes).

HSMs appeared in civilian applications starting from the late 1960s. At that time, driven by the explosion of the number of banking card forgery attacks, IBM (the main supplier of the computer systems of the banks) developed a system where the customer's PIN was computed from the account number placed on the card by encrypting it using a key called the *PIN derivation key*. Therefore, the protection of the *PIN derivation key* against both the bank employees and outside attackers became an importnat requirement. This led to the development of the IBM 3848 co-processor, which represenst the first generation of HSMs that were widely used in ATM networks later. Today, the application of HSMs is expanded, and besides the banking sector, they became widely used also in Public Key Infrastructures, in Automated Fare Collection systems, and generaly in electronic commerce.

The primary goal of attacking a HSM is to extract the secret data stored in it. The long list of potential attacks **[2]** starts with invasive attacks where the attacker physically penetartes the HSM and gains access to its internal parts, and it continues with non-invasive side channel attacks where the operational environment of the HSM (e.g., its timing and power consumption) is observed or manipulated. These attacks can be very effective, but at the same time, they often require expensive equipments. Finally, HSMs can also be attacked through their APIs by exploiting some design weaknesses in the API's logic. Being fully software based, this kind of attacks is much less expensive than physical and side-channel attacks, and depending on the weaknesses that are exploited, it may have devastating effects. This means that attacking HSMs through their APIs has a potentially high risk.

[1] Automatic Teller Machine
[2] Personal Identification Number

Many API attacks have been found against several widely-used, commercially available HSMs, which otherwise provide very strong physical protection **[3, 4, 5, 6, 7, 10, 11]**. Thus, discovering and patching security holes in APIs are required, ideally, still before the large-scale deployment of the HSMs. At the same time, APIs used in practice are complex, containing hundreds of functions, which renders their analysis difficult.

One promising approach of API analysis is to apply some formal verification method used in software engineering **[8, 9, 11, 12, 14, 16].** In this paper, we follow this approach, and propose an API verification method based on process algebra that seems to be extremely well-suited for the formal modelling of security APIs, the precise definition of the security requirements, and the rigorous analysis of the provided security properties. In particular, the here-in introduced method is based on the spi-calculus **[1]**, which was originally designed for analysing key exchange protocols. To the best of our knowledge, we are the first who use the spi-calculus for analysing security APIs.

In the rest of the paper, we first introduce API attacks against the Visa Security Modul in Section 2 for illustration and motivation purposes. Similar attacks also work against other HSMs. The subtlety of these attacks motivate the formal API analysis method introduced in Section 4. Our method is based on the spi-calculus, which is breifly reviewed in Section 3.

2. An API attack against the VISA Security Module

The primary function of the VISA Security Module (VSM) is to protect PINs transmitted over the ATM networks. VISA's goal in promoting this technology was to persuade member banks to connect their ATMs to VISA's network, so that a customer of one member bank could get cash from an ATM operated by another member bank. VISA wanted to minimize the loss that could be caused by dishonest or negligent employees at member banks. The goal was to ensure that no single employee of any bank in the network can learn the clear value of any customer's PIN. This means that PIN numbers should not simply be managed in the software running on the mainframes of the bank. Instead, PIN numbers are managed in a physically protected, tamper-resistant environment implemented by the VSM.

Due to the limitations of its internal memory size, theVSM only stores the most important master keys inside the module; other keys are stored outside secured under the master keys. The key storage method of the VSM follows a hierarchical structure [3] illustrated in Figure 1, which has the advantage of efficient key sharing. However, if a key at a top layer is compromised, every key below it in the hierarchy will be also compromised. The VSM uses five different master keys to encrypt other keys according to their relevancy and roles. The VSM supports nine key types to distinguish roles. As we can see, master keys are placed at the top layer of the hierarchy, and are illustrated as circles, and the nine key/data types are illustrated as rectangles at the lower layers. The keys that belong to a given layer and a given type are secured with the corresponding keys at the upper layers, except the master keys.

The master key *ZCMK* (Zone Control Master Key) is used to encrypt *ZCK* (Zone Control Key) keys. *ZCK*s are keys to be shared with other banking networks, used to protect the exchange of working keys. Working Keys (*WK*s) are used to protect trial PINs that customers have entered while they travel through the network on the way to the bank for verification, and are not used for intra-bank communications. Working keys are stored outside encrypted with the Working Master Key (*WMK*). Terminal Communications keys (*TCK*s) are for protecting control information going to and from ATMs, compute MACs of messages

exchanged between VSMs, and are secured with the Terminal Commucation Master Key (*TCMK*). The Terminal Master Key (*TMK*) and the PIN generation key (*P*) are very important keys and are considered as keys with the same relevancy. Thus, they are both encryted under master key *MK*, in other words, they are treated as the same key type. The *TMK* keys are shared between ATMs and used to protect all keys sent to an ATM. The PIN generation key is used to generate customer PINs, as we know . Finally, at the lowest layer we can find *user data* that are encrypted with the operational keys according to their type, where *X{ }* means that the user data is encrypted with a key of type *X*.

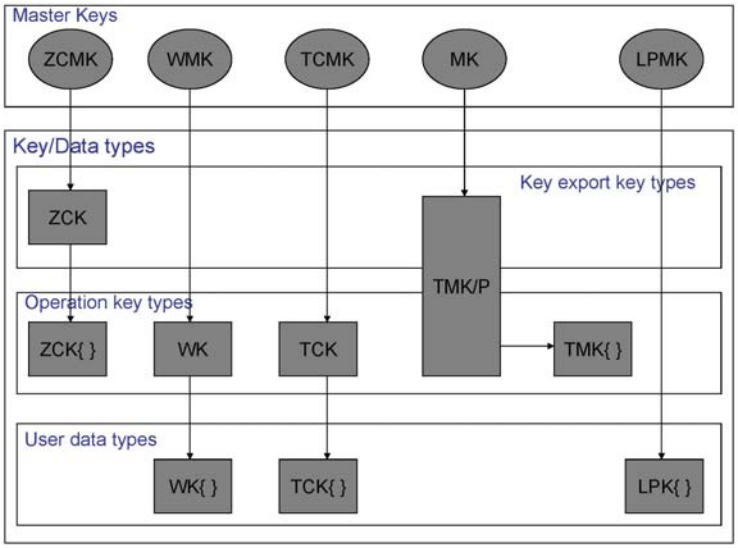

Figure 1. The key hierarchy of the VISA Security Module (VSM)

Before putting a new ATM in operation, the bank has to supply the ATM with every necessary key. To do this, first, a fresh *TMK* key is shared with the new ATM. All other keys are protected with this *TMK* during transmission to the ATM.

The generation of the key *TMK* is as follow: Function *GenerateKeyShares* of the VSM API is called by the Host:

$$Host \rightarrow VSM : "GenerateKeyShares"$$

The VSM generates a key part TMK_i, and at the same time, it prints the key part to a secure printer to which only authorized persons have access:

$$VSM \rightarrow SecurePrinter : TMK_i$$

Then, it returns the key part encrypted under the master key *MK* to the host.

$$VSM \rightarrow Host : \{TMK_i\}_{MK}$$

We assume that two key parts are required to construct the *TMK*. The key parts TMK_1 and TMK_2 printed by the secure printer are given to separate authorized couriers, who carry it to the new ATM and load it in. After receiving both parts of the key, the new ATM computes

the *TMK* key with XORing the two key parts, $TMK = TMK_1 \oplus TMK_2$. The same *TMK* key is produced at the bank with the *CombineKeyShares* command:

$$Host \rightarrow VSM : "CombineKeyShares", \{TMK_1\}_{MK}, \{TMK_2\}_{MK}$$
$$VSM \rightarrow Host : \{TMK_1 \oplus TMK_2\}_{MK} = \{TMK\}_{MK}$$

There exists an API attack that exploits the Terminal Master Key generation function above. Namely, instead of inputting $\{TMK_1\}_{MK}$ and $\{TMK_2\}_{MK}$, the host (or a programmer at the host) calls *Combine Key Shares* with inputing twice the same key token $\{TMK_1\}_{MK}$ (or $\{TMK_2\}_{MK}$).

$$Host \rightarrow VSM : "CombineKeyShares", \{TMK_1\}_{MK}, \{TMK_1\}_{MK}$$
$$VSM \rightarrow Host : \{TMK_1 \oplus TMK_1\}_{MK} = \{0\}_{MK}$$

Thus, the programmer can achieve that the all zero key becomes the *TMK*. He can then exploit this to produce customer PINs, since the PIN derivation key (*P*) is protected with the *TMK* key during transmission to the ATM for PIN verification. In other workds, the programmer can now easily decrypts $\{P\}_0$ with the key 0, and obtains *P* in clear. With the key *P,* he can generate the PIN of any customer.

There is an another attack that uses the *EncryptCommsKey* function of the API, which inputs a clear *TCK* key and returns the encrypted version under the master key *TCMK*. This key token is stored in an external storage.

$$Host \rightarrow VSM : "EncryptCommsKey", TCK$$
$$VSM \rightarrow Host : \{TCK\}_{TCMK}$$

As mentioned above, every key, including the *TCK* key, must be transferred to a new ATM. The transmission of the key *TCK* is also protected with the mastrer key *TMK*: $\{TCK\}_{TMK}$. The function *TranslateCommsKeytoTMK* ensures the generation of this key token:

$$Host \rightarrow VSM : "TranslateCommsKeytoTMK", \{TCK\}_{TCMK}, \{TMK\}_{MK}$$
$$VSM \rightarrow Host : \{TCK\}_{TMK}$$

The attack exploits that *TMK* and *P* are treated as having the same type. The malicious programmer calls *EncryptCommsKey*, but instead of inputting *TCK,* he inputs the customer's account number *PAN*:

$$Host \rightarrow VSM : "EncryptCommsKey", PAN$$
$$VSM \rightarrow Host : \{PAN\}_{TCMK}$$

Next, he calls *TranslateCommsKeytoTMK*, but instead of inputting $\{TCK\}_{TCMK}$, he inputs the resulted key token $\{PAN\}_{TCMK}$ of the previous step. Besides this, he inputs $\{P\}_{MK}$ instead of $\{TMK\}_{MK}$.

$$Host \rightarrow VSM : "TranslateCommsKeytoTMK", \{PAN\}_{TCMK}, \{P\}_{MK}$$

$$VSM \rightarrow Host : \{PAN\}_P = PIN$$

The returned value is the account number *PAN* encrypted under the PIN derivation key, which is exactly the PIN number of the account holder.

3. Overview of the spi-calculus

In this section, we give a brief overview of the spi-calculus **[1]**, an extension of the π-calculus **[13]** with cryptographic primitives. Similarly to the π-calculus, the spi-calculus can be seen as a programming language. Hence, the spi-calculus seems to be well-suited for modeling security APIs.

Syntax of the spi-calculus

In the spi-calculus, communication channels are represented with names. We assumme an infinite set of names. In addition, we assume an infinite set of variables that is important at initialization. Let *x*, *y*, and *z* range over variables, and let *m*, *n*, and *c* range over names. We distinguish *terms* and *processes*. Terms (messages, channel identification, keys, etc.) represent data, while processes describe behaviour. A term can be an atom, such as a constant or a variable, or it can be a complex term.

The set of terms is defined by the following grammar:

$L, M, N ::=$	*terms*
n	*name*
(M, N)	*pair*
0	*zero*
$suc(M)$	*successor*
x	*variable*
$\{M_1, M_2, ..., M_k\}_N$	*shared - key encryption*

As we can see, a term can be a name, a pair of terms, a constant zero, the successor of a given term, or a variable. We emphasize the term $\{M_1, M_2, ..., M_k\}_N$, which represents shared-key encryption, where N represents the key, and $M_1, M_2, ..., M_k$ terms represent the fields of the plaintext message.

The set of processes is defined by the following grammar:

$P, R, Q ::=$	*process*
$\overline{M}\langle N_1, N_2, \mathrm{K}\ , N_k \rangle.P$	*ouput* $(k \geq 0)$
$M(x_1, x_2, \mathrm{K}\ , x_k).P$	*input* $(k \geq 0)$
$P \mid Q$	$(parallel)$ *composition*
$(vn)P$	*restriction*
$!P$	*replication*
$[M \text{ is } N]P$	*match*
0	*nil process*
let $(x, y) = M$ *in* P	*pair splitting*
case M *of* $0 : P$ *suc*$(x) : Q$	*integer case*
case L *of* $\{x_1, x_2, \mathrm{K}\ , x_k\}_N$ *in* P	*shared - key decryption* $(k \geq 0)$

The above constructions of the spi-calculus have the following intuitive meanings:

- ***Output*** : Here, the term *M* represents a channel. This process is ready to output terms $N_1, N_2, ..., N_k$ on channel *M*. If a *reaction step* (see below) can occur, then terms $N_1, N_2, ..., N_k$ are sent on channel M and then process *P* runs.

- ***Input*** : This process is the pair of the ouput process. In a reaction step, an output process sends terms $N_1, N_2, ..., N_k$ as a message on channel *M*, and an input process inputs these terms from the same channel, and then process $P[N_1 / x_1, N_2 / x_2, \mathrm{K}\ , N_k / x_k]$ runs, where N / x represents the binding of variable *x* to term *N*. More precisely, variables are substituted with the inputted terms in process *P*.

- ***Composition*** (P|Q): This conctruction represents the parallel execution of processes *P* and *Q*. They can interact with each other via channels known to both, or they can interact with the outside world independently of each other.

- ***Restriction*** $(vn)P$: The process *P* creates a new local name n. This name cannot appear in other processes unless it has been sent explicitly during some communications. With this construction, we can model the generation of a new secret key.

- ***Replication*** $(!P)$: This construction represents an infinite number of copies of process P running in parallel.

- ***Match*** $([M \text{ is } N]P)$: This process behaves as *P* provided that terms N and *M* are the same; otherwise it is stuck, meaning that it does nothing.

- ***Nil process*** (0) : The nil process does nothing.

- ***Pair splitting*** $(let\ (x, y) = M\ in\ P)$: If $M = (N, L)$ holds, then process $P[N / x][L / y]$ will execute, otherwise the process will stuck.

- ***Integer case*** $(case\ M\ of\ 0 : P\ suc(x) : Q)$: This process behaves as P if term M is 0, and as $Q[N/x]$ if $M = suc(N)$. Otherwise, the process is stuck.

- ***Shared-key decryption***: Process *case L of* $\{x_1, x_2, ..., x_k\}_N$ *in P* attempts to decrypt the term L with the key N. If L is a ciphertext of the form $\{M_1, M_2, ..., M_k\}_N$, then the process will behave as $P[M_1/x_1, ..., M_k/x_k]$. Otherwise, the process is stuck.

As usual, there are some important assumptions made about cryptography and messages:

- The only way to decrypt an encrypted packet is to know the corresponding key.
- An encrypted packet does not reveal the key that was used to encrypt it.
- There is sufficient redundancy in messages so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected key.
- The attacker cannot find out or generate any secret data of the protocol.

Modeling secrecy property in the spi-calculus

In the spi-calculus the attacker is an arbitrary R process about which we assume only that at the beginning it does not have any secret data. The attacker process runs in parallel with the process that models the system, and they can interact (communicate) via public channels. The attacker attempts to obtain some secret data using only the information that he gets during the interaction.

Secrecy, which is a basic security property in the spi-calculus, is based on the *indistinguishability* of processes. Namely, the system P keeps data M secret, if for arbitrary data *M'*, the attacker process *R* cannot distinguish *P(M)* and *P(M')*.

A formal definition of indistinguishability in the spi-calculus is given by using the notion of *testing equivalence*. To make this clear, first we introduce some additional notions:

- **Free and bound variables:** Variable x is bound in process P if process P contains an input subprocess $m(x)$ (for arbitrary m). Variable x is free in process P if process P does not contain an input subprocess $m(x)$. Let $fv(P)$ denote the set of free variables in P.

- **Closed term/process :** We say that a term or process is closed if it has no free variables. In the spi-calculus, we assumed that the attacker process is closed.

- **Reaction step:** A reaction step arises from the interaction of an input process $m(x).Q$ and an output process $\overline{m}\langle M \rangle.P$. During the interaction the output process sends term M via channel m, while the input process receives it on channel m, and binds variable x to the received term. Then process Q runs with this term. Formally,

$$\overline{m}\langle M \rangle.P \mid m(x).Q \rightarrow P \mid Q[M/x]$$

- **Barb exhibiting:** Exhibiting a barb means that a process uses a given channel to send or receive messages. Barb exhibition is denoted by $\downarrow$. Exhibiting a barb is entirely independent from the content of the output or input messages. Barb exhibition is defined by the two axioms:

- **Barb In –** If a process *immediately* uses channel m to receive data, then it exhibits the barb m, namely, $m(x).P \downarrow m$.
- **Barb Out –** If a process *immediately* uses channel m to send data, then it exhibits the barb $\overline{m}$, namely, $\overline{m}\langle M \rangle.P \downarrow \overline{m}$.

- **Convergence**: *Convergence* intuitively means that a process does not definitely use a given channel immediately, but only after some reaction steps. Convergence is denoted by $\Downarrow$, and there are two related axioms:

  - If a process exhibits a barb β, then it will converge to β.
  - If a process *P* transforms to process Q that exhibits barb β, then process *P* will converge to barb β.

Next, after introducing the required notions, we give a formal definition of *testing equivalence*.

Definition (Testing equivalence): A test is a pair (R, β), where R is an arbitrary closed process and β is a barb (m or $\overline{m}$). Testing equivalence holds between *P* and *Q*, written as $P \approx Q$, if and only if $P \subseteq Q$ and $Q \subseteq P$ holds, where $P \subseteq Q$ holds if and only if $(P|R)\Downarrow\beta$ implies $(Q|R)\Downarrow\beta$ for any test (R, β).

Intuitively, $P \approx Q$ means that the behaviors of the processes *P* and *Q* are *indistinguishable* for any external observer *R*. More precisely, *P* and *Q* may have different internal structure, but a third process *R* cannot distinguish running in parallel with *P* from running in parallel with *Q*.

4. Modeling security APIs in the spi-calculus

Although the spi-calculus is designed for modelling key exchange protocols, we argue that it is also well-suited for modeling the interaction with a HSM via its API. This is because the interaction can be thought of as a set of two-party protocols, each describing an exchange of messages between the HSM and the user. We can model the entire API as the parallel composition of the replication of the processes that represent individual API function calls. We show an example in this section.

For this purpose, we first define a simplified security API. We assume that the security module has a master key, denoted by MK, which is stored inside the module. In addition, we distinguish two types of keys: data encryption keys (denoted by K_i), and key encryption keys (denoted by KEK_j), to which we link the type indicator constants $DataKey$ and $KEKKey$, respectively. Key tokens that contain a data encryption key K_i will carry a type indicator $DataKey$. Similarly, key tokens containing key encryption keys KEK_j will carry $KEKKey$ as a type indicator. We also tag encrypted data with the type indicator $TData$. In addition, we assume that the modul does not store K_i and KEK_j inside, instead it exports them in encrypted forms $\{DataKey, K_i\}_{MK}$ and $\{KEKKey, KEK_j\}_{MK}$ under the master key *MK*.

Our example API consist of four functions:

- **data-encryption:** This function inputs some data *Data* and some key token $\{DataKey, K_i\}_{MK}$. Then, it decrypts $\{DataKey, K_i\}_{MK}$ with the internally stored master key MK, and checks its type. If the type is $DataKey$, then it uses K_i to encrypt *Data*. Finally, it outputs the cipher $\{TData, Data\}_{K_i}$.

- **data-decryption:** This function inputs some encrypted data $\{TData, Data\}_{K_i}$ and some key token $\{DataKey, K_i\}_{MK}$. Then, it decrypts $\{DataKey, K_i\}_{MK}$ with the internally stored master key MK, and checks its type. If the type is $DataKey$ then it uses K_i to decrypt the cipher $\{TData, Data\}_{K_i}$. Finally, it checks if the type is $TData$, and if so, then it outputs *Data*.

- **data-key export:** This function takes two key tokens, $\{DataKey, K_i\}_{MK}$ and $\{KEKKey, KEK_j\}_{MK}$ as inputs. It decrypts both of them with the master key, and checks their types. If the types are $DataKey$ and $KEKKey$ respectively, then it encrypts $(DataKey, K_i)$ with KEK_j. It then outputs the key token $\{DataKey, K_i\}_{KEK_j}$. This token will be sent to another modul that may import key K_i.

- **data-key import:** This function takes two key tokens, $\{DataKey, K_i\}_{KEK_j}$ and $\{KEKKey, KEK_j\}_{MK}$ as inputs. It first decrypts $\{KEKKey, KEK_j\}_{MK}$ with the master key MK, and checks its type. Then it decrypts $\{DataKey, K_i\}_{KEK_j}$ with KEK_j, and checks its type. Finally, if the types are correct, it encrypts $(DataKey, K_i)$ with the master key, and outputs the key token $\{DataKey, K_i\}_{MK}$.

We can model the API defined above with the spi-calculus as follow: Let $MODULE^{ENC}$, $MODULE^{DEC}$, $MODULE^{EXP}$, $MODULE^{IMP}$ denote the data-encryption, data-decryption, data-key export and data-key import processes. Each process receives data (e.g., input arguments) via channels. The names $c_{enc}, c_{dec}, c_{\exp}, c_{imp}$ denote the communication channels through which the processes can receive data. Moreover, we define a channel $\overline{c_{user}}$ through which the processes output data to the environment. The formal definition of these processes is the following:

1. $MODULE^{ENC}(MK)$ @
$c_{enc}(x_{data}, x_{token0}).case\ x_{token0}\ of\ \{x_{typeK}, x_{K_i}\}_{MK}\ in\ [x_{typeK}\ is\ DataKey]$
$\overline{c_{user}} < \{TData, x_{Data}\}_{x_{K_i}} >$

2. $MODULE^{DEC}(MK)$ @
$c_{dec}(x_{token1}, x_{token2}).case\ x_{token2}\ of\ \{x_{typeK}, x_{K_i}\}_{MK},\ x_{token1}$
$of\ \{x_{typeData}, x_{Data}\}_{x_{K_i}}\ in\ [x_{typeK}\ is\ DataKey]\ [x_{typeData}\ is\ TData]$
$\overline{c_{user}} < x_{Data} >$

3. $MODULE^{EXP}(MK)$ @

$c_{\exp}(x_{token3}, x_{token4}).case\ x_{token3}\ of\ \{x_{typeK}, x_{K_i}\}_{MK}, x_{token4}$

$of\ \{x_{typeKEK}, x_{KEK}\}_{MK}\ in\ [x_{typeK}\ is\ DataKey]\ [x_{typeKEK}\ is\ KEKKey]$

$\overline{c_{user}} < \{DataKey, x_{K_i}\}_{x_{KEK}} >$

4. $MODULE^{IMP}(MK)$ @

$c_{imp}(x_{token5}, x_{token6}).case\ x_{token6}\ of\ \{x_{typeKEK}, x_{KEK}\}_{MK}, x_{token5}$

$of\ \{x_{typeK}, x_{K_i}\}_{x_{KEK}}\ in\ [x_{typeK}\ is\ DataKey]\ [x_{typeKEK}\ is\ KEKKey]$

$\overline{c_{user}} < \{DataKey, x_{K_i}\}_{MK} >$

Then, the API can be represented as the parallel composition of the replication of the above processes with an initial output of some key tokens. These key tokens are stored outside of the HSM, and thus, they are available to everyone (including the attacker).

$Sys_{API}(K_i, KEK_j)$ @

$$(vMK)\left(\begin{array}{l}\overline{c_{user}} < \{DataKey, K_i\}_{MK}, \{KEKKey, KEK_j\}_{MK} > . \\ \left(!MODULE^{ENC}(MK) \mid !MODULE^{DEC}(MK) \mid !MODULE^{EXP}(MK) \mid !MODULE^{IMP}(MK)\right)\end{array}\right)$$

It is possible to prove formally that this simplified API never leaks out keys in clear. In the formal proof we have to prove the following testing equivalences: $Sys_{API}(K_i, KEK_j) \approx Sys_{API}(K_i', KEK_j)$ and $Sys_{API}(K_i, KEK_j) \approx Sys_{API}(K_i, KEK_j')$, for every K_i, K_i', KEK_j, KEK_j'. The proof of this is based on induction. We assume that at first, the attacker process R does not has any key, that is, the system is in safe state. Then, we prove that if the system is in safe state, it will remain in safe state after any reaction step between process R and the system. This means that the attacker cannot extract any key from the system via its API. We omit further details of the proof here due to space limitations; the interested reader, however, can find the entire proof in **[15]**.

5. Conclusion

API attacks on hardware secutiy modules represent a serious risk. In this paper, we proposed a formal method for analysing security APIs. This method enables us to prove that an external attacker cannot extract any key from the modul via its API (given that indeed this is the case). A failed proof does not directly gives us an attack scenario, however, it often reveals the weak points of the API. The proposed method is based on the spi-calculus, which was originally designed for analysing key exchange protocols. In this paper, we showed that it can also be successfully used to analyise security APIs. Our experience shows that the spi-calculus is well-suited for this kind of analysis.

Acknowledgements

The work presented in this paper has been supported by the SeVeCom Project that receives funding from the European Commission withint the context of the 6th Framework Programme.

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the Spi calculus. Technical Report SRC RR 149, Digital Equipment Corporation, Systems Research Center, January 1998.

[2] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors – a survey . Technical Report UCAM-CL-TR-641, University of Cambridge, Computer Laboratory, August 2005.

[3] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the CHES 2001 Workshop*. Springer LNCS 2162, 2001.

[4] M. Bond. *Understanding security APIs*. PhD thesis, University of Cambridge, 2004.

[5] M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, October 2001.

[6] M. Bond and J. Clulow. Encrypted? Randomised? Compromised? (when cryptographically secured data is not secure). In *Proceedings of the Workshop on Cryptographic Algortihms and their Uses*, 2004.

[7] M. Bond and P. Zielinski. Decimalisation table attacks for PIN cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, January 2003.

[8] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19, July 2001.

[9] J. Clulow. The design and analysis of cryptographic APIs. MSc thesis, University of Natal, South Africa, 2003.

[10] J. Clulow. On the security of PKCS#11. In *Proceedings of the CHES 2003 Workshop*. Springer LNCS 2779, 2003.

[11] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of API-level vulnerabilities. In *Proceedings of the ACM/IEEE Conference on Software Engineering (ICSE)*, 2005.

[12] A. H. Lin. Automated analysis of security APIs. MSc Thesis, Massachusetts Institute of Technology, May 2005.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, September 1992.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC)*, June 2001.

[15] Ta Vinh Thong. Security API analysis with the Spi calculus. Student Scientific Conference (TDK), Budapest University of Technology and Economics, November 2007.

[16] P. Youn. The analysis of cryptographic APIs using the theorem prover otter. MSc Thesis, Massachusetts Institute of Technology, May 2004.