Defending the Bank with a Static Analysis *

Roberto Zunino

Dipartimento di Informatica, Università di Pisa, Italy
zunino@di.unipi.it

Abstract. The Common Crypto Arichitecture (CCA) defines the behavior of a small cryptographic device used in every automated teller machine (ATM). The CCA is meant to guarantee the confidentiality of the keys used internally by the device, while allowing common ATM operations to be performed. Some versions of the CCA were found to be vulnerable. An amended version was manually proved secure in [8], posing the question of whether fully automatic tools are up to the task. Here, we report about the automatic verification of the same fact.
Keywords: Common Crypto Architecture, ATM, API, verification, static analysis

1 Introduction

Automated teller machines (ATMs) are ubiquitous: their world-wide network allows any card owner to withdraw cash from them, anywhere, anytime. The volume of the transactions performed by ATMs every day is huge. Obviously, security is the main design concern for such a network.

The Common Crypto Architecture (CCA) [4,5] is the standard for the ATM architecture. The basic idea behind the CCA is to keep the trusted hardware and software base as small as possible. From this point of view, one should not trust the communication network between ATMs, since intercepting messages on it is feasible. Rather, only encrypted data should pass through this network. Also, an ATM is too large a system for it to be trusted. For instance, ATM software includes a component for the graphical user interface. ATMs also comprise a display, a printer, a card reader, a cash dispenser, and so on. These ATM components should have no access to the secret keys used for the actual transactions.

To this purpose, most of the ATM hardware and software is not trusted. Most ATMs are rather standard personal computers, equipped with a single small trusted device. An example of this device is the IBM 4758 coprocessor [5]. This device has a limited memory and processing power, just enough to perform the crypto operations involved in the

* Supported by the EU within the FETPI Global Computing, project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

ATM transactions. Further, the device is tamper-proof, so that physically breaking into it would trigger a self-destruct mechanism, making it unfeasible to retrieve the information stored in it. Accordingly, unencrypted secret keys are only stored *inside* the device. However, due the memory limitations, most keys are actually stored *outside* the device, protected by encryption. The encrypted keys are sent to the device on demand. The device will then decrypt the passed keys with its own keys, and proceed its computation.

The CCA specifies the Application Programming Interface (API), detailing the operations performed by the device. The API is meant to allow the common ATM transactions to be performed, while preserving the security of the ATM network. However, several attacks were found on the API. Bond [3] showed how to attack the CCA API, so to disclose the secret PIN of any card. Further attacks were discovered in [15]. After that, the known attacks were also automatically found by the theorem prover Otter [13]. Bond also provided a fixed version of the API, resilient to all known attacks. While the insecurity of the flawed API had been proved (even automatically), a correctness proof for the amended version was still missing. Courant and Monin [8] filled this gap, by providing a formal proof for it. The proof was written manually, and verified through the Coq proof assistant [7]. Coq surely offered a solid ground for the task at hand: indeed, having a machine-checkable proof makes one to be very confident about its correctness. The actual files comprising the proof contain about 2100 lines of hand-written definitions, lemmata, and theorems. This task required a non-trivial effort, and Courant and Monin themselves wondered if the same statement could be instead checked by a fully automated tool. They also note that, while the problem is similar to the ones arising in cryptographic protocol verification, the CCA API is more complex than the average protocol. While a protocol usually comprise a few, sequential, simple steps, the API has several, unordered, non trivial operations.

In this paper, we report on our success on validating the amended version, in a fully automated way through static analysis. To this purpose, we used the Rewrite tool [14].

Summary In Sect. 2 we define the operations underlying the CCA API. Then, we discuss Bond's attack in Sect. 2.2. The amended API is presented in Sect. 2.3. We consider the verification problem in Sect. 3. An abstract high-level specification for the API is checked in Sect. 3.2 and 3.3. In Sect. 3.4 a low-level refinement is presented and validated.

2 Background: The Common Crypto Architecture API

The Common Crypto API uses a fairly large number of constants, including *keys* and *type tags*.

- km is the *master key*, and is kept secret *inside* the device. Most other secrets are instead stored outside, in the ATM, and encrypted under km. More in detail, secrets are encrypted with both km and a *type tag* t: the encryption $\mathsf{enc}(x, \mathsf{xor}(t, \mathsf{km}))$ is meant to keep x secret and certify that x is of type t. Note that a XOR operation is used to combine the key km with the type tag t.
- $\mathsf{imp}, \mathsf{exp}, \mathsf{data}$ are type tags for *importation* keys, *exportation* keys, and *application data*, respectively. pin is a type tag for the p key (shown below).
- acc is the public account number for a card.
- p is a secret known by banks only. The secret PIN of any card can be computed with p as $\mathsf{enc}(\mathsf{acc}, \mathsf{p})$. So, it is very important for p to be kept secret. The key p is stored in the ATM as $\mathsf{enc}(\mathsf{p}, \mathsf{xor}(\mathsf{pin}, \mathsf{km}))$.
- kp is used to generate new tags. For any value x, $t = \mathsf{xor}(x, \mathsf{kp})$ is a type tag; this means that the actual key used for encryption is $\mathsf{xor}(x, \mathsf{kp}, \mathsf{km})$. Roughly, this tag t represents *incomplete* data of type x (as the parts of kek, shown below). The CCA API provides operations for performing some (limited) changes to incomplete data, i.e. to change the data under an encryption with the kp tag. An operation for completing the data and removing kp is also provided.
- kek is the "key encrypting key". It is known by both the ATM and the bank, and is used for confidential communication between them (e.g. exchanging p). The key kek is tagged as an importation key, and it is composed by three parts: $\mathsf{kek} = \mathsf{xor}(\mathsf{k1}, \mathsf{k2}, \mathsf{k3})$. This is to allow the transmission of kek using three distinct channels in a safe way: an adversary needs to intercept all the parts to learn kek. In normal CCA operations, the XOR between any two of these keys, e.g. $\mathsf{xor}(\mathsf{k1}, \mathsf{k2})$, is tagged as an incomplete importation key with kp and imp.
- $\mathsf{exp1}$ is an exportation key, and is tagged as such.

We can now define the API operations. To this purpose, we use term rewriting rules. We represent the operation as terms $\mathsf{f1}(-), \ldots, \mathsf{f6}(-)$.

$$\mathsf{f1}(X, Y, \mathsf{enc}(Z, \mathsf{xor}(X, \mathsf{kp}, \mathsf{km}))) \Rightarrow \mathsf{enc}(\mathsf{xor}(Z, Y), \mathsf{xor}(X, \mathsf{kp}, \mathsf{km}))$$
$$\mathsf{f2}(X, Y, \mathsf{enc}(Z, \mathsf{xor}(X, \mathsf{kp}, \mathsf{km}))) \Rightarrow \mathsf{enc}(\mathsf{xor}(Z, Y), \mathsf{xor}(X, \mathsf{km}))$$

These operations work on incomplete data Z, marked by the tag $\mathsf{xor}(X, \mathsf{kp})$. The first one, $\mathsf{f1}$, basically changes Z to $\mathsf{xor}(Z, Y)$. Note that there is no

restriction on Y, so the API user can perform arbitrary XOR operations on Z. The result is still marked as incomplete, as the kp tag still occurs in it. The second operation $\mathsf{f2}$, performs the same operation as $\mathsf{f1}$ but also removes the kp tag, thus forming a *complete* data. So, after a $\mathsf{f2}$ has been performed, it can not be applied again to its result.

The above operations are intended to be used to form the key kek from its components $\mathsf{k1}, \mathsf{k2}, \mathsf{k3}$. The XOR of them is performed by the API under the encryption layer, so that the owner of only one part can not learn anything about the others.

$$\mathsf{f3}(T, \mathsf{enc}(K, \mathsf{xor}(\mathsf{imp}, \mathsf{km})), \mathsf{enc}(X, \mathsf{xor}(T, K))) \Rightarrow \mathsf{enc}(X, \mathsf{xor}(T, \mathsf{km}))$$

The above operation *imports* a datum X. Initially, K is tagged as an importation key. Term X is instead tagged with T, but its encryption uses K instead of the usual km. The $\mathsf{f3}$ operation changes this encryption into the usual form, replacing K with km. So, this allows to use $\mathsf{enc}(K, \mathsf{xor}(\mathsf{imp}, \mathsf{km}))$ as a certificate, roughly meaning "K can be changed into km".

$$\mathsf{f4}(T, \mathsf{enc}(K, \mathsf{xor}(\mathsf{exp}, \mathsf{km})), \mathsf{enc}(X, \mathsf{xor}(T, \mathsf{km}))) \Rightarrow \mathsf{enc}(X, \mathsf{xor}(T, K))$$

The $\mathsf{f4}$ operation is the dual of $\mathsf{f3}$. It requires K be marked with the exportation tag. Then, replaces km with K in a tagged data X. Here the meaning of the certificate of K is "km can be changed into K".

$$\mathsf{f5}(X, \mathsf{enc}(K, \mathsf{xor}(\mathsf{data}, \mathsf{km}))) \Rightarrow \mathsf{enc}(X, K)$$
$$\mathsf{f6}(\mathsf{enc}(X, K), \mathsf{enc}(K, \mathsf{xor}(\mathsf{data}, \mathsf{km}))) \Rightarrow X$$

Operations $\mathsf{f5}, \mathsf{f6}$ instead consider keys K marked with the data tag. These kind of keys are for encrypting application data, so the device simply provides a mean for encrypting and decrypting with K. Note that K is still kept secret.

2.1 The CCA API Security Goals

The CCA API is supposed to keep several values secret. That is, no sequence of API calls should enable an adversary to learn these values. We summarize them here:

- km, the master key. Disclosing this would allow the adversary to read encrypted data meant only to be accessible by the device.
- $\mathsf{enc}(\mathsf{acc}, \mathsf{p})$ is the secret PIN of the card of account acc. Clearly, this would enable the adversary to freely withdraw money from ATMs, so it should be kept secret.

– p, the PIN-related key. Leaking p would enable the adversary to compute the PIN above, for any known account acc.
– kek, the "key encrypting key". The key p is encrypted with kek, so the latter should be kept secret.

Actually, the secrecy of the above values is subsumed by the secrecy of $\mathsf{enc}(\mathsf{acc}, \mathsf{p})$. However, considering the impact of the leak of the other secrets is insightful.

The adversary starts with an initial knowledge, comprising all the public known constants, as well as all the values that are stored outside the device.

– All the type tags: $\mathsf{data}, \mathsf{pin}, \mathsf{imp}, \mathsf{exp}, \mathsf{kp}$.
– The encrypted p, that is $\mathsf{enc}(\mathsf{p}, \mathsf{xor}(\mathsf{pin}, \mathsf{kek}))$.
– The encrypted exportation key $\mathsf{exp1}$. As usual, it is tagged with exp: $\mathsf{enc}(\mathsf{exp1}, \mathsf{xor}(\mathsf{exp}, \mathsf{km}))$.
– One part of the three components of $\mathsf{kek} = \mathsf{xor}(\mathsf{k1}, \mathsf{k2}, \mathsf{k3})$. Disclosing only one third of the kek key should not impact the API. Actually the API should be safe, even if *two* parts are leaked. Let the disclosed part be $\mathsf{k3}$.
– The encryption of the XOR of other two parts, as an incomplete importation key. That XOR would be $\mathsf{xor}(\mathsf{k1}, \mathsf{k2})$, or equivalently $\mathsf{xor}(\mathsf{kek}, \mathsf{k3})$. As done in [8], we choose the second form, since we can avoid introducing the constants $\mathsf{k1}, \mathsf{k2}$ in our specification. So, the initial knowledge also contains $\mathsf{enc}(\mathsf{xor}(\mathsf{kek}, \mathsf{k3}), \mathsf{xor}(\mathsf{imp}, \mathsf{kp}, \mathsf{km}))$.
– In [15], an additional incomplete importation and exportation key, $\mathsf{kek2}$, is also used, so we include it. The actual values known to the adversary are $\mathsf{enc}(\mathsf{kek2}, \mathsf{xor}(\mathsf{imp}, \mathsf{kp}, \mathsf{km}))$ and $\mathsf{enc}(\mathsf{kek2}, \mathsf{xor}(\mathsf{exp}, \mathsf{kp}, \mathsf{km}))$.

2.2 Bond's Attack

We now discuss Bond's attack [3] to the API defined above. First, we run $\mathsf{f2}$ as follows, using the known term $\mathsf{enc}(\mathsf{xor}(\mathsf{kek}, \mathsf{k3}), \mathsf{xor}(\mathsf{imp}, \mathsf{kp}, \mathsf{km}))$.

$$
\begin{aligned}
&\mathsf{f2}(\mathsf{imp},\\
&\quad \mathsf{xor}(\mathsf{k3}, \mathsf{pin}, \mathsf{data}),\\
&\quad \mathsf{enc}(\mathsf{xor}(\mathsf{kek}, \mathsf{k3}), \mathsf{xor}(\mathsf{imp}, \mathsf{kp}, \mathsf{km})))\\
&\Rightarrow \mathsf{enc}(\mathsf{xor}(\mathsf{kek}, \mathsf{pin}, \mathsf{data}), \mathsf{xor}(\mathsf{imp}, \mathsf{km}))
\end{aligned}
$$

The term produced above is unusual, in that normally only kek should have been tagged as an importation key. Instead, the importation key above also carries pin and data, XORed with kek.

We can then run f3, passing the above as its second argument:

$$\begin{aligned}&\mathsf{f3}(\mathsf{data},\\&\quad \mathsf{enc}(\mathsf{xor}(\mathsf{kek},\mathsf{pin},\mathsf{data}),\mathsf{xor}(\mathsf{imp},\mathsf{km})),\\&\quad \mathsf{enc}(\mathsf{p},\mathsf{xor}(\mathsf{data},\mathsf{xor}(\mathsf{kek},\mathsf{pin},\mathsf{data}))))\\&\Rightarrow \mathsf{enc}(\mathsf{p},\mathsf{xor}(\mathsf{data},\mathsf{km}))\end{aligned}$$

Note that the third argument above is indeed known to the adversary: by the XOR laws, is equivalent to $\mathsf{enc}(\mathsf{p},\mathsf{xor}(\mathsf{pin},\mathsf{kek}))$.

The result of f3 certifies p as a data key. This is indeed an unwanted result for the API, since now we can apply f5 to it.

$$\mathsf{f5}(\mathsf{acc},\mathsf{enc}(\mathsf{p},\mathsf{xor}(\mathsf{data},\mathsf{km}))) \Rightarrow \mathsf{enc}(\mathsf{acc},\mathsf{p})$$

Recall that $\mathsf{enc}(\mathsf{acc},\mathsf{p})$ is the secret PIN of the card of the account acc. So, the above actually computes the PIN of any card, given its public account number acc. Therefore, the CCA security was broken.

2.3 Bond's Amended API

One might argue that notations such as

$$\mathsf{xor}(\mathsf{data}, X)$$

are dangerous. The above formula suggests a fixed, rigid structure. Further, one might also be misled into believing that values of that form are somehow related to the constant data. Instead, *any* value Y may be easily written in the above form, as $Y = \mathsf{xor}(\mathsf{data},\mathsf{xor}(\mathsf{data}, Y))$. The attack of Sect. 2.2 exploits this fact. The API expects a value of the form above, intuitively relying on data being a "component" of it. The attack instead passes $\mathsf{xor}(\mathsf{kek},\mathsf{pin})$, leveraging the equation above.

Bond suggested an amended version of the API. The basic idea behind the fix is rather simple: do not use XOR in the second argument of enc. Using XOR does not ensure a rigid term structure in keys, and many algebraic rules are available to the adversary. Rather, one should use a primitive with as few algebraic rules as possible. Bond suggested using hashes for this. Below, we show the amended operations with a binary hash function.

$$\begin{aligned}&\mathsf{f1}(X, Y, \mathsf{enc}(Z, \mathsf{hash}(\mathsf{hash}(X,\mathsf{kp}),\mathsf{km}))) \Rightarrow \mathsf{enc}(\mathsf{xor}(Z,Y), \mathsf{hash}(\mathsf{hash}(X,\mathsf{kp}),\mathsf{km}))\\&\mathsf{f2}(X, Y, \mathsf{enc}(Z, \mathsf{hash}(\mathsf{hash}(X,\mathsf{kp}),\mathsf{km}))) \Rightarrow \mathsf{enc}(\mathsf{xor}(Z,Y), \mathsf{hash}(X,\mathsf{km}))\\&\mathsf{f3}(T, \mathsf{enc}(K, \mathsf{hash}(\mathsf{imp},\mathsf{km})), \mathsf{enc}(X, \mathsf{hash}(T,K))) \Rightarrow \mathsf{enc}(X, \mathsf{hash}(T,\mathsf{km}))\\&\mathsf{f4}(T, \mathsf{enc}(K, \mathsf{hash}(\mathsf{exp},\mathsf{km})), \mathsf{enc}(X, \mathsf{hash}(T,\mathsf{km}))) \Rightarrow \mathsf{enc}(X, \mathsf{hash}(T,K))\\&\mathsf{f5}(X, \mathsf{enc}(K, \mathsf{hash}(\mathsf{data},\mathsf{km}))) \Rightarrow \mathsf{enc}(X,K)\\&\mathsf{f6}(\mathsf{enc}(X,K), \mathsf{enc}(K, \mathsf{hash}(\mathsf{data},\mathsf{km}))) \Rightarrow X\end{aligned}$$

The API is unchanged, except for hashing having replaced XOR in the key arguments.

3 Towards Fully Automatic Verification

Courant and Monin [8], with their hand-written Coq-assisted proof, established the secrecy property of Sect. 2.1 w.r.t. the amended API. We now aim at achieving the same property, automatically.

For modelling the CCA API, we adopt (a slight variant of) the applied pi calculus [1], a simple process algebra [11] allowing processes to exchange arbitrary *terms*. In this calculus, terms are handled up-to some equivalence relation. Here, this equivalence is defined by the following *rewriting rules*, modelling the usual algebraic properties of XOR, as a binary function. These are the same properties used in [8].

$$\mathsf{xor}(X,Y) \Rightarrow \mathsf{xor}(Y,X) \qquad \mathsf{xor}(\mathsf{xor}(X,Y),Z) \Rightarrow \mathsf{xor}(X,\mathsf{xor}(Y,Z))$$
$$\mathsf{xor}(0,X) \Rightarrow X \qquad \mathsf{xor}(X,X) \Rightarrow 0$$

We also give rewriting rules for encryption and decryption.

$$\mathsf{dec}(\mathsf{enc}(M,K),K) \Rightarrow M \qquad \mathsf{enc}(\mathsf{dec}(M,K),K) \Rightarrow M$$

The first one is the usual cancellation rule. The second one is the *surjective encryption* rule, that is satisfied by some cryptosystems. The CCA does not require that rule to hold, in order to function properly. However, if it does happen to hold in the particular cryptosystem used in the API implementation, the adversary could use the rule to attack the API. So, by including it, we consider this stronger adversary model. Finally, we have no rewriting rules for $\mathsf{hash}(X,Y)$: the hash algebra is free.

The syntax of the applied pi processes is shown below: M, M' are terms, X, Y, Z are variables, π is a prefix, P is a process.

$$\pi ::= \mathsf{in}\ X \mid \mathsf{out}\ M \mid [M = M'] \mid \mathsf{let}\ X = M \mid \mathsf{new}\ X \mid \mathsf{repl}$$
$$P ::= \mathsf{nil} \mid \pi\ .\ P \mid (P|P)$$

Intuitively, nil is a process that performs no actions; $\pi.P$ executes the prefix π and then behaves as P; $P_1|P_2$ runs concurrently the processes P_1 and P_2. Prefixes perform the following actions: $\mathsf{in}\ X$ reads a term from the network and binds X to it; $\mathsf{out}\ M$ sends a term to the network; $[M = M']$ compares the terms M and M', stopping the process if they differ; $\mathsf{let}\ X = M$ locally binds X to the value of M; $\mathsf{new}\ X$ generates a

fresh value and binds X to it; repl spawns an unlimited number of copies of the running process, which will run independently.

As usual, a free variable is one that does not occur under an input, let, or new. A process P is *closed* iff P has no free variables.

3.1 Semantics

We now give a reduction semantics for the applied pi *closed* processes. First, we fix a constant base, and two unary function symbols next, val such that none of them occurs in the rewriting rules at hand. This always can be done, since we only use a finite set of rewriting rules. Hence, terms built only from base, next, val are unaffected by rewriting.

Then, we consider processes up to the abelian monoid laws:

$$\mathsf{nil} \mid P \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R)$$

We now define the reduction relation $P \to Q$. When a ground term M is disclosed by the process P, we label the corresponding reduction accordingly: $P \xrightarrow{M} Q$. Below, M and N are ground terms, and α denotes an optional output label.

Comm	$\mathsf{out}\ M\ .P \mid \mathsf{in}\ X\ .Q \to P \mid Q\{M/X\}$	
Out	$\mathsf{out}\ M\ .P \xrightarrow{M} P$	
Match	$[M = M].P \to P$	
Let	$\mathsf{let}\ X = M\ .P \to P\{M/X\}$	
New	$\mathsf{new}\ X\ .P \to P\{M/X\}$	where $M = \mathsf{genFresh}()$
Repl	$\mathsf{repl}.P \to P \mid \mathsf{repl}.P$	
Rew	$P\{M/X\} \to P\{N/X\}$	if $M \Rightarrow N$
Par	$P \mid R \xrightarrow{\alpha} Q \mid R$	if $P \xrightarrow{\alpha} Q$

Rule Comm makes two processes communicate, exchanging a term M. An output prefix may also be fired independently of an input, using rule Out: in this case M is disclosed, so we add it as a label of the transition. Rule Match allows a process to proceed only if the two (ground) terms are identical.

In rule New we generate a fresh value for X. This is done by the call to genFresh(), that chooses a fresh term value (not generated beforehand) in the sequence

$$\mathsf{val}(\mathsf{base}), \mathsf{val}(\mathsf{next}(\mathsf{base})), \mathsf{val}(\mathsf{next}(\mathsf{next}(\mathsf{base}))), \ldots$$

This peculiar sequence was chosen because its elements behave as distinct constants symbols. That is, it is not possible to compute a given

element from the others, through applications $x_1, \ldots, x_n \mapsto \mathsf{f}(x_1, \ldots, x_n)$ and rewritings. Unlike an infinite sequence of distinct constants symbols, the above uses only a finite number of symbols, making it amenable to static analysis.

Rule Repl spawns a new copy of the process P. Note that Repl does not consume the repl prefix, so it can be used to spawn an arbitrarily large number of processes. Rule Rew allows to rewrite any occurrence of a ground term inside a process, so that one effectively handle terms up to rewriting. Finally, rule Par allows a selected process P to act independently of its parallel processes, forming the rest of the system R.

3.2 A Specification for the CCA API

In Fig. 1, we provide a high-level specification for the (amended) CCA API. To this purpose, we use the function symbols $\mathsf{f1}, \ldots, \mathsf{f6}$. In Sect. 2.3, we gave the actual rewriting rules defining the semantics to all the API operations. We use the same rules here.

$$
\begin{array}{ll}
\mathit{System} = \mathit{Device} \mid \mathit{Init} \mid \mathit{DY} \mid \mathit{Test} & \\
\mathit{Device} = & \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.in}\ Z\ \mathsf{.out}\ \mathsf{f1}(X,Y,Z)\ \mathsf{.}() \\
& \mid \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.in}\ Z\ \mathsf{.out}\ \mathsf{f2}(X,Y,Z)\ \mathsf{.}() \\
& \mid \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.in}\ Z\ \mathsf{.out}\ \mathsf{f3}(X,Y,Z)\ \mathsf{.}() \\
& \mid \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.in}\ Z\ \mathsf{.out}\ \mathsf{f4}(X,Y,Z)\ \mathsf{.}() \\
& \mid \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.out}\ \mathsf{f5}(X,Y)\ \mathsf{.}() \\
& \mid \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.out}\ \mathsf{f6}(X,Y)\ \mathsf{.}() \\
\mathit{Init} = & \mathsf{out}\ 0\ \mathsf{.out\ data\ .out\ pin\ .out\ imp\ .out\ exp\ .out\ kp\ .out\ acc\ .out\ k3}\ . \\
& \mathsf{out\ enc(p, hash(pin, kek))}\ . \\
& \mathsf{out\ enc(xor(kek, k3), hash(hash(imp, kp), km))}\ . \\
& \mathsf{out\ enc(kek2, hash(hash(imp, kp), km))}\ . \\
& \mathsf{out\ enc(kek2, hash(hash(exp, kp), km))}\ . \\
& \mathsf{out\ enc(exp1, hash(exp, km))\ .}() \\
\mathit{DY} = & \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.out\ xor}(X,Y)\ \mathsf{.out\ hash}(X,Y)\ . \\
& \mathsf{out\ enc}(X,Y)\ \mathsf{.out\ dec}(X,Y)\ \mathsf{.out\ val}(X)\ \mathsf{.out\ next}(X)\ \mathsf{.}() \\
& \mid \mathsf{repl.new}\ \mathit{Nonce}\ \mathsf{.out}\ \mathit{Nonce}\ \mathsf{.}() \\
& \mid \mathsf{repl.in}\ X\ \mathsf{.repl.out}\ X\ \mathsf{.}() \\
\mathit{Test} = & \mathsf{repl.in}\ X\ \mathsf{.(\ let}\ Y = \mathsf{kek}\ \mathsf{.}[X = Y]\mathsf{.out\ forbidden}(X)\ \mathsf{.}() \\
& \qquad \mid \mathsf{let}\ Y = \mathsf{km}\ \mathsf{.}[X = Y]\mathsf{.out\ forbidden}(X)\ \mathsf{.}() \\
& \qquad \mid \mathsf{let}\ Y = \mathsf{p}\ \mathsf{.}[X = Y]\mathsf{.out\ forbidden}(X)\ \mathsf{.}() \\
& \qquad \mid \mathsf{let}\ Y = \mathsf{enc(acc, p)}\ \mathsf{.}[X = Y]\mathsf{.out\ forbidden}(X)\ \mathsf{.())}
\end{array}
$$

Fig. 1. CCA Specification in the applied pi calculus

Process *Device* is simple: it waits for inputs, performs an API operation, and then outputs the result. All the operations $\mathsf{f1}, \ldots, \mathsf{f6}$ are under a repl, so they are always available, and they can be invoked in any order.

Component $Init$ discloses all the initial knowledge of the adversary: the constant 0, all the tags, and all the encrypted keys (each one with its tag). The k3 key is also given to the adversary, as discussed in Sect. 2.1.

Component DY is the most powerful Dolev-Yao adversary [9]. Intuitively, this adversary tries any available option (building terms, invoking the API, etc.) in a *non-deterministic* fashion. The knowledge of this adversary is made of all the terms that processes output. More in detail, process DY is made of three parts. The first part (non-deterministically) builds terms using *all* the function symbols by applying them to *all* the known terms. Built terms are output, augmenting the knowledge of the adversary. Note that these terms can also be received by $Device$, so API operations are invoked with them. The second part of DY generates an unlimited number of fresh values, allowing the adversary to use them. The third part of DY replicates all the known terms indefinitely. This replication is needed, because otherwise inputs would *consume* known terms, enforcing an unwanted single-use policy on them. Further, DY can not send a term to a $Device$ so that it is received by a specific input, e.g. for f1 rather than for f2. Rather, communication is completely non-deterministic. This however does not make the adversary weaker, since we shall consider all the possible outcomes.

Finally, process $Test$ checks for secret leaks. The secret terms are those discussed in Sect. 2.1. Whenever a leak is detected, the witness term $\mathsf{forbidden}(X)$ is output, where X is the leaked secret. We also add a more general witness term fail, using the rewrite rule

$$\mathsf{forbidden}(X) \Rightarrow \mathsf{fail}$$

The secrecy goal for our specification can now be simply stated as

$$\neg\exists\alpha_1 \ldots \alpha_n(System \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \xrightarrow{\mathsf{fail}})$$

where the α_i denote optional outputs.

3.3 Static Analysis

We verified the secrecy goal with the Rewrite [14] tool. Rewrite performs a control flow analysis [12,2] on applied pi processes, using tree automata techniques [6,10] to finitely represent sets of terms. Terms are handled up to rewriting: the rewriting rules are given as an input to the tool, just as the process. In [16], we used Rewrite to check protocols involving complex cryptographic primitives, such as exponentiation. Here, we apply

Rewrite to the CCA API. We provided the tool with both process *System* and all the rewriting rules shown before. These include those for XOR, encryption, and the (amended) API operations.

The result of running Rewrite is an *over-approximation* for the set of ground terms output by process *System*. So, if there is no fail in the approximation, we can safely deduce that the secrecy result holds. It turns out this is indeed the case: inspecting the computed result revealed no fail term, so the property was established. Rewrite did not require any manual intervention during this computation. The time required for this was also acceptable: about three hours, on our desktop machine[1].

We do not provide here a detailed description of the actual approximation technique used in Rewrite, referring the reader to [16].

3.4 A Refinement of the CCA API Specification

We also experimented with a more detailed, low-level specification. Rather than describe the abstract API operations through rewriting rules, we can actually provide the code for an implementation of those. Basically, each operation can be implemented in a few steps: the keys given as input are decrypted, and then used to compute the result of the operation.

The abstract *Device* component can be changed as follows:

$$
\begin{aligned}
\mathit{Device} = \;& \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.in}\ Z\ \mathsf{.let}\ Z2 = \mathsf{dec}(Z, \mathsf{hash}(\mathsf{hash}(X, \mathsf{kp}), \mathsf{km}))\ .\\
& \mathsf{out}\ \mathsf{enc}(\mathsf{xor}(Z2, Y), \mathsf{hash}(\mathsf{hash}(X, \mathsf{kp}), \mathsf{km}))\ .()\\
|\ & \mathsf{repl.in}\ X\ \mathsf{.in}\ Y\ \mathsf{.in}\ Z\ \mathsf{.let}\ Z2 = \mathsf{dec}(Z, \mathsf{hash}(\mathsf{hash}(X, \mathsf{kp}), \mathsf{km}))\ .\\
& \mathsf{out}\ \mathsf{enc}(\mathsf{xor}(Z2, Y), \mathsf{hash}(X, \mathsf{km}))\ .()\\
|\ & \mathsf{repl.in}\ T\ \mathsf{.in}\ K\ \mathsf{.in}\ X\ \mathsf{.let}\ K2 = \mathsf{dec}(K, \mathsf{hash}(\mathsf{imp}, \mathsf{km}))\ .\\
& \mathsf{let}\ X2 = \mathsf{dec}(X, \mathsf{hash}(T, K2))\ \mathsf{.out}\ \mathsf{enc}(X2, \mathsf{hash}(T, K2))\ .()\\
|\ & \mathsf{repl.in}\ T\ \mathsf{.in}\ K\ \mathsf{.in}\ X\ \mathsf{.let}\ K2 = \mathsf{dec}(K, \mathsf{hash}(\mathsf{exp}, \mathsf{km}))\ .\\
& \mathsf{let}\ X2 = \mathsf{dec}(X, \mathsf{hash}(T, \mathsf{km}))\ \mathsf{.out}\ \mathsf{enc}(X2, \mathsf{hash}(T, K2))\ .()\\
|\ & \mathsf{repl.in}\ X\ \mathsf{.in}\ K\ \mathsf{.let}\ K2 = \mathsf{dec}(K, \mathsf{hash}(\mathsf{data}, \mathsf{km}))\ .\\
& \mathsf{out}\ \mathsf{enc}(X, K2)\ .()\\
|\ & \mathsf{repl.in}\ X\ \mathsf{.in}\ K\ \mathsf{.let}\ K2 = \mathsf{dec}(K, \mathsf{hash}(\mathsf{data}, \mathsf{km}))\ .\\
& \mathsf{let}\ X2 = \mathsf{dec}(X, K2)\ \mathsf{.out}\ X\ .()
\end{aligned}
$$

Now the specification does not use the $\mathsf{f1}, \ldots, \mathsf{f6}$ terms, since their semantics was embodied into process *Device*. Accordingly, we can discard the corresponding rewriting rules, and run the tool on the new specification. We ran Rewrite, and once more observed no fail term in its over-approximation. Therefore, we conclude this implementation preserves the security goals of Sect. 2.1. Computing the approximation took only a few minutes, showing a significant improvement w.r.t. the experiment of Sect.

[1] Reference: PowerPC G4 1.25GHz, 768MB RAM.

3.3, that took one hour. This is because we removed the complex rewriting rules of $\mathsf{f1}, \ldots, \mathsf{f6}$, leaving only the simpler ones.

For completeness, we also ran the tool on the flawed API. As expected, the over-approximation did include fail, detecting the possibility of a flaw.

4 Conclusions

We studied the verification problem of cryptographic APIs. As a real-world example, we considered the CCA API. This API was shown secure in [8], through a detailed manual proof. Here, we reported about the fully automatic verification of the same security statement. To the best of our knowledge, this is the first fully automatic proof of it. Moreover, our technique is general: we plan to apply it to similar APIs, including other versions of the CCA one.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of POPL'01*, pages 104–115, 2001.
2. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proc. of CSFW 2003*, pages 126–140, 2003.
3. M. Bond. Attacks on cryptoprocessor transaction sets. In *CHES*, pages 220–234, 2001.
4. IBM CCA basic services reference and guide, release 2.54.
5. IBM CCA devices. http://www-03.ibm.com/security/cryptocards/.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1rst 2002.
7. The Coq proof assistant. http://coq.inria.fr.
8. J. Courant and J. Monin. Defending the bank with a proof assistant. In *Proceedings of WITS 2006*, pages 87–98, 2006.
9. D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, 1983.
10. G. Feuillade, T. Genet, and V. V. T. Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33:341–383, 2004.
11. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
12. F. Nielson, H. Riis Nielson, and H. Seidl. Cryptographic analysis in cubic time. *ENTCS*, 62, 2002.
13. Otter: an automated deduction system. http://www-unix.mcs.anl.gov/AR/otter/.
14. The Rewrite protocol analysis tool. http://www.di.unipi.it/∼zunino/software.
15. P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. L. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, 2005.
16. R. Zunino and P. Degano. Handling $\mathsf{exp}, \times$ (and timestamps) in protocol analysis. In *Proceedings of FoSSaCS 2006*, volume 3921 of *LNCS*, pages 413–427, 2006.