Defending the Bank with a Proof Assistant

Judicaël Courant[1] and Jean-François Monin[1]

VERIMAG - Centre Équation, 2 avenue de Vignate, F-38610 Gières, France
{judicael.courant|jean-francois.monin}@imag.fr
http://www-verimag.imag.fr/~courant/

Abstract. We show how the proof-assistant Coq helped us formally verify security properties of an API. As far as we know, this is the first mathematical proof of security properties of an API. The API we verified is a fixed version of Bond's modelization of IBM's Common Cryptographic Architecture. We explain the methodology we followed, sketch our proof and explain the points the formal verification raised.

1 Introduction

1.1 Context

Application Program Interface (API) are everywhere in computer science: POSIX system calls and the Gtk toolkit are a few of them. They let the provider of a software component specify how this component should be accessed from the outside. Among them, *security API* provide security features such as authentication, electronic signature schemes, encryption schemes. They are often based on cryptographic primitives. The most well-known security API is the Secure Socket Layer (SSL) and is notably used to implement the HTTPS protocol (HTTP over SSL).

IBM's *Common Cryptographic Architecture* (CCA) is a security API found in every cash-machine (Automated Teller Machine, ATM) over the world. An ATM generally consists of a standard IBM-compatible PC, into which a specific PCI extension card has been plugged. This extension card contains an IBM 4758 coprocessor, implementing the CCA API. The IBM 4758 features a small internal memory, and is designed to be tamper-resistant to physical attacks.

Mike Bond [Bon01,Bon04] has shown that mistakes in the conception of a security API open security holes. In some cases, combining valid calls in unforeseen ways is enough for a user to get informations he was supposed to ignore. Regarding the IBM 4758, one may obtain secret values that were supposed to be confined inside the cryptographic module, using valid calls only.

Analyzing security API can be roughly compared to analyzing cryptographic protocols: Bond's model is close to the Dolev-Yao model in which cryptographic primitives (encryption, hashing) are supposed perfect and the adversary can listen to data transiting on the communication network, as well as modify them, erase them, and do some offline computations (encryption and decryption using known keys). The main difference here is that the adversary is not bound to follow the execution of a given protocol — he may, at will, call any API function with any messages he has built or captured — which raises the number of potential attacks. Moreover, whereas the number of messages exchanged during a single run of a protocol is quite low, the adversary may chain an arbitrary number of API calls. Security API analysis therefore seems more challenging than cryptographic protocol analysis.

Recently, Bond *et al.* proposed to use an automated theorem prover to find API attacks. On a simplified model of the CCA API, the automated prover Otter [Ott] could find all known attacks against CCA [YAB$^+$05]. Unfortunately, whereas the existence of an attack is shown by exhibiting a finite combination of API calls and offline computations leading to leakage of a secret, the absence of attack can only be shown through an infinitary argument proving that no combination of API calls and offline computation will ever lead to a leakage.

Therefore, the previous approach does not apply if we are interested in positive results, i.e., if we want to study the absence of attacks. This drove us to abandon the idea of using a fully automatic first-order prover, and to use instead a proof assistant with induction capabilities, namely Coq [HKP05,The05]. As a target of our validation, we used the model described in [YAB$^+$05], with slight modifications in order to prevent already known attacks.

1.2 Our Contributions

Our contributions are the following:

– We mathematically prove security properties of an API. Giving such a proof is generally considered significantly more difficult than exhibiting an attack. Thus, in order to design secure API, the published works we are aware of only propose methodological principles. As far as we know, our proof is the first mathematical proof of security properties of an API.
– Our methodology should be reusable with other security API.
– We checked our proof in a proof assistant. Hence we refute Bond's conjecture that the complexity of a formal proof of a security API is such that the treatment of security API would require a different approach from cryptographic protocols.
– We did not find our security proof *before* we checked it within Coq. Indeed, modeling the security API within the proof assistant and interacting with the proof assistant helped us find the proof, *while* modeling the CCA API: we gradually established conjectures; some of them could be proved without further ado; but during the proof process of the others, Coq helped us to point out problematic cases, suggesting intermediary lemmas or questioning the validity of our conjecture. As soon as we were modifying our model, we could see which ones among already written proofs would still hold without any modification, which ones would need some rewriting and we could focus on the ones which would be definitely broken. In that sense, our work suggests that using a proof assistant to *find* a proof could be a good idea.

1.3 Plan

The remainder of this paper is organized as follows: section 2 presents the CCA API, a known attack against it and introduces a fix; section 3 describes our formalization in Coq; finally, we conclude in section 4.

2 IBM 4758 and the CCA API

2.1 Introduction

In the following, if x is a datum and k a key, $\{x\}_k$ denotes x encrypted by k (the symmetric cryptographic algorithm used by the IBM 4758 is triple DES but that is not relevant here). In order to specify a cryptographic operation (whether provided by the API or computable offline), we use the standard notation $t_1, \ldots, t_n \to t$ to mean t can be obtained as soon as one knows $t_1, \ldots t_n$. Since the encryption algorithm is publicly known, the ciphering and deciphering operations can easily be done offline (without access to any IBM 4758) and are denoted by $x, k \to \{x\}_k$ and $\{x\}_k, k \to x$. We note $x \oplus y$ the bitwise exclusive or of x and y. Our terminology and the identifier names we use stick to the ones of [YAB+05].

2.2 IBM 4758

The IBM 4758 coprocessor was designed for the verification of confidential PIN code of credit cards at ATM, while preserving confidential data from potential eavesdroppers. It is supposed to resist external attacks (including physical ones), and even to attacks from insiders of the bank as well as to attacks from the programmers of the ATM.

As the CCA API is quite complex [IBM], we only deal with the simplified model of this API proposed in [YAB+05]. This API is already complex enough to be prone to actual attacks.

The CCA API features a check of PIN codes of credit cards. The PIN is $\{ACC\}_P$, where ACC is the account number of the owner of the card, and P is a secret key, known only by banks and identical over the whole ATM network. Hence it is of utmost importance that P is kept secret. The CCA API provides a call for checking a number is a correct PIN of some card, but it does not provide any call for directly computing this PIN. [1]

The IBM 4758 has little memory: enough for keeping a master key KM (randomly generated at the time it is set in operation) and to do some cryptographic operations.

Thus, P is not stored in the coprocessor, but in the hosting PC instead. Since this PC is insecure, P is stored in encrypted form $\{P\}_{KM}$. More precisely, encrypted values stored in the PC are labelled with a typing information: actually,

[1] Finding the code is still possible but needs a lot of guesses, hence computing time; this attacks is then risky, while the expected benefit is modest.

one does not store $\{P\}_{KM}$ but $\{P\}_{PIN \oplus KM}$, where *PIN* is a publicly known integer constant denoting the type of keys used for computing the PIN from an account number. In the following, we often refer to a value $\{P\}_{PIN \oplus KM}$ as "the datum P encrypted by *KM* under the type *PIN*".

Secrets (such as P) must be transmitted on a secure communication channel between the bank and the ATM. Such a channel is established by using a communication key *KEK* (*KEK* stands for "Key Encrypting Key"), known only by the ATM and the bank server. *KEK* is randomly chosen by the bank, and separately sent to the ATM as three secret shares K_1, K_2 and K_3, such that $KEK = K_1 \oplus K_2 \oplus K_3$. Therefore, an attack on *KEK* should involve the interception of each of the three shares.

As for P, *KEK* is not stored on the coprocessor but on the hosting PC, encrypted by *KM* under the type *IMP* of importation keys. Similarly, the intermediate values K_1 and $K_1 \oplus K_2$ are stored encrypted. The coprocessor indeed features the following operations to manage key parts:

$$x, y, \{z\}_{x \oplus KP \oplus KM} \rightarrow \{z \oplus y\}_{x \oplus KP \oplus KM} \tag{1}$$

$$x, y, \{z\}_{x \oplus KP \oplus KM} \rightarrow \{z \oplus y\}_{x \oplus KM} \tag{2}$$

Both these operations use encrypted data under the type $x \oplus KP$, which is the type of parts of keys of type x (in type theory parlance, the publicly known integer *KP* denotes a *parameterized* type). The first of them lets one add a value y to a key part z in order to get $y \oplus z$ under the type "key part of keys of type x". The second one, with the same inputs, yields instead $y \oplus z$ under the type x.

Given a value x encrypted by k under the type t and the encryption of k by *KM* in the type *IMP* (which proves k is an importation key), a CCA operation returns x encrypted by *KM* under t:

$$t, \{k\}_{IMP \oplus KM}, \{x\}_{t \oplus k} \rightarrow \{x\}_{t \oplus KM} \tag{3}$$

Thus, P is transmitted to the ATM as $\{P\}_{PIN \oplus KEK}$, which lets the PC get the value $\{P\}_{PIN \oplus KM}$ through one API call.

Symmetrically, the coprocessor is given a secret key EXP_1, typed in the type of exportation keys *EXP*. A CCA call converts an x encrypted by *KM* under the type t into an x encrypted by exportation key under the same type:

$$t, \{k\}_{EXP \oplus KM}, \{x\}_{t \oplus KM} \rightarrow \{x\}_{t \oplus k} \tag{4}$$

This latter call is irrelevant to attacks described in [YAB+05]. They mostly mentioned it in order to demonstrate that they are able to automatically find an attack even if more rules than needed are added. We show in the following, section 3.2, that this rule obliged us to give a subtler definition of sensitive terms than we initially thought. Finally, [YAB+05] also add a key KEK_2 typed both under importation keys and exportation keys. Once the previous difficulty with exportation key was solved, adding this rule to our proof development did not modify it.

The IBM 4758 also lets the PC to secure its own applicative data: in order to encrypt some data x, an application can generate an encryption key k; the API call $k \to \{k\}_{DATA \oplus KM}$ returns this key encrypted by KM under the type $DATA$ of user applicative data. CCA provides the following calls:

$$x, \{k\}_{DATA \oplus KM} \to \{x\}_k \tag{5}$$

and

$$\{x\}_k, \{k\}_{DATA \oplus KM} \to x \tag{6}$$

2.3 Attacks against the CCA API

Several attacks were found against CCA [Bon04]. The simplest one makes the cryptoprocessor imports KEK under the type $DATA$ instead of IMP, thus providing a way to decrypt all secret information received by the coprocessor to a malicious application. This attack is carried out in the following way: as soon as the last part K_3 of KEK is given, the attacker intercepts K_3 and instead of just applying operation (2) to K_3, it also applies it to $K_3 \oplus PIN \oplus DATA$, thus getting $\{KEK \oplus PIN \oplus DATA\}_{IMP \oplus KM}$ in addition to $\{KEK\}_{IMP \oplus KM}$. Then, as soon as $\{P\}_{PIN \oplus KEK}$ is received, rule (3) can be used in the standard way to import $x = P$ encrypted by $k = KEK$ under the type $t = PIN$, but it can also be used to import $x = P$ encrypted by $k = KEK \oplus PIN \oplus DATA$ under the type $t = DATA$ since $PIN \oplus KEK = DATA \oplus (KEK \oplus PIN \oplus DATA)$.

Thus, the attacker gets $\{P\}_{DATA \oplus KM}$, which allows him to compute $\{ACC\}_P$ for any account ACC by rule (5).

Therefore, by just intercepting the third key part (K_3), the attacker could get some sensitive pieces of information, although the API was supposed to resist the interception of any two of the three key parts K_1, K_2 and K_3.

Bond suggests that a solution would be to use a one-way function instead of the exclusive or for type labels (see [Bon01], section 5.1, page 230) but as far as we know, no proof of security of the thus corrected API was given. We address this challenge: The only difference between our API and the one presented in [YAB$^+$05] is the change Bond suggested. We replace occurences of encryption keys $T \oplus K$ by $H(T, K)$, where H is a one-way function. In order to avoid adding a pairing operation to our API and since H is only applied to pairs, we consider H to be two-arguments function.

3 Formalization

The general idea of the formalization is the following. We inductively define the predicate `known` for known values, that is, values which are initially known or which can be obtained from known values by application of computable operations such as $\oplus$, H, ciphering or operations provided by the cryptographic coprocessor 4758.

Our modeling deals with the case where K_1 and K_2 were not intercepted but where K_3 was. We thus suppose that we have `known`(K_3). Moreover, instead of introducing K_1, K_2 and K_3, and handling the expression $K_1 \oplus K_2 \oplus K_3$, we adopted the trick of [YAB+05] consisting in noting *KEK* the sum $K_1 \oplus K_2 \oplus K_3$. One notices that K_1 and K_2 occur in the modeling only under the form $K_1 \oplus K_2$ or $K_1 \oplus K_2 \oplus K_3$, that can respectively be written $KEK \oplus K_3$ and *KEK*, which makes it possible to remove all references to K_1 and K_2, and therefore to remove them from the modeling.

Similarly to the model of Dolev-Yao for cryptographic protocols, we suppose moreover that these known values are the only terms that the user of the IP can get, honest or not. We thus implicitly exclude from our study all attacks which would be based on attacks of cryptographic primitives themselves, such as attacks consisting in finding the key used in the ciphering of a set of known ciphered messages (when the corresponding clear messages can themselves be disclosed), in reversing the hash function on certain values or in finding collisions on the hash function (i.e. two distinct pairs (x_1, y_1) and (x_2, y_2) satisfying $H(x_1, y_1) = H(x_2, y_2)$). In other words, we suppose that these attacks are too difficult (need too much computing power) for a dishonest user.

The objective is then to show that, among known values, one finds neither *KEK*, neither *KM*, nor P, nor $\{ACC\}_P$. A simple examination of the various cases for `known` is not enough, because certain sub-cases in assumption contain terms larger than those of the conclusion (for example `K_decrypt` hereafter). One proceeds by introducing an over-approximation of `known` with good structural properties, namely `unc`, for *unclassified terms*, under the military meaning of the term: terms we estimate the public knowledge of would not jeopardize security. We will see that the precise definition of this predicate is more delicate than it may appear at first sight. The actual use of the proof assistant highlighted the insufficiencies of our first attempts and helped us to design a correct version of `unc`, letting us show that, in our model, any term computable by the user is indeed unclassified.

One of the main technical difficulties comes from the management of the operator $\oplus$: we have to reason on equivalence classes of terms modulo associativity, commutativity and involutivity of this operator. Indeed these algebraic properties play an essential role in this API, both negative and positive: in the case study considered, they are exploited to transmit *KEK* in three parts, but they also leave room to undesirable scenarios. Technically, we use canonical representatives for each equivalence class. We proceed into two phases:

- the core of the proof is carried out in the context of an abstract normalization function called `norm`;
- we separately build such a function by using a well-founded ordering on terms.

In what follows, only a subset of the elements related to the 4758 processor is mentioned—the elements already mentioned in the previous sections. The complete development includes all the elements considered in [Bon04].

3.1 Main statements

Syntax In Coq, a constructor—more generally: a function—with two arguments of type A and B and returning a C is declared with the type $A \to B \to C$. The right-associative operator $\to$ can actually be used in two ways: either for constructing functional types, or for the logical implication. As a logical connective, it is more primitive than $\wedge$. Hence a Horn clause, usually written $(P_1 \wedge P_2 \wedge \ldots P_n) \Rightarrow Q$, would be written here $P_1 \to P_2 \to \ldots P_n \to Q$.

Definitions We first define the enumerated types `secret_const` of secret constants and `public_const` of public constants. The former contains *KM*, *KEK*, *P*, KEK_2 and EXP_1. The latter contains in particular *DATA*, *PIN*, *IMP*, K_3, *ACC* and *KP*. The type of terms is then defined inductively using 5 constructors:

```
Inductive term : Set :=
  | Zero : term
  | PC : public_const → term
  | SC : secret_const → term
  | E : term → term → term        (* ciphering *)
  | Xor : term → term → term      (* bitwise exclusive or *)
  | Hash : term → term → term.    (* one-way function *)
```

In the following, we use the infix notations $\{x\}_y$ for (`E` x y), $x \oplus y$ for (`Xor` x y) and x `+*` y for (`Hash` x y). Computable terms are defined as follows.

```
Inductive init_known : term → Prop :=
  | K_Zero : init_known Zero
  | K_PC : ∀c : public_const, init_known c
  | P_KEK_PIN : init_known {P}(PIN +* KEK)
  | K3_KEK : init_known {norm (KEK ⊕ K3)}(IMP +* KP +* KM)
  | KEK2_IMP : init_known {KEK2}(IMP +* KP +* KM)
  | KEK2_EXP : init_known {KEK2}(EXP +* KP +* KM)
  | EXP1_EXP : init_known {EXP1}(EXP +* KM).

Inductive known : term → Prop :=
  | K_init : ∀x:term, init_known x → known x
  | K_Xor : ∀xy:term, known x → known y → known (x ⊕ y)
  | K_Hash : ∀xy:term, known x → known y → known (x +* y)
  | K_E : ∀xk:term, known x → known k → known {x}k
  | K_decrypt : ∀xk:term, known {x}k → known k → known x
  | K_key_import : ∀xyz, known x → known {z}(IMP +* KM) →
                     known {y}(x +* z) → known {y}(x +* KM)
  | K_key_part_import_completing :
      ∀xyz, known x → known y →
      known {z}(x +* KP +* KM) → known {z ⊕ y}(x +* KM)
  | K_key_part_import_notcompleting :
      ∀xyz, known x → known y → known {z}(x +* KP +* KM)
```

```
    → known {z ⊕ y}(x ** KP ** KM)
  | K_encrypt_using_data_key :
    ∀xy, known x → known {y}(DATA ** KM) → known {x}y
  | K_key_export :
    ∀xyz, known x → known {y}(x ** KM) →
    known {z}(EXP ** KM) → known {y}(x ** z)
  | K_decrypt_using_data_key :
    ∀xy, known {x}y → known {y}(DATA ** KM) → known x
  | K_Eq : ∀xy:term, known x → Eq x y → known y.
```

The rule K_init expresses the initial knowledge of an attacker. Rules K_Xor, K_Hash, K_E and K_decrypt describe the possible offline computations (the ciphering algorithm is supposed to be known). Other rules but the last describe API calls. The last one allows us to reason modulo equality of terms (the congruence generated by the laws governing the operator $\oplus$). The following table clarifies the correspondence between the constructors and the operations presented above. As announced, the value v encrypted with key k under the type t is $\{v\}_{(t{*}{*}k)}$ in our model, instead of $\{v\}_{t\oplus k}$ in the original CCA API.

Constructor	Corresponding operation
K_key_import	(3)
K_key_export	(4)
K_key_part_import_completing	(2)
K_key_part_import_notcompleting	(1)
K_encrypt_using_data_key	(5)
K_decrypt_using_data_key	(6)

Finally, we define forbidden terms.

```
Inductive forbidden : term → Prop :=
  | F_KEK : forbidden KEK
  | F_KM : forbidden KM
  | F_P : forbidden P
  | F_E_ACC_P : forbidden {ACC}P.
```

Security Theorem We prove that no forbidden term is computable. Formally:

```
Theorem security_results : ∀x, forbidden x → ¬ known x.
```

3.2 Proof

Context: axiomatic normalization We assume here the existence of a normalization function norm on terms. Two terms are equivalent if their normal forms are equal, which means they are equal modulo the algebraic properties of $\oplus$ (commutativity, associativity, involutivity). The predicate is_nf tells us whether a term is in normal form.

Unclassified terms An unclassified term is defined as a term whose norm satisfies the predicate `unc_nf` defined by the following inductive definition. Two auxiliary predicates are needed:

– The relation `sub_xor`(t_1, t_2), which indicates that a term t_2 has the shape $u_1 \oplus \dots u_n \oplus t_1 \oplus v_1 \oplus \dots \oplus v_m$ where the head of t_1 is not itself a $\oplus$;
– As noted in paragraph 2.2, the ciphered data of type *DATA* does not involve the same confidentiality problems as the other data ciphered by the coprocessor. Thus it will be necessary to distinguish in some places the type *DATA* from the other types. More precisely, it came into light, while looking for the proof, that this distinction between the types which contain *DATA* (i.e. *DATA* and types of the form " piece of ... piece of *DATA*") and the others should be generalized. This problem is detailed in paragraph *Unclassifying a simply encrypted term* below. Hence we define the predicate `contains_data`:

```
Inductive contains_data : term → Prop :=
  | CD_Data : contains_data DATA
  | CD_Hash : ∀x, contains_data x → contains_data (x +* KP).
```

```
Inductive unc_nf : term → Prop :=
  | U_Zero : unc_nf Zero
  | U_PC   : ∀c, unc_nf (PC c)
  | U_E    : ∀xy, unc_nf x → unc_nf y → unc_nf {x}_y
  | U_Data : ∀xy z, unc_nf x → is_nf y → contains_data y →
               is_nf z → unc_nf {x}_(y +* z)
  | U_Xor  : ∀xy, unc_nf x → unc_nf y → is_nf (x ⊕ y) →
               unc_nf (x ⊕ y)
  | U_Hash : ∀xy, unc_nf x → unc_nf y → unc_nf (x +* y)
  | U_crpt : ∀xyc1c2t, is_nf x → is_nf y → is_nf t →
             sub_xor (SC c1) x → sub_xor (SC c2) y →
             ¬contains_data t → unc_nf {x}_(t +* y).
```

```
Definition unc x := unc_nf (norm x).
```

Most clauses for `unc_nf` are quite natural. However, the clause about simple encryption (`U_E`) requires further explanation. We elaborate on this in paragraph *Unclassifying a simply encrypted term* below.

Main lemma We prove that computable terms are always unclassified. Formally, we have:

```
Theorem main : ∀x, known x → unc x.
```

The security theorem is obtained as a corollary: none of the clauses for `unc_nf` may yield one of the four prohibited terms (this is immediate for *KEK*, *KM*, *P*; as for $\{ACC\}_P$, only `U_E` could apply but this would require P to be unclassified which is not).

The proof of this main lemma is carried out by induction on the definition of `known`. That amounts to showing that each clause of `known` preserves the predicate `unc`. For example, in the case of clause `K_E`, one has to show the following lemma, which simply reduces to `U_E` after expansion of definitions and normalization:

`Lemma unc_e :` $\forall xy$, `unc` $x \rightarrow$ `unc` $y \rightarrow$ `unc` $\{x\}_y$.

Unclassifying a simply encrypted term As we were quite new to the field of cryptographic protocols, we first imagined that it is enough that x is unclassified for $\{x\}_y$ to become unclassified, and described `U_E` in that way. However, while trying our security properties with Coq, it soon turned out that such a possibility could make it possible to extract prohibited information. Indeed, the export rule (rule (4)) enables us to choose for exportation key any term x encrypted with EXP `+*` KM. If x is an exportation key, any given secret z of type t can be exported in the form $\{z\}_{t\texttt{+*}x}$. Then, if x is known, z can be known as well. A known key should thus never be allowed to be used as an exportation key, because that would make it possible to know all the secret keys contained in the coprocessor. (Actually, some attacks of this kind have been discovered a decade ago [LR92].) If x is known, one must then take special care not to unclassify $\{x\}_{\mathit{EXP}\texttt{+*}\mathit{KM}}$.

This leads us to the following remarks and rules:

- If x is unclassified, it should be checked that the knowledge of $\{x\}_y$ leaves no room to a bad use before unclassifying the latter.
- $\{x\}_y$ can be unclassified as soon as x and y are themselves unclassified, because the user can already compute $\{x\}_y$ from x and y by her or his own means (rule `U_E`).
- $\{x\}_{\mathit{DATA}\texttt{+*}\mathit{KM}}$ can be unclassified if x is unclassified because no significant use from the point of view of the safety of the secrecies of the coprocessor can be made from ciphered data in the type *DATA*: the type *DATA* corresponds only to data or keys at the application level (rule `U_Data`).
- Conversely, unclassifying a piece of data $\{x\}_{\mathit{DATA}\texttt{+*}\mathit{KM}}$ would undoubtedly be foolish if x is supposed to remain secret.
- Nevertheless, we want to unclassify pieces of data which have to leave the coprocessor. We thus accept to unclassify terms $\{x\}_{t\texttt{+*}y}$ typed in t insofar as x is secret (which can be denoted by the presence of a secret constant in the term), as y is secret (idem) and as t is not *DATA* (rule `U_crpt`).
- More precisely, the rule (2) for importing pieces of keys makes it possible to cast a data of type t `+*` KP to type t. Then if the user has access to the term $\{x\}_{t\texttt{+*}\mathit{KP}\texttt{+*}y}$, she or he will be able to obtain $\{x\}_{t\texttt{+*}\mathit{KP}}$. (Again, this problem was pointed out to us by the interaction with Coq.) Thus in the previous item, it is necessary to prohibit t to be equal not only to *DATA*, but also to *DATA* `+*` *KP*, as well as *DATA* `+*` *KP* `+*` *KP* because one could then get *DATA* `+*` *KP* then *DATA*. The role of predicate `contains_data` is precisely to take care of this. We also used this predicate to generalize the

declassification rule U_Data, although it is probably not necessary, in the absence of a rule allowing us to recover pieces of keys from a whole key.

3.3 Normalizing terms

Each term can be considered as an alternation of two kinds of layers: layers consisting of sequences of $\oplus$ and layers built with constructors different from $\oplus$. Normalizing a term consists in flattening it, (using the associativity of $\oplus$), then sorting the list thus obtained (using commutativity). The sorting is directed by a well-founded order involving

– an ad hoc concept of weight
– a lexicographic ordering for terms having the same weight. One shows that any set of terms whose weight is bounded by a given value is finite, and that any irreflexive and transitive relation on a finite set is well-founded.

In the course of the process, two identical terms will become adjacent and will be removed by involutivity of $\oplus$.

At the submission time of this paper, the gap between this part of the development and the axiomatic presentation of the standardization function is not filled yet. The checking of our development is then not complete, an error in the axioms of the normalization function may potentially challenge the validity of our results. However, we feel quite confident for the following reason:

– On the one hand, we reduced the complexity of the initial problem to that of the normalization of a term rewriting process. The probability that an error slipped into the axioms of this normalization is quite low, since they basically describe terms endowed with an associative-commutative operator ($\oplus$), a neutral element with respect to this operator and symbols for uninterpreted functions (hashing and coding). What makes this formalization difficult is more the lack of tools to manage such a rewriting in Coq than the originality of this rewriting relation.
– On the other hand, the advantage of an interactive tool such as Coq is that the user can see how and where the declared axioms are used. None of these uses seemed suspect to us. In other words, even if the axiomatization of the normalization function was flawed, we estimate that it would be relatively easy to correct it while preserving the essence of our proof. Conversely, in a completely automatic proof tool, an inconsistency in the axioms of a development may drive the tool to derive the absurd proposition, and then to use this absurd proposition to prove all the propositions stated by the user, whereas she or he does not realize the reason for which the tool validates her or his propositions.

4 Conclusion and Future Work

We could formalize and prove security properties of the CCA API in the proof-assistant Coq. Our development is available at `http://www-verimag.imag.fr/~courant/06/wits/`.

This development opens the way to several future works:

– Extending the results to a more realistic modeling of the CCA API;
– Modeling other security API ([Bon04] presents several of them, including the *Visa Security Module*);
– Developing a toolbox for such security proofs. Indeed, not only our methodology can be reused but we might also be able to share some Coq libraries between several developments: term comparison through polynomial interpretation and lexicographic ordering as we did in our development, normalization through rewriting, ...

Moreover, our development suggests that techniques already used in the context of security protocols [Mea92,Pau98] could be relevant also in the context of security API. Therefore, we would like to study links between API and protocol verification further.

Current techniques for securing API rely on heuristic design principles and management of failures such as attack detection, alert mechanisms, financial risk management, (possibly dishonest) fraud denial in order to prevent the whole banking structure from breaking down. We believe our approach provides an interesting, complementary way to secure API at design time.

References

[Bon01] Mike Bond. Attacks on cryptoprocessor transaction sets. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 2001.

[Bon04] Mike Bond. *Understanding Security APIs.* PhD thesis, University of Cambridge Computer Laboratory, June 2004.

[HKP05] Gérard Huet, Gilles Kahn, and Christine Paulin. *The Coq Proof Assistant Tutorial Version 8.0.* Logical Project, January 2005.

[IBM] IBM. *CCA Basic Services Reference and Guide*, release 2.54 edition.

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

[Mea92] Catherine Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.

[Ott] Otter: An automated deduction system. Web site : http://www-unix.mcs.anl.gov/AR/otter/.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[The05] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0.* Logical Project, January 2005.

[YAB+05] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, Computer Laboratory, August 2005.