Certification of an encryption algorithm with Coq proof assistant

Dhia Znaidi
Student
IMT Atlantique
Nantes, France
dhia.znaidi@imt-atlantique.net

Hervé Grall
Computer Science Department
IMT Atlantique
Nantes, France
herve.grall@imt-atlantique.fr

***Abstract*- With the rapid developement in the information field, data encryption and secruring information have drawn noteworthy regard. In fact, every information system has to be protected from cyberattacks and to be designed to yield a better security level. This solution can be found within Coq, an interactive proof assistant capable of verifying the correction of programs and ensuring that they meet the logical background behind their implementation. In this paper, we propose the implementation and the certification of an encyprtion algorithm with Coq Proof Assistant. The results and tests show that the designed algorithm is correct and satisfies the criteria of being well concieved and well coded.**

Keywords: Coq Proof Assistant · encryption · certification · One-time pad

1 Introduction

Data encryption consists of converting data to hide information in a coded language so that only people with a secret key or password can be able to access it. In other terms, it main aim is to prevent unauthorized access to information. Its purpose is to ensure a confidential and correct transmission of information between users. Data encryption needs to satifsy some other criteria for instance the authenticity of information, data privacy, protection against alteration and a non-repudiated transfer of information[1]. However, the transmission of messages in an open-access environment can lead to the most critical and challenging security issues. As a result, the design of an effective data encryption algortihm remains an important challenge.

Divers data encryption algorithms have been proposed and they can be divided into two categories ; symmetric-key and non symmetric-key cryptography. In this work, we chose to certify an algorithm that belongs to the first category and called the one-time pad, but the major question that arises is how to ensure its correctness and if this algorithm is well coded. To handle this matter, we aim in this work to code an algorithm, with Coq proof assistant, that will be able to guarantee that the coded encryption satisfies the requirements. The rest of this paper is organized as follows. In section 2, Coq proof assistant evironment is explored. A summary and characteristics of the one-time pad encryption algorithm is given in section 3. Section 4 the mathematical tools that were used in this work .Section 5 states the results. Finally, section 6 and concludes the paper.

2 Coq Proof Assistant

The Coq proof assistant is a Type Theory-based interactive proof assistant developed at INRIA. It has a strong user base both in the field of software and hardware verification and in the field of formalized mathematics. The whole logic of Coq is based mainly on the Curry-Howard correspondence [2] which illustrates the direct relationship between computer programs and mathematical proofs.In this correspondance, formulas are seen as types and proofs as programs. But, in these programmes, programmers can separate the purely logical parts and informational parts. Coq is not a tool dedicated to software verification but a general purpose environment for developing mathematical proofs. However, it is based on a powerful language including basic functional programming and highlevel specifications. As such it offers modern ways to literally program proofs in a structured way with advanced data-types, proofs by computation, and general purpose libraries of definitions and lemmas. Coq is well suited for software verification of programs involving advanced specifications. Several approaches exist to program certification. For example, it is possible to prove *a posteriori* properties on existing programs, possibly annotated. This can be done by analysing the source code (the form originally written by the programmer), the machine code (the version understood by the computer) or any intermediate version that compilation (transformation of source

code into machine code) may produce. Another approach is to express the desired behaviour in a more abstract formalism than the usual programming languages, and then automatically derive programs that satisfy the desired properties *a priori*. The programs generated in this way are in usual programming languages and can even be used as source code for some of them, but the reasoning is done in the initial formalism and/or on the generation process itself. It is rather in this framework that this work is situated.

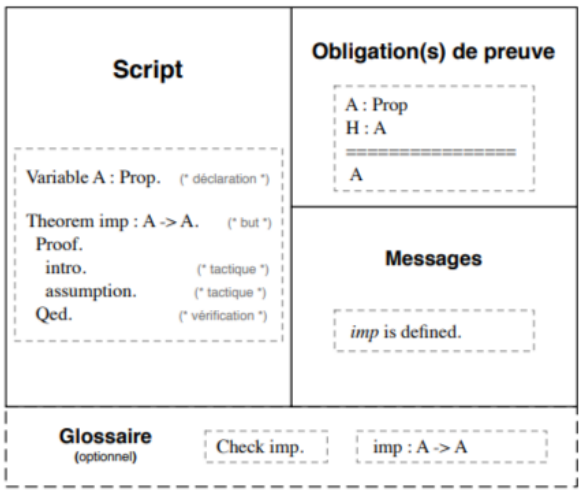


Figure 1: CoqIDE Interface [6]

3 One-time pad encryption algorithm

In the present article, we chose to work on symmetric-key encrpytion algorithm, more specifically the one-time pad encryption algorithm.
In the following we denote addition modulo TA as $\bigoplus$ and subtraction modulo TA as $\bigominus$, where TA=255 is the size of Ascii codes . We denote the function composition $f \circ g$ as : `(g;;f)`.
Essentially, the one-time pad (OTP) alogrithm conducts a modular addition of the clear message M with a key K to obtain the encrypted C:

$$C = M \bigoplus K$$

The deciphering process conducts a modular addition of the encrypted message C with a decryption key K' to obtain the clear message M :

$$M = C \bigoplus K'$$

where K' is generated from the encryption key K.

The key, in this process, must satisfy very strict conditions that make the encryption algorithm theoretically unbreakable: it must be at least as long as the plaintext, random, never be reused in whole or in part and kept completely secret by the communicating parties. Given all these information, the general scheme of the encryption algorithm can be resumed as :

$$msg\_clair \longrightarrow \left.\begin{array}{c} msg\_clair \\ + \\ cle\_chiffrement \end{array}\right\} \longrightarrow msg\_chiffre$$

$$msg\_chiffre \longrightarrow \left.\begin{array}{c} msg\_chiffre \\ + \\ cle\_dechiffrement \\ \uparrow \\ {\scriptstyle cle\_chiffrement} \end{array}\right\} \longrightarrow msg\_clair$$

4 Implementation and mathematical tools

Given that designing and ensuring the reliability of a encryption algorithm remains a challenging process, many questions would arise mainly about ensuring its correctness, wether it is well concieved and well done or not. In order to alleviate this problem, we choose to work, in the present article, with Coq proof assistant, whose main characteristics are detailed above, and we divided our program in two parts : the frontend and the backend.
Using Coq, we have two possibilities of transformations : lists or vectors. Vectors, in the Coq library, are lists with known length, wether lists don't give any direct information about their length[3,4]. Thus, there were several logical problems with using lists because it is always necessary to ensure that the two lists have the same length through the proposition `P : length l = length l1`, but the main flaw is we cannot define a cartesian product of lists. This flaw appears in this lemma provided in the Coq library of lists :

```
Lemma combine_length :  forall (l:list
A)(l':list B), length (combine l l') =
      min (length l) (length l').
```

For these reasons, and in order to ensure the logical correctness of the program, we decide to work with vectors.

4.1 The backend

In this part of the program, we start by defining the `Modulo TA` library in Coq.
`Nat` is recursively defined as :

```
Inductive nat :  Set := O : nat | S :
```

`nat -> nat`
It can be defined as a group with the + operation. A quotient group or factor group is a mathematical group Instead of working with numbers, we choose to define the `Modulo TA` which is a quotient group of the `Nat` obtained by aggregating its similar elements using the equivalence relation $\mathcal{R}$ defined as :

$$\forall a, b \; a\mathcal{R}b \iff a \equiv b \quad (modTA)$$

We use the function below to operate the transformation :
```
Fixpoint modulo_morphismeNat(d : nat)(n
: nat) : Modulo d.
```
We define the opposite transformation as well :
```
Definition modulo_valeurd : nat(m :
Modulo d) : nat.
```
The next step is to verify that when composing these two functions, the result will be the remainder of the Euclidian division of a given n by the successor of d:
```
Lemma modulo_morphismeNat_valeur :
forall d, forall n, modulo_valeur
(modulo_morphismeNat d n) = reste n (S
d).
```
This lemma proves that $\mathcal{R}$ groups all the numbers by their respective remainders, hence dealing with the set $\{\overline{0}, \overline{1}, \ldots, \overline{TA}\}$. The group structure of `Nat` is preserved in `Modulo TA`. It has the structure of a group possessing an associative, commutative and additive law, a neutral element `modulo_zero` ,opposite (equal to the remainder of the successor of the complementary). Now that we have defined the necessary structure to work with, we proceed to the definition of the encrypting and decrypting functions that we respectively denote as f and g :
`f :(Modulo TA)*(Modulo TA)` $\rightarrow$ `modulo TA`
`g :(Modulo TA)*(Modulo TA)` $\rightarrow$ `modulo TA`
By noting the clear message $\overline{M} = (\overline{m_1}, \overline{m_2}, \cdots, \overline{m_n})$, the encrypted $\overline{C} = (\overline{c_1}, \overline{c_2}, \cdots, \overline{c_n})$ and the encryption key $\overline{K} = (\overline{k_1}, \overline{k_2}, \cdots, \overline{k_n})$ where all their respective elements are of type `Modulo TA`, f and g do the following transformation :

$$\forall i \in \{1, \ldots, n\}, \quad \begin{array}{c} f(\overline{m_i}, \overline{k_i}) = \overline{c_i} \\ g(\overline{c_i}, \overline{TA - k_i}) = \overline{m_i} \end{array}$$

These two functions only operate on single couple of `(Modulo TA)*(Modulo TA)`, but the main aim of the algorithm is to implement the modular addition $\bigoplus$ defined in the section 3. For this reason, we implement the mathematical tool called a functor $\mathcal{F}$ [5] that has the property of conserving the cartesian products between two categories i.e:

$$\forall A, B \quad \mathcal{F}(A) * \mathcal{F}(B) \xrightarrow{iso} \mathcal{F}(A * B)$$

We proceed to the definition of this isomophism in Coq :
```
Definition
vecteur_preservationProduit_vecteurCouples{A
B : Type}{n : nat} : ((Vecteur A n) *
(Vecteur B n)) -> Vecteur (A * B) n :=
vecteur_biApplication (@id _).
Definition
vecteur_preservationProduit_coupleVecteurs{A
B : Type}{n : nat} : (Vecteur (A * B)
n) -> (Vecteur A n) * (Vecteur B n) :=
produitCartesien_produitFonctionnel
(vecteur_application fst)
(vecteur_application snd).
```
And then this functor $\mathcal{F}$ operates on the function f (resp. g):

$$\mathcal{F}(A * B) \xrightarrow{\mathcal{F}(f)} \mathcal{F}(A)$$

Given all these information, we define the following functor :

$$\mathcal{G}(f) = (iso; \mathcal{F}(f))$$

and we denote it as `Vecteur _ n`, thus $\overline{M}$, $\overline{K}$ and $\overline{C}$ are of type `Vecteur (Modulo TA) n`. We define, as well, the `chiffrement` and `dechiffrement` functions respectively as $\mathcal{G}(f)(\overline{M}, \overline{K})$ and $\mathcal{G}(g)(\overline{C}, \overline{K})$:

```
Definition chiffrement{d n : nat}
: (Vecteur (Modulo d) n) * (Vecteur
(Modulo d) n) -> (Vecteur (Modulo d) n)
:= vecteur_biApplication masquage.

Definition dechiffrement{d n : nat}
: (Vecteur (Modulo d) n) * (Vecteur
(Modulo d) n) -> (Vecteur (Modulo d) n)
:= vecteur_biApplication demasquage.
```

In this work , `vecteur_application` and `vecteur_biApplication` comprises applying a transformation function to the vectors :

```
Fixpoint vecteur_applicationA B : Typen
: nat(f : A -> B) (v : Vecteur A
n)struct n : Vecteur B n.
Fixpoint vecteur_biApplicationA B C
: Typen : nat(f : A * B -> C) (c :
(Vecteur A n) * (Vecteur B n))struct n :
Vecteur C n.
```

Given all these information, we obtain the final transformation scheme :

$$\begin{array}{rcc} \overline{C} & = & \mathcal{G}(f)(\overline{M}, \overline{K}) \\ & = & (f(\overline{m_1}, \overline{k_1}), \ldots, f(\overline{m_n}, \overline{k_n})) \\ & = & (\overline{c_1}, \overline{c_2}, \cdots, \overline{c_n}) \end{array}$$

$$\begin{aligned}\overline{M} &= \mathcal{G}(g)(\overline{C},\overline{K'}) \\ &= (g(\overline{c_1},\overline{TA-k_1}),\ldots,g(\overline{c_n},\overline{TA-k_n})) \\ &= (\overline{m_1},\overline{m_2},\cdots,\overline{m_n})\end{aligned}$$

4.2 The frontend

In this part of the program, we intend to operate the transformation of the clear message M of type `String` into a vector of ascii $(m_1, m_2, \cdots, m_n)$ where n is its length.

```
Fixpoint ascii_ChaineVecteur_retraction(n
: nat)(s : string) : Vecteur ascii n.
```

We define as well the opposite tranformation where the vector of ascii with a given length is transformed to a string with the same length.

```
Fixpoint ascii_VecteurChaine_plongement{n
: nat}(v : Vecteur ascii n) : string.
```

We proceed then to the verification that composing the first and second transformation on a given string generates the same string , in other terms the composition is equal to the identity function.

```
Proposition
ascii_VecteurChaine_plongementRetraction
: forall n, fonction_egalite
((@ascii_VecteurChaine_plongement n)
;; (ascii_ChaineVecteur_retraction n))
id.
```

In order to conduct the modular addition described in previous section, we operate a second transformation of the given clear message turned into a `Vecteur ascii n` to another vector of type `Vecteur (Modulo TA) n`.

In order to conduct this transformation and its reciprocal, we start by defining the functions that will operate on the ascii characters :

```
Definition ascii_traduction_Ascii_Modulo
: ascii -> Modulo TA := nat_of_ascii ;;
(modulo_morphismeNat TA).
Definition ascii_traduction_Modulo_Ascii
: Modulo TA -> ascii := modulo_valeur
;; ascii_of_nat.
```

After defining these two functions, we proceed to the definition of these transformations for `Vecteur ascii n` and `Vecteur (Modulo TA) n` using `vecteur_application` detailed in the previous subsection 4.1 :

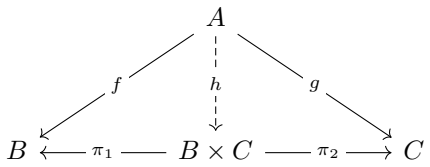
```
Definition
ascii_traduction_Ascii_Modulo_vecteur{n
: nat} : (Vecteur ascii n) -> (Vecteur
(Modulo TA) n) := (@vecteur_application
_ _ n ascii_traduction_Ascii_Modulo).

Definition
ascii_traduction_Modulo_Ascii_vecteur{n
: nat} : (Vecteur (Modulo
TA) n) -> (Vecteur ascii n) :=
(@vecteur_application _ _ n
ascii_traduction_Modulo_Ascii).
```

4.3 The corectness of the encryption algorithm

This property representes the fundamental aim of this work, because it will ensure that the given algorithm satisfies the logical rules thoroughly verified with Coq. The whole demonstration aims at proving that independently of the message to be encrypted and the encryption key, the composition of the encryption and decryption functions authentically reproduces the original message.

Preserving the f and g functions defined this the previous section, we cannot simply apply the composition $g \circ f$ because the encryption key will be lost in the first transformation as $f(\overline{m_i},\overline{k_i}) = \overline{c_i}$, thus losing the parameter $\overline{k_i}$ in this process.

To alleviate this problem, we resort to the product in category theory detailed in this following scheme :



As the scheme tells, we can see that both π_1 and π_2 are defined as projection functions and h can be defined as (f, g). Consequently, we ought to define : $\tilde{f} = (f, \pi_2)$ where :

$$\forall i, \quad \forall \overline{m_i},\overline{k_i} \quad \tilde{f}(\overline{m_i},\overline{k_i}) = (\overline{c_i},\overline{k_i})$$

Denoting that :

```
Notation "f # g" :=
(chiffrementSymetrique_sequence f g)
```

where :

```
Definition chiffrementSymetrique_sequence{M
P : Type}(f g : M * P -> M) :
M * P -> M := fonction_sequence
(produitCartesien_produitFonctionnel
f snd) g.
```

is the translation , in Coq, of the mathematical function :

$$g \circ \tilde{f}$$

and :
```
Definition idCS{M P : Type} :=
(@chiffrementSymetrique_identite M P).
```
where :
```
Definition chiffrementSymetrique_identite{M
P : Type} :  M * P -> M := fst.
```
is the translation , in Coq, of the mathematical function defined in the scheme above :

$$\pi_1$$

We obtain the fundamental theorem ensuring the corectness of the OTP algorithm :
```
Theorem chiffrement_masquage_correction :
forall (d n :  nat), @fonction_egalite
((Vecteur (Modulo d) n) * (Vecteur
(Modulo d) n)) (Vecteur (Modulo d) n)
(chiffrement # dechiffrement) idCS.
```
The demonstration of this theorem is based on the following theorem which uses algebraic properties on the group `Modulo TA`:
```
Theorem masquage_correction :  forall d,
@fonction_egalite ((Modulo d) * (Modulo
d)) (Modulo d) (masquage # demasquage)
idCS.
```
The mathematical demonstration of this theorem is that :

$$\begin{array}{lccc} & (\overline{m_i}+\overline{k_i}) & + & (-\overline{k_i}) & \\ = & \overline{m_i} & + & (\overline{k_i}+(-\overline{k_i})) & \textit{associativity} \\ = & \overline{m_i} & + & \overline{0} & \textit{inverse element} \\ = & \overline{m_i} & & & \textit{neutrality} \end{array}$$

The demonstration of the theorem above reveals the puissance of Coq proof assistant because it remains valid independently from the message and the encryption key and draws the origin of the demonstration from rather abstract logical mathematical concepts.

5 Results

In this section, we try to test the algorithm on some examples in order to verify its coherence and that it is functioning properly. For these reasons, we start by defining :
```
Definition testGenerique (n :  nat)(k m
:  string) :  Prop := ((chiffrer n) ##
(dechiffrer n)) (m , k) = Some m.
```

```
  Definition testErreurGenerique (n
:  nat)(k m :  string) :  Prop :=
((chiffrer n) ## (dechiffrer n)) (m ,
k) = None.
```
The first test ensures that the algorithm correctly operates the encryption and the decryption of the message in parameter whilst the second don't operate anything on the message and generates nothing, thus having the `None` as the outcome.

These results are purely logical as they derive from these verifications of the compatibility of the length of the clear message and the encryption key :
```
Theorem testGenerique_verification :
forall n k m, ((length m = n) /\(length
k = n)) -> testGenerique n k m.

Theorem testErreurGenerique_verification
:  forall n k m, ~((length m = n)
/\(length k = n)) -> testErreurGenerique
n k m.
```
Once all these requirements fulfilled, we proceed to the tests :
```
Example test5 :  testGenerique 10
"0123456789" "voiliers f".
compute.
reflexivity.
Qed.

Example testE5 :  testErreurGenerique
10 "0123456789" "voiliers fg".
compute.
reflexivity.
Qed.
```

6 Conclusion

In this paper we presented the Coq prrof assistant and used it to implement the One-Time pad encryption algorithm. We chose to verify and ensure the correctness of the algorithm by defining and proving various properties. We found that we had to look for many abstact mathematical structures for instance groups, functors, quotient groups etc and redefine all the mathematical properties and to prove that they satisfy the logical rules.

References

[1] Daniel Barsky and Ghislain Dartois "*Cryptographie Paris 13*", p 9-11,2010.

[2] Morten Heine B. Sørensen, Pawel Urzyczyn "*Lectures on the Curry-Howard Isomorphism*", p 57-72, 1998.

[3] `https://github.com/coq/coq/blob/master/theories/Lists/List.v`

[4] `https://github.com/coq/coq/blob/master/theories/Vectors/VectorDef.v`

[5] `https://en.wikipedia.org/wiki/Functor`

[6] Christine Paulin-Mohring, "*Mini-guide Coq*" `https://www.lri.fr/~paulin/MathInfo/`