Formal verification of an implementation of CRT-RSA Vigilant's algorithm

Maria CHRISTOFI
Joint work with Boutheina CHETALI, Louis GOUBIN and David VIGILANT

PROOFS 2012, September 13^{th} 2012

Introduction

- Implementations of cryptosystems can be sensitive to physical attacks, such as fault attacks
- Improved attack methods $\Rightarrow$ more attack paths
- Design more and more complex countermeasures
- No proof of flaw absence in the implementation
- This talk : Formal verification of cryptographic implementations
  - Example : Resistance of CRT-RSA Vigilant's algorithm against fault attacks

Formal verification of a cryptographic implementation

Formal Verification :

Use of formal methods (and the associated tools) to verify the correctness of an algorithm against its specification or/and a specific property

Two approaches :

- formalize the specifications and prove properties on the formal model of the specification $\Rightarrow$ What about the implementation ?
- "formalize" the source code $\Rightarrow$ That's what we talk about in this talk !

Verification techniques

How to achieve a formal verification

- **Mathematical proof :** completely manual
- **Theorem Proving :** mathematical reasoning mechanization
  - infinite models, partially automatic, human interaction
- **Model checking :** systematic and exhaustive exploration of the mathematical model
  - combinatoric exploration, finite model, completely automatic
- **Static analysis :** Software analysis with symbolic execution of the program
  - partially automatic

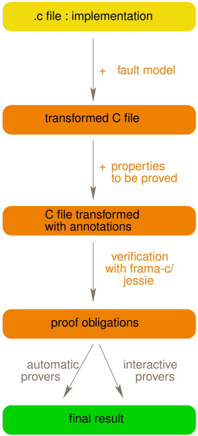
Some of the existing tools for source code analysis

- **VeriFast :** C and java program verifier. Programs first annotated with pre and post conditions (theorem proving)
- **Frama-C :** Platform dedicated to source code analysis of C programs (theorem proving & static analysis)
- **CertiCrypt** / **EasyCrypt :** Verification using games sequence
- Tools oriented protocols : **ProVerif, CryptoVerif**, etc

Global view

Aim :

Given an implementation of a cryptographic algorithm with countermeasures, define an attack model (here based on fault model) and formally verify that this implementation is resistant to this attack model.



Global view

fault model

Fault model

Classifying faults

- number of faults authorized per code execution
- faults on instructions VS faults on data
- fault types

	Precise Bit Fault Model	Single Bit Fault Model	Byte Fault Model	Random Fault Model	Arbitrary Fault Model
control on location	complete (chosen bit)	loose (chosen variable)	loose	loose	loose/no
control on timing	precise	no	no	no	no
number of affected bits	1	1	8	random	random
fault type	bit set or reset	bit flip	random	random	unknown
persistence	permanent and transient	permanent and transient	permanent and transient	permanent and transient	permanent and transient

Inject fault model

- If $NextType(var, i) \in \{write, \emptyset\}$ an attack on *var* injected on line *i* is useless and equivalent to the initial code.
- If $NextType(var, i) \in \{read, read/write\}$ and *j* the line that presents the next use of *var*, an attack on *var* injected on the interval $[i, j]$ has exactly the same effects on *var* with an attack injected on line *j* , but it has no effect between lines *i* and $j - 1$.

Example : we are interested in variable *a*

```
int   example(int a, int b){



      int x = 0 ;



      a = a + 1 ;




      x = a + b ;




      return x ;
}
```

Inject fault model

- If $NextType(var, i) \in \{write, \emptyset\}$ an attack on *var* injected on line *i* is useless and equivalent to the initial code.
- If $NextType(var, i) \in \{read, read/write\}$ and *j* the line that presents the next use of *var*, an attack on *var* injected on the interval $[i, j]$ has exactly the same effects on *var* with an attack injected on line *j* , but it has no effect between lines *i* and $j - 1$.

Example : we are interested in variable *a*

```
int    example(int a, int b){
         switch(f){
          case 1 : a = 0 ; break ;
          }
        int x = 0 ;
         switch (f) {
          case 2 : a = 0 ; break ;
         }
        a = a + 1 ;

         switch(f) {
           case 3 : a = 0 ; break ;
         }
        x = a + b ;

         switch(f) {
          case 4 : a = 0 ; break ;
         }
        return x ;
}
```

Inject fault model

- If $NextType(var, i) \in \{write, \emptyset\}$ an attack on *var* injected on line *i* is useless and equivalent to the initial code.
- If $NextType(var, i) \in \{read, read/write\}$ and *j* the line that presents the next use of *var*, an attack on *var* injected on the interval $[i, j]$ has exactly the same effects on *var* with an attack injected on line *j* , but it has no effect between lines *i* and $j - 1$.

Example : we are interested in variable *a*

```
1 :  int  example(int a, int b){        /* NextType(a,1) = read/write */
2 :         switch(f){
3 :          case 1 : a = 0 ; break ;
4 :          }
5 :        int x = 0 ;
6 :         switch (f) {
7 :          case 2 : a = 0 ; break ;
8 :         }
9 :        a = a + 1 ;                    /* Type(a,9) = read/write */
10 :                                      /* NextType(a,9) = read */
11 :         switch(f) {
12 :           case 3 : a = 0 ; break ;
13 :         }
14 :        x = a + b ;                    /* Type(a,14) = read */
15 :                                      /* NextType(a,14) = ∅ */
16 :         switch(f) {
17 :          case 4 : a = 0 ; break ;
18 :         }
19 :        return x ;
20 :  }
```

Inject fault model

- If $NextType(var, i) \in \{write, \emptyset\}$ an attack on *var* injected on line *i* is useless and equivalent to the initial code.
- If $NextType(var, i) \in \{read, read/write\}$ and *j* the line that presents the next use of *var*, an attack on *var* injected on the interval $[i, j]$ has exactly the same effects on *var* with an attack injected on line *j* , but it has no effect between lines *i* and $j - 1$.

Example : we are interested in variable *a*

```
1 :  int    example(int a, int b){          /* NextType(a,1) = read/write */
2 :           switch(f){
3 :            case 1 : a = 0 ; break ;
4 :            }
5 :          int x = 0 ;
6 :           switch (f) {
7 :            case 2 : a = 0 ; break ;
8 :           }
9 :          a = a + 1 ;                    /* Type(a,9) = read/write */
10 :                                          /* NextType(a,9) = read */
11 :           switch(f) {
12 :             case 3 : a = 0 ; break ;
13 :           }
14 :          x = a + b ;                     /* Type(a,14) = read */
15 :                                          /* NextType(a,14) = ∅ */
16 :           switch(f) {
17 :            case 4 : a = 0 ; break ;
18 :           }
19 :          return x ;
20 :  }
```

Inject fault model

- If $NextType(var, i) \in \{write, \emptyset\}$ an attack on *var* injected on line *i* is useless and equivalent to the initial code.
- If $NextType(var, i) \in \{read, read/write\}$ and *j* the line that presents the next use of *var*, an attack on *var* injected on the interval $[i, j]$ has exactly the same effects on *var* with an attack injected on line *j* , but it has no effect between lines *i* and $j-1$.

Example : we are interested in variable *a*

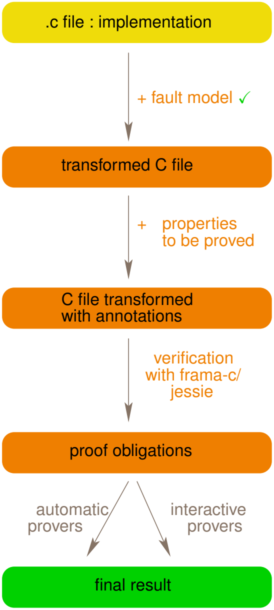
```
 1 :  int  example(int a, int b){        /* NextType(a,1) = read/write */
 2 :
 3 :
 4 :
 5 :         int x = 0 ;
 6 :          switch (f) {
 7 :           case 1 : a = 0 ; break ;
 8 :          }
 9 :         a = a + 1 ;                  /* Type(a,9) = read/write */
10 :                                       /* NextType(a,9) = read */
11 :          switch(f) {
12 :            case 2 : a = 0 ; break ;
13 :          }
14 :         x = a + b ;                  /* Type(a,14) = read */
15 :                                       /* NextType(a,14) = ∅ */
16 :          switch(f) {
17 :           case 3 : a = 0 ; break ;
18 :          }
19 :         return x ;
20 :  }
```

Inject fault model

- If $NextType(var, i) \in \{write, \emptyset\}$ an attack on *var* injected on line *i* is useless and equivalent to the initial code.
- If $NextType(var, i) \in \{read, read/write\}$ and *j* the line that presents the next use of *var*, an attack on *var* injected on the interval $[i, j]$ has exactly the same effects on *var* with an attack injected on line *j* , but it has no effect between lines *i* and $j - 1$.

Example : we are interested in variable *a*

```
 1 :  int   example(int a, int b){           /* NextType(a,1) = read/write */
 2 :
 3 :
 4 :
 5 :          int x = 0 ;
 6 :           switch (f) {
 7 :            case 1 : a = 0 ; break ;
 8 :           }
 9 :          a = a + 1 ;                      /* Type(a,9) = read/write */
10 :                                           /* NextType(a,9) = read */
11 :           switch(f) {
12 :             case 2 : a = 0 ; break ;
13 :           }
14 :          x = a + b ;                      /* Type(a,14) = read */
15 :                                           /* NextType(a,14) = ∅ */
16 :
17 :
18 :
19 :          return x ;
20 :  }
```


.c file : implementation
+ fault model ✓
transformed C file
+ properties to be proved
C file transformed with annotations
verification with frama-c/ jessie
proof obligations
automatic provers
interactive provers
final result

properties to be proved

Properties to be proved

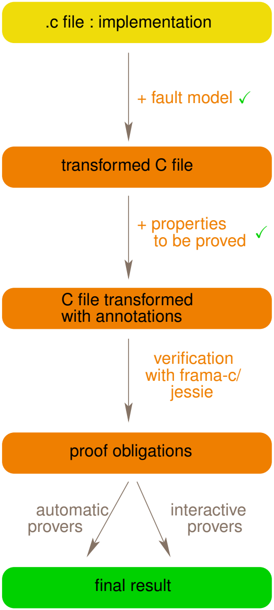
Informally,

we want to check whether any possible attack can be detected by the defined set of countermeasures

Formally,

Let $f \in \{0\} \cup F$, where F is the set of faults for the current implementation and $f = 0$ the original execution of the implementation (without injected faults). Let also *res* be the output of the implementation, $x_1, ..., x_n$ be the n variables of the input of the implementation and g a function. Then :

$$[(f = 0) \Rightarrow (res = g(x_1, ..., x_n))] \text{ AND}$$
$$[(\forall f \in F) \Rightarrow ((res = ERROR)\ OR\ (res = g(x_1, ..., x_n))]$$


.c file : implementation
+ fault model ✓
transformed C file
+ properties
to be proved ✓
C file transformed
with annotations
verification
with frama-c/
jessie
proof obligations
automatic
provers
interactive
provers
final result

verification with frama-c/ jessie

Verification with Frama-C / WHY / Jessie

Frama-C

- a platform for analyzing a C program
- includes different techniques of static analysis

Why / Jessie

- Why :
  - proof obligations generator
  - input : programs + first logic assertions
  - output : logic assertions + proof obligations on the chosen prover language
- Jessie :
  - Why plug-in
  - based on weakest precondition computation techniques

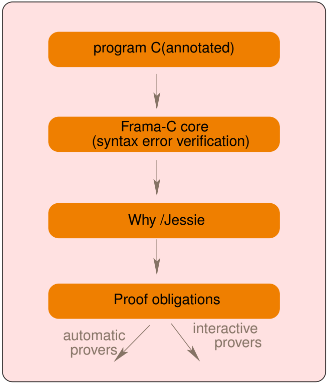


FIGURE: Frama-C platform

To sum up

- Describe the implementation to verify
- Define the fault model
- Inject faults on the original code
- Describe the properties to be proved
- Proceed to the verification
- Exploit the results

Let's see a concrete example...

Case study

- **Algorithm :**
  - CRT-RSA algorithm
- **Countermeasure :**
  - Vigilant's countermeasure
- **Implementation :**
  - pseudo code published in Vigilant's paper :
    "RSA with CRT : A New Cost-Effective Solution to Thwart Fault Attacks"
    CHES 2008

CRT-RSA algorithm

parameters

public key : (N, e)

private key : (p, q, d_p, d_q, i_q)
such that :

$$N = p \cdot q \ (p, q \text{ large primes})$$
$$\gcd(p-1, e) = 1$$
$$\gcd(q-1, e) = 1$$
$$d_p = e^{-1} \mod (p-1)$$
$$d_q = e^{-1} \mod (q-1)$$
$$i_q = q^{-1} \mod p$$

CRT-RSA algorithm

Input : $m \in \mathbb{Z}_N, p, q, d_p, d_q, i_q$

Output : $m^d \in \mathbb{Z}_N$

$S_p = m^{d_p} \mod p$

$S_q = m^{d_q} \mod q$

$S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$

return $S \mod N$

Vigilant's countermeasure

- Choose a random r , s.t. $\gcd(N, r^2) = 1$
- We want : Exponentiation modulo N ($m^d \mod N$).
- Instead, compute exponentiation modulo Nr^2 ($m'^d \mod Nr^2$).

$$m' \equiv \begin{cases} m \mod N \\ 1 + r \mod r^2 \end{cases}$$

  - Verification of the exponentiation result consistency modulo r^2 $((m'^d \mod r^2) = (1 + dr))$
- Same principle for computation of S_p and S_q
- Exponentiation result reduced modulo N

Verification of CRT-RSA Vigilant's algorithm

Fault model

- inject one fault per execution
- modify the value in memory by setting the value of a variable to 0
- inject both transient and permanent faults to any variable
- modify only data (not the code execution)
- cannot modify the boolean result of a conditional check

Property to prove

- $(f = 0) \Rightarrow$
  $((\mathit{output} \mod p = m^{d_p} \mod p) \; \mathit{AND} \; (\mathit{output} \mod q = m^{d_q} \mod q))$
- $(f \in F) \Rightarrow$
  $((\mathit{output} = \mathit{ERROR})$ OR
  $((\mathit{output} \mod p = m^{d_p} \mod p) \; \mathit{AND} \; (\mathit{output} \mod q = m^{d_q} \mod q)))$

Results

Faults with success probability 1

- faults on random variables
- output : the real signature
- no information about the secret parameters is obtained
- depending to the fault model this may give information on the faulty variable. It is the case for our model.

Faults with a weak success probability

- output : a faulty signature
- probabilities manually calculated : $2^{-2|r|+1}$, $2^{-(|p'|-1)} ln2$ and $2^{-(|q'|-1)} ln2$

Faults with a high success probability : 1

- faults on d_p and d_q during the computation of d'_p and d'_q
- output : a faulty signature
- attacker can extract information about the secret data
- no danger for the original fault model

Summary

- Method :
  - Select a fault model
  - Inject faults to the original code (w.r.t. the chosen fault model)
  - Verify using frama-C
- Verify methodically cryptographic implementations
- Increase confidence to our implementations
- Eliminate flaws due to countermeasures weaknesses

Questions / Remarks / Propositions are more than welcome ! !

maria.christofi@gemalto.com