Attack Model for Verification of Interval Security Properties for Smart Card C Codes

P. Berthomé
Centre-Val de Loire Université
Ensi de Bourges, LIFO
88 bd Lahitolle
18020 Bourges, France
pascal.berthome@ensi-bourges.fr

K. Heydemann
UPMC / LIP6
BC 167
4 place Jussieu
75252 Paris Cedex, France
karine.heydemann@lip6.fr

X. Kauffmann-Tourkestansky
Oberthur Technologies
Ensi de Bourges, LIFO
71-73 rue des Hautes-Pâtures
92726 Nanterre, France
x.kauffmann@oberthurcs.com

J.-F. Lalande
Centre-Val de Loire Université
Ensi de Bourges, LIFO
88 bd Lahitolle
18020 Bourges, France
jfl@univ-orleans.fr

ABSTRACT

Smart card programs are subject to physical attacks that disturb the execution of the embedded code. These attacks enable attackers to steal valuable information or to force a malicious behavior upon the attacked code. This paper proposes a methodology to check interval security properties on smart card source codes. The goal is to identify critical attacks that violate these security properties. The verification takes place at source-level and considers all possible attacks thanks to a proposed source-level model of physical attacks. The paper defines an equivalence relation between attacks and shows that a code can be divided into areas where attacks are equivalent. Thus, verifying an interval security property considering all the possible attacks requires to verify as many codes as the number of equivalence classes. This paper provides a reduction algorithm to define the classes i.e. the minimal number of attacked codes that covers all possible attacks. The paper also proposes a solution to make the property verification possible for large codes or codes having unknown source parts.

Categories and Subject Descriptors

I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*model validation and analysis*; B.8.1 [**Hardware**]: Performance and Reliability—*reliability, testing, and fault-tolerance*

PLAS '10, June 10, 2010 Toronto, Canada

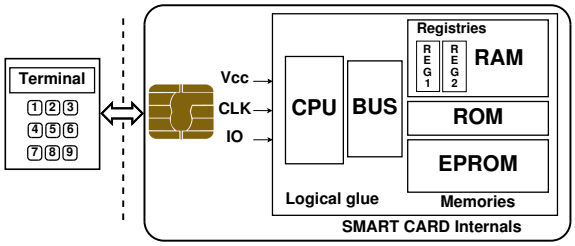

Figure 1: Terminal and smart card components

Keywords

Smart card, physical attacks, verification

1. INTRODUCTION

This paper addresses the security of embedded software on smart cards. Embedded applications are protected by the difficulty for attackers to access the program and the data that are manipulated. Chips on smart cards are small processors with strong hardware protection capabilities such as cryptographic hardware routines or unalterable register memories that guarantee a high and reliable level of security. Nevertheless, attackers are still able to perform physical attacks against the card. This is a well known issue [2] and is one of the hypothesis of this paper.

A payment transaction is an example of software executed on a smart card. The smart card will interact with a payment terminal in order to authenticate the user and perform a bank transfer. Embedded software can also deal with authentication, health care or mobile applications that will be executed when the card is powered until it is disconnected. Such an execution that requires high and reliable level of security will be called a transaction.

Figure 1 shows a smart card with its components and its communication with a terminal. Data sent by the terminal will go through the input/output pad (IO) and will be pro-

cessed in the logical glue, for example it will be stored in a buffer. Code in ROM memory can then instruct the CPU to read this data from the buffer. The data will then transit through the bus to be loaded in a RAM register. Data from EPROM memory can be similarly loaded in a RAM register. A function can then compare these two registers to check if their values are equal and therefore that the data from the terminal is the same as the one embedded in the smart card. This is an oversimplified use case for the sake of comprehension, adding security greatly complicates the process.

When performing physical attacks, for example with a laser pulse, an attacker may succeed in disturbing the behavior or the data of executed programs. Even if the attacker does not have the source code of the program, he may have an idea of the operations performed during the transaction. For example, a cryptographic computation could be detected by measuring the smart card's power consumption. Even if the attacker has no information on what he succeeded in inducing during a transaction, physical attacks can be blindly launched and their consequences, if any, logged.

Most of the time, the performed attack does not produce any interesting result for the attacker. The card may turn into an unstable state that will be detected by the defensive countermeasures of the card. In this case, the card should prevent any further transaction. But in some cases, the card will continue to work even after the attack. The attacker has then the opportunity to perform the same attack on the next transaction with some small changes or improvements and observe the impact on a measured output. Measured outputs are for example power consumption or data exchanges with the terminal. An attack could benefit the attacker if he obtains some data he is not supposed to extract from the card, or if it modifies some data that should not be.

In order to defend against such physical attacks, smart card manufacturers implement software protections to protect the secrets buried in the card. The simplest example is an access control with a pin code that must remain confidential in the card. Even if the code manipulates the pin code using variables, it should not be sent to the terminal, even if an attacker disturbs the program. For example, a physical attack could succeed in copying the pin code into another variable that is used for data transfer with the terminal.

There are several difficulties to protect the card from such issues. First, the software mechanisms that check the correct behavior of the program could be bugged. Under some circumstances that never happened during the tests campaigns, a new information flow could be created disclosing sensitive data. Second, it is impossible to predict all the possible physical attacks and their consequences due to the large number of possible attacks. Thus, software developers have no idea if the implemented protections will resist to any code disruption. One of the goals of this paper is to solve the impossibility of taking into account all the possible attacks and their consequences.

This paper presents a methodology to check simple security properties on the source code of a transaction. The goal of this methodology is to identify critical attacks that violate these security properties. A software developer could then add software countermeasures for the identified attacks and rerun the identification process until the properties are guaranteed. The verification takes place at C level and considers all possible attacks using a physical attack model. This model includes different types of attack, their duration characteristics and C codes that simulates them. Verification of the security properties is processed first on the initial code and then on the codes where all the possible attacks are injected.

The paper defines an equivalence relation between attacks and shows that the code can be divided into areas where attacks are equivalent. Thus, taking into account all the possible attacks to verify a given security property requires the verification of as many codes as the number of equivalence classes. For codes with sequential control flow, the number of classes is the minimal number of codes to be verified to ensure a property and to cover all possible attacks. But for other codes, we show that this number can be reduced by using a notion of attack coverage defined in the paper. The paper gives the way to build and the number of codes to be verified to ensure a property while covering all possible attacks.

Moreover, the verification may fail due to lack of information, the paper proposes a solution to bypass this limitation by performing a preliminary dynamic analysis step to retrieve the information needed for the verification to succeed.

The paper is organized as follows: Section 2 gives an overview of past works on physical attacks against smart cards; Section 3 presents physical attacks and proposes a corresponding C model; Section 4 recalls the principles of verification using a static verifier and introduces the designed approach for checking partial smart card codes; Section 5 shows how to build and reduce the number of attacked codes that have to be checked by the verifier.

2. STATE OF THE ART

This section deals with works that are dedicated to physical attacks targeting smart cards. The studied attacks are described and each approach is commented. It tackles the goals of attackers and the security properties that are involved. At the end of the section, the difficulties of code verification are briefly presented.

2.1 Attacks against smart cards

In [12], the authors give a hierarchical classification of attackers and their methodologies. Social attacks, that are out of scope of this paper, are distinguished from physical and logical attacks. A physical attack can be an active one when physically targeting the chip. A passive attack analyzes the electrical power of the components. This is one of the first works that present attacks and discusses the risk level and costs of such attacks.

Two categories of attacks are proposed in a survey that focuses on attacks against cryptographic algorithms [3]. An attack can be either transient or permanent. A transient attack occurs only once during the transaction while a permanent one affects the program during the whole transaction. An example shows how to exploit an attack that injects a fault in RSA and DES algorithms. The consequences of such an attack is characterized describing how the EPROM and bus are affected.

In [14], the author presents a methodology to simulate physical attacks. An automation tool is proposed to simulate several attacks. First, NOP opcodes can be injected simulating a laser pulse that overwrites an assembly operation. Second, the memory data can be replaced by special values (0x00 or 0xFF) that simulate a physical attack on the

EPROM. Third, an assembly operation can be replaced, for example an INC can be turned into a DEC, a STORE into a LOAD. The main difference with the attack model that will be proposed in this paper is that [14] considers that faults occur randomly. It is more a safety point of view that focuses on stability of the program execution rather than on security.

2.2 Attacker's goals

Classical attacks considered in the literature try to bypass implemented security measures in order to get illegal access to sensitive data buried in the chip. In [10] several means of attack are presented. They range from the use of electric current to disrupt the memory of the chip to the use of laser cutters to manipulate the electric circuits. Hardware countermeasures to defend against these attacks are also presented and authors show that from a hardware point of view most of the attack methods are known and that countermeasures might be bypassed as described in [2]. This raises another issue as to how to best secure the program embedded in the chip from a software point of view. From a higher standing point, [1] describes the sensitive assets in a card, its life cycle, and presents the various security requirements that have to be met in order to pass the Common Criteria certification. The goal of an attacker will be to reverse engineer the program embedded in the chip, using side channel for example, in order to understand its critical computation and to eventually physically attack the chip to gain access to sensitive information. Logical attack patterns, as shown in [13], are taken into account when specifying the software to weight their cost and feasibility and then design the program accordingly. These patterns and the difficulty to assess them under the influence of physical attacks is an important issue when securing embedded software.

Classically, a security measure might protect a sensitive asset like the pin code of a card. Such a sensitive data should remain confidential and be protected against modification attempts. A physical attack using laser beams disrupting the physical components of the card may try to bypass the assembly code that checks the pin code entered by the user. This is also true for cryptographic algorithms: the first academic fault attack against a public RSA signature has been described in [6]. By using transient fault attacks to provoke faulty computations in the algorithm it is possible to recover the private key of the party trying to authenticate itself without triggering any countermeasures. Using a statistical approach to improve this theory, the authors of [5] have shown that such attacks are possible and can be used to break the DES algorithm.

Attackers also try to combine their attacks with side channel observations. Electrical patterns of known algorithms can be recognized when observing the electrical consumptions of the card. For example, a cryptographic function may be identified when a certain electrical variation is seen. In [9], the authors describe how side channel observations can be used to disclose some sensitive information from the card and propose some countermeasures against these possible attacks.

Fortunately these attacks are most of the time useless, due to the instability of the smart card system once it has been tampered with. For example, a physical perturbation may cause the program to jump to an unknown address or a C pointer to point to a non functional memory information. In many situations the system does not go back to a plausible state and thus is useless for the attacker. Moreover, it is difficult to reproduce an attack. The card may adopt a different behavior for each execution, for example by introducing random temporal delays. So, the attacker may get an interesting result during one try and not be able to reproduce it later.

Attackers have the following global goals when performing physical attacks against smart cards: 1) break the confidentiality of some data; 2) break the integrity of some data; 3) obtain a non authorized behavior of the program. These are standard security properties that have been widely studied for operating systems or databases for example. For smart cards, there is no formal model of security to express the data and program behaviors to be protected at source level. This is one of the contribution of this paper: an attack model is given and a basic security property (interval confinement of a variable) are studied in depth, analyzing the interactions between the modeled attacks and the security property.

2.3 Verification

Due to memory size constraints, C language is very commonly used in smart cards' operating system and embedded applications. It offers a higher abstraction level than assembly without the heaviness of high level languages such as Java which needs a virtual machine in order to work. Nevertheless, the C language suffers from several drawbacks: 1) it is not strongly typed, 2) the memory can be freely accessed, 3) it does not provide an object oriented programming language. These characteristics prevent verification techniques from deeply analyzing C programs. Moreover, the card registers are accessed using dedicated assembly instructions that are not recognized by static C verifiers which are usually limited to the ANSI C standard. Nevertheless, they can provide advanced features like a memory model that can describe the behavior of pointers during the analysis [7].

3. PHYSICAL ATTACK MODEL

This section describes an attack model at C level which is easier to understand. Moreover, there is no attack model at assembly level in the literature. Most of the time research papers only show the effects of a precise attack on assembly instructions [14]. The goal of the proposed model is to specify types of attack using general rules on C instructions.

Several abstraction levels can be considered to characterize attacks on the card:

1. Assembly level: concretely it is at this level that attackers tamper with registers and memory cells.

2. C level: at this level attacks are easier to specify and security countermeasures are also implemented in C. However attacks conceptualized at this level are the result of physical attacks on assembly code.

3. Hardware level: specific or specialized components of the chip can be physically tampered with.

Any performed attack is a physical perturbation of a chip's component. In this paper, how to perform attacks and resulting physical perturbations are not described: the paper focuses on the consequences of attacks. In this section, we

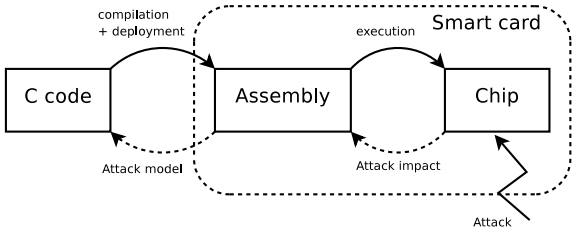

Figure 2: Abstraction levels for hardware attacks

show how the observed consequences of some attacks at assembly level can be simulated at C level in order to obtain a convenient way of studying these attacks (exhaustive projection of all possible assembly attack into the equivalent C code level is out of scope of this paper). Figure 2 sums up the relationships between the abstraction levels. The source code is compiled in assembly and deployed on the smart card. The card is able to execute this assembly code. In the event of a physical attack, the hardware components of the chip are disturbed and the consequences can be represented on the assembly code and then simulated at C level.

3.1 Assembly attack to C attack

A physical perturbation of the hardware can lead to a modification of assembly registers. In practice, the attacker wants to set interesting values from which he can take advantage. This value can be: fixed, the attacker succeeds in choosing the value that should be read by the processor; random, the attacker succeeds in setting a random value that is read by the processor.

The modification of a register can change the effect of a sequence of assembly instructions that accesses a memory variable in EPROM: this kind of attack is called a "memory attack". If the perturbation modifies the type of an assembly instruction (for example deletes a jump), the attack is called a "code attack". The next two sections give some examples of translations of memory and code attacks from assembly level to C level.

3.1.1 Memory attack

A memory attack example could consist in attacking the following assembly/C code:

```
CALL check_pin
MOV R0,#pin
MOV A,R7
MOV @R0,A
MOV R7,#pin
RET
```

```
pin = check_pin(1234,0000);
return pin;
```

If the second MOV is attacked, the executed assembly code and the corresponding equivalent C code become:

```
CALL check_pin
MOV R0,#pin
MOV A,#0F0H
MOV @R0,A
MOV R7,#pin
RET
```

```
pin = check_pin(1234,0000);
pin = 0x0F0;
return pin;
```

3.1.2 Ignore code attack

Another example could consist in attacking the code of Section 3.1.1 by replacing the first instruction by a NOP to bypass the check_pin operation. The obtained code and the corresponding C code is given below:

```
NOP
MOV R0,#pin
MOV A,R7
MOV @R0,A
MOV R7,#pin
RET
```

```
return pin;
```

The effect of an ignore code attack depends on the sequence of instructions produced by the compiler, so it depends on the optimization level that is set. Furthermore the binary code that is interpreted by the hardware can also vary between chip manufacturers. For example, jump instruction code can be mapped to 22 or 23. Consequently the effect of an attack depends on the value mapping of opcodes. So, the effects of physical attacks on each architecture have to be studied independently.

3.1.3 Operation injection attack

At C level, instructions are separated by semi colons and interpreted sequentially. At assembly level it is different because during execution the binary code is read byte after byte even if the command part of an instruction, the opcode, or the arguments of an instruction has been modified by an attack.

```
MOV 22,R0
```

Listing 1: Bytewise interpretation

In Listing 1 the MOV opcode will be read and interpreted then the two following bytes are read, they correspond to the first argument (22) and the second one (R0). Dangerous attacks are attacks that have a high probability of keeping the flow of the program in a stable state but with a "silent" fault. For example, the code of Listing 1 can be attacked by an ignore code attack making the processor interpret the resulting two instructions as a jump to the R0 address while resuming the flow of the program without memory address shifts:

```
MOV 22,R0 → NOP 22,R0 → JMP R0
```

In order to take into account all the possible attacks at assembly level, around 40 component-dependent opcodes have to be studied (average number of opcodes for two 8051 development platforms). This work is out of scope of this paper. Nevertheless, a lot of these attacks have a similar consequence to the memory attack presented in Section 3.1.1, or the ignore attack presented in Section 3.1.2 or the jump attack presented in this section. All these types of attack will be conceptualized at C level in Section 3.3.

3.2 Attack duration

An attack also has temporal effects during the execution of the code. An attack falls into one of the three following duration categories explained in this section: 1) transient: the attack occurs once during a transaction; 2) transient repetitive: the attack occurs several times in the same way and on the same spot; 3) permanent: the consequence of the attack is everlasting.

A transient attack occurs at a precise place on the code and only once during the execution. It is performed using a single short laser pulse. It means that the consequence is limited in time if we consider the C code. For example in a

```
int attack_trigger = false; /* is the attack on? */
int time_is_reached = false; /* synchronization condition */
int time_is_over = false; /* stop condition */
int main() {
  ...
  if (attack_trigger && time_is_reached && !time_is_over)
  {
    /* perform the attack */
    attack = !attack;
}}
```

Listing 2: C model for transient attacks

loop, a variable can be disrupted during one iteration and not during the others. Another example can be the introduction of a jump to a function due to a glitch attack that occurs only once during the execution. This is the most difficult attack to model, as the corresponding C modification must not be permanent in the code. This can be represented with a boolean variable attack_trigger that is flipped when the attack has been performed and a boolean variable time_is_reached that models the synchronization condition that triggers the attack, as shown in Listing 2. For the end of the attack, the same model is applied using the time_is_over boolean.

A transient repetitive attack is a transient attack that is repeated several times during the execution of the transaction but localized at the exact same place in the code. It means that the synchronization condition triggering the attack is the same for each laser pulse that is sent (represented by the boolean attack_trigger in Listing 2). For example, a C variable can be disrupted for several iterations of a loop.

Finally, a permanent attack is a transient repetitive attack that occurs during the whole time of the transaction. It means that the simulation of the attack introduces a permanent C instruction in the code. For example, a permanent pulse of laser on an EPROM register will fix the value of one or more C variables to an arbitrary value. Several variables can be impacted because they will be loaded in the same register that is under attack by a permanent pulse. This is represented by time_is_reached = true and attack_trigger=true.

3.3 C model of attacks

This section gives the model of simulation that is used to inject attacks as C instructions. The different assembly attacks are represented using two classes: a modification of a variable and a jump in the code. In the next sections, the simulated attacks are illustrated considering the code of Listing 3.

3.3.1 Variable modification

When a physical attack disturbs the value of a register, for example into the EPROM, any variable of the program whose value is loaded from the corresponding EPROM location is affected. The C model that is proposed introduces a new C instruction that assigns an arbitrary value to this variable as shown in Listing 4 between lines 3 and 4. This type of attack is the easiest to study. This paper will focus on this kind of attacks and the proposed methodology described in Section 4 aims at guaranteeing a security property for a code attacked by such a value modification attack. The described attack in the next section completes the C model of attack. Our intention is to give in Section 3 an as complete as possible C model of simulation of possible attacks.

```
int func() {                          1
...                                   2
target_val = 0x00;                    3
...                                   4
...                                   5
if (!security(target_val))            6
  secret = authorize();               7
return secret;                        8
}                                     9
```

Listing 3: Example

```
int func() {                          1
...                                   2
target_val = 0x00;                    3
target_val = 0xF0;                    4
...                                   5
if (!security(target_val))            6
  secret = authorize();               7
return secret;                        8
}                                     9
```

Listing 4: Modification

```
int func() {                          1
...                                   2
target_val = 0x00;                    3
...                                   4
... goto label;                       5
if (!security(target_val))            6
  secret = authorize();               7
label:                                8
return secret;                        8
}                                     9
```

Listing 5: Jump attack

```
int func() {                          1
...                                   2
target_val = 0x00;                    3
...                                   4
...                                   5
if (!security(target_val))            6
  secret = authorize();               7
                                      8
return secret;                        8
}                                     9
```

Listing 6: Ignore attack

3.3.2 Jump attack

As presented before in Section 3.1.2, disturbing an instruction can lead to modify the nature of the assembly instruction. Even if controlling such an attack is really difficult, an attacker may succeed in introducing a jump. This assembly instruction is simulated by a *goto* C instruction, as shown in Listing 5 where the attack is inserted between lines 4 and 5.

The modification of an assembly instruction could also lead to a NOP operation, which may result in ignoring a condition. The example of Listing 6 shows a line that has been deleted. This deletion could be simulated by using a *goto* to line 7. So, ignoring a C line is included in the *goto* model of attacks.

3.4 Attack injection examples

Let us consider an example of C code, shown in Listing 7. It represents a banking software that performs some operation and computes a result. The main code is not given here to simplify the reading: it is a 8051 C code that calls the method *banking()* during the transaction.

In such a code, the attacker could first focus on the authentication mechanism, in line 20. When the wrong pin code is entered, the transaction is aborted. The attacker could try to disturb the *if* condition to obtain the execution of the function *work()* for any entered pin code. In the proposed model, a *goto* instruction would be injected before line 21 and a label would be added before line 22.

Second, the attacker could focus on the function *work* that computes a value (for example a money amount) before returning this value. The attacker could try to disturb one of the components of the addition of line 12. In this case, the attacker can disturb the x local variable or one of the global variables $number1$ or $number2$ to obtain the wanted result. For example, the attack *x=255;* can be injected before line 12. To obtain the result, several attacks are possible. Even worse, if the function *work()* has a large number of instructions, it gives more time for the attacker to perform the attack.

This example shows that a formalization of the data that

```
#include <stdio.h>                                                    1
typedef unsigned int word;                                           2
word number1; /* 2 global variables initialized elsewhere */        3
word number2;                                                        4
extern word scanp(); /* PIN code input prompt */                     5
                                                                     6
word work() { /* Work when authenticated */                         7
  word result;                                                       8
  word x = 0;                                                        9
  if (number1 > 10)                                                  10
    x++;                                                             11
  result = number1 + number2 + x;                                    12
  return result;                                                     13
}                                                                    14
                                                                     15
void banking() { /* Main code for the banking software */           16
  word res;                                                          17
  word pin_input;                                                    18
                                                                     19
  pin_input = scanp();                                               20
  if (pin_input == 1234) {                                           21
    res = work();                                                    22
  }                                                                  23
}                                                                    24
```

Listing 7: Example of C code for a banking software

should be protected is needed. This is what is achieved in Section 4 where the interval security property of a sensitive variable is presented before moving to the description of the verification process. This section aims at expressing the wanted security and at showing how to verify it, with or without attack injection using the C attack model.

The model of attack is a simulation of what happens at assembly level. This model requires to choose places where attacks have to be injected. The number of locations for possible variable modifications or jump attacks is very large. This issue is addressed in Section 5 where we prove that we are able to enumerate the minimal number of attack places to be considered and that others are equivalent to the enumerated ones.

4. SECURITY VERIFICATION

This section presents the kind of properties the paper focuses on, and how the verification of such properties is performed by a verification tool. Finally, the limitation and issues relative to the verification steps are described before giving a solution to be able to succeed in the verification.

4.1 Interval variable verification

The considered property is the interval consistency of a sensitive variable. If x is identified as a sensitive variable, the developer may express that x should be, at a point of a transaction, in a defined interval $[x_{min}, x_{max}]$. Of course, the goal of this work is not only to check with a verification tool if this property is true, but also to guarantee that the property stays true for any injected attack. If the property is not verified, the smart card developer might be able to implement countermeasures and check the new code again.

DEFINITION 1. *$Interval(x, i, x_{min}, x_{max})$ is defined as the property that says if x is in the interval $[x_{min}, x_{max}]$ at line i of the program.*

This property is useful to check if a piece of code is consistent with a variable, when manipulating this variable. For the case of the pin code, the developer may want to check

```
#include <stdio.h>                                                    1
typedef unsigned int word;                                           2
word number1; /* 2 global variables initialized elsewhere */        3
word number2;                                                        4
extern word scanp(); /* PIN code input prompt */                     5
                                                                     6
word work() { /* Work when authenticated */                         7
  word result;                                                       8
  word x = 0;                                                        9
  if (number1 > 10)                                                  10
    x++;                                                             11
  /* Check property 2 */                                             12
  /*@ assert 0 <= x <= 1; */                                         13
  result = number1 + number2 + x;                                    14
  return result;                                                     15
}                                                                    16
                                                                     17
void banking() { /* Main code for the banking software */           18
  word res;                                                          19
  word pin_input;                                                    20
                                                                     21
  pin_input = scanp();                                               22
  /* Check property 1 */                                             23
  /*@ assert 0 <= pin_input <= 9999; */                              24
  if (pin_input == 1234) {                                           25
    res = work();                                                    26
  }                                                                  27
  /* Check property 3 */                                             28
  /*@ assert 0 <= res <= 100; */                                     29
}                                                                    30
```

Listing 8: Banking C code with ACSL annotations

$Interval(pin\_code, 40, 0000, 9999)$ if some treatments on the pin code occurs after line 40.

4.2 Verification of interval properties

When an attack is performed at line i of a program the new source code is built including the simulated attack using the process described in Section 3.4. Then, in both cases, for the original code and the attacked code, the property is verified using a semantic abstract interpreter. In our experiments, we have chosen the Frama-C tool [7] that is able to natively performs value analysis checks on variables for C codes [8].

The property is expressed using the ACSL language [4] which annotations are included in C comments. Let us consider the example of Listing 7 and the three following properties:

1. The pin code should be in $[0, 9999]$ before the *if*;
2. Variable x should be in $[0, 1]$ before computing *result*;
3. At the end of *banking()*, the result should be in $[0, 100]$.

Note that properties are requirements that are to be specified by the developer or the security advisor of the software. The obtained code using the ACSL annotation is given in Listing 8. The verification process of such a code gives different results for each property. The partial output of Frama-C is given in Listing 9.

Frama-C first recognizes global variables, *number1* and *number2* in the example, and assigns them an unknown value interval. Then, the analysis is performed and Frama-C reaches the different assert statements. First, Property 1 is checked at line 24. This property cannot be verified as it depends on *scanp*[1] whose code has not been given to Frama-C.

[1] *scanp* is a kind of *scanf* for the smart card as it reads the pin code as input on the terminal.

```
[value] ====== INITIAL STATE COMPUTED ======
[value] Values of globals at initialization
          number1  ∈  [−−..−−]
          number2  ∈  [−−..−−]
[kernel] No code for function scanp, default assigns generated
banking−input2.c:24:[value] Assertion got status unknown.
banking−input2.c:13:[value] Assertion got status valid.
banking−input2.c:29:[value] Assertion got status unknown.
[value] ====== VALUES COMPUTED ======
[value] Values for function work:
            result  ∈  [−−..−−]
            x {0; 1; }
[value] Values for function banking:
            res  ∈  [−−..−−] or UNINITIALIZED
            pin_input  ∈  [0..9999]
```

Listing 9: Output results from the Frama-C tool

Second, the property at Line 13 is checked. This property is easily verified by Frama-C: the output indicates that the status of this property is valid. Third, Property 3 at Line 29 is checked and got an invalid status. This is due to the unknown result returned by the function *work()*. Note that at the end of the value analysis, Frama-C outputs that the *pin_input* variable is in $[0, 9999]$. This is because each ACSL assert annotation is first verified and, even for a unknown status, is supposed to be true for the next steps of the verification process. Thus, the important output of Listing 9 is the assertion validation part that gives the actual status of the assertions.

4.3 Limitations

As shown in the Listing 9 representing the output verification of the code example of Listing 8, some properties have not been verified by Frama-C. This is due to two reasons.

First, the analyzed code is partial as the code of some functions like the *scanp* function is not given to the analysis tool. Even if the source code of the *scanp* function was given to the analyzer, it could not determine any valuable information as the function gets the input numbers from the terminal. Moreover, for large smart card applications it is a fastidious work to give the whole source code to the analyzer. The source code includes non-ANSI C calls, pieces of assembly codes that disturb the analyzer. Patching the source code to get it usable for the analysis tool is an impossible work.

Second, Frama-C can be confused by complex C code, for example if C pointers are involved in the analysis. This second issue is not solved in this paper, as it deals with the limitations of semantical analyzers.

Nevertheless, a proposal is given in the next section to deal with the analysis of partial source code for two particular problems: the global variables that may be unknown or changed by external function calls; the call to an unknown function that defines a local variable. *scanp* is a typical example of such a function.

4.4 Instrumentation for source code analysis

In this section, a technical solution is proposed to help the analysis tool to deal with global variables and external function calls that may disturb the analysis. The solution is based on code instrumentation in order to observe unknown global variables and results returned by the functions during one execution of the code. The collected information will then be reused in order to help the analysis.

```
/* AUDIT BEGIN */
char * buf_1 = ( char * ) malloc (20 * sizeof ( char ) ) ;
sprintf ( buf_1 , "%i" , number1 ) ;
char temp2_1 [ 400 ] = "@10 : number1 = " ;
strcat ( temp2_1 , buf_1 ) ;
fprintf ( dump , "%s\n" , temp2_1 ) ;
/* AUDIT END */
if (number1 > 10)
```

Listing 10: Patch for global variable auditing

A Java software has been developed to analyze a C file and to determine which instructions should be audited. Then, the program patches the original code in order to add code snippets in charge of producing a dump of the audited values in a log file during the execution. In a third step, the dumped values are collected and transformed into ACSL assertions that are then injected as new assumptions for Frama-C.

4.4.1 Patched code

The Java program parses the lines of a C file and searches for all the declared global variables, since a global variable may have an impact on the sensitive variables that are considered. For example in Listing 8, the value of *number1* variable impacts the conditional and may trigger the *x++* instruction. Then, for each found global variable, the Java program searches for any line that uses this variable for a *read* operation (a formal presentation of *read* and *write* operations is given later in Section 5.2). Before each line that performs such a read on the global variable, a patch is added to dump the value of this variable in a log file. An example of the patch on variable *number1* is given in Listing 10.

A similar process is applied for functions that Frama-C cannot analyze. For example, for the function *scanp*, the Java program will get the name of the variable that receives the assignment from *scanp*. Then, a patch is added just after the *scanp* call to dump the value of the impacted variable. After an execution, the resulting dump file could be:

```
@22 : pin_input = 1234
@10 : number1 = 10
@14 : number1 = 10
@14 : number2 = 0
```

4.4.2 Injecting the new information

The collected information during the execution is read in the log file and new ACSL annotations are inserted at line numbers provided by the log file, as shown in Listing 11. For global variables, an *assert* annotation is added before the use of the variable. For unknown functions like *scanp*, an *assert* annotation is added just after the function call for the variable that has been assigned. Note that in case of multiple executions that collect different values, the annotation can reflect it: */*@ assert number1 == 10 || number1 == 11; */*

4.4.3 Verification with the new information

Frama-C is called to re-check the new code with the added informations. The output is shown in Listing 12. The assert annotations that correspond to the injected new information may have an unknown or valid status (lines 23, 10, 14) but then become new hypothesis for the semantic analyzer. Thus, valid status is obtained for all security properties (lines 24, 13, 29).

```
#include <stdio.h>                                                    1
typedef unsigned int word;                                           2
word number1; /* 2 global variables initialized elsewhere */        3
word number2;                                                        4
extern word scanp(); /* PIN code input prompt */                     5
                                                                     6
word work() { /* Work when authenticated */                         7
  word result;                                                       8
  word x = 0;                                                        9
  /*@ assert number1 == 10; */ /* AUDIT INSERT */                   10
  if (number1 > 10)                                                 10
    x++;                                                            11
  /* Check property 2 */                                            12
  /*@ assert 0 <= x <= 1; */                                        13
  /*@ assert number1 == 10; */ /* AUDIT INSERT */                   14
  /*@ assert number2 == 0; */ /* AUDIT INSERT */                    14
  result = number1 + number2 + x;                                   14
  return result;                                                    15
}                                                                   16
                                                                    17
void banking() { /* Main code for the banking software */          18
  word res;                                                         19
  word pin_input;                                                   20
                                                                    21
  pin_input = scanp();                                              22
  /*@ assert pin_input == 1234; */ /* AUDIT INSERT */               23
  /* Check property 1 */                                            23
  /*@ assert 0 <= pin_input <= 9999; */                             24
  if (pin_input == 1234) {                                          25
    res = work();                                                   26
  }                                                                 27
  /* Check property 3 */                                            28
  /*@ assert 0 <= res <= 100; */                                    29
}                                                                   30
```

Listing 11: ACSL injected instrumentation results

4.4.4 *Verification including an attack*

Let us consider a physical attack against the sensitive variable x just before the line 14 that computes the result of the function. The injected attack could be: $x = 255$. The obtained output from Frama-C clearly states that the status of the third property is invalid, as shown in Listing 13.

5. VERIFICATION REDUCTION

In this section, an attack is a C code that simulates the attack. The difficulty is that this may produce as many injected codes as the number of possible attacks, even worse if considering the different attack durations. Nevertheless, the model of attack proposed in Section 3 represents an attack as a block of C code that is activated for a given duration using boolean variables: injecting such a code at a given line in a loop covers all the possible temporal ranges of attacks. So,

```
banking−input3.c:23:[value] Assertion got status unknown.
banking−input3.c:24:[value] Assertion got status valid.
banking−input3.c:10:[value] Assertion got status unknown.
banking−input3.c:13:[value] Assertion got status valid.
banking−input3.c:14:[value] Assertion got status valid.
banking−input3.c:14:[value] Assertion got status unknown.
banking−input3.c:29:[value] Assertion got status valid.
[value] ====== VALUES COMPUTED ======
[value] Values for function work:
            result ∈ {10; }
            x ∈ {0; }
[value] Values for function banking:
            res ∈ {10; }
            pin_input ∈ {1234; }
```

Listing 12: Frama-C analysis with new information

```
banking−input3.c:23:[value] Assertion got status unknown.
banking−input3.c:24:[value] Assertion got status valid.
banking−input3.c:10:[value] Assertion got status unknown.
banking−input3.c:13:[value] Assertion got status valid.
banking−input3.c:14:[value] Assertion got status valid.
banking−input3.c:14:[value] Assertion got status unknown.
banking−input3.c:29:[value] Assertion got status invalid (stopping
      propagation)..
banking−input3.c:43:[kernel] warning: non termination detected in function
      banking
[value] ====== VALUES COMPUTED ======
[value] Values for function work:
            result ∈ {265; }
            x ∈ {255; }
[value] Values for function banking:
            NON TERMINATING FUNCTION
```

Listing 13: Frama-C analysis with a simulated attack

the number of possible places for attacking is proportional to the size of the program.

However, injecting one simulated attack produces a lot of source codes to be verified. Even worse, the verification process described in Section 4.2 induces many sensitive variables to attack in order to simulate all the possible consequences. In this section, we show that the number of codes to check can be reduced. Two attacked codes can be considered equivalent when the effect of both injections gives the same consequences in the executed code. For example, using the function *work()* of Listing 8, the injection of an attack against *number2* has the same effect from line 8 to line 14. On the contrary, the injection of an attack against *number1* impacts the condition at line 10. Thus, in this example two classes of equivalence for the attack of *number1* should be considered: the attacked codes where the attack occurs before line 10 and the attacked codes where the attack occurs after line 10.

In the following, using notions of equivalent codes and of coverage of an equivalence class by another, the minimum set of attacked codes is computed. This set is then provided to the verification step and the output guarantees the coverage of all the possible attacks. The definitions and algorithms given in this section rely on the determination of all read and write operations on a sensitive variable in the considered application. This corresponds to the use-def data flow and related dependencies analysis needed by compilers to perform some code optimizations [11].

5.1 Considered attack

In the rest of the paper we assume only one sensitive variable and consider only memory attacks modifying x. Basically, such an attack code α is $x = value;$. Note that to cover all temporal cases, α is enclosed in a C-block guarded with boolean variables as explained in Section 3.

DEFINITION 2. *$Attack(code, \alpha, i, x)$ denoted $code_{\alpha,i}$ is a copy of the original code where the attack instruction α targeting x is injected between line $i-1$ and i.*

We consider that a source code of size n contains as many C instructions as number of lines, that is n. So, the source code resulting from the injection of an attack using the function *Attack* has $n+1$ lines. This changes the source code line numbering which may confuse the explanations in next sections. So, we consider that the injected attack is inserted at line $i-\frac{1}{2}$ and does not impact the source code numbering.

Sensitive variable: x	Local	Global
y = f(x);	read	rw
x = f(y);	write	rw
proc(&x);	rw	rw
proc(y);	∅	rw

Table 1: Type rules for function calls

5.2 Code and types

Considering a code and a variable x, we define the type *read* or *write* of a source code line regarding what is performed on x. In some cases, a C line can perform both operations, like $x{+}{+}$ that will be typed as rw.

DEFINITION 3. *Let $Type(code, x, i)$ be defined as follows: the line i of the code is typed by read or write or rw or $\emptyset$ for a variable x. The read (resp. write) type means that the considered code line reads (resp. write) the x variable. The $\emptyset$ type means that no operation is performed on x.*

In the sequel, we restrict the analysis of the code to the lines that define a function f, noted $code_f$. The analysis is local to a block function and the goal is to determine the next use or definition of a variable, after an attack has been performed. As we do not want or may not be able to analyze the whole source code, the analysis focuses on a single block function, and does not enter function calls.

However, we need to type any function call with a sensitive variable as parameter. Table 1 gives the typing rules for such function calls. An interprocedural analysis could refine the types of function calls. For example, depending on the body of f, the call to proc(&x) could be type as *read* or *write*, or possibly $\emptyset$. Table 1 is only a safe over-approximation of the type for function calls that is useful for the case where the code is not available or difficult to analyze.

DEFINITION 4. *Let $TypeAfter(code_f, x, i)$ be the next found typed line using x, defined recursively as follows:*

$$\begin{cases} TypeAfter(code_f, x, last) = \emptyset \\ TypeAfter(code_f, x, i) = \\ \quad \begin{cases} Type(code_f, x, i) \text{ if } Type(code_f, x, i) \neq \emptyset \\ TypeAfter(code_f, x, i+1) \text{ otherwise} \end{cases} \end{cases}$$

where last is the last line of the block function $code_f$.

$TypeAfter(code_f, x, i)$ returns no type if the function does not use x after line i, including. If a line of the block code of the function uses x after line i (or can possibly affect x in one of the cases of Table 1), $TypeAfter(code_f, x, i)$ returns the type of the next use of x.

DEFINITION 5. *Let $LineTypeAfter(code_f, x, i)$ be the number of the line that defines $TypeAfter(code_f, x, i)$.*

This function will be useful to find the line that is just before the next use of variable x, or the last line of the block function if x is not used after line i. A code example is given in Listing 14 that illustrates the both previously defined functions.

5.3 Equivalence classes

In this section, we define the notion of equivalence classes. We focus on a source code with sequential control flow, i.e. the considered source code does not contain any loop nor conditional: the code can be seen as a linear sequence of C instructions as assignments, function calls and variable manipulations. For such a code, we show that the equivalence classes correspond to the minimal number of code to verify to ensure a security property. The general case for a code with conditionals and loops is studied in Section 5.4.

```
word f() {                  // TypeAfter(x,1)=w                    1
  word result;              // Type(x,2)=∅                         2
  word y = 0;                                                      3
  word x = 0;               // Type(x,4)=w                         4
  result  = 0;              // TypeAfter(x,5)=rw                   5
  x = x + 1;                                                       6
  result  = result  + y;    // TypeAfter(x,7)=r                    7
  y−−;                                                             8
  send(x);                                                         9
  y−−;                      // LineTypeAfter(x,10)=12              10
  result  = 2 ∗ y;                                                 11
  result  = result  + x;                                           12
  y = 0;                    // TypeAfter(x,13)=∅                   13
  return result ;           // LineTypeAfter(x,14)=15              14
}                                                                  15
```

Listing 14: Code example illustrating type functions

DEFINITION 6. *$Attack(code_f, \alpha, i, x) \equiv Attack(code_f, \beta, j, x)$ iff the values of x are equal between the code $Attack(code_f, \alpha, i, x)$ and the code $Attack(code_f, \beta, j, x)$ for each read of x during the execution.*

Note that this equivalence relation can be extended to the initial code. In this case, the equivalence class corresponds to all the attacked codes for which the attack is useless.

LEMMA 1. *If $TypeAfter(code_f, x, i) \in \{write, \emptyset\}$ then $Attack(code_f, \alpha, i, x) \equiv code_f$.*

PROOF. In the first case, $TypeAfter(code_f, x, i) = write$. The next operation on x after line i is of type *write*, say line j. Thus, it does not exist any *read* operation between lines i and j. Any attack that modifies x in this interval is useless because x is overwritten at line j. In the second case, $TypeAfter(code_f, x, i) = \emptyset$ means that there is no *read* nor *write* operation in the remaining operations. Thus the attack has no effect. □

For example, using Listing 14, Lemma 1 implies that injecting an attack against x at line 1 (before the execution of the first instruction of the function) or at line 5 has no effect on the code: these attacked codes are equivalent to the original code.

LEMMA 2. *If $TypeAfter(code_f, x, i) \in \{read, rw\}$ then, $Attack(code_f, \alpha, i, x) \equiv Attack(code_f, \alpha, LineTypeAfter(code_f, x, i), x)$.*

PROOF. After line i, the next operation concerning x is of type *read*, at line $j = LineTypeAfter(code_f, x, i)$. Any attack that modifies x in the interval $[i, j]$ has the same consequence on x at line j but has no consequence between lines i and $j-1$. Thus, all these attacked codes are equivalent to the code where the attack is performed at line j. □

For example, using Listing 14, Lemma 2 implies that injecting an attack against x at line 7 is equivalent to injecting the attack at line 9. These two lemmas highlight the important attacks that have consequences and those that have not. The main idea is to only study the attacks that have consequences and to reduce them to the minimum.

THEOREM 1. *If there are n read and read-write operations on x in $code_f$ then the minimal number of equivalence classes is $n+1$: $code_f \cup \{Attack(code_f, \alpha, i, x), \forall i \ s.t. \ Type(code, i, x) \in \{read, rw\}\}$*

Indeed, there are $n+1$ codes to be verified that are the initial $code_f$ plus the attacked codes where an attack has been injected at the read or read-write operation places.

To divide into equivalent classes a code function that contains loops and tests, the next section has to reconsider the $TypeAfter$ and $LineTypeAfter$ functions that have to take into account the fact that each line may not be executed only once and in the order of the line numbering.

5.4 Conditionals and loops

In the previous section, we have considered only source codes without loops and without conditionals. We could redefine functions $TypeAfter$ and $LineTypeAfter$ to divide a code function with loops and conditionals into regions where attacks are equivalent. However, this would give a large number of equivalence classes and the effects of attacks of some classes are included into the effects of attacks from another class from the point of view of the verification goal. As we aim at reducing the number of injected codes to be verified, we propose to use the notion of attack coverage, defined in this section and illustrated below:

```
...                                                                    1
if (cond){                                                             2
                        attack1  ∈ attack2                             3
                  but   attack1  ∉ attack2                             4
z = x;            so    attack1  ≠ attack2                             5
  ← attack1                                                            6
}                 equivalent  regions : {{1,2},{3,4,5},{6,7},{8,9}}    7
  ← attack2                                                            8
y = x;            attack2 covers attacks of regions {6,7},{8,9}        9
```

DEFINITION 7. *$Attack(code_f, \alpha, i, x) \in Attack(code_f, \beta, j, x)$ iff the effects of $Attack(code_f, \alpha, i, x)$ on x values are included in effects of $Attack(code_f, \alpha, j, x)$ on x values. In the remainder, for two such attacks, we say equivalently that $Attack(code_f, \alpha, i, x)$ is covered by or included in $Attack(code_f, \beta, j, x)$.*

In the following, we generalize the $TypeAfter$ function for generic parts of code. We also redefine the $LineTypeAfter$ function in order to deal with the coverage notion in case of more generic portions of code. $LineTypeAfter(code, i, x)$ returns a line number such that:

$$Attack(code_f, \alpha, i, x) \in Attack(code_f, \beta, LTA(code, i, x), x)$$

Both functions are extended to any C blocks i.e. between balanced braces. We assume that the numbering of the lines is global to the whole source code and that $first$ and $last$ are two functions on a C block that respectively provide the line numbers of the first and last line of the considered block.

5.4.1 Conditionals

This section deals with the conditional instructions, namely the general `if-then-else` structure. A conditional can be decomposed in three distinct parts: the condition `cond`, the `then-block` and the `else-block`. This latter block might be empty. Let *block−then(block, i)* be the function that returns the "then" block if an *if* instruction is located at line number i or if i belongs to this "then" block. Otherwise, NULL is returned. Let *block−else* and *block−if* be defined in the same way. Let *cond* be the function that return the condition of a conditional block `if-block`.

```
Type TypeAfter(block, x, i) {                                          1
  if (i== last(block))                                                 2
    return ∅;                                                          3
  if (Type(block, x, i) != ∅)                                          4
    return Type(block, x, i);                                          5
  Block bif = block−if(block, i);                                      6
  if (bif != NULL) { /* i ∈ bif */                                     7
    Block bthen = block−then(block, i);                                8
    Type TypeThen = TypeAfter(bthen, x, first(bthen));                 9
    Block belse = block−else(block, i);                                10
    Type TypeElse = TypeAfter(belse, x, first(belse));                 11
    if (i == line(cond(bif)){ /* line of the condition of bif */       12
      if (Type(block,x,line(cond(bif))) != ∅)                          13
          return Type(block,x,line(cond(bif)));                        14
      if (TypeThen == ∅ && TypeElse == ∅)                              15
          return TypeAfter(block, x, last(belse)+1);                   16
      return TypeThen ∪ TypeElse;                                      17
    }                                                                  18
    else if (i < last(bthen)                                           19
            if (TypeThen == ∅)                                         20
            return TypeAfter(block, x, last(belse)+1);                 21
            else return TypeThen                                       22
    else if (TypeElse == ∅)                                            23
            return TypeAfter(block, x, last(belse)+1);                 24
          else return TypeElse;                                        25
                                                                       26
  return TypeAfter(block, x, i+1);                                     27
}                                                                      28
```

Listing 15: TA function for conditionals

With this new definition[2], Lemma 1 remains valid.

PROOF. If $TypeAfter$ returns $\emptyset$ it means that code analysis has crossed the *if* blocks without finding any *read* or *write* operation. If the $TypeAfter$ function returns *write*, it means that a *write* operation has been found in or after one of the *if* blocks. If the *write* operation has been found after the *if* blocks, then the operation will cancel any variable modification attack. If the *write* operation has been found in, for example, the `then-block`, an attack could be canceled by the *write* operation or propagated by the `else-block`. In this last case, this attack will be considered in the equivalence class of the attacks injected after the *if* block. □

Two *read* operations can occur in both `then-block` and `else-block`. To cover all attacks impacting these reads, instead of injecting one attack at both corresponding lines, we can inject the attacks before the *if* block when no operation on x are performed between the condition of the *if* and both these reads. In other cases, the place to inject an attack is the line where the read is performed.

DEFINITION 8. *Let $LineTypeAfter(code, x, i)$, equivalently denoted $LTA(x, i)$ in the rest for sake of space and readability, defined as Listing 16 for lines i such that $TypeAfter(code, x, i) \in \{read, rw\}$.*

An *if* code example with the values of $TypeAfter$ (TA) and $LineTypeAfter$ (LTA) is given in Table 2. This example shows a factorization for the first read on x in both branches of the conditional. For the other cases, the value of LTA is the number of the line where a read is performed.

5.4.2 Loops

The following listings illustrate the two cases to distinguish when code contains a loop and show the lines where

[2]Note that if there is no `else-block` then *last (belse)* return the value of *last (bthen)*.

```
if (there is one line L determining this type)                                  1
   if (L and i ∈ a same else−block or then−bloc of a conditional                2
         bif and TypeAfter(bif,first(bif),x)∈{read, rw} and there are
          two lines defining TypeAfter(bif,first(bif),x), L included)
      LTA(x,i) = first(bif)                                                     3
   else                                                                         4
      LTA(x,i) = L                                                              5
if (there is two lines L and L' determining this type)                          6
  if (Type(block,x,L)∈{read,rw} && Type(block,x,L')∈{read,rw})                  7
     LTA(x,i) = first(cond(block−if(block,L))                                   8
  else if (Type(block,x,L) ∈ {read,rw})                                         9
     LTA(x,i) = L                                                               10
  else                                                                          11
     LTA(x,i) = L'                                                              12
```

Listing 16: Definition of LineTypeAfter

L	Code	TA(x)	LTA(x)	TA(y)	LTA(y)	TA(z)	LTA(z)
1	word f()	r	3	r	3	rw	6
2	{	r	3	r	3	rw	6
3	if (y == 0) {	r	3	r	3	rw	6
4	y = 48;	r	3	w	×	rw	6
5	y = x;	r	3	w	×	rw	6
6	z = x + z;	r	6	∅	×	rw	6
7	} else {	r	3	w	×	∅	∅
8	y = x + 1;	r	3	w	×	∅	∅
9	}	∅	×	∅	×	∅	∅
10	}	∅	×	∅	×	∅	∅

Table 2: Example of LTA values with a conditional

attacks must be injected to cover all the possible attacks for both cases. Only *while* loops are considered as others can be rewritten using a *while*.

```
← Attack point
while ( ... x ... )
{
    ...
← Attack point
}
```

```
while ( .... ) {
    ...
← Attack point
    ... x ...
    ...
}
```

To include loops in the previous definition of $TypeAfter$ the line 27 of Listing 15 must be patched with Listing 17.

```
Block bwhile = block_while(block, i);                                           1
if (bwhile != NULL) // i is in a loop                                           2
  // does x occur until the end of the loop ?                                   3
  Type TypeEndLoop = TypeAfter(bwhile, x, i);                                   4
  if (TypeEndLoop != 0)                                                         5
    return TypeEndLoop;                                                         6
  // does x occur in the condition of the loop ?                                7
  Type TypeCondLoop = Type(bwhile, x, line(cond(bwhile)));                      8
  if (TypeCondLoop != 0)                                                        9
    return TypeCondLoop;                                                        10
  // does x occur from the beginning of/after the loop ?                        11
  Type TypeBeginLoop = TypeAfter(bwhile, x, first(bwhile)+1);                   12
  Type TypeAfterLoop = TypeAfter(block, x, last(bwhile)+1);                     13
  return TypeBeginLoop ∪ TypeAfterLoop;                                         14
else                                                                            15
  return TypeAfter(block,x,i+1);                                                16
```

Listing 17: Added code for the definition of TA

With this modified definition, Lemma 1 remains valid.

PROOF. Since the condition is evaluated after each internal parts of a while block, any attack at line i inside such a while may impact two distinct parts of the code: first, the part right after i, then the part from the while condition to i. If the TA function returns $\emptyset$, any attack has no impact on the rest of the code. If the TA function returns *write*, an injected attack is canceled by a *write* operation in the first part of the loop or after the while. □

Finally, the definition of $LineTypeAfter$ becomes:

```
if (there is one line L determining this type)                                  1
   if (L and i ∈ a same else−block or then−bloc of a conditional                2
         bif and TypeAfter(bif,first(bif),x)∈{read, rw} and there are
          two lines defining TypeAfter(bif,first(bif),x), L included)
      LTA(x,i) = first(bif)                                                     3
   else if (i∈bwhile and L∈bwhile)                                              4
      if (i == first(bwhile) && i==L)                                           5
         LTA(x,i) = {L, last(bwhile)}                                           6
      else if (L == first(bwhile))                                              7
         LTA(x,i) = last(bwhile);                                               8
      else LTA(x,i) = L                                                         9
   else LTA(x,i) = L                                                            10
if (there is two lines L and L' determining this type)                          11
  if (Type(block,x,L) and Type(block,x,L')∈{read,rw})                           12
    if (block−if(block,L) != NULL && block−if(block,L) ==                       13
          block−if(block,L'))
      LTA(x,i) = first(cond(block−if(block,L))                                  14
    else                                                                        15
      LTA(x,i) = {L,L'}                                                         16
  else if (Type(block,x,L)∈{read,rw})                                           17
    LTA(x,i) = L                                                                18
  else                                                                          19
    LTA(x,i) = L'                                                               20
```

Listing 18: Final definition of LineTypeAfter

5.5 Attacks coverage

The previous two sections have defined $TypeAfter$ and $LineTypeAfter$ to divide a code function into areas where attacks are covered by an attack injected at places defined by the LTA function. Then, to simulate all the possible attacks it is then only needed to consider the codes where an attack is injected at the place defined by LTA.

THEOREM 2. *The reduced set of attacked codes that covers all the possible attacks against variable x is the initial $code_f$ plus the attacked codes where an attack is injected at the place defined by the LTA function:*

$$code_f \cup \bigcup_{j \in |program|} Attack(code_f, \alpha, LTA(code_f, x, j), x)$$

Let n be the number of sensitive-read of variable x in the code. Among those n sensitive-reads, the wl sensitive-reads which are included in the condition of a while loop require to generate $2wl$ injected codes. Also, the ite full if-then-else blocks that have a sensitive-read as first operation on x in both branches and no operation on x in the if-condition represent $2ite$ sensitive-reads but require only ite places to inject attacks. Using these notations, the number of injected codes to be generated and to be given to the verification tool in addition to the initial code is equal to: $(n - 2ite - wl) + 2wl + ite$ i.e. $n - ite + wl$.

5.6 Experiments

The reduction proposed in Section 5 have been implemented in a Java prototype. This prototype takes a C file and detects automatically all variables. Then, it detects all the read operations on these variables and computes the lines pointed by LTA that will receive the attacks. Then for each attacked line, the software generates a new code. For the code example of Listing 7 and the considered variables x, *pin_input* and *res*, it will create:

```
/home/jf/smart_card/pin_input/attack_line_21.c
/home/jf/smart_card/x/attack_line_11.c
/home/jf/smart_card/x/attack_line_12.c
... (and so on for number1, number2...)
```

In the next paragraphs, we show the effect of the reduction on the example of Listing 7. There are 3 sensitive variables to consider: *pin_input*, x and *res*. Without the reduction provided by Theorem 2, the number of codes to generate (the attack space) is roughly the number of sensitive variables multplied by the number of lines of the considered function, that is 6 (x can be attacked inside work() on 6 lines) + $2 * 5$ (*res* and *pin_input* in banking()) = 16 codes.

5.6.1 Reduction for pin_input

If we apply the reduction for variable *pin_input*, the line 21 is tagged as *read* by the TA function. So, for the injection of attack, only the line 21 is concerned because $TA(i \in [16..20]) = write$, $TA(i \geq 22) = \emptyset$, $TA(i \in [7..15]) = \emptyset$). Thus, applying the reduction for *pin_input* introduces only 1 attack at line 21.

5.6.2 Reduction for x

If we apply the reduction for variable x, the lines 10 and 11 are tagged as rw by the TA function and the line 12 is tagged as *read*. For the injection of attacks, two lines are concerned because $\forall i \in [10..11], LTA(i) = 11$ and for $i = 12$, $LTA(i) = 12$ (similarly as for pin_input the other lines have the type $\emptyset$ or *write*). Thus, applying the reduction for x introduces two attacks at lines 11 and 12.

5.6.3 Reduction for res

In this example, *res* is never read and no attack can have an effect on *res*. This is clearly a pedagogical example because *res* is not used. In practice it would be tested, returned or be a parameter of a function call. In these cases, *res* would be read implying a possible attack.

5.6.4 Reduction result

As a conclusion, applying the methodology for *pin_input*, x and *res* reduces the number of attacks to 3: one attack against *pin_input* at line 21; two attacks against x at lines 11 and 12; no possible attack against *res*.

6. CONCLUSION AND PERSPECTIVES

In this paper, we have proposed a methodology to verify interval properties on smart card codes based on a source-level model for attacks and on the enumeration of a reduced but safe number of injected codes to cover all possible attacks for such properties.

This work is a first step to a global verification solution for smart card codes. The whole set of security property types needs to be investigated and modeled. Each type of a security property may be threatened by one type of the presented attacks.

The automation of the attack injections has to be developed to additionally cover code attacks. The difficult challenge is then to develop verification methodologies in order to check any security property type under the influence of one of the simulated attacks.

Another key point is to be able to analyze real smart card codes. Again this paper is a first step towards a global solution of instrumentation in order to help in the verification process. Issues related to pointers and arrays still remain to be solved as the static analyzer may not be able to infer enough precise information for sensitive variables and their dependencies.

7. ACKNOWLEDGMENTS

This work was supported by engineering students in computer science for their master degree from ENSI de Bourges, France: F. Assoudi, F.-Z. Bouam, V. Dumand, M. Ougier. The authors would also like to thank F. Chamberot from Oberthur Technologies.

8. REFERENCES

[1] Smartcard Integrated Circuit Protection Profile, September 1998. PP/9806.

[2] R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In The second USENIX Workshop on Electronic Commerce, pages 1–11, Oakland, California, November 1996.

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE, 94(2):370–382, February 2006.

[4] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C specification language. Technical report, CEA LIST and INRIA, 2009-2010. Preliminary Design (v 1.4).

[5] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In Advances in Cryptology - CRYPTO '97, volume 1294 of Lecture Notes in Computer Science, pages 513–525, Santa Barbara, California, USA, August 1997. Springer Berlin / Heidelberg.

[6] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In Advances in Cryptology - EUROCRYPT '97, volume 1233 of Lecture Notes in Computer Science, pages 37–51, Konstanz, Germany, January 1997. Springer Berlin / Heidelberg.

[7] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. Frama-C User Manual. CEA LIST, 2010.

[8] P. Cuoq and V. Prevosto. Frama-C's value analysis plug-in. Technical report, CEA LIST, 2010.

[9] E. Hess, N. Janssen, B. Meyer, and T. Schütze. Information leakage attacks against smart card implementations of cryptographic algorithms and countermeasures – a survey. In Eurosmart Security Conference, pages 55–64, Marseilles, France, June 2000.

[10] O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In USENIX Workshop on Smartcard Technology, pages 9–20, Chicago, Illinois, USA, May 1999.

[11] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1998.

[12] W. Rankl and W. Effing. Smart Card Handbook. John Wiley And Sons, 2003.

[13] B. Schneier. Attack trees. Dr. Dobb's journal, 24(12):21–29, 1999.

[14] P. Teuwen. How to Make Smartcards Resistant to Hackers' Lightsabers? In J. Guajardo and B. Preneel and A.-R. Sadeghi and P. Tuyls, editor, Foundations for Forgery-Resilient Cryptographic Hardware, pages 1–8, Dagstuhl, 2010.