Deliverable D1.2.1

Specification of the Use Cases, first release

Editor	Jean-Louis Lanet
Authors	by alphabetical order:
	Thierno Barry, CEA
	Damien Couroussé, CEA
	Philippe Jaillon, ENSMSE
	Jean-Louis Lanet, XLIM
	Bruno Robisson, ENSMSE

Version	Revision: 159
Date	lun., 13 oct. 2014 09:09:52 +0200
CEA ref.	V13DACLE014 – 14-0575

Contents

1 Executive Summary **3**

2 Cryptographic Algorithm: AES **4**

2.1 Description 4

2.2 Use of code polymorphism 5

3 Secure Algorithm: PIN Code verification **6**

3.1 Motivation 6

3.2 Description 7

3.3 Use of code polymorphism 7

4 General Purpose Algorithm: the JVM **9**

4.1 Motivation 9

4.2 Description 9

4.3 Use of code polymorphism 11

5 References **12**

List of Figures

1 An implementation example of the secure algorithm in C 6

1 Executive Summary

This report focuses on the different use cases of the COGITO project. We have chosen different algorithms that have different characteristics in terms of usage frequency, security level and side channel signature. The first one is the cryptographic algorithm AES; its side channel signature is of prime importance while its usage is moderate but could be complex to implement. The second one, the PIN code verification algorithm, is rarely used; its side channel signature is very important and it has a very low complexity. The last one, the memory prefetch in a Java Virtual Machine, is very frequently used, presents a very low complexity but is rarely protected against side channel and security aspect is lower.

The case studies within the COGITO project contribute to the gathering of evidence for the benefits of using code polymorphism. The results are aimed at raising awareness of the benefits of using the `deGoal` technology in the smart card domain. The objective is to evaluate the advantages and drawbacks of using code polymorphism in different phases of the smart card development. For that purpose, beside the description of the algorithms we describe the evaluation process and the related metrics we want to collect in order to asses if this technology is affordable for this application domain.

This deliverable is intended as the first release of the Use Case report. We plan to have a second release of this deliverable.

2 Cryptographic Algorithm: AES

2.1 Description

AES has become a kind of benchmark reference when it comes to evaluate the effectiveness of a generic protection scheme.

The AES cipher executes a number of round transformations on the input plain-text, where the output of each round is the input of the next one. The number of round r is determined by the initial key length: a 128-bit key uses 10 rounds, a 192-bit uses 12 and 256-bit key uses 14. Overall the implementation of AES is achieved with only three types of operations: `xor` operation, table lookups and 1-byte shifts. Each step operates on 128-bit blocks of data viewed as a 4×4 matrices of bytes.

```
input  : A plaintext $p$ of length 128 bits
input  : A key $k$ of length 128 bits
output: A ciphertext $c$ of length 128 bits
$\langle k_0, k_1, \ldots, k_r \rangle \leftarrow$ KeySchedule($k$);
$c \leftarrow$ AddRoundKey ($p, k_0$);
for $i \leftarrow 1$ to $r$ do
    $c \leftarrow$ SubBytes ($c$);
    $c \leftarrow$ ShiftRows ($c$);
    $c \leftarrow$ MixColumns ($c$);
    $c \leftarrow$ AddRoundKey ($c$);
end
$c \leftarrow$ SubBytes ($c$);
$c \leftarrow$ ShiftRows ($c$);
$c \leftarrow$ AddRoundKey ($c$);
```

Algorithm 1: The AES encryption algorithm

Algorithm 1 describes AES encryption. Each round but the latter is composed of four processing stages:

- `AddRoundKey`: this stage perform a bytes-addition in $\mathbb{F}_2$ between the plain-text and the round key meaning a byte-`xor` operation.
- `SubBytes`: each byte in the plain-text is substituted by another one from the look-up table (S-Box).
- `ShiftRows`: each row from $i = 0$ to $i = 3$ of the 4x4 *state* matrix is cyclically i-bytes left shifted
- `MixColumns`: the four bytes of each column of the *state* are combined using an invertible linear transformation.

Each round is composed of the same steps, except for the first that starts with an extra addition of a round key and the last where the `MixColumns` operation is skipped.

2.2 Use of code polymorphism

Each processing stage in AES can be provided with a polymorphic implementation. From the point of view of DPA attacks, the weakest points in AES are the execution of the first `SubBytes` and the last `AddRoundKey` [MOP07]. In this use case, we will consider the overhead cost of using code polymorphism for each of the sub-stages of AES with regards to security aspects. In particular, considering the flexibility of the COGITO protection scheme, it is possible to selectively apply code polymorphism on the code portions that are more likely to be under attacks. Otherwise said: on the one hand, a full polymorphic implementation is expected to provide the highest degree of security but with a higher overhead because of the execution time required for code generation. On the other hand, carefully selecting the portions of AES where code polymorphism is applied is likely to reduce the protection cost at the risk of providing lower security.

Several usage scenarii will be considered, considering a full polymorphic implementation of AES, or selective application of polymorphism. We will provide for each scenario figures for the security performance and for the protection overhead.

3 Secure Algorithm: PIN Code verification

3.1 Motivation

The PIN code verification procedure is one of the most attacked. Its purpose is to verify if the proposed PIN value is equal to the stored one. Many attacks have been elaborated to retrieve the code including attacks on the terminal itself [CC11] using terminal sensors. But the most current attack are timing attack against the verification algorithm. While the algorithm compares the proposed value byte by byte one should infer analysing the behaviour of the target on which byte the algorithm stops the verification. Hereafter is a naive verification algorithm.

At the beginning (line 8) the program checks if the user did not more than 3 trials before. Then it compares line 13 byte by byte if the stored value `pin` is equal or not to the proposed value `buffer`. If it does not match, the trial counter is decremented and the program memorizes this state and returns a negative answer.

```
 1  #define maxTries 3
 2  int triesLeft = maxTries;
 3
 4  boolean verify (short[] buffer, short ofs, short len)
 5  {
 6    // No comparison if PIN is blocked
 7    authenticated[0]= false;
 8    if (triesLeft < 0)
 9      return false ;
10    // Main comparison
11    for(short i=0; i < len; i++)
12    {
13      if (buffer[ofs+i] != pin[i])
14      {
15        triesLeft-- ;
16        return false ;
17      }
18    }
19    // Comparison is successful
20    triesLeft = maxTries ;
21    authenticated[0]= true ;
22    return true ;
23  }
```

Figure 1: An implementation example of the secure algorithm in C

Of course, the time needed by the algorithm to evaluate each digit increases with time and thus offers a side channel to the attacker who needs just to evaluate the response time of the algorithm to infer if the proposed digit was correct or not.

Such a basic attack on a naive implementation of the PIN verification algorithm is very easy to set up. For this reason smart card manufacturer have developed algorithms that are resistant to timing attacks but also fault attacks. A fault attack needs to be synchronized, so often the first step of this attack is to reverse the algorithm in order to have the know-how on the precise instant to fire the laser beam.

We propose with this use case to evaluate the usage of `deGoal` to generate code that will avoid this synchronization process. This algorithm is rarely used only once per session except while the card is under attack. The time needed to generate the code is not a constraint.

3.2 Description

The algorithm that we use for this project is an improved version of the naive C version proposed in the previous section. In particular, we have an integrity protection of the different fields (mainly redundancy with bit inversion), which are checked before each usage. Of course PIN trial incrementation before use is the basic coding rule. We implement also the secure conditional to detect transient fault. The secure condition test twice the same value; if the second test is different from the precedent without having modified any variable it implies that the environment (a fault attack) did it. We implement also step counters to avoid control flow transfer due to a fault. If the program counter is incremented by the environment (fault attack) it can jump anywhere and potentially bypass some security tests. For this reason we mark important step in the algorithm and we verify before returning that all steps have been executed. We also use constant time evaluation procedure to eliminate the timing attack and secure constants to prevent an attack targeting boolean value. A false value is obtained with all the bit of the cell to zero, and a true value is obtain if the contain of the cell is different of zero. Of course an external event can modify a false value easily with a true value. We transfer the sensitive fields into the RAM at the beginning of the algorithm which allows us to work only on the `C` stack. We ensure consistency of the different fields used in the algorithm to avoid DoS attacks.

Such an algorithm is highly resistant to several attacks. Normally we have to introduce random computations to desynchronize the traces. Code polymorphism will be used for that purpose but also to avoid reverse engineering of the algorithm taking care that the other protections are not affected by code polymorphism. We present hereafter the secure algorithm.

Each time an abnormal behaviour is detected a countermeasure must be taken. This part is out of the study but can be blocking the current application by modifying its life cycle or blocking completely the card (card is mute).

3.3 Use of code polymorphism

In the secure algorithm, the evaluation section, *i.e.* the main comparison, should not be polymorphic until we have the guarantee that the constant time evaluation is not altered. All the other parts of the algorithm can be impacted by polymorphism.

The evaluation process of the code polymorphism will take into account the ability to recognise patterns by side channel but also the level of randomisation brought by polymorphism. For that purpose, we will evaluate the EM traces with or without polymorphism in order to assess the benefits. The memory footprint and the code generation process will be also evaluated.

4 General Purpose Algorithm: the JVM

4.1 Motivation

Recently, side channel analysis has become of interest to be used for reverse engineering purposes (e.g. [VWG06], [OSS+]). Reverse engineering of software is primarily known as the process of discovering the source code from the software binaries or executables. It often involves detailed analysis of the program and uses many methods such as analysis through observing information exchange, disassembling or decompiling. There are many tools available on-line that provide all of these functions and that even combine them to acquire the source code. Reverse engineering of Java Card applets is much more difficult because the attacker does not have access to the binary files.

Power or EM analysis can be used to acquire parts of bytecode in order to be reverse engineered. Once the collection of execution traces have been recovered by one of the attack methods, it can be analysed. Then, various techniques may be constructed to affect its function on the card or reveal sensitive information. Of course there is no guarantee that the collection of power traces covers the whole software to analyse. The attacker has to exercise all the data input in order to generate different traces to obtain a high coverage of the traces.

The traces can then be analysed with correlation or pattern recognition. To mitigate such an attack, a solution is to change the pattern of each bytecode in order reduce the probability of recognition. The pattern can be changed in the time axis including random delay or in the form of the pattern thanks to code polymorphism.

4.2 Description

To successfully reverse engineer an unknown Java Card applet from a smart card, first, a dictionary of patterns must be set up using a reference card. The process of reverse engineering begins with the identification of bytecode instructions in the collected power traces. Since the reference card is programmable, it allows the attacker to run testing applets that repeat one known instruction or that repeat a small sequence of known instructions multiple times and then reveal a repeating pattern in the power trace. By comparing the individual parts of the power trace that represent one instruction to each other, a unique template that defines an instruction by its power trace can be constructed. A template is usually constructed as an average power trace of multiple measurements of the same instruction. Moreover using correlation analysis one can recognise the common part of each instruction.

In fact a virtual processor acts as a real one with the same sequence of prefetch, decode and execute cycle. In the JCVM interpretation loop, as described below, these sequences are clearly identifiable: the first part is the preamble, is to say the 'prefetch - decode cycle of a virtual processor, then the second part represents the execute cycle of bytecode, followed by a postamble that depends on the type of bytecode being executed.

```
while (true) {
  bc_item = NULL; /* Preambule */
  handler = bytecode_table[*vm_pc]; /*Prefetch + decode*/
  vm_pc++;
```

```
    bc_action = handler();          /* Execute */
    if (bc_action < 0) {
       if (!handle_excep())
         return false;
  }
    switch (bc_action)     {/* postambule */
      case 0:       continue;
      case ACTION_RETURN:
                         i = handle_return(init_frame);
                         if (i == RUN_RETURN)
                             return true;
                         go = (bool)(i != RETURN_FAIL);
                         break;
      case ACTION_INVOKE:
                         exec_method = (method_t *)bc_item;
                         break;
      case ACTION_NATIVE:
                         go = handle_native();
                         break;
      case ACTION_NEW:
                         go = handle_new();
                         break;
      case ACTION_THROW:
                         go = handle_throw((ref_t *)bc_item);
                         break;
     }
}
```

The handler is a function pointer defined as: `typedef int16 (*bc_handler)(void);`. The system table `bytecode_table` associates a bytecode value to a function pointers of type `bc_handler`:

```
const bc_handler bytecode_table[256] =
{
       BC_nop,                        /* 0                  */
       BC_iconst_0,                   /* 1      BC_aconst_null */
       BC_iconst_m1,                  /* 2                  */
       BC_iconst_0,                   /* 3                  */
       ...
       }
```

Each of these functions describes the behaviour of the instructions.

```
int16 BC_iconst_0(void)
{
       return _iconst(0);
}
```

When executing the preamble, the processor handles two parameters that are attached to the bytecode being read: (1) the index `vm_pc` (virtual machine program counter) in the table of bytecodes,

(2) the address of the function handler to be executed. These two parameters are exploitable by an attacker to recover information about the program under execution. `vm_pc` is the pointer to the array of bytecodes representing the method, so it is a pointer to an array of bytes. By using side channel analysis, the attacker can obtain numerous information: the variation in the preamble power curves indicates the value of `handler` or `bc_action`. The first one gives directly the instruction, the second gives the address of the instruction. Then, the execution of the bytecode provides a trace easily identifiable with pattern recognition analysis; the execution time is also highly characteristic of the bytecode value. So an attacker has in each bytecode execution four correlated information about the instruction to be recognised.

4.3 Use of code polymorphism

We will limit the use of polymorphism on the prefetch cycle only. The difficulty with this case study is the high frequency of the code execution. Indeed, considering that an attacker needs a very low number of traces to recover information about the program under execution, it would be desirable to change the form of a polymorphic prefetch at *each* execution. However, considering the overhead of code generation with regards to code execution of the polymorphic code generated, this solution is hardly acceptable in terms of performance. This issue can be a limit of our approach that we want to investigate. We have backup solutions if the ratio compilation time *vs.* execution time is too high.

5 References

[CC11] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *HotSec*, 2011.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer, 2007.

[OSS$^+$] David Oswald, Daehyun Strobel, Falk Schellenberg, Timo Kasper, and Christof Paar. When reverse-engineering meets side-channel analysis – digital lockpicking in practice.

[VWG06] Dennis Vermoen, Marc Witteman, and Georgi N. Gaydadjiev. Reverse engineering of java card applets using power analysis. Technical report, WISTP'2007, LNCS 4462, 2006.