Using Side Channel Information for Improving Data Partitioning Strategy to Test Smart Cards

Putt Benjamin (benjamin.putt@etu.unilim.fr)*
Putt Ethan (ethan.putt@etu.unilim.fr)*
Lanet Jean-Louis (jean-louis.lanet@unilim.fr) *

Abstract: The fuzzing approach is a fast and simple way to use generated or modified inputs to evaluate the resistance of an application. But it has several limitations due to the models precision which is used for the input generation: random and/or simple models cannot reach all of the states and significant values, but higher model precisions can result in a combinatorial explosion of test cases. In this paper, we propose a new method based on the combination of timing attacks with the use of fuzzing techniques to discover and classify the behavior of the system under test. This new technique aims to solve the test space explosion and to simplify the fuzzing process configuration. We will show the use of this approach to discover the behavior of an applet loaded in a Java Card.

Keywords: Security, Software testing, Fuzzing, Side Channel attacks, Java Card

1 Introduction

The development of most systems must verify that the implementation of their functionalities are correct, often thanks to testing. Security dedicated systems must also check that no vulnerability remains in the system before deployment. Such software needs to be tested with regard to these two aspects. The main goal of functional software testing is to find discrepancies between the actual behavior of the system's functions and the expected one described in the functional specification. Vulnerability testing aims to find a behavior that violates security policies. A security policy is a set of rules defining the acceptable and rejectable values of a system as its state changes through time. Policies define what is allowable or desirable in the system and hence the notion of computer vulnerability ultimately depends on the notion of policy.

Software vulnerability comprises of several forms: access control, state space and fuzzy. Access control [RD82] consists in defining through an access matrix a partition for a given state of authorized commands and unauthorized states. A state space vulnerability [BB96] consists in reaching an unauthorized state from valid state using a valid command. Fuzzy vulnerability has been introduced by Amoroso [Amo94] who defines a vulnerability as an event that allows a threat to potentially occur. Such a policy is often not specified and consists in checking that what is not specified should not occur. Thus, verifying that a non expected command should not trigger an event in the system needs to test all the non specified values of the input command. It moves the challenge from a test activity to a

* University of Limoges, Computer Science Department

proof activity (exhaustive search with the variable domain). The cost of the latter is often non affordable.

The issue is to generate test cases that evaluate the presence of hidden commands without exploring the entire state space. A solution has been proposed by [SFL13] but it requires to build a detailed model of the system. A less costly technique is the fuzzing, which tries to exercise different parts of the program. We propose in this paper to improve the data generation of a fuzzer by using leaked information from the processor. This allows to infer the behavior of the program even if the response provided does not leak information, and thus improve the data partitioning.

The rest of the paper is organized as followed in the first section we introduce the fuzzing technique and the next section the specific domain of the smart card. The third section introduces the smart card test environment. In a forth section, we present the side channel attacks and in particular timing attacks. Then, we develop our solution which is then evaluated. In the last section, we conclude this work.

2 Discovering Vulnerabilities with Fuzzers

Fuzzing is a testing technique based on the analysis of the software behavior when its inputs are fed with particular data (*e.g.* invalid or random data). The fuzzing presents an interesting approach to software testing and has attractive benefits: it is an automated or a semi-automated process and it is lighter than other testing methods. It was invented in 1988 by Barton Miller who realized that some Unix command line tools would crash when random inputs are given [TDM08, Tak09]. A fuzzer can be considered as a testing technique used to uncover a variety of issues like: coding errors; security vulnerabilities; Buffer Overflow and so on, using unexpected, malformed, random data as program inputs or unexpected commands in the case of stateful system.

- Data Generator: this module is responsible for generating the inputs that will be used to drive the System Under Test (SUT). The generated abstract tests are provided to the SUT by the delivery module. The Data Generator concentrates the power of the fuzzer tool.
- Delivery Mechanism: the delivery mechanism get the abstract tests from the data generator and adapts them to be sent to the SUT. This module is in charge of adapting the test cases to the target *i.e.* packing the protocol data into a transport protocol.
- Monitoring System: the monitoring system observes the behavior of the SUT as it executes the test cases and collects the output of the system. The log can then be analyzed either manually or automatically if the specification is precise enough.

2.1 Existing Techniques

Here we will only detail the approaches that do not rely on the source code. Fuzzing is an approach in the context of security assessment, for software testing in a black box testing approach. Testing applications with negative test cases must face the problem of too many cases to exercise. The concept of a fuzzer is that the input data of a command is tampered in order to test a target application. Fuzzing handles this by automating the testing, allowing the evaluator to focus on other tasks. Choosing randomly the data to send is often inefficient. To make testing more efficient, only certain boundaries or patterns may be used. There exists two main techniques for determining the data used through test cases: data generation and data mutation.

The mutation-based fuzzers are the simplest. They use an existing valid data session and apply several transformations before sending it to the System Under Test (SUT). These transformations are most of the time minor substitutions of the input stream with random data. The behavior of the SUT is logged and analyzed. This technique is almost exclusively used for testing network protocols and parsers because it is very easy to save a session and replay it on these softwares. This solution might be more appropriate when fuzzing very large or unknown protocols.

The generation-based fuzzers are smarter than mutation based ones. They generate the input data from the scratch based on the specification or a command format. They provide a set of tools for describing the software under test inputs, data and state machine. This means that a specification needs to be written for each piece of software to fuzz. This model allows the data generator to output specific data targeting the software and thus, enhance the test coverage.

The main challenge resides in the choice of input data that can reveal software errors. An intelligent fuzzer does not just randomly change fields to produce invalid data. It uses the knowledge about the specification or the format to produce data. Intelligence might also include operations such as calculating and appending cryptograms. The trend is to design generic fuzzing frameworks which provide basic components for building specific fuzzers targeting the software under test. These frameworks can be used to design a fuzzing process which combines both previous approaches, and therefore can be used to fuzz a wide range of software. Peach [Edd04] and Sulley [Ami04] are two open source examples of such frameworks. They provide a language to model inputs and states of a system and generate test data automatically. The main benefit with this fuzzing technique is the high ratio of automation to manual work. Each input data or field of a protocol is described in term of domain and the fuzzer has several heuristics to choose the value in the domain.

2.2 Limitations

Current fuzzing frameworks have three major issues. We describe them in this section. The first issue is inherent to the fuzzing technique: how to deal with the test case combinatorial explosion. Fuzzing process is not deterministic because it uses random test data. To gain more control over this process, several strategies are used to decrease the testing set. One of them will only generate some values around the input boundaries. However, these strategies have also the effect of limiting the test coverage. The other approach consists in partitioning the input domain of a function being tested and selecting test data from each partition.

Various methods for creating test partitions are discussed in the literature [RC85], [OB88] but they are based on the knowledge of the specification or the implementation. None of these methods are available when working in a black box context. It is likely to choose loose test generation strategies (i.e. wider bounds) nowadays, as the number of tests per second ratio is very high for common software. It increases the test coverage but also consequently the difficulty of results analysis. This work is cumbersome and most of the fuzzing frameworks lacks of tools to ease this process. There are some works in this domain to solve this issue by coupling the fuzzer process with an oracle [MPRB10] or by using several existing implementations of a specification to automatically classify the test results [YCER11].

To improve the test generation quality and enhance the fuzzing process, fuzzing frameworks use models to describe the software under test inputs. However this approach is really cumbersome when the software under test is somewhat complex. Moreover, the grammar used by the fuzzer is sometimes too limited for describing the inputs of the software under test as described in [GMICL11].

3 Testing Smart Card Application

Smart cards provide a tamper resistant environment and are widely deployed in sensitive applications like mobile phone identification (USIM), banking system *etc.* Smart cards have small computation and memory capacity, but these limited capabilities are provided in highly embedded and secure components. Nowadays most of the USIM cards are based on a Java Card Virtual Machine (JCVM). Java Card is a smart card that implements the standard Java Card 3.0 (Sun, 2010) in one of the two editions *Classic Edition* or *Connected Edition*. Such a smart card embeds a Virtual Machine (VM), which interprets application byte codes already romized with the operating system or downloaded after issuance. There are some works dedicated on testing smart cards using a model based approach. Model-based test case generations has turned out to be a viable alternative to hand-written tests for smart card applications, but it involves additional cost in order to construct the model and the test case specifications. Such a technique is dedicated for functional testing not for vulnerabilities search.

3.1 Java Card Application

An applet receives commands, makes computations and responds; the card acts as a server. The Application Protocol Data Unit (APDU) is used for the communication between a software running on a computer and an applet running on a smart card. An APDU is built by concatenating several bytes corresponding to each APDU field. In its simplest form, an APDU command takes one byte for the instruction class (CLA, which corresponds to P_0 in the algorithm), one byte for the instruction code (INS, *i.e;* P_1), one byte for the first parameter (P1, *i.e;* P_2) and one byte for the second parameter (P2, *i.e;* P_3) followed, if needed by data and finally by the number of expected data in the response APDU. This latter is even simpler: it takes a stream of bytes (which is the applet response) followed by two status bytes. Successful execution of a command can depend on the previous command history (adequate rights have been presented to perform the current command). Designing a Java Card applet must take into account the probability of an attack (mainly physical), leading the design more complex due to the redundancy needed for the computation or the verification of the internal data integrity. The fact that this technology is derived from the main stream Java implies that only a few dedicated tools are available. For example, there is no tool for computing the code coverage of test suites. Often tools have to be redesigned or adapted for this platform.

3.2 Fuzzing Java Card Application

Prior works have been done for applying fuzzing on smart cards and in particular on Java Card applications. One of the challenges for testing these type of platforms, is the adaptation of the delivery mechanism and monitoring system. The first mechanism is used to transmit test cases provided by the data generator to the card. The second mechanism is in charge of recording the actual response of the system under test for future analysis. But the most important issue to solve is related to the domain of the input parameters and the policy to choose a concrete value for each parameter.

In [BBKL11], the authors explain how they adapted the Peach framework for testing Java Card Web Server enabled smart cards and how they enhanced the Peach configuration file to infer automatically the oracle configuration from it. They have used their smart card dedicated fuzzer to study several smart card protocol in particular the BIP (Bearer Independent Protocol). In [Lan11], the author describes how he used the Sulley fuzzing framework to test several implementations of the EMV protocol running on different smart cards. He uses a reference implementation and an implementation under test and uses the first one as an oracle in case of discrepancies. In [Guy10] the author shows how simple it is to discover all the commands accepted by a smart card application. He proposes a naive solution based on using a combination of the two parameters (P1 and P2) of the command to improve the resistance against a fuzzer. As Barreaud demonstrated by fuzzing also these parameters, such a counter measure does not resist to any fuzzer. Alimi presented another fuzzer in his PhD [Ali12] with a new approach based on genetic algorithms to chose the input data. He applied the fuzzer on an EMV application, but did not solve the state problem.

All these fuzzers bring to the fore an inherent problem of the fuzzer the input data choice policy. None of these approaches have been able to overcome the parameters combinatorial explosion. Thus they are inefficient in detecting hidden commands.

4 Side Channel Attacks

Side channel attacks is a hot topic in the smart card security field. They are almost exclusively used to recover public cryptosystem secret (*i.e.* private key). In 1996, P. Kocher demonstrated that knowing the time needed to perform private key operations on RSA or Diffie-Hellman protocol may result in breaking a cryptosystem [Koc96]. In 1998, Dhem *et al.* [DKL$^+$98] implemented a timing attack able to obtain the 512-bit RSA key of a smart card chip in a couple of minutes. Brumley and Boneh [BB03] managed to attack the SSL protocol by measuring the time an OpenSSL server takes to respond to decryption queries. The attacker managed to extract a RSA private key stored in a server by using approximately a million queries in a couple of hours. Timing attacks use the time spent by a processor to perform some computation to infer some information about data being processed. A processor will take different amounts of time to process different inputs according to the control flow graph. The most common factors are the following:

- instructions are executed in different amount of processor cycle (*e.g.* multiplication and division will take more processor cycle than addition and subtraction),

- compiler optimization (for example, short-circuited conditional checking),

- cache hits or misses.

To mitigate timing attack, algorithms should be designed to run for a constant time regardless of the input data. But it is very hard to build a software which will respect this principle (due to compiler optimizations amongst other things). Moreover, on secure elements, the CPU does not have cache components which add noise to the measures. Recently smart card development, countermeasures have been integrated at different levels to prevent these attacks. But all these counter measures are only at the lower level (assembly language) not at the Java language level. As far as we know there is no platform implementing counter measures at the virtual machine level.

5 Enhancing the Data Generator with Data Partitioning

In order to fix the fuzzing issues, we have chosen to combine both previous techniques to design a fuzzer which uses a timing attack on a Java Card based. This combination of existing tools brings a new approach to software testing. We have developed in Java a proof of concept for testing an applet running on a Java Card platform. The input domain of each parameter is the cardinal of the type `byte`, so the number of tests is the combination of all these parameters. In our approach, we have two steps to set up the different partition before fuzzing. The complexity of each of the two phases is only the cardinal of the type `byte` which reduce drastically the complexity. Then the fuzzing step can occur which will perform the combinatorial permutation of each element in the partition. And at that step the complexity is the combination of the number of partition. The partition analysis using the side channel information is the major contribution of this method.

5.1 *Creating the partitions*

The goal of data partitioning is to make the input data split in such a way that our tool selects test cases based on subsets which are a good representation of the entire domain. The partition process divides the data domain into sub-domains with the property that within each sub domain either all elements produce the correct result or all elements produce an incorrect result. The idea is to measure the response time t_i for each specified behavior. The algorithm has four steps has shown hereafter.

```
   **Data**: P the set of input parameters
   **Result**: C the set of data partition
1  $Min \leftarrow \emptyset$; $C \leftarrow \emptyset$; $InitParameter()$;
2  for $i \leftarrow 0$ to $MAXPARAMS$ do
3      $P_i \leftarrow NON\_VALID\_VALUE$;
4      $t_{i,j} \leftarrow Send(P_{i,j}, data)$;
5  end
6  $SortedOrder \leftarrow Sort(t_{i,j})$;
7  for $i \in 0$ to $MAXPARAMS$ using the SortedOrder do
8      for $j \leftarrow 0$ to $MAXBYTE$ do
9          $P_i \leftarrow j$;
10         $t_{i,j} \leftarrow Send(P_{i,j}, data)$;
11     end
12     $InitParameter()$;
13     $Min_i \leftarrow Sort(t_i)$;
14     for $j \leftarrow 0$ to $MAXBYTE$ do
15         $Mean(t_{i,j})$;
16         if $t_{i,j} > Mean(t_{i,j}) + \delta$ then
17             $C_i \leftarrow CreateNewPartition(j)$;
18         end
19     end
20 end
21 $Sort(Min)$;
22 $DefineCFG(Min)$;
   // Fuzzing step
23 for $i \leftarrow 0$ to $MAXPARAMS$ do
24     for $j \in$ each subdomain of $C_i$ do
25         $Res_{i,j} \leftarrow Send(P_{i,j}, data)$;
26     end
27 end
```

Algorithm 1: Partitioning algorithm

Step 1 : Get the Evaluation Order We initialize the set of partition C to empty and the set Min of the minimum value for each parameter which will be used to determine the control flow graph. We initialize each parameter to a fixed value, valid with respect to the specification (line 1). In the algorithm given below, the function $InitParameter()$ is in fact the result of a preprocessing of the input domains of all the parameters from the specifications. Then for each parameter, we choose a non valid value of one parameter (line 3). On the host, the program sends commands to the reader which in turn sends the commands to the card and measures the time spent to respond and send it back to the host (line 4). At that step, after sorting the response times (line 6), we know the evaluation order of each parameter by the program under test. The complexity of this step is $O(n)$ with n equal to the number of parameters

Step 2 : Search for Hidden Commands According to the evaluation order obtained in the previous phase, we can exercise each parameter with non defined values (line 10) and we measure the time spent to respond and send it back to the host (line 10). At that step, we have obtained the different behavior of the program for the different parameters without any combination. We can detect here hidden commands that are not based on the combination of different parameters. The complexity of the second loop is $O(nb\_Param * byteDomain)$.

Step 3 : Build the Partition We create the partition. We sort in an increasing order the values (line13) and we compute the current mean (m) of the values (line 15) to eliminate minor deviation of the collected metrics. If the value is different, we add a new subset to the set C representing the partition. Whatever the returned value are, we can discriminate with the measured time the modification of the behavior. The response time for each command can have some slight differences. These differences are mainly related to the acquisition system but also we believe that some smart card operating systems implements software caches. We have observed on some cards a deterministic delta (a two values delta) when a given command is executed after other commands. For that reason, we need to define a threshold acceptable to discriminate the command response times. In a preliminary learning step in white box, we have defined the value of δ then we set up the interval $[m-\delta; m+\delta]$ and we make a new cluster for each values out of it (line 17). At that step all the sub-domains of C have been identified. The complexity of this step depends on the program, in the worst case any value of each parameter has a different behavior. In fact, we decompose into two different steps, but it is done in the previous loop.

Step 4 : Fuzz it ! In a last step, we sort the Min value of all parameters (line 21) in order to deduce the evaluation order of the parameters (line 21) and to rebuild the control flow graph of the application. This could have been done in the first step, but we have to redo it according to the detected hidden commands.Then the fuzzing step can start. For each parameter we chose one value in each sub-domain (line 24). We send the commands to the reader collecting the data and the status word of the card (line 25) testing all the possible permutations.

Collecting the response time needs to avoid any bias in the measurements. The time measured with the workstation has too much variation due to the different processes executed on it. For that purpose, we use a specific reader able to measure precisely the time, the MP300 TC3 from Micropross. We can observe that the response time to a command can differs revealing groups of different behaviors as shown in figure 1. We can infer that this parameter has three different treatments plus a default treatment which leads to four sub-domains.

5.2 *Example: a One Time Password*

We use the OTP (One Time Password) example that generates a keyed hash using the DES algorithm and an implementation is given is presented in Listing 1. The specification states that there is only one CLA valid (P_0= 0), three INS (P_1 = 0x34, 0x36, 0x38) , P_2 is always `null` and the value of P_3 is between 0x50 and 0x57 for the get or put data command and either 0 or 1 for the verify command. It has an internal state, *i.e.* getting an `otp` is only possible if the user PIN code has been verified and all the parameters have been initialized.

Listing 1: The one time password code

```java
public void process(APDU apdu) throws ISOException {
  if (selectingApplet()) return; // called by the select apdu
  byte[] cmd_apdu = apdu.getBuffer();
  Util.arrayFillNonAtomic(wy, (short) 0, LEN_WY, (byte) 0);
  if (cmd_apdu[ISO7816.OFFSET_CLA] == ISO7816.CLA_ISO7816) {
    switch(cmd_apdu[ISO7816.OFFSET_INS]) {
      case INS_VERIFY: cmdVERIFY(apdu); break;
      case INS_PUTDATA: cmdPUTDATA(apdu); break;
      case INS_GETDATA: cmdGETDATA(apdu); break;
      default :
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    } // end switch
  }    else {
    ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
  } // end else
} // end process
```

Some of the valid commands are:

- VERIFY: 0x00, 0x34, 0x00, 0x01, (payload) 0x04, 0x01, 0x02, 0x03, 0x04; it checks the validity of the user PIN code which has a length of 4 bytes and the code is 1234.
- PUTDATA: 0x00, 0x36, 0x00, 0x56, (payload) 0x2, 0x00, 0x54; this command initialize the `otp` counter with two bytes having the value 100,
- GETDATA: 0x00, 0x38, 0x00, 0x57, (no payload) 0x04. This command requests to obtain an `otp` having the length of 4 bytes.

Of course, there are other commands and the state requires also that all the initialization must have been done and the user is authenticated before sending an `otp`. The first step of our algorithm is to evaluate the evaluation order of each parameter. Each parameter is set up to its initial valid value (step 1), for example we start with the second command PUTDATA 0x00, 0x36, 0x00, 0x56, 0x2, 0x00, 0x54 with valid parameters. Then each element of the command receives a wrong value (out of its specification) and the response time is collected for the instances of this command *i.e.* $t_0 \leftarrow$ **0x02**, 0x36, 0x00, 0x56, 0x2, 0x00, 0x54, then $t_1 \leftarrow$ 0x00, **0x76**, 0x00, 0x56, 0x2, 0x00, 0x54 and so on. Then we compare all the response times for this command and we can sort the evaluation order. With the code given in Listing 1 we obtain: $t_1 < t_2 < ... < t_n$ says CLA is tested first, then the INS field is tested then the control flow enters the method $cmdGETDATA()$. There the evaluation starts with P1 and the interval for P2 after the content of P2 is evaluated a second time in the *switchcase*. Then the state data are evaluated first a previous command should have validated the user PIN, second the initialization phase is complete. Then there is no more control flow.

Then we search for hidden command and we build the partition. We know the evaluation order, here the lowest response time corresponds to an erroneous CLA. We can can try all the 255 values of this parameter. We discover only two partitions one corresponding to CLA equals 0 and one for the rest. We do the same evaluation for the other parameters.

Then the fuzzing can start using only one value in each partition. The complexity of this last part is related to the combination of the partition domains of all parameters.

Listing 2: generating an OTP

```java
private void cmdGETDATA(APDU apdu) {
    byte[] cmd_apdu = apdu.getBuffer();
    if (cmd_apdu[ISO7816.OFFSET_P1] != 0) {// check if P1=0
      ISOException.throwIt(ISO7816.SW_WRONG_P1P2);    } // end if
    short tag = (short) (cmd_apdu[ISO7816.OFFSET_P2] & (short) 0x00FF);
    if ((tag < 0x50) || (tag > 0x57)) {
      ISOException.throwIt(ISO7816.SW_WRONG_P1P2);    } // end if
    short le = (short)(cmd_apdu[ISO7816.OFFSET_LC] & 0x00FF);
    switch(tag) {
      case (byte) 0x50:     ...        break;
      case (byte) 0x51:     ...   break;
 ...
      case (byte) 0x57:       // get a new NSU
         if (le != (byte)8) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
         if (userpin.isValidated() == false) {
    ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);}
         if (stateMachine != INIT_DONE)
     ISOException.throwIt((short)(ISO7816.SW_DATA_INVALID+2));
         generateNSU();
         Util.arrayCopy(wy,(short) (INDEX_MAC),wy,(short) (0),(byte )8);
         break;
    default :
      ISOException.throwIt(SW_DATA_NOT_FOUND);
    }
    apdu.setOutgoingAndSend (short) 0, (short) le);
 }
```

5.3 Evaluation

We have tried our method on several Java Card applications. The content of a first set of application was known thus we worked in white box. At that step we fixed the value of δ. We have developed a code coverage API (such a tool does not exist for Java Card) to verify the ability of our partition algorithm to cover all the branches of the application. We can see in the Figure 1 that this applet responds to three instructions value: 0x34, 0x36 and 0x38 with no hidden commands. And several runs (different colors) provide the same result. Then in a next step we will exercise the tool on hostile Java Card applications to verify the ability to discover hidden commands.

From the previous results, we discovered that we can infer some properties of the applet under test. Indeed, we can also deduce the order in which each APDU fields are tested inside the applet: the APDU field generating the smallest mean value on the sample is the first field to be tested. This is only due to the a particular way of programming Java Card which requires to cancel any command for which the header is unspecified by throwing an exception. Of course this can not *a priori* be generalized to other application domain. This gives valuable information about the treatment done inside the card and in particular the control flow graph of the application. By iterating, we can deduce a tree representing the values of APDU commands used in the applet and the order in which they are tested.

Testing the entire commands space which would have resulted in a combinatorial explosion $(2^8(P0) * 2^8(P1) * 2^8(P2) * 2^8(P3) = 268.435.456$ tests. Within our framework,

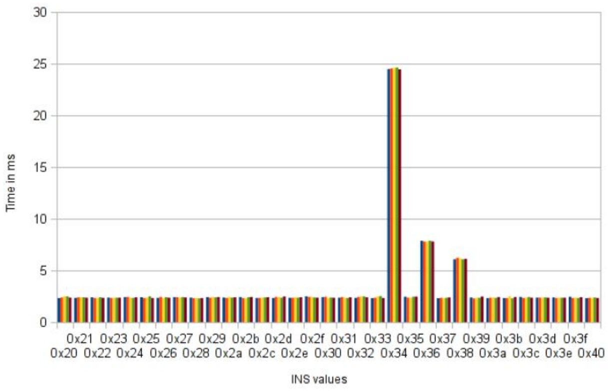

Fig. 1: The INS values.

the number of tests is limited to the number of partitions of each parameter. If there is no hidden command $dom(P1) = \{0, 1\}, dom(P2) = \{0x34, 0x36, 0x38\}, dom(P2) = \{0, 1\}, dom(P3) = \{0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57\}$. For example, the whole number of test of the `otp` application requires to automatically generate $(2^2(P0) * 2^3(P1) * 2^2(P2) * 2^7(P3) = 144$ tests and any hidden command will be discovered.

To ease the development process, we have checked our testing framework against an applet for which we had the source code during the phase of design. At the end of the development process, to test our approach in a black-box environment, we have checked our framework against several applets for which we do not know anything except a minimal specification (used to create a minimal oracle). Our framework was able to guess the APDU commands used in these applets and the order in which the applet was testing APDU command fields. Moreover, these results were obtained in less than a minute for typical applets. All the developments have been written in Java and will be available as an open source project.

5.4 Limits of the Approach

The first limit is inherent with measurement precision. If an application is coded in constant time, we will not discover hidden command. Having a precise knowledge on the execution time of a Java Card virtual machine is difficult. As far as we know, there are no Java Card applications written in constant time.

The second limit is inherent with fuzzing and security applications. If you try a command in an unexpected state or a ill formed command in a correct state, there is a risk to kill the card (message card is mute). So once a card is unresponsive, it needs to evaluate the context and either remove the dangerous command or change the card.

It is possible to extend the approach to the data field (the payload) of the command to some extends. But it requires to have cryptographic computation capabilities within the fuzzer and access to the keys to generate correct cryptograms. In a pure black box approach, this is not affordable.

It is possible to apply this technique to other devices under some hypotheses: the execution time must be repeatable. If there is hardware memory caches, they will be a bias in the measurement and thus will not be applicable. More over if the operating system of the target supports multithreading, it generates another bias. The fact that we are able to rebuilt the control flow graph is mainly due to the smart card application development process where an exception is thrown when the conditions are not satisfied which restrict also the applicability of this technique.

6 Conclusion and future works

In this paper, we have presented a new method for testing software and the associated framework that we designed to validate the approach. It allow to define classify the input data into sub-domains according to the behavior of each parameters. Then, with our data generator that uses side channel information, fuzzing the SUT allows to evaluate the combination of all the parameter swhich was not possible until now. The framework is thus able to discover hidden commands, that is to say unspecified commands which will trigger computations in the tested software. Due to the specific nature of the application (the domain of the parameters is the byte) and its programming model we are also able to retrieve the control flow graph of the application. The limit of the approach consist in writing applications with constant execution time: balancing all the branches of the program. If this is possible at the assembly language (some assembly instructions) it is not affordable at the virtual machine level (hundred or thousand of assembly instructions). When we use the oracle capabilities, we have a testing framework which do a semi-automatic functional testing: we still need to express the domain of the input variables from the specification. In order to verify the quality of the generated test suites we have had to develop our own code coverage tool. This work has been oriented into detecting hidden commands but we plan to continue this work, especially by extending the test to other APDU fields. There are still some issues related with the internal state which can lead to different behaviors, the effect of the software or hardware caches that need to be investigated.

We are evaluating other side channel information leakages like electromagnetic field. Especially when the smart card writes into the EEPROM it generates a visible pattern on an oscilloscope acquisition curve. This information can be analyzed and provided to the fuzzer. Writing in EEPROM means that the system has written a state variable (persistent information) *i.e.* the behavior of the system could be different after this command. We are working on the reverse of Java Card applet using template recognition of EM signatures. It appears that method invocation and return are patterns easy to recognize: building and destroying the Java frame. Thus we plan to use this information to obtain a more accurate control flow graph.

References

[Ali12] V. Alimi. Contribution au déploiement des services mobiles et à l'analyse de la sécurité des transactions. Master's thesis, University of Caen, 2012.

[Ami04] P. Amini. Sulley fuzzing platform. 2004.

[Amo94] E. Amoroso. *Fundamentals of Computer Security Technology.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[BB96] M. Bishop and D. Bailey. A critical analysis of vulnerability taxonomies. 1996.

[BB03] D. Brumley and D. Boneh. Remote timing attacks are practical. In *In Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[BBKL11] M. Barreaud, G. Bouffard, N. Kamel, and J.L. Lanet. Fuzzing on the http protocol implementation in mobile embedded web server. 2011.

[DKL$^+$98] J. F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater, and J. l. Willems. A practical implementation of the timing attack, 1998.

[Edd04] M. Eddington. Peach fuzzing platform 3. 2004.

[GMICL11] A. Gauthier, C. Mazin, J. Iguchi-Cartigny, and J.-L. Lanet. Enhancing fuzzing technique for okl4 syscalls testing. In *ARES*, pages 728–733. IEEE, 2011.

[Guy10] V. Guyot. Smart card the invisible bullet. *Proceeding of the 9th European Conference on Information Warfare and Security*, 2010.

[Koc96] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[Lan11] J. Lancia. Un framework de fuzzing pour cartes a puce: application aux protocoles. *SSTIC*, 2011.

[MPRB10] L. Martignoni, R. Paleari, Giampaolo F. Roglia, and D. Bruschi. In Paolo Tonella and Alessandro Orso, editors, *ISSTA*, pages 171–182. ACM, 2010.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communication of the ACM*, 31(6):676–686, June 1988.

[RC85] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Trans. Softw. Eng.*, 11(12):1477–1490, December 1985.

[RD82] D. Robling and E. Dorothy. *Cryptography and Data Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[SFL13] A. Savary, M. Frappier, and J.-L. Lanet. Detecting vulnerabilities in java-card bytecode verifiers using model-based testing. In *IFM*, pages 223–237, 2013.

[Tak09] A. Takanen. Fuzzing: the past, the present and the future. 2009.

[TDM08] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance.* Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[YCER11] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 283–294. ACM, 2011.