When Time Meets Test

Jean-Louis Lanet, Hélène Le Bouder, Mohammed Benattou, Axel Legay

International Journal of Information Security manuscript No.
(will be inserted by the editor)

When Time Meets Test

Jean-Louis Lanet · Hélène Le Bouder · Mohammed Benattou · Axel Legay

Received: date / Accepted: date

Abstract One of the main challenges in system's development is to give a proof of evidence that its functionalities are correctly implemented. This objective is mostly achieved via testing techniques, which include software testing to check whether a system meets its functionalities, or security testing to express what should not happen. For the latter case, fuzzing is considered as first class citizen. It consists in exercising the system with (randomly) generated and eventually modified inputs in order to test its resistance. While fuzzing is definitively the fastest and the easiest way for testing applications, it suffers from severe limitations. Indeed, the precision of the model used for input generation: a random and/or simple model cannot reach all states and significant values. Moreover, a higher model precision can result in a combinatorial explosion of test cases.

In this paper, we suggest a new approach whose main ingredient is to combine timing attacks with fuzzing techniques. This new approach, which is dedicated to work on Java Card, allows not only reducing the test space explosion, but also to simplify the fuzzing process configuration.

The technique has been implemented and we present the results obtained on two applets loaded in a Java Card.

Hélène Le Bouder · Jean-Louis Lanet · Axel Legay
INRIA-RBA, LHS-PEC,
Campus de Beaulieu, 263 Avenue du Général Leclerc,
35042 Rennes
E-mail: firstname.lastname@inria.fr

Mohammed Benattou
Laboratoire LARIT, University Ibn Tofail
Kenitra, Morocco
E-mail: mbenattou@yahoo.fr

Keywords Security · Software testing · Fuzzing · Timing attacks · Smart card · Java Card

1 Introduction

1.1 Motivation

The main objective of this paper is to propose a new testing technique for Java Card. Our objectives are 1. to ensure that the functionalities of the system are properly implemented, and 2. to check that no vulnerability remains in the system before it is deployed. This leads to two main testing issues:

1. **Functional software testing**. The main goal of functional software testing is to find discrepancies between the actual behavior of the system functions and the expected behavior as described in the functional specifications.
2. **Security testing**. Security testing expresses what a system should not do, or what should not happen. A security policy is a set of rules defining the acceptable values of a system as its state changes through time. Security testing aims to find a behavior that violates security policies. It implies to check that all security rules are enforced by the software.

The literature and know-how [1] offer numerous ways to perform functional testing [2]. This includes among others Structural testing [3], Limit testing, or Model based testing [4]. On the other hand, security testing is more based on know-how and a few methodologies are available. Fuzzing is one of the most used techniques to search for vulnerabilities.

1.2 State of the art

This paper offers an elaborated security testing solution for black box Java Card applications. It relies on combining security testing and timing attacks. While most of the related work dedicated to Java Card vulnerability assessment is delayed to Section 3.3, we hereafter briefly summarize some state-of-the-art of fuzzing and timing attacks.

1.2.1 Security Testing

Amoroso [5] defined a vulnerability as an event that allows a threat to potentially occur. With this definition, a threat is any event that can have an undesirable effect on the assets. As a consequence, it refers to a violation of the expectations of users, administrators, and designers. Practically, there are several reasons for a software vulnerability to happen. Among those, one finds access control weakness [6], that is where the system performs a command it should not have access to at a particular moment of time. Vulnerability can also be defined with respect to the state of the system. Such a state space vulnerability [7] consists in reaching an unauthorized state from valid state using a valid command.

Detecting vulnerabilities is a difficult task, and several strategies can be used in order to mitigate the security vulnerabilities. These solutions include auditing the source code, performing formal software verification, applying experience-based testing based on the analysis of known attacks. In this paper, we focus on fuzzing that is a testing technique which analyses the software behavior when its inputs are fed with particular data (*e.g.* invalid or random data). This approach, proposed by B. Miller [8,9], is definitively one of the most adopted testing approaches as it can be (semi-)automatized, and it is considered to be lightweight in the development process.

1.2.2 Timing attack

Timing attacks, which are based on observing the circuit during computation time, belong to the family of Side Channel Attacks (SCA). In recent years, they have been a hot topic in the smart card security research. Such attacks exploit the fact that some physical values of a circuit depend on intermediary values of the computation. This is the so-called information leakage of the device.

The most classic leakages are power consumption [10, 11], electromagnetic emissions [12] and timing. In this paper, we focus on timing attacks which use the time spent by a processor to perform some computation. We observe that a processor takes different amounts of time to process different inputs according to the Control Flow Graph (CFG). The reasons are:

- instructions are executed in different amounts of processor cycles (*e.g.* multiplication and division will take more processor cycles than addition and subtraction),
- compiler optimization (for example short-circuited conditional checking),
- cache hits or misses.

Timing attacks were first used for the cryptanalysis. In 1996 when P. Kocher has demonstrated that knowing the time needed to perform private key operations on RSA or Diffie-Hellman protocols may result in breaking a cryptosystem [13]. In 1997, Dhem *et al.* [14] implemented a timing attack to obtain the 512-bit RSA key of a smart card chip in a couple of minutes. In [15] the authors presents how a PIN code should be protected against these attacks. Brumley and Boneh [16] managed to attack the SSL protocol by measuring the time an OpenSSL server takes to respond to decryption queries. In [17], Bernstein has shown that cache-miss induced timings can be used to recover an AES secret key. This last attack showed that there is a timing dependency created by the size of the AES structures (SBox look-up table notably) with respect to the cache sizes. It is therefore very difficult for the developer to overcome this vulnerability. Finally, Felten and Schneider [18] carried out several kinds of timing attacks on web systems. The cache-based timing attack is based on the presence of information in the cache of the browser such that once a user visits a static page, her local cache contains a copy of the page causing the page to load faster on subsequent visits. By measuring the time the browser takes to load a given page, they showed that a malicious website can determine whether the user visited the page before.

1.3 Contributions

In this paper, we propose a new method of security testing of Java Cards. At the root of our approach, there is a combination of fuzzing and timing attack. This new testing scenario is based on observing several executions of the system and by freezing some of its parameters in order to establish a partial order on their timing evaluation. This form of data dependence can latter be used to reduce the state-space explosion on input tests. Albeit partitioning of inputs at boundaries have already been proposed for white box systems, we are the first to

study them in the context of black-box Java Card applications. The absence of knowledge on the code of the system is compensated by the timing information. This new approach allows not only to reduce the test space explosion, but it also simplifies the fuzzing process configuration. We also show how our approach can be used to reverse the behavior of an applet loaded in a Java Card. Finally, we have developed a coverage toolkit and have used it to assess the performances of our methodology on a complex credit card application and a one time password application.

A preliminary proof of concept was developed with two students [19]. The aim of this proof of concept was to check the capability to differentiate precise timing leakage. In this revisited work, we have developed the theoretical foundation, the partitioning algorithm has been redesigned and evaluated with real world applets. We have investigated the capability to discover uncovered commands and we have clearly defined the limits of this approach. Instead of using dedicated reader, we choose in this development to use external measurement means with an EM probe and an oscilloscope. This equipment is in charge to recognize on the fly the incoming command, to store the traces and to send back to the PC the elapsed time measures. The prototype has been entirely redeveloped due to the new architecture.

1.4 Organization

The paper is organized as follows. Section 2 poses the context and summarizes most of the challenges in guaranteeing security of Java Card applications, while Section 3 outlines the main difficulties in fuzzing such cards. The data partitioning problem at the root of our approach is presented in Section 4. Finally, Section 5 and 6 present experiment/implementation and conclusion, respectively.

2 Context

As already said, testing security products such as smart cards not only requires to ensure that the functionalities are met, but also that there is no vulnerability in the product. Those requirements are guaranteed by a certification center which guarantees that the product satisfies a claimed certification level.

2.1 Certification Process

Over the past years, one has observed an increasing demand for certification of security sensitive products. Such certificates are obtained by an independent laboratory which performs a security evaluation based on a certification scheme [20]. In this context, the certification process is done by establishing the confidence in the security of the evaluated product. The degree of assurance that a security target has met is classified via the Evaluation Assurance Level (EAL). EAL ranges from EAL1 to EAL7, and each EAL level has a set of assurance components issued from the various assurance classes.

Certification process is intensively used by the smart card industry where vulnerability remains the main issue. There, the AVA assurance class that defines requirements for identifying the exploitable vulnerabilities plays a particular role. This class has four levels to potentially identify vulnerabilities introduced at development time:

1. *Developer analysis*, that checks for obvious exploitable weaknesses.
2. *Independent analysis*, which uses an independent entity, searches for vulnerabilities that could be exploited by an attacker with low attack potential.
3. *Moderately resistant*, that establishes resistance to attackers possessing a moderate attack potential but mandates the evaluator to perform and document a systematic search for vulnerabilities.
4. *Highly resistant*, that exploits independent penetration testing and resistance to attackers possessing a high attack potential.

It must be understood that the evaluator has to perform a systematic search to determine the relevance of security measures and the security weaknesses. This search is often divided in the following steps: search for potential vulnerabilities, developing intrusion tests, performing intrusion tests and reporting the results.

Over the past, the ability to find vulnerabilities was mostly based on the know-how of the evaluator but can be improved with some tools. Recently, certification centers have paid a particular attention to the fuzzing technique for performing the search.

2.2 Java Card

2.2.1 Definition

Java Cards are smart cards based on the Java environment from Oracle (formerly Sun Microsystems). Such cards come together with a tiny operating system which provides a Java Card Virtual Machine capable of running applets written in Java with the Java Card API. An applet receives commands, makes computations and

responds. 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, a command APDU has only a header made of four bytes: instruction class (CLA), instruction code (INS), first parameter (P1) and the second parameter (P2). Data can follow the header. The response issued from APDU is even simpler: it takes a stream of bytes (which is the applet response) followed by two bytes: the status word (SW). For example 0x90 0x00 is the simplest APDU response meaning that the processing was done correctly. The list of APDU commands and responses for an applet is a part of its specification.

2.2.2 Java Card applets

Java Card applications have one main entry point that is the `process` method. The other entry points are the `install` method (that is executed once), the `select` and the `deselect` methods. The `process` method decodes the header and according to the fields performs different actions.

The CFG of a card represents its different execution paths along the different basic blocks of the application. The connection between the blocks depends on the evaluation of the parameters of the basic blocks. Concretely, each basic block ends with either the evaluation of a conditional expression or a return, and is thus a leaf of the program. Due to the nature of the embedded applications, several tests are done at run time and if a state does not correspond to the expectation, the program generates a security exception.

3 Discovering Vulnerabilities with Fuzzers

A functional test of applications such as the Europay Mastercard Visa (EMV) [21] command requires to generate test cases for all the nominal behaviors of the command. A nominal behavior leads either to a success *e.g.* PIN is verified or a failure *e.g.* Try Counter reaches its limit. The objective of security testing could be to check that the parameter P2 has only two valid values: 0x80 and 0x88. That is no hidden value can be exploited. For that purpose, the fuzzing technique can be used. In the rest of this section, we briefly summarize the fuzzing approach.

3.1 Definition

A fuzzer is a testing entity 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 command in the case of stateful system. Fig. 1 presents a typical test bench architecture.

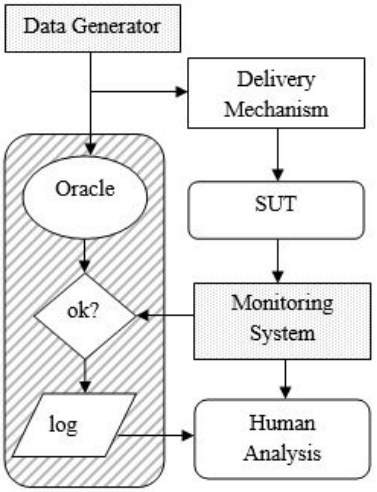

Fig. 1: Typical fuzzing framework architecture.

The main ingredients of this architecture are the following:

- Data Generator: This module generates the inputs according to a given policy (random, bounds...) that are used to execute the System Under Test (SUT). To be independent of the transport layer, it generates abstract tests. These are transformed in a concrete form adapted to the transport layer, to the SUT by the delivery module. The Data Generator is the core module of the fuzzer tool.
- Delivery Mechanism: The delivery mechanism is in charge of adapting the test case to the target *i.e.* packing the protocol data into a transport protocol or directly invoking a program through unit testing mechanism.
- Monitoring System: This module observes the behavior of the SUT while it executes the test cases. It also collects the output of the system.
- The oracle and log mechanism in the dashed box are optional but help greatly during result analysis. They allow to distinguish if the result matches with the expected one.

The above architecture emphasizes the importance of the Data Generator where data are manually generated or automatized via the fuzzing process.

One of the challenges with fuzzing is in fact to evaluate the effectiveness of generated fuzzed inputs. There are fuzzers that use an oracle in order to decide what should be the expected answer when passing the value

to the system. Such an answer is often *a priori* unknown and the input data lead to a crash of the observed system. We also observe that if the protocol or the system to fuzz is a stateful system, meaning only a subset of commands can be sent at a given time, then the fuzzer must be able to handle an internal state machine. Otherwise, there is a risk that most of the commands sent are not accepted leading to poor results in the discovery of vulnerabilities. This does not mean that the fuzzer should not try unexpected commands in a given state, but rather that the search of error must be driven by its internal state machine.

In this paper, we focus on fuzzers that do not rely on the source code. In the rest of this section, we briefly mention the existence of other fuzzers that do exploit this code. Such white box fuzzers use the information included in the source code to find the vulnerabilities. As an example, the Dowser fuzzer [22] generates test inputs by combining concrete and symbolic execution. This technique is also known as *concolic* execution. It combines taint tracking, program analysis and symbolic execution to find buffer overflow vulnerabilities. It drives the search to paths that have a potential for erroneous pointer arithmetic. It can thus be combined with code coverage tools which are used to estimate the fuzzer quality. This sort of fuzzers is much more efficient but needs access to the source code.

3.2 Fuzzing techniques

The main idea of a fuzzer is to generate input data (Data Generator) of a command in order to test a target application. The high level of automation in the data generation and a complete architecture allows to automate the testing. Nevertheless, the policy of the Data Generator is important. If data are chosen randomly, the result is often inefficient. Indeed, many data may only cover a small portion of the state space if not selected properly. To make testing more efficient, only certain boundaries or patterns might be used. There exist two main techniques for determining the data used through test cases: data generation and data mutation.

Mutation-based fuzzers use an existing valid data session and apply several transformations before sending it to the SUT. These transformations are often minor substitutions of the input stream with random data. The behavior of the SUT is logged and analyzed. Mutation-based fuzzer is mostly used for testing network protocols and parsers as it is very easy to save a session and replay it on these software. It is not adequate for fuzzing very large or unknown protocols.

The generation-based fuzzers are smarter than mutation based ones. They generate the input data from scratch based on the specification or a command format. They provide a set of tools to describe the software under test, input 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.

An open loop fuzzer does not rely on information inferred from the response and its results are often poor but well dedicated to a black box approach. On the other hand feedback fuzzers reuse information provided by the SUT to improve the data generation. They have much better results in particular for code coverage but also in activating rare events. But they rely on a collaboration with the SUT, so needing a white box approach.

The main challenge resides in the choice of input data that can reveal software errors. Elaborated fuzzers do not randomly change fields to produce invalid data. Rather, they use the knowledge about the specification or the format to produce such data. This form of 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 [23] and Sulley [24] 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 the fuzzing technique is the high ratio of automation to manual work. Each input data or field of a protocol is described in terms of domain and the fuzzer has several heuristics to choose the value from the corresponding domain.

3.3 Java Card Applet Fuzzing

Prior works have been done for applying fuzzing on smart card and in particular on Java Card application. One of the challenges for testing this type of components, is the adaptation of 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 SUT for future analysis. However, above those two challenges, the main issue that remains to be solved is related to the domain of the input parameters and the policy to choose a concrete value for each parameter.

This difficulty is largely emphasized in the literature on the topic. In [25], 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 protocols in particular the BIP (Bearer Independent Protocol). In [26], Lancia describes how he used the Sulley fuzzing framework to test several implementations of the EMV protocol running on different smart cards. The approach proposed by the author is to use a reference implementation and an implementation under test and to consider the first one as an oracle in case of discrepancies. In [27] Guyot shows how simple it is to discover all the commands accepted by a smart card application. He proposes a basic solution based on using a combination of the two parameters (P1 and P2) of the command to improve the resistance against fuzzer. As Barreaud demonstrated by fuzzing also these parameters, such a counter measure does not resist to any fuzzer. Alimi has presented another fuzzer in his PhD [28] with a new approach based on genetic algorithms to choose the input data. He applied the fuzzer on an EMV application. Any of those approaches rely on how input data are selected. Providing a solution to this question is the objective of this paper.

3.4 Limitation of the Current Approaches

Current fuzzing frameworks have three major issues:

1. the test case combinatorial explosion,
2. the difficulty of result analysis,
3. the limitations induced by models.

The first issue is to deal with the test case combinatorial explosion. Fuzzing process is not deterministic as it uses random test data. To gain more control over this process, several strategies are used to decrease the testing set. One of them is to generate only some values around the inputs 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 have been studied in the literature [29,30]. All of them are based on the knowledge of the specification or the implementation. None of these methods are available when working in a black box context. In this latter context, existing approaches are in fact likely to choose loose test generation strategies (*i.e.* wider bounds) as nowadays the ratio of the number of tests per second is very high for common software. It increases the test coverage and consequently the difficulty of exploiting the observed results.

This work is cumbersome and most of the fuzzing frameworks miss the tools to ease this process. There are some works in this domain to solve this issue by coupling the fuzzer process with an oracle [31] or by using several existing implementations of a specification to automatically classify the test results [32].

To improve the test generation quality and enhance the fuzzing process, fuzzing frameworks use models to describe the SUT inputs. However, this approach is really cumbersome when the SUT is somewhat complex. Moreover, grammars used by fuzzers are sometimes too limited for describing the inputs of the software under test as described in [33].

All these limitations hinder a wider and more effective use of the fuzzer technique. In many cases, the means available to inject faults in the generated input are restrictive and do not include methods to specifically generate inputs that would likely trigger well-known, target-specific attacks. Furthermore, support for testing complex, stateful protocols is generally lacking; thus, requiring the tester to manually bring the system to the desired state before starting the actual test. Finally, the language adopted to describe how fuzzing should be performed is often very primitive, and, as a consequence, the activity of specifying fuzzing tests can require significant efforts.

4 Data Partitioning using Timing Measurements

One of the major problems with exploiting fuzzing for security testing is that, contrary to functional testing, security failures exploit combinations of variable's value that are not specified. One thus has potentially to explore the entire combinatorial of the domains.

To overcome this issue, we exploit the particular nature of a Java Card implementation. Consider Fig. 2 which presents an implementation with two decisions: `c1` and `c2` and three basic blocks: `bb1`, `bb2` and `bb3` which lead to three measurements `t1`, `t2` and `t3`. The fact that $\texttt{t1} > \texttt{t3} > \texttt{t2}$ does not imply that condition `c1` is evaluated before `c2`. The response time reflects the observable behavior not the structure of the program.

4.1 The programming model of Java Card

The situation changes with Java Cards. There, each command is made of a header having four byte-fields and potentially some data. The header is often used as a first control part of the command, while the data field is used to process the command. There is a particular way of programming a Java Card which requires

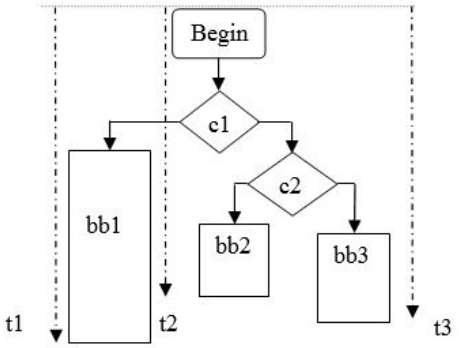

Fig. 2: Partial Order of Time Measurements

to cancel any commands for which the header or a data field is unspecified. This is done by throwing an exception using the method `throwIt()`. This programming paradigm gives valuable information about the treatment done inside the card. For example, the time needed to execute the basic block `bb1` is negligible *vs.* the execution time of `bb2` as shown in Fig 3.

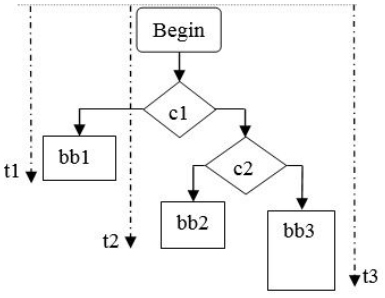

Fig. 3: Total Order of Time Measurements

Consequently, by measuring the response time of different parameters we can infer the evaluation order of the blocks. If a given parameter provides the smallest response time `t1` then it is evaluated first in the algorithm. This specificity is the base for the design of our data partitioning algorithm, whose intuition is discussed in the next section.

Our Java Card applet corpus contains 47 different applets from Telcom applications to banking and transportation applications. The control part (header parameters) of these applets always follows this programming scheme. This has been confirmed by our industrial partner in its own corpus. Moreover, we have also seen that often the control flow related to the data part of the command follows also this scheme but not necessary. In this paper, we consider only the control part related to the header. The data part is one of our future works.

Remark 1 There are techniques that can be used to mitigate timing attacks. Such counter-measures have the form of algorithms that are designed to run in constant time. While efficient, it is well-known that respecting this principle is hard and hence limited to particular cryptographic algorithms at a native level. To the best of our knowledge, there exists no such implementation in the Java Card world at the application level.

Remark 2 Programming in constant time is difficult. The following example shows a fragment which seems to be executable in constant time. Its translation to Java is obvious.

```
1 if x > y then
2 |  x ← 20;
3 else
4 |  x ← 40;
5 end
```

Algorithm 1: Constant time execution at source level

But once compiled, the code presents some differences according to the path used. The first branch

```
1 ifne L2;
2 L1 :bspush 20;
3 sstore_3;
4 goto L3;
5 L2 : bspush 40;
6 sstore_3;
7 L3 : return;
```

Algorithm 2: Non constant time execution

takes more time to be executed compared to the second. This is due to the presence of the jump to join the label *L3*. To be executed in constant time one has to add in the second branch also the instruction **goto** *L3* which in that case is useless in term of functionality but necessary to resist to a timing attack.

4.2 Side Channel Fuzzer by intuition

Our algorithm starts from the specification of the implementation, that is a set of functionalities that can be triggered for specified values of the parameters. As an illustration, let us consider the 7 following functions.

- $f_1 : \mathrm{P0} \leq 80$
- $f_2 : \mathrm{P0} > 80 \wedge \mathrm{P1} > 5 \wedge \mathrm{P1} \leq 71$

- $f_3 : \mathrm{P0} > 80 \wedge \mathrm{P1} \leq 5 \wedge \mathrm{P2} = 0$
- $f_4 : \mathrm{P0} > 80 \wedge \mathrm{P1} \leq 5 \wedge \mathrm{P2} \neq 0 \wedge \mathrm{P3} \leq 10$
- $f_5 : \mathrm{P0} > 80 \wedge \mathrm{P1} \leq 5 \wedge \mathrm{P2} \neq 0 \wedge \mathrm{P3} > 10$
- $f_6 : \mathrm{P0} > 80 \wedge \mathrm{P1} > 71 \wedge \mathrm{P2} \leq 90$
- $f_7 : \mathrm{P0} > 80 \wedge \mathrm{P1} > 71 \wedge \mathrm{P2} > 90$

Observe that those functionalities split the domains of the parameters P0, P1, P2, P3 as follows:

$$Dom(\mathrm{P0}) = \{0 \cdots 80\} \cup \{81 \cdots 255\}$$
$$Dom(\mathrm{P1}) = \{0 \cdots 5\} \cup \{6 \cdots 71\} \cup \{72 \cdots 255\}$$
$$Dom(\mathrm{P2}) = \{0\} \cup \{1 \cdots 90\} \cup \{91 \cdots 255\}$$
$$Dom(\mathrm{P3}) = \{0 \cdots 10\} \cup \{11 \cdots 255\}$$

In what follows, $D^1_{\mathrm{P}x}$ is used to denote the first sub-domain of Px.

In order to perform testing, one has to fuzz values of parameters among those that are accepted by the command. Unfortunately, as this solution is not tractable (the number of test cases is the product of the domain of each parameter), one generally proposes to fuzz with respect to a few representatives per sub-domain. The danger of this approach is that one specific (but not selected) value in a sub-domain could launch an unspecified command. Another danger is that this approach does not permit to reason on those values of the parameters that are not specified by the command.

To leverage this problem, we propose to evaluate the timing response of each functionality. For doing so, we first fix a nominal value for each parameter and run the command. For simplicity, let us assume that the response time of f_i is smaller than that of f_j for $i < j$. From this information and from the nature of Java Card implementation, we can infer that the control flow graph should correspond to the one in Fig. 4. In this figure and by convention, the gray boxes indicate the leafs of the program *i.e.* an exit which sends back a status word to the reader. The choice for the next block depends on the value of the parameter passed to the command. The specification of this application expresses the behavior of the program if the parameters belong to certain values. That is, *e.g.*, it specifies the value of P1 if P0 is greater than 0x80, but not in the other cases. The CFG defines 12 basic blocks that are (a, b, c, d, e, f, g, h, i, j, k, l). The successive conditional evaluations specify seven paths:

$c_1 = \{\mathrm{a}, \mathrm{b}\}$
$c_2 = \{\mathrm{a}, \mathrm{c}, \mathrm{i}\}$
$c_3 = \{\mathrm{a}, \mathrm{c}, \mathrm{d}, \mathrm{e}\}$
$c_4 = \{\mathrm{a}, \mathrm{c}, \mathrm{d}, \mathrm{f}, \mathrm{g}\}$
$c_5 = \{\mathrm{a}, \mathrm{c}, \mathrm{d}, \mathrm{f}, \mathrm{h}\}$
$c_6 = \{\mathrm{a}, \mathrm{c}, \mathrm{j}, \mathrm{k}\}$
$c_7 = \{\mathrm{a}, \mathrm{c}, \mathrm{j}, \mathrm{l}\}$

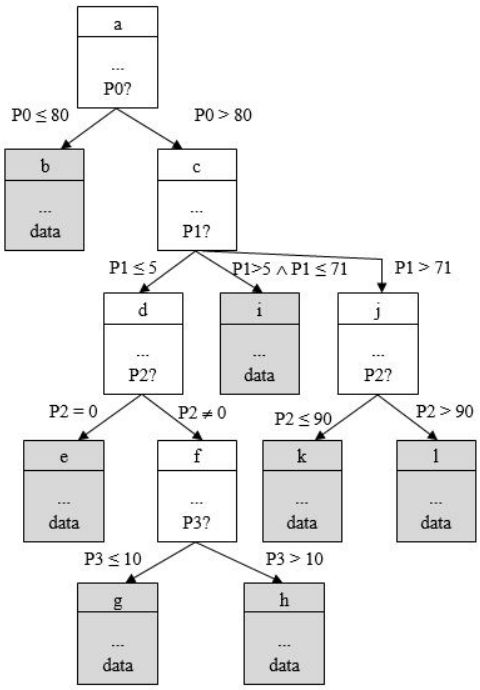

Fig. 4: CFG of the application under test

There, the sub-domain of each variable for the test seems to be defined and a solution to fuzzing could simply be to select a value in each sub-domain. While this solution suffices for functional testing (*i.e.* to testing what is specified), it is not enough for security testing that requires to also test what is not specified. We thus need to focus on uncovering commands which are not part of the specifications. Let us illustrate this principle. Assume for example that the implementation of the specification has three undocumented behaviors {uc1, uc2, uc3} that correspond to security failures. This situation is illustrated in Fig. 5.

As those behaviors are not part of the specification, a functional testing approach that would consider one nominal value for each sub-domain would not detect them. Consequently, the only solution would be to fuzz the entire combinatorial. Our algorithm proposes to overcome those difficulties by exploiting the timing evaluation of the parameters.

The implementation in Fig. 5 outlines two important features that are:

1. One element of a sub-domain (here P2 = 100) is used to launch a hidden command (uc3);
2. The use of a parameter (P3) where it is not expected, to launch uncovered commands (uc1 and uc2).

Our algorithm proposes to discover those commands as follows. For uc3, we exploit the nominal order be-

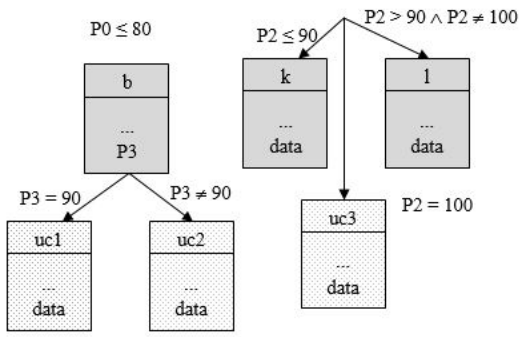

Fig. 5: Uncovered commands in the application under test

tween the parameters obtained above. That is, we start from P0 that we variate between 0 and 255 while fixing all the other parameters. The response time of the corresponding calls should be divided into two values. If more than two values appear, then we can conclude that there is a hidden command. When P0 is done, we evaluate the next parameter P1. We thus fix a value of P0 for which P1 is evaluated (*i.e.* P0 $> 0x80$) and we restart the process described above. Observe that exploiting the order of evaluation allows us to reason on 255 values at a time. The result is a coarser partition of the domain that we fuzz with a classical approach, that is we fix either one or three representatives per sub-domain according to the fuzzing strategy to avoid the combinatorial blow up. The approach works well for discovering uc3 but not to directly find uc1 and uc2. For this, we still have to fuzz the obtained sub-domains.

Param.	sub-domains		
P0	D^1_{P0}	D^2_{P0}	
P1	D^1_{P1}	D^2_{P1}	D^3_{P1}
P2	D^1_{P2}	D^2_{P2}	D^3_{P2}
P3	D^1_{P3}	D^2_{P3}	

Table 1: Sub-domains for activating paths

A second strategy for the first step is described with the Table 1, where a value of each sub-domain is used for the fixed parameters. If a parameter is part of the path, the initial choice is not changed. Else, evaluate the other domains of these parameters. For example, the path c_1 depends on P0 as shown in Table 2. So, in the first strategy, one sub-domain is fixed for P1, P2 and P3. In this second strategy, a value of each sub-domain is tested.

For the path c_1 we need to get the response time by fuzzing the entire domain of P0 eight times for one value of the sub-domain of each other parameter says D^1_{P1}, D^2_{P1}, D^3_{P1}, D^1_{P2}, D^2_{P2}, D^3_{P2}, D^1_{P3} and D^2_{P3}. The number of tests is slightly increased but with this strategy, the covert commands uc1 and uc2 will be discovered.

Responses	P0	P1	P2	P3
r_1	D^1_{P0}	$\forall$	$\forall$	$\forall$
r_2	D^2_{P0}	D^2_{P1}	$\forall$	$\forall$
r_3	D^2_{P0}	D^2_{P1}	D^1_{P2}	$\forall$
r_4	D^2_{P0}	D^1_{P1}	$D^2_{\text{P2}} \cup D^3_{\text{P2}}$	D^1_{P3}
r_5	D^2_{P0}	D^1_{P1}	$D^2_{\text{P2}} \cup D^3_{\text{P2}}$	D^2_{P3}
r_6	D^2_{P0}	D^3_{P1}	D^2_{P2}	$\forall$
r_7	D^2_{P0}	D^3_{P1}	$D^2_{\text{P2}} \cup D^3_{\text{P2}}$	$\forall$

Table 2: Sub-domains for activating paths r_1 to r_7

4.3 The challenge of measuring response time

As said above, the idea for creating a parameter partition is to measure successive response times $r(f_i)$ for a series of parameter's values. For each parameter, we compute the response time for its possible values (*i.e.* from 0 to 255), while the value of the others is taken in their expected domains (*i.e.* the one defined by the specification). This approach is supposed to refine the domain provided by the specification.

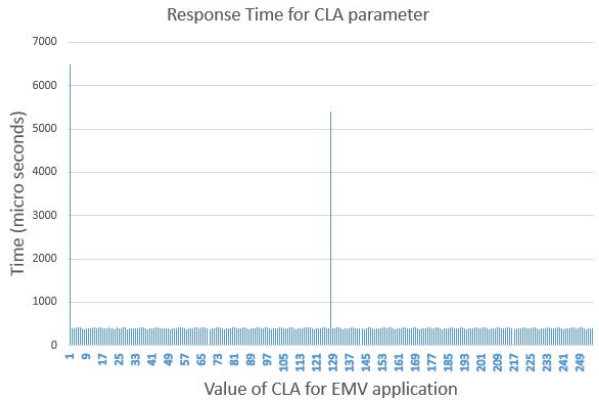

Fig. 6: Response time for P0 parameter

It can be observed that the response time to a command can differ revealing groups of different behaviors as shown in Fig. 6. One has to remark, that this parameter has two different treatments plus a default treatment which lead to three sub-domains. As an example, the EMV specification states that the CLA byte can have four values: $0n, 8n, 9n$ and En. The values $9n$ and En are manufacturer proprietary reserved values. We can observe that this implementation does not use these specific values but only value 00 and 0x80.

The low nibble of the parameter is used for the logical channel. We can deduce that this implementation uses a mask on the CLA byte to be inter-operable on every logical channels.

Collecting the response time requires to avoid any bias in the measurements. The time measured with the workstation has too much variations due to the different processes executed on it. In our proof of concept, we used a MP300 TC3 from Micropross as a specific host reader.

Now on the host, the program sends commands to the reader which are measured with an EM probe. The oscilloscope triggers on the command sent and detects the response sent back by the card. Once the response is acquired by the reader, the program requests the measurement to the oscilloscope. This change in the architecture of the side channel fuzzer offers a new possibility to also disassemble the code executed inside the card [34] such that we have a better understanding of the code executed. The link between the two approaches (fuzzing and disassembler) is part of the future works.

Of course, the response time for each command can have some slight differences. If the response time of the VERIFY command with the nominal parameters is measured, different values from 5959μs to 6193μs are obtained. In this context, it needs to define a threshold acceptable to discriminate the command response times. In order to create the partitions, the mean (m) of the sample and its standard deviation (s) is computed. Then, we define the interval $[m-s; m+s]$, and a new cluster is built, for each value out of it.

4.4 The Algorithm

As sketched above, 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 for each specified behavior. The Partitioning algorithm, which is presented in Algorithm 3, has four steps that are described hereafter.

The notation P_i denotes the i^{th} parameter, $t_{i,j}$ the response time of the i^{th} parameter for the value j, Min is a set for storing the response times, m is the cumulated mean, $D_{i,k}$ the k^{th} partition for the parameter P_i. The function $Send(\mathrm{P}_i, data)$ transmits a command with the parameters vector and the data, having changed only the parameter i in the command. The function $InitParameter()$ initializes each parameter with a value corresponding to one valid command. We reuse the previous example with only one command, four parameters P0..P3 and seven different behaviors according to the combination of the parameters.

```
   Data: P the set of input parameters
   Result: D the set of data partition
   // Step 1: Get the Evaluation Order
 1 Min ← InitParameter();
 2 D ← ∅;
 3 for i ← 0 to MAXPARAMS do
 4 |   P_i ← NON_VALID_VALUE;
 5 |   t_i ← Send(P_i);
 6 |   Min ← Min ∪ {t_i};
 7 end
 8 Min ← Sort(Min);
 9 SortedOrder ← InferOrder(Min);
   // Step 2: Search for Hidden Commands
10 for i ∈ SortedOrder do
11 |   Min ← ∅;
12 |   for j ← 0 to MAXBYTE do
13 |   |   P_i ← j;
14 |   |   t_{i,j} ← Send(P_i);
15 |   |   Min ← Min ∪ {t_{i,j}};
16 |   end
   |   // Step 3: Build the Partition
17 |   Min ← Sort(Min);
18 |   m ← Min_0;
19 |   index ← 0;
20 |   k ← 0;
21 |   for j ← 1 to MAXBYTE do
22 |   |   m ← 1/(1+j−index) * Σ_{x=index}^{j} Min_x;
23 |   |   if Min_j > m + δ then
24 |   |   |   CreateNewPartition(k);
25 |   |   |   D_{i,k} ← Min_j;
26 |   |   |   m ← Min_j;
27 |   |   |   index ← j;
28 |   |   |   k ← k + 1;
29 |   |   else
30 |   |   |   D_{i,k} ← Min_j;
31 |   |   end
32 |   end
33 end
   // Step 4: Fuzz it !
34 Min ← Sort(Min);
35 SortedOrder ← InferOrder(Min);
36 for i ← 0 to MAXPARAMS do
37 |   for j ∈ each sub-domain of D_i do
38 |   |   Res_{i,j} ← Send(P_i);
39 |   end
40 end
```

Algorithm 3: Partitioning algorithm

Step 1: Get the Evaluation Order.
(**Algorithm 3** lines 1 to 9)

On the host, the program sends commands to the reader which in turn sends the commands to the card.

The time spent to get a response is provided by the oscilloscope which sent it back to the host. For the clarity of the algorithm, the interaction with the scope is hidden. The measure t_i is provided by a request to the oscilloscope after sending the command. Each timing value t_i memorizes the vector value associated.

We first initialize a set Min that assigns a timing value to each command (line 1) using *InitParameter()*. This is done by applying the function with one of its nominal parameters. It will be exploited to extract the control flow graph and discover hidden commands in step 2. As a result, at that step Min contains seven timing values $\{t_0, t_1, ..., t_6\}$ corresponding to $f_1, ..f_7$.

For each function, we also measure the time for value of the parameters outside those that are specified (line 4-6). The objective is to make sure that the function tests the value of its parameters. The value MAXPARAMS corresponds to the number of parameters (fields of the APDU to be fuzzed) minus one. We add it to the set Min which contains now $\{t_0, ..., t_6, t_7, t_8, t_9, t_{10}\}$.

At that step (line 8), after sorting the response times, the set Min has for example the following value:
$\{t_7, t_8, t_9, t_{10}, ..., t_1, t_2, t_3, t_4, t_5, t_6\}$.

If t_7 (the smallest response time which is related to the path $c_1 = a, b$ of our example) corresponds to the parameter P0 outside its range we can infer (using the function $InferOrder()$) that this parameter is evaluated first. Then the set $SortedOrder$ used at line 10 contains $\{0, 1, 2, 3\}$. The complexity of this step is $O(i)$ with i equal to the number of parameters ranging from 0 to MAXPARAMS.

Step 2: Search for Hidden Commands.
(**Algorithm 3** lines 10 to 16)

The objective here is to discover commands like uc3. For doing so, each parameter is fuzzed by exploiting the set order obtained in the previous step. We reuse the set Min for collecting the response time for each parameter, thus Min is reinitialized at line 11.

Each parameter, even those with undefined values (line 14), is exercised in and the time spent is measured. We collect all the response times ($t_{i,j}$) for each parameter (P_i) that we store into Min. The complexity of this second loop is: $O(\text{nb\_Param} * \text{MAXBYTE})$.

If we use the example provided in Fig. 4, we know thanks to step 1 that the parameter P0 is evaluated first. Then, at line 16 we have collected all the $t_{i,j}$ for this parameter and Min contains now:

$\{t_{0,0}, t_{0,1}, .., t_{0,255}, t_{1,0}, .., t_{1,255}, .., t_{3,0}, ..t_{3,255}\}$.

The measurements are stored as a structure (index, value). We collect 256 time measurements, some of them having a low value for example 180, 182, 178,... micro-seconds for the indexes ranging from 0 to 80. They reflect the behavior c_1 with some fluctuations. After that, the values are increasing says 2500, 2523, 2489,... micro seconds for the remaining indexes that reflect the behavior of another command depending on the fixed parameters.

Step 3: Build the Partition.
(**Algorithm 3** lines 17 to 33)

At this step, the new partitions of sub-domains are created. The values are sorted in an increasing order (line 17). In our example, the set Min contains:
$\{178, 180, 182, ..., 184, 2489, 2500, ..., 2523\}$.
The response time for each command has some slight differences. These differences are mainly related to the acquisition system, but we also postulate that some smart card operating systems implements software caches. On some cards, we have observed a deterministic standard deviation δ (δ takes two values) when a given command is executed after other commands. For that reason, a threshold acceptable is required to discriminate the command response times. In a preliminary learning step in white box, the value of δ is defined for example with the value 20.

We need to detect the difference between the values 184 and the one that follows 2489 which implies a new partition, but not between the first 178 and the second 180. For that purpose, we use in the loop, the cumulated mean m of the value (line 22). This eliminates minor deviation of the collected metrics. At the first iteration, m is initialized with the first measurement (line 18) equals to 178.

Then, at line 23 the comparison is false and we remain in the same set. We store the measurement in the same partition $D_{i,k}$. At the second index (remind that here the index j is the index of the ordered set), $m = 179$, the measure remains also in the same partition.

When reaching the 81st measure, the mean has the value for example 2489. Then, the evaluation at line 23 becomes true, there is a new subset to be added to the set D_{P0} representing the partition for the parameter P0. We create the k^{th} partition for the parameter i and we initialize the mean m with the new value of $t_{i,j}$. Due to the fact that the value of the parameter is stored in the structure of the measurement, we are able to generate a new set for the value of the parameter for example 81. The partition $D_{P0,0}$ contains the values $0 \ldots 80$ and the partition $D_{P0,1}$ the values $81 \ldots 255$.

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.

Step 4: Fuzz it ! (**Algorithm 3** lines 34 to 40)

In a last step, Min value is sorted of all parameters (line 34-35) in order to deduce the evaluation order of the parameters (line 36). This has been done in the first step, but we have to do it again according to the detected hidden commands. Therefore, the fuzzing step can start. For each parameter, one value is chosen randomly in each sub-domain (line 37). After that, the command to the reader collecting the data and the status word of the card (line 38) are sent, for testing all the possible permutations.

In this paper, we consider two strategies to fuzz our application. The first one, which we call the random one, consists in fuzzing with choosing randomly one parameter from each sub-domain. Given x_i sub-domains per parameter, the total number of tests is thus $\prod_i x_i$. The second strategy, which is refereed to as the random and bound one, consists in extending the random approach by adding the fuzzing of both the lower and upper value of each sub-domain. In case the sub-domain contains less than three values, we do consider all of them. Observe that the random value is assumed to be different from the minimal and maximal bound.

In the example above, for the random strategy, we have two sub-domains for P0, three sub-domains for P1 and P2, and two for P3. This gives us a total of $2 \cdot 3 \cdot 3 \cdot 2 = 36$ tests. For the second strategy, each sub-domain now gives rise to three possible values. In this strategy, the set $\{0, rnd_1, 80, 81, rnd_2, 255\}$ is taken for P0; while for P1 we take the set: $\{0, rnd_3, 5, 6, rnd_4, 71, 72, rnd_5, 255\}$ and $\{0, rnd_6, 10, 11, rnd_7, 255\}$, for P3.

For P2 the situation is slightly complicated as one of the domains is a singleton where the three possible values coincide, leading to the set $\{0, 1, rnd_8, 90, 91, rnd_9, 255\}$. This gives rises to a total of $6 \cdot 9 \cdot 7 \cdot 6 = 2268$ tests.

5 Experiment the Timing Fuzzer on Applets

In this section, we illustrate the power of our approach on two concrete case studies, which are the EMV application and a one time password applet. We then discuss the limit of the approach.

5.1 Fuzzing EMV Applets

For our first experiment, we consider the EMV application. EMV is an open-standard set of specifications for smart card payments and acceptance devices. It appeared in 1995 in order to ensure security, global interoperability and a secure control of off-line credit card transaction approvals. Fig. 7 presents a partial view of the state machine of the protocol.

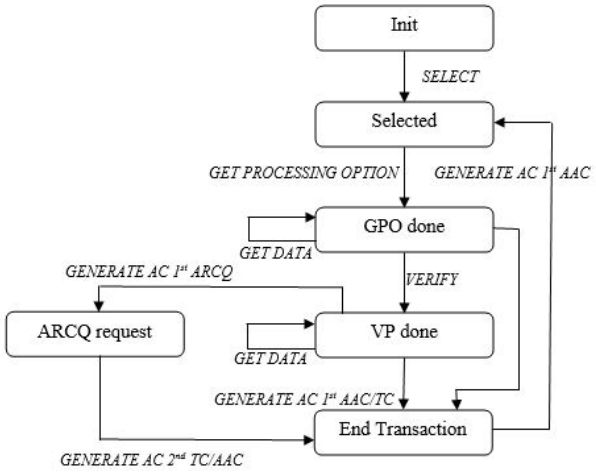

Fig. 7: EMV partial state machine.

EMV is a stateful protocol which means that some actions can only be performed if some previous commands have been processed. For example, the verification of the PIN (Personal Identify Number) holder can only occur if the terminal has sent the GET PROCESSING OPTION command. The terminal receives the card profile which indicates that the card supports card-holder verification. Then, the terminal can read the list of verification methods supported by the card using the command GET DATA. According to the result provided by the command, the terminal can perform one of the supported verification methods using the VERIFY command. The specification defines the header of the command and the data field. It indicates the verification that must be executed on the header and data while processing the command without expressing the order of the different tests.

Code	Value
CLA	00
INS	20
P1	00
P2	Qualifier of the reference data {0x80, 0x88}
LC	Size of the Data
Data	Transaction PIN Data

Table 3: Specification of the VERIFY command

We particularly focus on the VERIFY command that is coded as shown in Table 3. When the comparison between the Transaction PIN Data and the reference PIN data performed by the VERIFY command succeeds the status word is `0x9000`. Otherwise, the card shall return SW = `0x63Cn`, where n represents the number of retries still possible. When the card returns `0x63C0`, no more retries are left, and the card shall be blocked. Any subsequent VERIFY command applied in the context of that application shall then fail with status word = `0x6983`. The Table 4 gives the obtained metrics.

Condition.	Execution time μs min - max
Normal Behavior	5959 - 6193
Incorrect CLA	389 - 406
Incorrect INS	155 - 425
Incorrect P1	5959 - 6193
Incorrect P2	682 - 725
Wrong PIN length	734 - 793
Wrong PIN	8823 - 9032
Not GPO processed	520 - 580

Table 4: Execution Time of the EMV VERIFY command

We first deduce the evaluation order of the parameters:

$$\begin{aligned} \text{t(CLA)} < \text{t(INS)} < \text{t(GPO)} < \text{t(P2)} < \text{t(PIN length)} \\ < \text{t(Normal)} < \text{t(Wrong PIN)} \end{aligned}$$

At Step 1, we first test P1 with an incorrect value (*i.e.*, different from 0) and observe that the response time of the function is the same than for the acceptable value (0). We can thus conclude that the command **does not test the value of its P1 parameter**. At Step 2, we fuzz all the values of P1, which comfort our observation and thus reveals a weakness in the command. We did not discover hidden commands while executing the steps 3 and 4.

A side effect of our approach is that we have also been able to verify that the internal state is effectively checked. That is, VERIFY cannot be executed before applying GET PROCESSING OPTION.

Quality of the approach From an experimental point of view, it is necessary to assess the coverage of our technique at the binary code level. The main idea is to use a white box approach to quantify the code coverage via instrumentation of the byte code. The collecting data process must be done inside the card while analyzing the results has to be done on a laptop.

Our tool first adds an array whose size corresponds to the number of basic blocks. This array is used to count the number of accesses to each block. At the end of the test, the array is retrieved from the card with the help of a specific method. This tool is an extension of the JaCoCo [35] framework for Java Card which inserts automatically probes at the class level (prior to the conversion process). With our tool, we have been able to obtain the code coverage at source level after the fuzzing phase. As a conclusion, it shows that all the branches at source level have been covered with our side channel fuzzer.

5.2 Second Evaluation: the One Time Password applet

In this section we consider the OTP (One Time Password) used to generate a keyed hash using the DES algorithm. Its implementation is given in **Algorithm** 4.

```
1  public void process(APDU apdu)
2  throws ISOException
3  {
4  if selectingApplet() then
5  |   return ;
   |   // called by the select APDU
6  end
7  byte[]cmd_ apdu = apdu.getBuffer();
8  Util.arrayFillNonAtomic(
   wy, (short)0, LEN_WY, (byte)0);
9  if  cmd_apdu[ISO7816.OFFSET_CLA] =
   ISO7816.CLA_ISO7816 then
10 |   switch cmd_apdu[ISO7816.OFFSET_INS] do
11 |   |   case INS_VERIFY
12 |   |   |   cmdVERIFY(apdu);
13 |   |   |   break;
14 |   |   endsw
15 |   |   case INS_PUTDATA
16 |   |   |   cmdPUTDATA(apdu);
17 |   |   |   break;
18 |   |   endsw
19 |   |   case INS_GETDATA
20 |   |   |   cmdGETDATA(apdu);
21 |   |   |   break;
22 |   |   endsw
23 |   |   case Default
24 |   |   |   ISOException.throwIt(
25 |   |   |   ISO7816.SW_INS_NOT_SUPPORTED);
26 |   |   endsw
27 |   endsw
28 end
29 else
30 |   ISOException.throwIt(
   |   ISO7816.SW_CLA_NOT_SUPPORTED);
31 end
32 }
```

Algorithm 4: The One Time Password code

The specification states that:

- there is only one CLA valid (P0= 0);

- 3 values for INS (P1=0x34, 0x36, 0x38);
- P2 is always `null`;
- the value of P3 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.

Some 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 correct value is 1234;
- PUTDATA: 0x00, 0x36, 0x00, 0x56, (payload) 0x2, 0x00, 0x54; this command initializes 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 initializations must have been done and the user is authenticated before sending an `otp`. The first step of our algorithm evaluates the order of each parameter. Indeed, each parameter is set up to its initial valid value (step 1), for example the second command is started with:

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, all the response times are compared for this command and the evaluation order can be sorted.

With the code given in **Algorithm** 5 we obtain:

$t_1 < t_2 < \cdots < t_n$

which indicates that the CLA is first tested, then it is the turn of the INS, and then the control flow enters the method cmdGETDATA(). The evaluation starts with P1 and the interval for P2. The content of P2 is evaluated a second time in the *switchcase*. Then, for a request of a new password (tag value is 0x57), the state data are evaluated. The internal state consists in two boolean values. The first one is the validation of the PIN thanks to a previous command VERIFY, and the second one ensures that the initialization phase is complete. At this step, the hidden command is not considered.

```
1  private void cmdGETDATA(APDU apdu) {
2  byte[] cmd_apdu=apdu.getBuffer();
   // check if P1=0
3  if cmd_apdu[ISO7816.OFFSET_P1] ≠ 0 then
4  |  ISOException.throwIt(
5  |  ISO7816.SW_WRONG_P1P2);
6  end
7  short tag = (short)
8  (cmd_apdu[ISO7816.OFFSET_P2] & 0x00FF);
9  if (tag < 0x50) || (tag > 0x57) then
10 |  ISOException.throwIt(
11 |  ISO7816.SW_WRONG_P1P2);
12 end
13 short le = (short)
14 (cmd_apdu[ISO7816.OFFSET_LC]&0x00FF);
15 switch tag do
16 |  case (byte) 0x50
17 |  |  ··· break;
18 |  endsw
19 |  case (byte) 0x51
20 |  |  ··· break;
21 |  endsw
22 |  ···
   |  // get a new NSU
23 |  case (byte) 0x57
24 |  |  if le ≠ (byte)8 then
25 |  |  |  ISOException.throwIt(
26 |  |  |  ISO7816.SW_WRONG_LENGTH);
27 |  |  end
28 |  |  if userpin.isValidated() = false then
29 |  |  |  ISOException.throwIt(ISO7816.SW_
30 |  |  |  SECURITY_STATUS_NOT_SATISFIED);
   |  |  |
31 |  |  end
32 |  |  if stateMachine ≠ INIT_DONE then
33 |  |  |  ISOException.throwIt((short)
34 |  |  |  (ISO7816.SW_DATA_INVALID + 2));
35 |  |  |  generateNSU();
36 |  |  end
37 |  |  Util.arrayCopy(wy,(short) (INDEX_MAC),
38 |  |  wy,(short)(0), (byte)8);
39 |  |  break;
40 |  endsw
41 |  case default
42 |  |  ISOException.throwIt(
43 |  |  SW_DATA_NOT_FOUND);
44 |  endsw
45 endsw
46 apdu.setOutgoingAndSend((short) 0, (short) le);
47 }
```

Algorithm 5: Generating an OTP

Thus, the hidden command is found and the partition is built. The evaluation order is known, here the lowest response time corresponds to an erroneous CLA. All the 256 values of this parameter are tried. Only two partitions are discovered, one corresponding to CLA

equals 0 and one for the rest. Similarly, the evaluation process is applied 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.

5.2.1 Results

Working in a gray box approach for validating our method allows us to retrieve the complete CFG as described in the specification of the applet. In order to obtain the OTP, we first need to verify the PIN code and then generate it. We observe that the different data evaluation times and the evaluation order correspond to the source code under evaluation. We then verify that no modification is made by the card loader to the code in terms of evaluation order. Then, we can precisely infer the time needed to process each part of the code.

Regarding the coverage of the test suites, we have observed that it not possible to generate all the test cases if choices are made during the installation phase which corresponds to different configuration of the application. We thus need to modify our coverage tool in such a way it can install/uninstall several times the application to cover all the cases of the constructor in a gray box approach. When using the tool in a black box manner, *i.e.* the code is already installed in the card, this functionality is useless.

5.3 Limit of the Approach

This approach has several limitations. The first limit is inherent to measurement precision. That is, if an application is coded in constant time, hidden commands cannot be discovered. However, we observe that having a precise knowledge on the execution time of a Java Card virtual machine is difficult, and we have never met such application.

The second limit is inherent with fuzzing and security applications, that is you may kill the card when trying an incorrect command or reaching an incorrect state or triggering a security threshold.

The third limit corresponds to object allocation at run time. That is, the time to allocate a memory block depends on the memory allocation strategy and on the previously allocated objects. This could affect the repeatability of the measurements. This effect is mitigated by the coding rules for smart card applications which imply to always reserve the algorithms (Java factory paradigm) and memory in the constructor method.

In addition, the approach 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.

We can apply our technique to other devices under the hypothesis that the execution time must be repeatable. However, if there are hardware memory caches, then there will be a bias in the measurement and thus will not be applicable. Similar problem arises if the operating system of the target supports multi-threading. Finally, observe that the fact that we are able to rebuild 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.

6 Conclusion

In this paper, we offer a new method for testing software based on timing observations. The root of our technique is to exploit timing information to classify the input data into sub-domains according to the behavior observed for specific values of the parameters. Our approach is able to discover hidden unspecified commands that may 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 can also retrieve the control flow graph of the application. The limits of the approach have been identified, and it has been tested on two applications. Validation via a coverage tool has been established.

As a future work, we plan to extend our approach to other side channel information leakages such as those found in the 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. Our first results on the reverse of Java Card applet using template recognition of EM signatures demonstrate that method invocation and return instruction are patterns which are easy to recognize. We will use this information to obtain a more accurate control flow graph.

References

1. Paul C Jorgensen. *Software testing: a craftsman's approach*. CRC press, 2013.
2. Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.
3. André Baresel, Hartmut Pohlheim, and Sadegh Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary

algorithms. In *Genetic and Evolutionary Computation—GECCO 2003*, pages 2428–2441. Springer, 2003.
4. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
5. Edward G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
6. D. Robling and E. Dorothy. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.
7. Matt Bishop and David Bailey. A critical analysis of vulnerability taxonomies. Technical report, DTIC Document, 1996.
8. Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.
9. Ari Takanen. Fuzzing: the past, the present and the future. *SSTIC*, 2009.
10. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.
11. Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In *Information Security and Cryptology—ICISC 2002*, pages 343–358. Springer, 2003.
12. Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and countermeasures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer, 2001.
13. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.
14. Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *International Conference on Smart Card Research and Advanced Applications*, pages 167–182. Springer, 1998.
15. Denis Foo Kune and Yongdae Kim. Timing attacks on pin input devices. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 678–680. ACM, 2010.
16. David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.
17. Daniel J Bernstein. Cache-timing attacks on aes, 2005.
18. Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.
19. B. Putt, E. Putt, and J.-L. Lanet. Using side channel information for improving data partitioning strategy to test smart cards. In *SAR-SSI conference*, 2014.
20. Common Criteria. *Common Criteria for Information Technology Security Evaluation-version 3.0 Rev. 2*, 2005.
21. Integrated circuit card specifications for payment systems, book 3 : Application specification, version 4.3 ed., emvco. `https://www.emvco.com/specifications.aspx`.
22. Istvan Haller, Asia Slowinska, and Herbert Neugschwandtner, Matthiasand Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 49–64, 2013.
23. Michael Eddington. Peach fuzzing platform 3, http://peachfuzzer.com/. 2004.
24. Pedram Amini. Sulley fuzzing platform, https://github.com/openrce/sulley. 2004.
25. M. Barreaud, G. Bouffard, N. Kamel, and J.-L. Lanet. Fuzzing on the http protocol implementation in mobile embedded web server. In *Caesar*, 2011.
26. J. Lancia. Un framework de fuzzing pour cartes a puce: application aux protocoles. *SSTIC*, 2011.
27. V. Guyot. Smart card the invisible bullet. *Proceeding of the 9th European Conference on Information Warfare and Security*, 2010.
28. V. Alimi. *Contribution au déploiement des services mobiles et à l'analyse de la sécurité des transactions*. PhD thesis, University of Caen, France, 2012.
29. 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.
30. T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988.
31. Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 261–272. ACM, 2009.
32. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 283–294. ACM, 2011.
33. Amaury Gauthier, Clement Mazin, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Enhancing fuzzing technique for okl4 syscalls testing. In *ARES*, pages 728–733. IEEE, 2011.
34. Mohammed Amine Kasmi, Mostafa Azizi, and Jean-Louis Lanet. Reversing bytecode of obfuscated java based smart card using side chanel analysis. *International Journal of Security and Its Applications*, 9(11):347–356, 2015.
35. Jacoco java code coverage. `http://eclemma.org/jacoco/`.