Protection of systems against fuzzing attacks

Léopold Ouairy, Hélène Le Bouder, Jean-Louis Lanet

Protection of systems against fuzzing attacks

Léopold Ouairy[1], Hélène Le-Bouder[2], and Jean-Louis Lanet[3]

[1] INRIA, Rennes, France leopold.ouairy@inria.fr
[2] IMT-Atlantique, Rennes, France helene.le-bouder@imt-atlantique.fr
[3] INRIA, Rennes, France jean-louis.lanet@inria.fr

Abstract. A fuzzing attack enables an attacker to gain access to restricted resources by exploiting a wrong specification implementation. Fuzzing attack consists in sending commands with parameters out of their specification range. This study aims at protecting *Java Card* applets against such attacks. To do this, we detect prior to deployment an unexpected behavior of the application without any knowledge of its specification. Our approach is not based on a fuzzing technique. It relies on a static analysis method and uses an unsupervised machine-learning algorithm on source codes. For this purpose, we have designed a front end tool *fetchVuln* that helps the developer to detect wrong implementations. It relies on a back end tool *Chucky-ng* which we have adapted for *Java*. In order to validate the approach, we have designed a mutant applet generator based on *LittleDarwin*. The tool chain has successfully detected the expected missing checks in the mutant applets. We evaluate then the tool chain by analyzing five applets which implement the *OpenPGP* specification. Our tool has discovered both vulnerabilities and optimization problems. These points are then explained and corrected.

Keywords: unsupervised machine-learning; k-Nearest-Neighbors; vulnerability detection; fuzzing attacks, Java Card, Chucky

1 Introduction

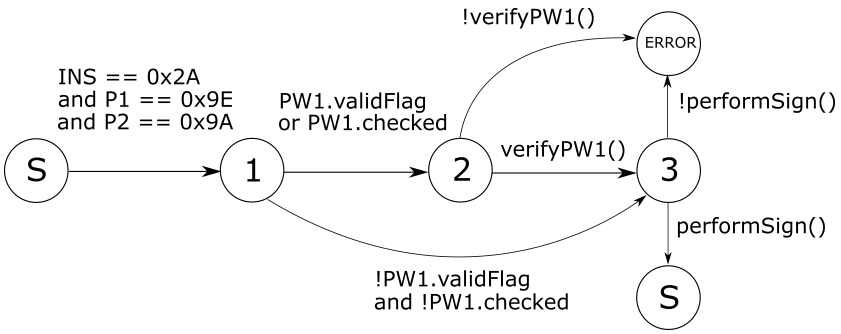
A fuzzing attack aims at sending crafted messages to a running program in order to test all the possible paths of its control flow. With this method and according to the system response, an attacker can detect a deviation of the program's expected behavior, in response to his message. This same crafted message can perturb the program's state machine and change its current state. By modifying the state of the program, an attacker can illegally gain access to resources stored onto the *Java Card*. One of the reasons for this illegal transition in the state machine can be related to the absence of input validation tests. Such a forgotten condition is called a *missing-check*. With this work, we aim at detecting those *missing-checks* before a fuzzing attack.
Our approach is not based on a fuzzing attack, but on a static analysis of the source codes. To achieve this task, we have created three tools. The first one, *ChuckyJava* is an adaptation of *Chucky-ng*[17,8] for *Java*. The second tool we have designed is *fetchVuln*. It is a front-end layer above *ChuckyJava* which aims at automating tests of *ChuckyJava*. Moreover, it gathers all the outputs that *ChuckyJava* generates and it processes them to produce a report about vulnerable methods of the applet under analysis.

The last tool is *LittleDarwinJC* which is based on *LittleDarwin*[10]. Its objective is to generate *Java* mutant applets and it enables us to characterize and evaluate the ability of *fetchVuln* to detect those mutants. We have tested *fetchVuln* on an applet set implementing *OpenPGP*. This evaluation has brought to the fore two optimization problems while we have discovered two vulnerabilities in the implementations.
The context is presented in section 2. The state of the art is exposed in section 3. Section 4 explains the functioning of *Chucky-ng*. The adaptation phase is shown in section 5. We expose our methodology in section 6. Then, we describe our results and discuss the evaluation of the performances in section 7. The limitations of our tool are exposed in section 8. To finish, we present our conclusion in section 9.

2 Context

A program with a state machine accepts only a set of commands from a specific state. Such commands enable the program to change its state from one to another. A specification clarifies both states and transition links that shall be implemented for a program. *OpenPGP* is an open version of the Pretty Good privacy (*PGP*) standard defining encryption formats. Fig. 1 shows a fragment of the *OpenPGP* state machine specification with the command COMPUTE_DIGITAL_SIGNATURE. It allows or denies the data to be signed before being written to the card. The *S* node (state *Selected*) is where the program accepts messages. During the process, the state checks the value of the incoming message in order to determine which command to execute. This is represented by the test on transition from state *S* to the state *1*. The state machine then checks the parameter *PW1* (*Password 1*) if necessary. If *verifyPW1()* fails, then the state machine enters the *ERROR* state. If the test succeeds, the applet can perform the signature operations. By entering in the state *S*, the program waits for another command. To illustrate a *missing-check*, suppose that the condition *verifyPW1()* from node *2* to node *3* is missing does not exist. It is possible for an attacker to enter state *3* with a wrong *PW1*.

Fig. 1: Partial state machine of the *COMPUTE_DIGITAL_SIGNATURE*. There are conditions to transition from a state to another. States are represented by the nodes. The *S* node is the *Selected* state. Source from the *OpenPGP* specification[11]



3 State of the art

Dynamic and hybrid protection Dynamic and hybrid protection techniques consist in filtering user controlled inputs. We have found this kind of input cleaning on web application domains. As an example, the XSS attack mitigation which uses (*UrlEncoder* or *HtmlEncoder*[9] for example) to filter user controlled inputs. This step has to ensure that the message format and its application domain are correct. One drawback of this mitigation method is the possibility of missing cases against a new attack and it needs to be updated to improve the filtering accuracy. If an attacker has access to the source code of the library, then he can adapt its inputs to bypass the secure filter. In her PhD[5], Kamel proposes to implement a filtering library *API JCXSSFilter*. She chooses to adapt it to fit in the *OWASP*'s *ESAPI*[1] open-source web application. Added to this, the use of such secure libraries add an extra overhead in the system.

Formal methods The tools *Z*[14], *VDM*[3] rely on formal methods. They aim to mathematically specify the expected behavior of a sequential system by using sets, relations and functions. Burdy *et al.*[2] present an experiment on formal validation of *Java Card* applets. To specify a behavior, the user has to annotate his *Java* classes with the *Java Modeling Language*. One of the drawbacks of this method is the necessity to specify the right behavior for all classes. Depending on the size of the project, this step can be difficult. Added to this, if the specification of the applet changes, the developer has to adapt his code in order to fit it to the new expected behavior.

Static Analysis A concolic analysis performs both concrete and symbolic analysis for a given source code. This is the case for the tool *JDart*[7] which relies on this kind of static analysis. For instance, the symbolic execution aims at discovering the paths a program can follow, while the concrete one proposes valid inputs to use in order to follow a specific path.
In the static analysis field, taint analysis techniques aims at following the evolution of an input through a program. To illustrate this method, suppose that one wants to follow a parameter. To do this, the tool has to taint this variable. Then, every variable which uses this tainted parameter gains the same taint color. *Pixy*[4] uses taint analysis. It aims at propagating limited string value information in order to handle some of *PHP*'s most dynamic features. One drawback of a concolic analysis is that if there is a lot of paths, then the analysis could never finish its execution.

Text mining Text mining is a technique which consists in extracting the terms (words) present in software components (files) and their frequencies. Then, the term frequencies are correlated in order to build a model which predicts if a given software component is vulnerable. This method is implemented in the tool of Scandariato *et al.*[13].

Machine learning The tool entitled *Chucky-ng*[8], based on *Chucky*[17], relies on machine-learning to discover vulnerabilities in a source code. It extracts and compares functions with a unsupervised machine-learning algorithm in order to flag the vulnerable ones. *Chucky-ng* relies on the *k-Nearest-Neighbors*[12] algorithm. An important advantage for this tool is that *Chucky-ng* does not require to calibrate the tool with any sort of training step.

3.1 Our contribution

At the best of our knowledge, no tool for mitigation against fuzzing attacks on *smart cards* is publicly available. We do not want to create a *smart card* fuzzer to extract vulnerabilities for two reasons. The first one is the required time to send a command from a terminal to a *smart card* is time expensive. We want our tool to perform an analysis in the best delays. The second reason is that *smart cards* contains Non Volatile Memory (*NVM*). Such memory has a short life expectation. Therefore, sending too many commands to the card would trigger many write functions and then prematurely aging the card.
We have created both *fetchVuln* and *ChuckyJava*. The former is the front-end of *Chucky-Java*. The latter allows the parsing of *Java* source codes and perform on them the unsupervised machine-learning technique of *Chucky-ng*.

4 Description of *Chucky-ng*

Chucky-ng takes three inputs:

- the folder of applet's source codes to analyze,
- one or more *API* symbols, such as variable names, parameter names or method names,
- the number of neighbors to select *k*.

The *API* symbols are the element that *Chucky-ng* searches in every method of the source code. An analyst can add several *API* symbols in the analyze queue. Therefore, in order to be added to the methods to analyze, this same method must manipulate all of these *API* symbols. The necessity of the *k* parameter is explained in the *Neighborhood discovery* step. Once *Chucky-ng* succeeds its execution, it returns an anomaly score for each function containing at least one of the *API* symbol. The tool processes the anomaly score in four distinct steps: The parsing, the neighborhood discovery, the reduction of the vector's dimensions and the anomaly detection.

4.1 Parsing

The tool named *Joern*[16], first parses *Java* files in the applet's folder source code to analyze. Based on this, it creates a *Code Property Graph*[17]. This graph gathers all nodes and links of the abstract syntax tree, the control flow graph and the data flow graph into one single graph. This same graph focuses mainly on the representation of functions from the source code.

4.2 Neighborhood discovery

During this step, *Chucky-ng* creates a set of functions which contain at least one *API* symbol under analysis. This is done by running through the *Code Property Graph*. From this set, *Chucky-ng* picks one function. Then, it represents every function of the set as

vectors of dimension $|$*API* $symbols|$. It uses the machine-learning algorithm *k-Nearest-Neighbors*[12] on these vectors in an unsupervised way, in order to gather the most k similar neighbors to the picked one. This value of k is required as input and it determines the number of similar functions to gather. The *k-Nearest-Neighbors* algorithm is split into two steps.

The first step aims at increasing the selection of neighbor's accuracy. To do so, *Chucky-ng* uses an approach based on *Inverse Document Frequency (IDF)* in order to discriminate rares *API symbols*. It insists on the fact that rare symbols are meaningful compared to others used by the vast majority of the function set. Two functions using the same rare *API* symbols should be more similar than two functions using only regular *API* symbols. *Chucky-ng*'s set of vectors now contains these new values based on the *IDF*.

The second step consists in gathering the *k* most similar functions from the one picked at the beginning of the *Neighborhood discovery*. To do so, *Chucky-ng* processes the vectors using the cosine distance metric where x and y are vectors *API* symbols values restricted by the *IDF* filtering:

$$cos(\boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}|| \cdot ||\boldsymbol{y}||}. \tag{1}$$

The use of this distance metric is twofold. Since *Chucky-ng* needs to consider rare *API* symbols, this metric takes into account both the orientation and the *Euclidean* distance of vectors.
Since *Chucky-ng* has the similar methods set to the one picked, it does not need the vectors anymore. Instead, *Chucky-ng* uses new vectors whose dimensions are the number of the total number of expressions used in every method of the set. Mind that the *Neighborhood discovery*, the *Reduction of vector's dimension* and the *Anomaly detection* steps are repeated as many times as there are methods in the set.

4.3 Reduction of the vector's dimension

This step objective is to reduce the dimension of the method set. It achieves this by removing tests of the program which does not manipulate any of the *API* symbols. For example, if a control structure does not uses this *API* symbol either in its condition or body, *Chucky-ng* discards this control structure for the rest of the algorithm. It reduces the total number of expressions and therefore the dimension of vectors. To do so, *Chucky-ng* relies on the *Code Property Graph* in order to taint the *API* symbol evolution through the source code.
Added to this, *Chucky-ng* manages to reduce the dimensions of the vectors by normalizating the remaining expressions. It suppresses minor syntactical differences. As an example, the binary relational operators ($\leqslant$, $<$, $\geqslant$, $>$) are now replaced by $CMP expression. It affects the arguments, the return value of callees (*callees*) and the conditions of control structures. For example, the expression $if(ident \leqslant 1)$ is normalized as $if(ident \quad \$CMP \quad \$NUM)$. It reduces the impact of small syntactical differences.

4.4 Anomaly detection

During this step, *Chucky-ng* creates a model of normality. This object is a vector whose dimensions are each of the remaining expressions. For each dimension, the value is the average presence for this expression in all the functions of our set as shown in equation (3). Let E be the normalized expression set and X the set of neighbor functions. $\varphi(x)$ is the mapping function that transforms neighbors in a vector space. The function $I(x, e)$ is equal to 1 if neighbor x contains the expression e. Otherwise I is equal to 0.

$$\varphi : X \to \mathbb{R}^{|E|}, \varphi(x) \mapsto (I(x, e))_{e \in E}. \tag{2}$$

Let μ be the vector of normality and N the neighbor set.

$$\mu = \frac{1}{|N|} \sum_{n \in N} \varphi(n), \mu \in \mathbb{R}^{|E|}. \tag{3}$$

Chucky-ng then creates a distance vector d as in equation (4) which corresponds to the values of the normality model minus the values of our function vector under test. The distance vector is in fact the list of the anomaly scores for each expression.

$$d = \mu - \varphi(x), d \in \mathbb{R}^{|E|}. \tag{4}$$

To finish, the anomaly score for the function under analysis is the maximum value of the distance vector as shown in equation (5). Its range is from [-1.00, 1.00]. If the anomaly score is closer to 1.00, our function is more likely to omit an expression, compared to the other functions. On the contrary, if it is closer to -1.00, our function has an expression that none of the others perform.

$$Score = max(d). \tag{5}$$

Even if *Chucky-ng* precises the anomaly score for a whole function, in practice, we observe all of its expression's anomaly scores. For example, if a function's anomaly score is set to 1.00 for an expression, all scores ranked under 1.00 are hidden by the result of *Chucky-ng*. To be efficient, an analyst has to read the whole distance vector including the -1.00 expressions. As we have discovered in the methodology (section 6), an anomaly score of -1.00 is meaningful in our case: it corresponds to an *extra-check*. In other words, the applet may accept an unwanted additional command in the state machine.

5 Adaptation

5.1 From *C/C++* to *Java*

Java shares notions that are similar to those in *C++*. For instance, the *class abstraction* and *virtual methods* are similar. While such notions are shared with *C++*, we have syntactically adapted them in order to fit to the *Joern*[16], the parser. To achieve this task, we have modified *Joern* and *Chucky-ng* in order to link them together to the functionalities we have implemented. We now explain how we handle those specificities.

Abstract classes Methods which are declared in an abstract class in *Java* do not have definition. Since such methods are defined in another *Java* source file, *ChuckyJava* discards methods only declared. This prevent the use of duplicates methods.

Virtual methods If a class inherits of methods defined in the *Java API*, than *ChuckyJava* cannot parse them. To prevent this, we have created a tool that gathers both used and defined classes in the source code, and then it warns the user if a class is used but is never declared. This can happen if the user calls an object constructor from the *Java's API*.

New expressions We have to take into account new expressions or control flow structures. As an example, in *C++*, the *try/catch* clause does not have the *finally* block. Since this last one is executed either if the *catch* is triggered or not, we have decided to treat it as code block, outside of the control flow of the *try/catch*. Other ways to iterate exists in *Java* and not in *C++* such as the iteration over a list. For example, *for(Element e: elements) { [...] }*.

New operators We have implemented two binary operators which exist in *Java* but not in *C/C++*. Those operators are the structure comparison operator of strict equality $===$ and the bit shift to the right, which includes the sign byte $>>>$.

Switch/cases *Chucky-ng* does not handle and detect missing *cases* in a *switch/case* structure. *ChuckyJava* takes into account the missing *cases* in order to detect this kind of *missing-check*. It means that *ChuckyJava* can now return an anomaly score for such tests. Every applet has to declare a *process* method in order to handle the reception of a message from an external source. In many cases, they use a *switch/case* to handle the instruction byte and launch the expected operations. In *Java Cards*, *switches* are important since they inform us about the allowed or denied commands of an applet.

5.2 ChuckyJava

We modify the *ChuckyJava*'s algorithm in order to improve its accuracy during the *Neighborhood discovery* step.

Object Type For the Neighborhood Discovery step, we now take into account two new *API symbols*. An object's cast type and the type of an object if it is created as a parameter of a method call. To illustrate the cast, from the expression *(byte) 0x0F*, we now extract the *byte* type and we include it in the vector's dimension. For the second new *API* symbol, the expression *call(new OwnerPin(0x03,0x04))* corresponds to the object creation inside a method call. From now on, the type *OwnerPin* would be included in the vector's dimension too. These modifications improve *ChuckyJava*'s ability to gather similar functions with a better accuracy.

6 The methodology

This section presents the test framework we have created. It includes the three tools *fetchVuln, LittleDarwinJC, ChuckyJava*. Fig. 2 shows the relation between those tools.

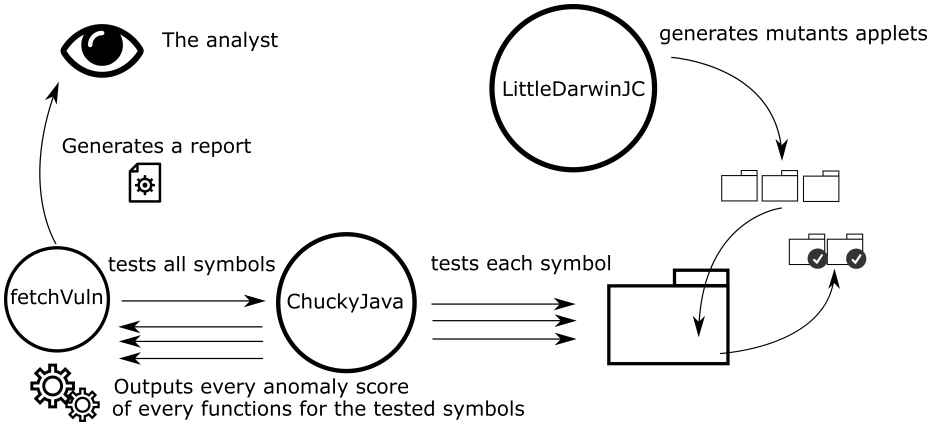


Fig. 2: *fetchVuln* requests *ChuckyJava* to test every *API* symbols. For each analyzed symbols, the output from *ChuckyJava* is stored and processed by *fetchVuln*. *LittleDarwinJC* produces different sets from the original ones, each of them containing one single mutation

6.1 FetchVuln

We have created the tool *fetchVuln*. This front-end's objective is to help an analyst to discover vulnerabilities easier. This tool operates in two distinct steps. It can be seen as a top layer over *ChuckyJava*.

Firstly, it lists all *API* symbols used within all the applets and their usage. Based on this, the tool is able to perform an analysis with *ChuckyJava* for each of these symbols. This is useful since *ChuckyJava* requires to specify one or multiple *API* symbols. Once *ChuckyJava* returns its output containing the anomaly score for each selected methods, *fetchVuln* stores it. It is stored as a list of triplets: method, *missing-check* (or feature) and anomaly score.

Secondly, when the execution of *ChuckyJava* for each *API* symbol is done, it outputs a report. Based on a configurable output filter value, this report contains the anomaly scores, the method names, the locations in the source code and the *missing-check* associated. This is readable by an analyst and it sorts the result from the highest anomaly score 1.00 to the defined output filter value. Moreover, our tool feedbacks statistics. For instance, it is able to output the number of *API* symbols for which we do not have enough similar neighbors. This can happen if a precise parameter name is used in too few functions. Another example of statistics that *fetchVuln* outputs is the number of functions with anomalies compared to total number of functions.

6.2 LittleDarwinJC

Thanks to *fetchVuln*, we are now able to output automatically a vulnerability report for an applet folder. We validate the adaption of *Chucky-ng* to *Java*. Secondly, we want

to verify *fetchVuln* capacity to detect single variation of conditions in a source code. It enables us to characterize our tool. To do this, we have chosen to adapt the tool *LittleDarwin*[10]. We base our tool *LittleDarwinJC* on it since it generates mutations on the source code instead of the bytecode of a *Java* file. Then, we create a folder of one applet[4]. This same applet is duplicated five times inside the folder. *LittleDarwinJC* creates many copies of the original applets folder as there are existing conditions in the applet. On each folder, it removes one different single condition. Therefore, each applet folder is now a mutant and none of these mutants is duplicated. Then, a temporary tool we have created requests *fetchVuln* to perform an analysis on every mutant applet folders. If its final report shows an abnormal method with an anomaly score set to 1.00, then the mutant has been found by *fetchVuln*. We conclude from our tests that *fetchVuln* is able to detect *missing-checks*.
An *extra-check* is the anomaly where only one applet of the set performs a tests, but none of the others. A *missing/extra-assignment* is similar to *missing/extra-checks*, but on variable assignments. After this first step, we have tested the tool ability to discover other varieties of anomaly. From our results, *fetchVuln* is able to detect *extra-checks* and *missing/extra-assignments* too. Moreover, since we implemented the *switch-case* as conditions in *ChuckyJava*, *fetchVuln* is able to detect *missing-checks* within the *cases*.

7 Evaluation

7.1 Vulnerability results

Description of the dataset Our dataset is made of five different *Java Card* applets, all implementing the *OpenPGP*[11] specification. Among these applets, we can find two different versions of *OpenPGP*: the 2.0.1 one, and the 3.3.1 one. These applets and their *OpenPGP* versions are listed in Table 1.

Applet name	OpenPGP version
FluffyPGP	2.0.1
JCOpenPGP	2.0.1
MyPGPid	2.0.1
OpenPGPCard	2.0.1
SmartPGP	3.3.1

Table 1: *OpenPGP* applets and their implementation versions

To communicate from a card terminal to the *Java Card*, one has to send an *APDU* object which contains the communication information. Among the bytes sent during this process, we present three *APDU*'s header bytes: the CLA byte, the INS byte and the P1 byte. The CLA (Class) byte is used to define the interindustry class. The second one is the INS (Instruction) which enables the applet to determine which command the user wants to perform. The last one, the P1 (Parameter 1) byte, is a parameter for the instruction command. We analyze the results of *fetchVuln* and therefore *ChuckyJava*.

[4] https://github.com/FluffyKaon/OpenPGP-Card

A useless check We have executed an analysis on the callee *JCSystem.makeTransient-ByteArray*. We set the value of the number of neighbors to select (parameter k) to 4. Since all applets implement a *process* method, this value of k enables us to gather all the *process* functions in order to compare them with *ChuckyJava*. Listing 1.1 shows the output we obtain with *ChuckyJava* for the *PGPKey* constructor in the *PGPKey.java* file.

Listing 1.1: The anomaly score, the expression, the function and file concerned

```
-1.00    "( null $CMP $RET )" PGPKey PGPKey.java:38
```

Since the anomaly score is evaluated to -1.00, it means that the test is performed in this method while it is not in other similar methods. We can verify this assumption with this code snippet available in the appendix A. At this point in the applet, *tmpBuf* has been declared but no memory has been allocated, this it points on null. The evaluation is always *true*. *ChuckyJava* detects here a useless test which can be eliminated.

Dead code detection We are able to detect dead code with *ChuckyJava*. Listing 1.2 shows the output we obtain for analysis of the *process* method of the *MyPGPid* applet.

Listing 1.2: *ChuckyJava* output for *MyPGPid*

```
-1.00    "case ISO7816.INS_SELECT"          process MyPGPid.java
```

ChuckyJava returns an anomaly score of -1.00. It means that the *process* function performs a *case ISO7816.INS_SELECT* which is not in other *process* functions. By analyzing the code of appendix B, we can see a test performed at the beginning: *selectingApplet()*. Both this test and the *case ISO7816.OFFSET_INS* check the value of the *APDU*'s INS byte. Moreover, both of them returns immediately if this byte's value is *0xA4*. This explains why the second test in the *switch* can not be reached and is therefore not necessary.

A misuse of the CLA byte By analysing the *ChuckyJava* output for the *process* method, we are able to detect a misuse of the CLA byte. This same byte has to be checked before usage, as requested in the *OpenPGP* specification[11]. In the *MyPGPid* applet, this byte is tested by a bitwise *AND* and the value 0xFC, but not in the other applets.

Listing 1.3: misuse of the CLA byte

```
-1.00    "buffer[ISO7816.OFFSET_CLA] = (byte)(buffer[ISO7816.OFFSET_CLA] & (byte) 0
    ↪ xFC)   process MyPGPid.java:347
```

We have discovered that this assignation performed, but the value of *buffer[ISO7816.-OFFSET_CLA]* is pratically never checked in the source code. The value is verified once in a method. Nearly every values for the CLA byte are possible for this applet. An attacker could be able to exploit it to make the program unexpectedly enter in a new state. The *OpenPGP* specification stipulates that this CLA byte shall be verified in order

to allow only specific values (most of the time: 0x00, 0x0C, 0x10 or 0x1C according to the current state).

A Missing-Check for the P1 byte We have discovered an anomaly for the P1 byte. We have investigated the *verify* functions of our applet set. The output of *ChuckyJava* in listing 1.4 warns us about an anomaly existing in the *OpenPGPApplet.java* file. This anomaly shows the *verify* function which does not use nor check the P1 byte while the others do it.

Listing 1.4: Missing-check of P1 byte

```
0.67    "buffer[ISO7816.OFFSET_P1] $CMP (byte)($NUM)"    verify   OpenPGPApplet.java
    ↪ :413
```

According to the specification for the *verify* method, the P1 byte shall be set to 0x00 for this precise command. This is a *missing-check* because it is performed in the *verify* functions of the other applets of the set. The code snippet for *OpenPGPApplet* is available in appendix C. The anomaly score of 0.67 instead of 1.00 is due to a limitation which we discuss in section 8.

7.2 Performance evaluation

A quality assurance tool (*QA*) in *smart cards* aims at verifying if all the specified commands are implemented (conformance testing). Our tool *fetchVuln* is not a *QA* tool. It detects if the applet has more functionalities than expected. To illustrate this purpose, imagine an applet which is perfectly implementing the *OpenPGP* specification. Each command a user send to the applet returns the expected output. We now suppose that a back-door exists in the applet. A *QA* tool is not able to detect it, because the applet performs perfectly according to its specification. However, since *fetchVuln* detects *extra-checks*, it is able to discover such an anomaly in an applet.

7.3 Time overheads

This performance evaluation focuses on the advantages of using *fetchVuln* instead of a physical verification.
To test a *Java* applet, an analyst can first generate several different messages. Then, he sends them individually to the *smart card* to obtain the output. Finally, he verifies that the combination input/output is conform to the specification. The drawback of physically testing the inputs is the communication cost between a *smart card* and the terminal. It is time expensive. We have tested the method and the process has last roughly one second to transmit a single command to the applet *FluffyPGPApplet*. The time required with such an analysis, increases with the number of states which the specification plans. As a comparison, *fetchVuln* requires about 3 minutes to analyze a set of five applets implementing the *OpenPGP* specification, including *FluffyPGPApplet*. This analyzes was performed on a *Intel i7-7600U* 2.8GHz CPU, using two of the four threads available in a virtual environment.

8 Limitations

8.1 Number of missing-checks

During the fourth step, *ChuckyJava* creates vectors which contain information about the presence or not of a normalized expression only. For example, the condition *if (k == 3)* is normalized as *(k $CMP $NUM)*. If there are multiple conditions comparing *k* with a number, the tool normalizes them with the same expression. Then, the value for the dimension *(k $CMP $NUM)* is equal to one regardless of the number of comparison. We can see in Listing 1.5 that two tests are executed. However, only one is performed in Listing 1.6. After the normalization step, both vectors representing the code snippet have the dimension *(k $CMP $NUM)*. Its value is equal to one in both vectors. This leads to a non-detection of some *missing-checks*.

Listing 1.5: Two comparisons of number with *k*

```
public void test(int k)
{
  if(k <= 1)
        callee(1);
  else if(k <= 2)
        callee(2);
}
```

Listing 1.6: Vulnerable code

```
public void test(int k)
{
  if(k <= 1)
        callee(1);
  //Missing check...
  callee(2);
}
```

8.2 Missing distinction between variables and constants

We are able to observe that in some cases it is not possible for *ChuckyJava* to distinguish a variable from a constant. Listing 1.7 shows an initialization with a variable. This same variable could be uninitialized at this point but there is no test to verify it. In Listing 1.8, the same function call uses a constant as a parameter. Since it is a constant, it is more likely to be defined in one of the project classes. In this example, its value is set to *10*. It is not necessary to test its value before using it. However, if the other applets use a variable as like in Listing 1.7, then *ChuckyJava* detects a wrong *missing-check* here for code snippet in Listing 1.8.

Listing 1.7: Local variable

```
if(my_variable > 0)
        myInitMethod(my_variable);
```

Listing 1.8: Constant

```
private static final int MY_CONSTANT = 10;
myInitMethod(MY_CONSTANT);
```

Added to this, *ChuckyJava* is not able to handle differences between identifiers. For example, we want *ChuckyJava* to analyze the identifier *buffer*. Then, all similar identifiers (*tmpBuff*, *buf*, etc.) are discarded. To perform an accurate analyze with our tool, we first have to normalize the used identifiers, which is one of our current research directions.

9 Conclusion

We have designed a tool chain which includes *ChuckyJava*, *fetchVuln* and *LittleDarwinJC*. It aims at detecting incorrect implementations of specification within *Java Card* applets. We have improved the original tool by adding various features. *fetchVuln* has successfully detected mutant applets generated with *LittleDarwinJC*. Added to this, we have discovered that our tool is able to detect *extra-checks* and *missing-assignements* too. In real conditions, *fetchVuln* is able to detect wrong implementations specification. However, the tool has two limitations. We are currently working on the identifier problem since it is the most restrictive one. Because it is a known problem, we have found different methods to start with. We are heading to source code de-obfuscation and source codes merging techniques[6,15] to solve this limitation.

References

1. Owasp enterprise security api (2009) 3
2. Burdy, L., Requet, A., Lanet, J.: Java applet correctness: A developer-oriented approach. International Symposium of Formal Methodes Europe (2003) 3
3. Jones, C.: Systematic software development using vdm. vol. 2. Orentice-Hall Englewood Cliffs, NJ (1986) 3
4. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities. Security and Privacy, IEEE Symposium (2006) 3
5. Kamel, N.: Sécurité des cartes à puce à serveur web embarqué. PhD thesis, Université of Limoges (2012) 3
6. Kuhn, A., Ducasse, S., Girba, T.: Semantic clustering: Identifying topics in source code 13
7. Luckow, K., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., Raman, V.: Jdart: A dynamic symbolic analysis framework 3
8. Maier, A.: Assisted discovery of vulnerabilities in source code by analyzing program slices 1, 3
9. OWASP: Xss (cross site scripting) prevention cheat sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet 3
10. Parsai, A., Demeyer, S., Murgia, A.: Littledarwin: a feature-rich and extensible mutation testing framework for large and complex java systems 2, 9
11. Pietig, A.: Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems 2, 9, 10

12. Sayad, S.: K nearest neighbors. http://chem-eng.utoronto.ca/ datamining/Presentations/KNN.pdf 3, 5
13. Scandariato, R., Walden, J., A., H., W., J.: Predicting vulnerable software components via text mining 3
14. Spivey, J.: Understanding z: a specification language and its semantics, vol. 3. cambridge university press 1988 3
15. Tairas, R., Gray, J.: Phoenix-based clons detection using suffix trees 13
16. Yamaguchi, F.: Joern. http://mlsec.org/joern/ 4, 6
17. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: Exposing missing checks in source code for vulnerability discovery 1, 3, 4

A PGPKey constructor

```
private static byte[] tmpBuf;
[...]
public PGPKey() {
  key = new KeyPair(KeyPair.ALG_RSA_CRT, KEY_SIZE);
  fp = new byte[FP_SIZE];
  Util.arrayFillNonAtomic(fp, (short) 0, (short) fp.length, (byte) 0);
  Util.setShort(attributes, (short) 1, KEY_SIZE);
  Util.setShort(attributes, (short) 3, EXPONENT_SIZE);
  //The useless check
  if(tmpBuf == null) {
   tmpBuf = JCSystem.makeTransientByteArray((short) (KEY_SIZE_BYTES / 2), JCSystem
       ↪ .CLEAR_ON_DESELECT);
  }
 }
```

B Process function of MyPGPid applet

```
public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    short lc;
    boolean status = false;

    // ignore the applet select command dispached to the process
    if (selectingApplet()) {
        return;
    }

    buffer[ISO7816.OFFSET_CLA] = (byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);

    if (buffer[ISO7816.OFFSET_INS] == GET_RESPONSE) {
        if (remainingDataLength <= 0) {
            ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
        }
        else { sendData(apdu, tmpData, remainingDataLength);}
        return;
    } else {
        remainingDataLength = 0;
        remainingDataOffset = 0;
    }

    switch (buffer[ISO7816.OFFSET_INS]) {
        case ISO7816.INS_SELECT:
            return;
        case GET_DATA:
```

```
        getData(apdu);
        return;
    case PUT_DATA:
        putData(apdu);
        return;
    case PUT_DATA_CHAINING:
        putDataChaining(apdu);
        return;
    case VERIFY:
        if (buffer[ISO7816.OFFSET_P1] != 0) { ISOException.throwIt(ISO7816.
            ↪ SW_WRONG_P1P2); }
        lc = apdu.setIncomingAndReceive();
        if (lc == 0) {
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }
        switch (buffer[ISO7816.OFFSET_P2]) {
            case (byte)0x81:
                if (chv1.getTriesRemaining() == (byte)0) {
                    ISOException.throwIt(SW_PIN_BLOCKED);
                }
                status = chv1.check(buffer, (short)ISO7816.OFFSET_CDATA, (byte
                    ↪ )lc);
                break;
            case (byte)0x82:
                if (chv2.getTriesRemaining() == (byte)0) {
                    ISOException.throwIt(SW_PIN_BLOCKED);
                }
                status = chv2.check(buffer, (short)ISO7816.OFFSET_CDATA, (byte
                    ↪ )lc);
                break;
            case (byte)0x83:
                if (chv3.getTriesRemaining() == (byte)0) {
                    ISOException.throwIt(SW_PIN_BLOCKED);
                }
                status = chv3.check(buffer, (short)ISO7816.OFFSET_CDATA, (byte
                    ↪ )lc);
                break;
            default:
                ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        if (!status) { ISOException.throwIt(ISO7816.
            ↪ SW_SECURITY_STATUS_NOT_SATISFIED);}
        return;
    case GENERATE_ASYMMETRIC_KEY_PAIR:
        generateAssymetricKeyPair(apdu);
        return;
    case PERFORM_SECURITY_OPERATION:
        performSecurityOperation(apdu);
        return;
    case CHANGE_REFERENCE_DATA:
        /* Fall through */
    case RESET_RETRY_COUNTER:
        changeResetChv(apdu);
        return;
    case INTERNAL_AUTHENTICATE:
        if (buffer[ISO7816.OFFSET_P1] != 0 || buffer[ISO7816.OFFSET_P2]
                != 0) {
            ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        if (!chv2.isValidated()) {
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }
        lc = receiveData(apdu, tmpData);
        sig.init(keyAuth.getPrivate(), Cipher.MODE_ENCRYPT);
        lc = sig.doFinal(tmpData, (short)0, lc, tmpData, (short)0);
        sendData(apdu, tmpData, lc);
        return;
    case GET_CHALLENGE:
```

```
            if (buffer[ISO7816.OFFSET_P1] != 0 || buffer[ISO7816.OFFSET_P2]
                    != 0) {
                ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
            }
            lc = apdu.setOutgoing();
            random.generateData(tmpData, (short) 0, lc);
            apdu.setOutgoingLength(lc);
            apdu.sendBytesLong(tmpData, (short) 0, lc);
            return;
        //case EXPORT_KEY_PAIR:
            //exportKeyPair(apdu);
            //return;

        case INS_CARD_READ_POLICY:
            ReadPolicy(apdu);
            return;
        case INS_CARD_KEY_PUSH:
            KeyPush(apdu);
            return;

        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
}
```

C Verify function of OpenPGPApplet

```
private void verify(APDU apdu, byte mode) {
 if (mode == (byte) 0x81 || mode == (byte) 0x82) {
  // Check length of input
  if (in_received < PW1_MIN_LENGTH || in_received > PW1_MAX_LENGTH)
   ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

  // Check given PW1 and set requested mode if verified succesfully
  if (pw1.check(buffer, _0, (byte) in_received)) {
   if (mode == (byte) 0x81)
    pw1_modes[PW1_MODE_NO81] = true;
   else
    pw1_modes[PW1_MODE_NO82] = true;
  } else {
   ISOException
   .throwIt((short) (0x63C0 | pw1.getTriesRemaining()));
  }
 } else if (mode == (byte) 0x83) {
  // Check length of input
  if (in_received < PW3_MIN_LENGTH || in_received > PW3_MAX_LENGTH)
   ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

  // Check PW3
  if (!pw3.check(buffer, _0, (byte) in_received)) {
   ISOException
   .throwIt((short) (0x63C0 | pw3.getTriesRemaining()));
  }
 } else {
  ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
 }
}
```