Confiance: detecting vulnerabilities in Java Card applets

Léopold Ouairy, Hélène Le Bouder, Jean-Louis Lanet

Confiance: detecting vulnerabilities in Java Card applets

Léopold Ouairy
INRIA
Rennes, France
Email:leopold.ouairy@inria.fr

Hélène Le Bouder
IMT-Atlantique
Rennes, France
Email: helene.le-bouder@imt-atlantique.fr

Jean-Louis Lanet
INRIA
Rennes, France
Email: jean-louis.lanet@inria.fr

***Abstract*—This study focuses on automatically detecting wrong implementations of specification in Java Card programs, without any knowledge on the source code or the specification itself. To achieve this, an approach based on Natural Language Processing and machine-learning is proposed. First, an oracle gathering methods with similar semantics in groups, is created. This focuses on evaluating our approach performances during the neighborhood discovery. Based on the groups automatically retrieved, the anomaly detection is based on Control Flow Graph of programs of these groups. In order to benchmark its ability to detect vulnerabilities, another oracle of vulnerabilities is created. This oracle knows every anomaly the approach should automatically retrieve. Both the neighborhood discovery and the anomaly detection are benchmarked using the precision, the recall and the F1 score metrics. Our approach is implemented in a tool: *Confiance* and it is compared to another machine-learning tool for automatic vulnerability detection. The results expose the better performances of *Confiance* over another approach in order to detect vulnerabilities in open-source programs available online.**

I. Introduction

A fuzzing attack aims at covering the most possible paths of a program's control flow in order to make the program crash or enter an unexpected state. To perform this, an attacker sends crafted messages to a running program, according to this program's message policy. Based on the program's output, the attacker generates the next message. A developer is able to mitigate a fuzzing attack, by testing user controlled inputs. This is useful before using these user controlled inputs sensible method parameters. Such tests are performed accordingly to a specification. A test stated in the specification but not implemented in the program is called missing-check. Such missing-checks are the cause of program crashes or illegal state changing. On the contrary, extra-checks are additional tests such as optional features or sometime back-doors.

This study aims at automatically detecting missing-checks in Java Card source codes, without knowing the program's source or specification. An oracle is automatically created based on the corpus of methods of the source code. Then, for each method under test, methods of similar semantics are retrieved from this oracle. Finally, research for differences using distance metrics is performed. To reduce the number of comparisons, a selection of similar functions in the corpus is done. Thus, our approach performs in two steps.

- The first step focuses on gathering methods of similar semantics, called neighbors. This is performed by using Natural Language Processing (NLP) techniques such as the Bag-of-words and the Latent Semantic Analysis (LSA) in order to transform methods of the source code in vectors. Then, the distance between each of these vectors is computed. According to a threshold, methods with distance under this threshold are flagged as neighbors.
- Then, the anomaly detection focuses on detecting missing-checks within these neighbors. This is achieved by comparing distances between these neighbors. The detection is based on methods only. As a result, if a test is performed in the caller of a method, it might lead to a false missing-check detection. To mitigate this, the exploration of the program's Control Flow Graph (CFG) is performed.

This paper starts with the context of this research in section II. The state of the art and the previous work is presented in section III. Then, the design of our approach is explained in section IV. The configuration and results of our approach's implementation: Confiance, against ChuckyJava are exposed section V, with our approach's limitation. Finally, the conclusion is drawn in section VI.

II. Context

A. The Java Card environment

The Java Card software technology allows Java programs, called applets, to be run on smart cards. Because it is designed for small memory devices, Java Card environment uses a subset of the Java language. It is widespread on SIM cards and ATM cards. Multiple applets can be embedded on the same Java Card. Because smart cards contain sensible information about the card owner, it is mandatory to secure them. As a result, despite the resources constraints of a smart card, Personal Identification Number (PIN) code and asymmetric or symmetric cryptography are available in the Java Card API. The PIN code is a number used for authenticating the card owner, before performing sensible operations. Asymmetric and symmetric keys use secret keys, allowing the card owner to cipher, decipher and sign documents. An asymmetric cryptosystem, such as RSA, uses a public key associated each secret key. The objective of an attacker is to retrieve any sensible information in the applet, such as the PIN code, the secret keys or any personal information. To achieve this, an

attacker can install a malicious applet, leading him to illegally extract sensible information in the smart card. As a result, securing applets before their deployment mitigates such attack vector.
In order to communicate from a terminal to the smart card and vice versa, applets can use a buffer which is divided in byte fields, as described in the ISO 7816, part 4 [1]. The sender writes values to this buffer and the applet reads these values in the buffer. Each byte of this buffer is structured in a header and a data part. All the data in the buffer transmitted to the applet is user controlled and therefore it cannot be trusted.

B. Fuzzing attacks in general

The System Under Test (SUT) is the running program to analyze. Generally, a fuzzing attack consists in sending messages (test cases) to this SUT in order to detect implementation errors. Such wrong behaviors can lead an attacker to exploit the program to disclose secrets. A typical fuzzing framework is composed of three main components.

- Message generator: it generates the messages to send to the SUT. These messages are crafted according to the program's policy.
- Message transmitter: it communicates the generated message to the SUT.
- Oracle: for a given message, it assesses if the behavior of the program is expected or wrong. According to the SUT outputs, it helps the message generator in order to craft the next message.

C. Example from the OpenPGP's specification

Once the applet receives a communication, it checks the fields values of the buffer. Depending on these received values and on the internal state, the corresponding operations described in the specification are performed. The byte buffer b at position n is defined as b_n. For example, in order to generate a private key in the OpenPGP version *2.0.1* [2] specification, the applet has to check if the following precondition for `privateKey`, as in (1), is respected.

$$\begin{aligned} PRE(\texttt{privateKey}) = (b_0 = 00 \vee 0C \vee 10 \vee 1C) \\ \wedge b_1 = DB \wedge b_2 = 3F \wedge b_3 = FF \end{aligned} \quad (1)$$

This precondition can be split among the CFG in different methods. A wrong implementation leading to an anomaly is either.

- A missing-check: there is a missing test for the precondition stated by the specification, but the command is still called. As a result, the condition for the command generating a private key is weakened.
- An extra-check: the precondition is respected so the command is called, but an extra test is present, even if not stated in the specification. As a result, this situation is sometime an optional feature, an implementation particularity or a back-door.

The objective of this study is to detect both missing-checks and extra-checks before the applet is embedded in Java Card smart cards.

III. State of the art

A. Security testing against Java Card

In [3], the authors are able to discover flaws in the bytecode verifier component, by using a fuzzing attack. Such component is embedded in the card and it is responsible for checking if non-illegal instructions are added between the compilation and the installation of the applet on the card. To do this, it verifies parameter's types to all the bytecode instructions of the compiled applet. They have achieved such attack by Crafting Compiled Applet files (CAP) and send them to this bytecode verifier. Each CAP file is mutated based to a single byte deletion, insertion or transformation. As a result, authors are able to execute a generic function from any point of the smart card.
A drawback of a fuzzing attack is the number of test cases to generate in order to cover the most possible paths of a program. To mitigate this problem, Lanet *et al.* [4] propose to combine a timing attack with a fuzzer working for Java Card. Next test cases to be sent to the program are generated according to time delay responses of the previous ones. As a result, the approach reduces the test space exploration by observing the time delay response of the SUT.
Lancia [5] proposes a framework for fuzzing smart card implementing the payment protocol Europay Mastercard Visa (EMV) [6]. A reference implementation is used as an oracle. Output for each command sent to the implementation under test is compared to the expected one of the oracle. As a result, implementation errors are detected. Such an approach requires a correct reference implementation, and might require modification when new versions of EMV's specification are released.
In his thesis, Savary [7] propose an approach based on test cases mutation in order to detect vulnerabilities in robust program models. Mutations are generated based on the required pre-conditions in order to negate a condition. However, the approach does not provide any metrics to assess the cost of deploying such a solution.

B. Static analysis approaches

Approaches aiming at detecting missing-checks are able to achieve this, based on a source and sink model. For example, a source is a user controlled input, and the sink is a sensible method, like writing data in the program for example. Before calling the sensible method, the user controlled input has to be sanitized to prevent crashes or non-allowed values. The tools mentioned in the state of the art follow the source and sink model.

a) Vanguard: Vanguard [8] is a missing-check detection tool based on the source and sink model. Based on a `C/C++` source code and a configuration file. The tool performs in three steps. First it locates security-sensitive operations with a configuration file describing the function's arguments, return

values, etc. Then, it judges argument assailability by exploring the CFG, inter-procedurally and intra-procedurally. Finally, it assesses insufficient protections.

b) Crix: Crix [9] is a missing-check discovery tool for `C` source codes. The tool is open-source and based on the source and sink model. *Crix* infers the "criticalness" of a variable. It is either a return value, called state variable, or it is used in sensitive calls, called critical-use variable. The "criticalness" of a variable is assessed based on a pattern. Briefly, if the return value of a function is tested and an Unix-like kernel error handling function (`BUG()`, `panic()`, and so on) is used, then the function returning is critical. Once the sensible sources and sinks are known, *Crix* constructs the peer slices in order to gather similar semantics and context. The tool has been tested on the Linux kernel.

c) ChuckyJava: ChuckyJava [10] a machine-learning tool which aims at detecting missing-checks in source codes based on the source and sink model. It has been adapted from *Chucky* [11] in order to analyze Java source codes. *ChuckyJava* performs in two steps: the neighborhood discovery and the anomaly detection. The neighborhood discovery, used as specification-mining, aims at gathering methods performing similar operations, called neighbors. In order to improve its neighborhood discovery performances, the source code can be modified as in [12], to homogenize identifier names, or replace the algorithm for *JavaNeighbors* [13].

C. Synthesis of the state of the art

a) Fuzzing approach: An approach based on fuzzing attacks might be time consuming. This is because the communication between the terminal and the card can be slow, as data are transmitted serialized. Stressing the memory of a Java Card with a lot of requests can destroy the card. In addition, a fuzzing attack might not cover all paths of the program's CFG. So an approach performing in the best delays is preferable.

b) Missing-checks detection approach: In Java Card source codes, the source and sink model might not be able to perform. Depending on the specification, a Java Card applet might test parameters which do not lead to a sink. In section II, for example, in the OpenPGP specification, the command to generate a private key states that the input has to be tested against specific values. In some cases, the private key generation uses the Java Card API method `genKeyPair` of the `KeyPair` class. However, this method does not require parameters thus it is not considered as a sensitive method by tools based on the source and sink model. As a result, the tests leading to this method are discarded. Automatically detecting sensitive methods might be quite a task, as it requires the following of source variable in the overall data set, which is time consuming and error-prone. As seen on applets source codes available online, a Java Card API method which writes data in the smart card can be used outside of a *try* and *catch* clause. They return value, considered sensible in source and sinks approaches, are not necessarily always checked in applets. In addition, write methods are used regardless if the inputs are user controlled or not. As a result, the method might be bypassed during the sensible method detection. Checking only sensible method is not enough in the Java Card context. *ChuckyJava* is able to detect extra-checks, where *Crix* and *Vanguard* are not able to achieve this.

It is crucial to mitigate fuzzing attacks before the applet is deployed in smart cards. To achieve this, we propose a static approach for fuzzing applets source codes, detecting missing-checks and extra-checks.

D. Previous work: ChuckyJava

1) ChuckyJava's approach: ChuckyJava is a specification-mining tool used to detect missing-checks in Java source codes. Since Java Card is a subset of Java, *ChuckyJava* can be used for the analysis of applets. In order to detect missing-checks, *ChuckyJava* extracts identifiers (method call, variable or field names) and performs for each a neighborhood discovery and the anomaly detection. The neighborhood discovery is useful in order to reduce the number of method to analyze during the anomaly detection. As a result, it mitigates wrong detection of anomalies. In *ChuckyJava*, this discovery is performed as follows for every identifier of a method.

1) The methods which are not using the same identifier are discarded.
2) Bag-of-words: it consists in extracting terms from each method to represent such methods as vectors. *ChuckyJava* extracts API symbols: parameter types, variable types and method call names, and it gathers them in a method by term matrix M.
3) Then, the cosine distance, exposed in (7), between the method under observation's vector x is computed pairwise with all other method vectors y remaining. The more the distance is close to 0.00, the more both methods are considered semantically similar: they are neighbors.

The anomaly detection takes the neighbor methods for a single identifier as input, and performs the following.

1) **Lightweight tainting**. It is the first step of the anomaly detection. It follows the identifier under observation through the source code. *ChuckyJava* discards expression blocks not using the identifier. This taint analysis is not inter-procedural, stopping at method boundaries.
2) **Abstraction**. The remaining expressions for each neighbor is abstracted as follows.
   - Comparison operators are abstracted and the negation operator is discarded.
   - Numerical values or numerical expressions are abstracted by a generic value.
   - Arguments and return values of calls are abstracted to a generic value.
3) **Model of normality**. So far, each method is represented as a subset where each element is a term t. $\mathbb{T}$ is the set of all expressions present in every method's subset, and $|\mathbb{T}|$ the cardinal of this set. As a result, each method

is now represented as a vector of size $|\mathbb{T}|$. For each t in $\mathbb{T}$, if the method contains at least one time t, then the value associated to t is 1.0. On the contrary, if the method does not contain t, the value associated to t is 0.0. Finally, the tool computes the vector of the model of normality of size $|\mathbb{T}|$. Its values for each dimension t is the mean of t's value in neighbor's vector.

4) **Anomaly score computation**. A distance vector is created. The value for each dimension t in $|\mathbb{T}|$, is the value of t in the model of normality minus the value of t in the method's vector under observation. As a result, the values in the distance vector are between -1.00 and 1.00. An analyst is able to interpret the results as follows.
   - An anomaly score of -1.00 represents a test performed only in the method under observation, but not in any neighbors.
   - An anomaly score of 1.00 represents a test which is not performed by the method under observation.

2) ChuckyJava's limitations: *ChuckyJava* has limitations. One of those limitations is the abstraction of every numerical value by a generic value. Two different numerical tests on an identifier with the same name are abstracted as the same expression. It only verifies if at least one check is performed. *ChuckyJava* does not output an anomaly for the expression, even if the tests do not even check the same values. Another limitation is about the identifier name selection during the neighborhood discovery step. Some programs may have optional features and might use extra identifiers. If not every program implements such features, then the *ChuckyJava* is assessing it cannot analyze it since it has not enough methods to analyze. This depends on the value of the expected neighbors during the anomaly step, and requires to be tweaked depending on the context. This reduces the automation of the analysis.

IV. Our contribution

This study aims at automatically detecting missing-checks, based on static analysis of programs. The approach first needs to gather neighbor methods by using a neighborhood discovery algorithm such as JavaNeighbors. This step is useful to extract a subset of methods with similar semantics in order to limit the number of methods to analyze during the anomaly detection. Then, a simplification of the normality model [10] is used to detect missing-check. Because the tool performs with methods as the base unit, the analysis of the CFG of the programs is necessary to verify if a check is performed in the method's caller. The methodology is aware of *ChuckyJava*'s limitations and it mitigates some of them to improve the missing-checks detection. The design of the approach is summarized in Fig. 1.

A. First step, source code gathering

It consists in gathering source codes to analyze as a data set. In our case, these are *Java Card* applet sources. According to JavaNeighbors [13], the applets have to implement the same specification.

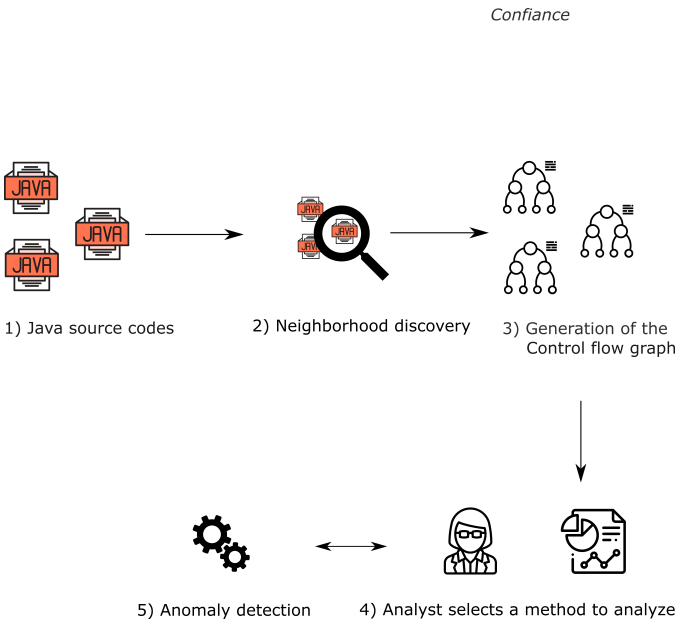

Fig. 1: Our approach performs in 5 steps. (Symbols are from *flaticon.com*).

B. Second step, neighborhood discovery

```
1  class Test{
3      private static final byte CONSTANT_VALUE = (byte) 0;
       private Object globalObject = new Object();
5      private KeyPair key = new KeyPair();
7      public static void myMethod01(Object o) {
           //Retrieve the communication buffer
9          byte[] b = o.getBuffer();
11         if(b[0] == (byte) 0xCA)
               key.genKeyPair(); //generates a key pair
13
       }
15     public static void myMethod02() {
           //Retrieve the communication buffer
17         byte[] buffer = globalObject.getBuffer();
19         //missing-check of 0xCA
           key.genKeyPair(); //generates a key pair
21
       }
23
       public static void myMethod03(byte b) {
25         if (b == CONSTANT_VALUE)
               //Throws an error
27             ISOException.throwIt(SW_CLA_NOT_SUPPORTED);
       }
29
   }
```

Listing (1) Example of 3 different Java methods. `myMethod01` and `myMethod02` are neighbors. However, `myMethod03` does not have neighbors.

Constant field names and values among files in the applet are gathered. Then, constant field identifiers are replaced by their value in the applet source code (according to the field modifiers, package, etc.) Then, the neighborhood discovery is performed. The objective is to reduce the number of methods

to compare in the anomaly detection, by gathering methods of similar semantics first. This is achieved in 4 distinct steps.

1) Bag-of-words: The Bag-of-words [14] consists in extracted key terms in a method. For example, terms in source codes such as Java API calls, tests and loop keywords are extracted. If a method is called from an instance, then this instance's identifier name is replaced by its class. Tests are represented in the same form. As an example, in Listing (1), `myMethod01` and `myMethod02` aims at generating a key pair for the instance `key` of the class `KeyPair`. In this example, they are supposed to implement the same specification. This same specification requires the value of the first byte of the communication buffer (represented by the parameter `Object o` or the global variable `Object globalObject`) to be set to `0xCA` before being able to generate the key pair. However, `myMethod02` does not perform such a test. The difficulty of the neighborhood discovery is to be able to detect neighbors, even if they do not have the exact same expressions and in different implementations of a same specification. The terms extracted for `myMethod01` are:

- `Object.getBuffer` (t_1),
- `Test:0x00CA` (t_2), and
- `KeyPair.genKeyPair` (t_3).

For `myMethod02` (m_2), the terms extracted are:

- `Object.getBuffer` (t_1), and
- `KeyPair.genKeyPair` (t_3).

Finally, the terms extracted for `myMethod03` (m_3) are:

- `Test:0x0000` (t_4) and
- `ISOException.throwIt` (t_5).

Methods such as m_1, ..., m_n are gathered in the corpus $\mathbb{C}$, of cardinal $|\mathbb{C}|$. Term such as t_1, ..., t_n are gathered in the set $\mathbb{T}$, of cardinal $|\mathbb{T}|$. Finally, every method is represented as a vector of terms of the same dimension $|\mathbb{T}|$. Each dimension's value in a vector is the frequency of that term in the method. As a result, a matrix method by term M of dimension $|\mathbb{C}| \cdot |\mathbb{T}|$ is created, as in (2).

$$\mathbf{M} = \begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array} \tag{2}$$

2) Weighting scheme adjustments: The weighting scheme of terms in M are adjusted. This allows to adjust the importance of certain terms in this matrix. This helps during the distance calculation to gather methods using similar terms. To achieve this, a local, a global and a normalization weighting scheme have to be configured:

- the local weight, for adjusting the weight of a term based on the document term frequencies,
- the global weight, for adjusting the weight of a term based on the corpus term frequencies,
- the normalization, for adjusting the weight of a term based on document length for example.

One common configuration in NLP is to combine the term frequency for local weight with Inverse Document Frequency (IDF) [15]. The objective behind this is to give more weight for terms appearing less often, as they are considered more meaningful compared to terms occurring in every methods. In the example, if the local weight is binary, as in (3), combined to probabilistic inverse as in (4) without any normalization. The function $freq(t_n)$ is the frequency of the term t_n, $|\mathbb{C}_{t_n}|$ is the number of methods in the corpus $\mathbb{C}$, where $binary(t_n) > 0$. Finally, M is adjusted as in (5).

$$binary(t_n) = \begin{cases} 1, & if freq(t_n) > 0 \\ 0, & otherwise \end{cases} \tag{3}$$

$$prob\_inverse(t_n) = \begin{cases} 0, & if |\mathbb{C}| - |\mathbb{C}_{t_n}| = 0 \\ log\left(\dfrac{|\mathbb{C}| - |\mathbb{C}_{t_n}|}{|\mathbb{C}_{t_n}|}\right), & otherwise \end{cases} \tag{4}$$

$$\mathbf{M} = \begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & t_4 & t_5 \end{array} \\ \begin{bmatrix} -0.69 & -0.69 & -0.69 & 0 & 0 \\ -0.69 & 0 & -0.69 & 0 & 0 \\ 0 & 0 & 0 & 0.69 & 0.69 \end{bmatrix} \end{array} \tag{5}$$

3) Dimension reduction: The curse of dimensionality happens when analyzing data in high-dimensional spaces. In such spaces, the data become so rare that it becomes sparse. It can lead to incorrect dissimilarity between data. In order to mitigate this phenomenon, the NLP technique entitled Latent Semantic Analysis (LSA) [16] is used. Its role is to discover correlated terms in M and group them as a concept. LSA requires a number k in order to reduce the number of terms of M into k concepts, with an adjusted weighting. It can discard terms if one does not bring any information, like a term used in every method. As an example, let say the number of k is set to 2. LSA decomposes the matrix, it discards less relevant terms and it returns a reduced matrix M'. In this example, LSA gathers as concept k_1 the terms t_1, t_3 (appearing together) and t_2. The value of k_1 in both m_1 and m_2 is adjusted depending on t_2. The second concept k_2 groups t_4 and t_5 as they appears together. Every concept is gathered in the set $\mathbb{K}$, of cardinal $|\mathbb{K}|$. LSA has transformed M from (5) to M', as in (6), of dimension $|\mathbb{C}| \cdot |\mathbb{K}|$.

$$\mathrm{M'} = \begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{cc} k_1 & k_2 \end{array} \\ \begin{bmatrix} 1.16 & 0.0 \\ 0.9 & 0.0 \\ 0.0 & 0.98 \end{bmatrix} \end{array} \tag{6}$$

4) Distance computation: Finally, the distance between methods is computed using a distance metric. Each row of M' represents a method vector. The cosine distance, as represented in (7) is often used in NLP for literature texts similarity. The Bray-Curtis dissimilarity [17] (sometime entitled Sorensen), as exposed in (8) measure seems to fit for source code similarity [13]. In both (7) and (8), x and y are two vectors of methods (m_1 and m_2 for example), $\mathbb{T}$ is the set of every term extracted, and $|\mathbb{T}|$ its size. As a result, if the distance between two vectors is below a threshold: the corresponding methods are neighbors. This distance is bounded from 0 to 1. The more the distance is close to 0, the more the methods are similar.

$$cosine(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|} \tag{7}$$

$$Bray - Curtis(x, y) = \frac{\sum_i^{|\mathbb{T}|} |x_i - y_i|}{\sum_i^{|\mathbb{T}|} |x_i + y_i|} \tag{8}$$

The distance calculation is the distance metric applied over the matrix $M' \cdot M'^t$, returning the distance matrix D of size $|\mathbb{C}| \cdot |\mathbb{C}|$. D is displayed in (9), where the distance metric used is the Bray-Curtis dissimilarity measure. For a distance threshold of 0.45, m_1 (`myMethod01`) and m_2 (`myMethod02`) are neighbors, because the distance between them is below the threshold. On the contrary, m_3 (`myMethod03`) has a distance to m_1 of 1.0, which is over the threshold and it cannot be considered as neighbors. Idem for m_3 and m_2.

$$\mathrm{D} = \begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\ \begin{bmatrix} 0.0 & 0.33 & 1.0 \\ 0.33 & 0.0 & 1.0 \\ 1.0 & 1.0 & 0.0 \end{bmatrix} \end{array} \tag{9}$$

The difficulty is to succeed to gather neighbor methods, even if they are missing test terms, and with many implementation differences.

C. Third step, CFG generation

The inter-procedural CFG is generated for every applet in the data set. This CFG is composed of.

- Nodes, labeled with expressions such as instructions, control flow indications (end of *try/catch*, end of *if*, etc.) or method body definition.
- Edges, which have one label corresponding to the program flow (*true* or *false* outgoing from a test node for example).

Inter-procedural calls are resolved in an iterative way, until no modification of the CFG is required. The maximum number of iterations has to be set (to prevent infinite iteration for recursive calls). However, Java Card strongly recommends recursion to be avoided due to RAM limitation. For a method call, there is any corresponding method definition, then all nodes and links are duplicated to the method call location. As performed in the second step, every constant field names and values among files in the applet are gathered and replaced in the source code. The identifiers of instance objects are replaced by their object type. Since *Java Card* works with `short` and `byte` values, every numeric value encountered is translated to a `short`. It helps the analyst to understand the output when fetching for a specific missing-check value in the source code. Then, expressions in nodes are transformed or abstracted. The CFG is constructed to handle two Java object oriented concepts: the inheritance and the exception handling.

a) Inheritance in Java: In a Java class, a constructor is a method which instantiates an object of the class once called. Every class inherits from the root `Object` class. As a result, if a constructor is called, before performing the instructions within this call, the `Object` constructor is called first. This is always performed, either by calling `super()` or implicitly. Now suppose a class `Class1` inherits from another class `Class0` itself inheriting the `Object` class. Then, calling the constructor `Class1()` of `Class1` leads in calling the constructor `Class0()` before performing `Class1`'s constructor instructions. Next, before computing the instructions of `Class0()`, the *Object*'s constructor `Object()` is called. This example is illustrated in Fig. 3.

b) Exception handling in Java: The exception handling is another key mechanism of Java. A method definition can declare throwable checked exceptions. Such exceptions have to be handled within a `try` and `catch` structure. Since exceptions are classes, a `catch` clause needs to precise the specific exception class it handles. In case of an exception occurring at runtime, then the control flow of the program is modified to flow to the first corresponding `catch` clause. If no corresponding `catch` capturing the exception class can be found, then the program flows to the `catch` capturing the *Exception* class. As a result, the CFG is generated to implement such Java mechanism. To achieve this, a list of known Java Card API method with the exceptions they can throw is created. During the CFG generation, corresponding Java Card API calls have extra exception nodes linked to the first corresponding exception `catch` clause. If case the method is

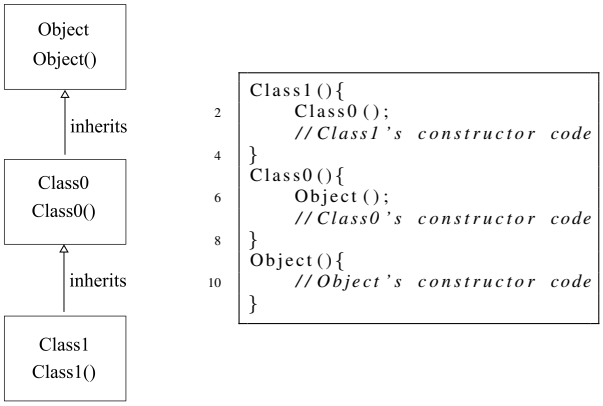

Fig. 3: The figure on the left corresponds to the inheritance chain. On the right is the implicit source code equivalent of this chain. `Class1` inherits of `Class0` which inherits of `Object`. This is called a cascading constructor calls.

defined in the corpus, then exception nodes are replaced by the `throw exception` declaration. This declaration is the precise location where an exception is in fact thrown. Another particularity of Java Card are the exception classes calling `throwIt`. In this case, if the method is not defined, a node exception `throw` with the exception class name is created. Such exceptions are connected to their corresponding catches too. Fig. 4 proposes an example of `try` and `catch` handling.

```
static void myMethod(Object o) throws Exception {
    if(o == null)
        throw new Exception();
}

public static void main(String args[]){
    Object o = null;
    try{
        myMethode(o);
         //This code is not executed
        System.out.println("Everything is ok.");
    }catch(Exception e){
        //This code is executed
        o = new Object();
    }finally{
        System.out.println("End try-catch-finally.");
    }
}
```

Fig. 4: Example of exception of type "*checked*" in *Java*.

D. Fourth step, interaction with the user

All the neighbors are known at this point. An analyst has to chose the method to test in order to detect its anomalies. It is possible to automatically perform the analysis on the overall data set.

E. Fifth step, anomaly detection

To perform an anomaly detection, the transformed tests are extracted from the CFG for the method under analysis. Those tests are gathered in a Bag-of-words, such as the neighborhood discovery. The approach uses the Bag-of-words technique instead of the CFG for a method. The latter is sensible to the order of the tests performed, which wrongly increases the dissimilarity between methods. All different tests used for both the method and its neighbors are gathered in a new set $\mathbb{T}$, of cardinal $|\mathbb{T}|$. Then, method under analysis and its neighbors are gathered in a new set $\mathbb{N}$, of cardinal $|\mathbb{N}|$. Each are represented as vectors such as $\mathbb{M}$, as in (10). For each test, if the method contains at least once the test, then the value associated to it is 1.0. On the contrary, if the method does not contain the test, then the value associated to it is 0.0.

$$\mathbb{M} \rightarrow [\![0, 1]\!]^{|\mathbb{T}|} \tag{10}$$

Our approach then consists in iterating over the method under observation's tests vectors. For each test, two cases occur for the anomaly detection.

- **Missing-check**: for a test, every neighbor has a value of 1.0 for it, but its value for the method under observation is 0.0. For every path from the applet entry point to the method.
  - If the test is present, then the shortest path containing this test is reported to the analyst. As a result. The analyst has to assess if the test found is the expected one.
  - If the test is not found in at least one path, then it is assessed that there is a *missing-check*, reported as the shortest path to the method not containing the test.
- **Extra-check**: for a test, every neighbor has a value of 0.0 for it, but its value for the method under observation is 1.0. For all neighbors, if at least one of them has the test in all its paths from its applet entry point, then the anomaly is not displayed to the analyst. On the contrary, the extra-check is displayed.

In the example of Listing (1), the neighbor of `myMethod01` (m_1) is `myMethod02` (m_2). This anomaly detection example focuses on the analysis of m_1. Both m_1 and m_2 methods are gathered in a neighbor set $\mathbb{N}$, of cardinal $|\mathbb{N}|$, and they are represented as test vectors. In the example, $\mathbb{T}'$, of cardinal $|\mathbb{T}'|$, is composed of only one test term `Test:0x00CA` (t_1), which is performed in only one method of $\mathbb{N}$. The resulting test matrix corresponds to the tests performed for a method under analysis and its neighbors. This test matrix, R, of dimension $|\mathbb{N}| \cdot |\mathbb{T}'|$, is represented in (11). Because m_1 performs the test t_1, its value is equal to 1. On the contrary, m_2 does not perform the test t_1. Its value is equal to 0. The result of the analysis exposes that m_1 performs an extra-check, t_1, while m_2 does not. In such case, *Confiance* verifies if there is such a test in all paths of the CFG, from the entry point of the applet, to the method m_2. If there is such test, the value for t_1 in m_2 is set to 1, so that there is no extra-check detected, preventing a false extra-check detection. On the contrary, the value for t_1 in m_2 is not modified, and an extra-check is reported to the analyst. Because m_1 and m_2 are neighbors only together, there is no need to analyze m_2, as the extra-check is going to be detected as missing-check. The result for the analysis of m_1 exposes an extra-check for m_1, meaning m_2 is missing a

Implementation	Number of method
OpenPGPApplet (OP)	66
JCOpenPGPApplet (JP)	23
MyPGPId (MP)	23
FluffyPGP (FLP)	25
Total	137

TABLE I: OpenPGP Java Card implementations (about 6,291 lines of code)

test. In cases where a method has more than two neighbors, it is necessary to analyze all of them. However, m_3 does not have any neighbor in the data set, so it cannot be analyzed for missing-check or extra-check.

$$\mathrm{R} = \begin{matrix} \\ m_1 \\ \\ m_2 \end{matrix} \begin{matrix} t_1 \\ \left[\begin{matrix} 1 \\ \\ 0 \end{matrix}\right] \end{matrix} \tag{11}$$

After the analysis of a method, one can analyze another method, by returning to the fourth step. The analysis can perform for the overall data set automatically too.

V. Experiment

A. An OpenPGP applet data set

1) OpenPGP: OpenPGP v2.0.1 [2] is a standard for signing, ciphering and deciphering messages. Its specification describes commands in order to allow the PIN code verification within the applet. Different commands for performing security operations are described. As an example, the commands for signing a message, ciphering a message, deciphering a message are described. This specification relies on symmetric keys for secured communications (because of the contactless aspect) but on asymmetric keys for security operations. This specification supports Triple-DES and AES for symmetric operations and RSA for asymmetric.

2) Data set: This data set is the same one used in [13]. Each of the four applets are available on Github. The content of this data set is exposed in Table I.

3) Oracle of neighbors and anomalies: Our data set is composed of four OpenPGP applets available online and implementing the v2.0.1 specification. The oracle of neighbors contains a neighbor list for each methods. As an example, a signing method shall be neighbors to signing methods of the three other applets. As specified, some features are optional. As a result, some methods might not have any neighbors. Implementations have sensible differences, and contains missing-checks or extra-checks, which can lead the neighborhood discovery step to be a difficult task. The neighborhood discovery configuration has find the most possible correct neighbors despite these implementation choices and errors in order to detect those in the anomaly detection step.

The oracle of anomalies set is composed of every test performed in the applets. For each test, the corresponding missing-check in every method is considered to be existing. If two methods are wrongly flagged as neighbors, then each test of the first one is a missing-check in the other one. As a result, the data set is composed of a total of 99,495 tests to classify. In other words, the data set contains 2 different classes: anomaly or benign, and both tools have to classify each test in one of those classes. The oracle of anomalies contains 55 anomalies (missing-check and extra-checks). The missing-checks and extra-checks can be either:

- Optional features such as secured communication and optional data objects for example.
- Weak precondition verification to write or to read objects.
- Data objects which can be illegally written.
- Non-specified commands, potentially back-doors.

In our case, a True Positive (TP) is an anomaly test, correctly flagged as anomaly. A False Positive (FP), is a benign test classified as an anomaly test. A False Negative (FN) is anomaly test wrongly flagged as benign. Finally, a True Negative (TN) is a benign test correctly classified as benign.

Metric	Formula
Precision	$precision = \frac{TP}{TP+FP}$
Recall	$recall = \frac{TP}{TP+FN}$
F1 score	$F1 = \frac{2 \cdot precision \cdot recall}{precision+recall}$

TABLE II: Metrics used for classifiers using True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN)

B. Experiment methodology

The experiment is exposed in two steps. The first one consists in displaying vulnerabilities that *Confiance* is able to detect where *ChuckyJava* can not. The second part of the experiment present the scalability of both approaches against the overall data set in terms of detection and execution.

1) Metrics: In classification, there a three main metrics to compute to assess the efficiency of a classifier: the precision, the recall and the F1 score [18]. These metrics are computed for a value from 0.0 to 1.0. The best result for the metrics are a value of 1.0. Table II displays these metric equations.

- Precision: computes the ability of the classifier to return only expected results.
- Recall: computes the ability of the classifier to return all of the expected results.
- F1 score: used in majority in machine-learning for classification. It computes the harmonic mean between the precision and the recall.

C. Tools configurations

1) Confiance:

a) Neighborhood discovery: There are different configurations possible for the neighborhood discovery. Such configurations focus on tweaking the weighting scheme of the document by term matrix, the distance metric, the distance threshold and the number of concepts for LSA.
In [13], authors have created a configuration performing better than *ChuckyJava* for the neighborhood discovery. They recommend to configure JavaNeighbors by using:

- local weighting scheme: the term frequency,
- global weighting scheme: the inverse document frequency,
- normalization: none,
- distance metric: Bray-Curtis,
- distance threshold: 0.3, and
- $k = 40\%$ of $|\mathbb{T}|$, for LSA.

Such configuration results in an F1 score for the neighborhood discovery is equal to 0.37. In [19], the authors propose a configuration based on their student data set. The study is based at the Java file level, while our approach is based at the method level. As a result, such configuration does not fit in this case.
In order to find the best F1 score, different configurations of JavaNeighbors are tested against the OpenPGP oracle of neighbors. There are 43,700 different combinations tested for the neighborhood discovery:

- five different local weighting schemes,
- six different global weighting schemes,
- with or without the cosine normalization,
- four different distance metrics,
- distance threshold: from 0.0 to 1.0, steps of 0.05, and
- k from 10% to 90% of $|\mathbb{T}|$, steps of 10% for LSA.

Finally, the configuration for the neighborhood discovery resulting in the best F1 score is:

- local weighting scheme: binary,
- global weighting scheme: probabilistic inverse,
- no normalization,
- the dissimilarity measure: Bray-Curtis,
- distance threshold: 0.45, and
- $k = 40\%$ of $|\mathbb{T}|$, for LSA.

The F1 score of the neighborhood discovery is the best among the the 43,700 combinations tested and is equal to 0.51. The precision and the recall are balanced with a value of at least 0.51 for each.

b) Anomaly detection: During the anomaly detection, *Confiance* requires a maximum depth to find a path from the entry point of the applet to the method under analysis. Not defining such maximum depth can lead the program to never terminate its execution as every path are being explored. Increasing this value exponentially increases the computation time. A maximum path depth of 60 links gives a fair trade-off for a computation time and reachable methods.

2) ChuckyJava: In [12], the authors precise that *ChuckyJava* returns way too many entries. This is because it decomposes every test it encounters. As an example, after the lightweight tainting, if there are two different identifiers used in an test, two additional expressions are created with only each identifier and the same anomaly score. As a result, some results might be displayed multiple times at different locations during the analysis. A filter showing only tests is applied. In addition, *ChuckyJava* requires a number of neighbors k during the neighborhood discovery. This value is set to 3, as a method under analysis has in theory 1 neighbor in each applet. Finally, *ChuckyJava* requires an anomaly score threshold to assess if a test is an anomaly. This threshold is set to 1.0. As like *Confiance*, *ChuckyJava* is configured to perform an analysis on every method of the data set.

D. Results

1) Case study: OP `put data` command: Confiance has successfully gathered the neighbors for the `put data` command of the OP applet (i.e the `put data` commands of the other applets). As mentioned in the OpenPGP specification, the `put data` command enables a user to write values in data objects (DO) embedded on the smart card. The parameters of the communication buffer b_2 and b_3 precise the DO code to overwrite. As an example, the specification for the command allows to overwrite the DO name, the DO language preferences, the DO gender of the card holder, the DO fingerprint data, and so on. According to the OpenPGP specification, the gender of the cardholder can be either of the three values:

- `0x31` for a male,
- `0x21` for a female, and
- `0x39` if the gender is not announced.

In this data set, only the OP applet performs these tests before writing the value to the DO, as shown in Listing (2) of Fig. 5. Listing (3), (4) and (5) of Fig. 5 represent the `put data` handling for respectively applets MP, JC and FLP. This is possible because during the anomaly detection steps, the test terms are extracted in an inter-procedural way.
ChuckyJava can not detect such missing-checks, as the numerical values for $0x31$, $0x32$ and $0x39$ are abstracted to a generic value.

2) Result examples: Three different method results are exposed as examples. For each, a sentence describing briefly if the test is missing or in extra is added, as returned by *Confiance*. Then, the test and anomaly score is represented as `[test : anomaly score]`. As explained in section II, once the applet receives a communication buffer, it needs to check the value of its byte fields. In order to understand the following results, the signification of the first four bytes are of the communication buffer are.

- b_0, it corresponds to the instruction class of the command (proprietary, for example).
- b_1, it is the instruction code to execute (generation of private key for example).
- b_2 and b_3, those are the first and the second parameter bytes.

```
case TAG_GENDER:
    if (in_received != 1) //Size received tested
        //Exception
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    // buffer[0] contains the new value for DO gender
    if (buffer[0] != (byte) 0x31
            && buffer[0] != (byte) 0x32
            && buffer[0] != (byte) 0x39)
            //Throws an error
        ISOException.throwIt(ISO7816.SW_WRONG_DATA);

    sex = buffer[0];
    break;
```

Listing (2) DO gender tests performed in OP

```
//sex is an instance of DataObject
//setData does not evaluate 'data'
//for 0x31, 0x32 nor 0x39

case TAG_GENDER:
    sex.setData(data, dataLen);
    return;
```

Listing (3) Missing-checks for DO gender tests performed in MP

```
case TAG_GENDER :
    checkPw3();
    byteRead = apdu.setIncomingAndReceive();
    if (byteRead!=1) //Size received tested
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    //ret_getBuffer[ISO7816.OFFSET_CDATA] contains
    //the new value for DO gender
    sex = ret_getBuffer[ISO7816.OFFSET_CDATA];
    break;
```

Listing (4) Missing-checks for DO gender tests performed in JC

```
//buffer variable 'apdu' contains new value for DO gender
//storeFixedLength does not
//evaluate 'apdu' for 0x31, 0x32 nor 0x39

case TAG_GENDER:
    storeFixedLength(apdu, sex, (short) 0, (short) 1);
    break;
```

Listing (5) Missing-checks for DO gender tests performed in FLP

Fig. 5: DO gender writing code snippets from `put data` commands in the applets. OP is the only one to perform tests on the value before writing the corresponding DO.

Those bytes are the one the results expose. However, other byte fields exists but they do not appear in our results.

a) JC process command: Listing (6) exposes 4 out of the 10 anomalies discovered by *Confiance* for the `process` method of the applet JC. The `process` method is called when a message is received by the smart card for the specific applet. It is the entry point of an applet.

Line 1 exposes a missing-check for the value `84`. It corresponds to the test for the `get challenge` command. This command is optional according to the specification, but the three others applets implement it. It consists in the generation of a random number. This is useful since smart cards often embed certified hardware for generating random numbers.

```
1 Missing-check :  [ TEST:0x0084 : 1]

3 Missing-check : [ TEST:0x3fff : 1]

5 Missing-check : [ TEST:0x0093 : 1]

7 Extra-check [ TEST:0x001c : 0] ]
```

Listing (6) Anomaly score results for the `process` method of JC

Line 3 corresponds to the `put data` command, in a specific case. This special case is executed for importing a private key on the card. As specified in the documentation of OpenPGP, in the case of an "odd b_1" for the command `put data`, the specification states that b_2 and b_3 have to be set respectively to `3F` and `FF`. However, these checks are never performed in the applet. It weakens the preconditions to write private keys. This is a missing-check for the value `3FFF`. This is the vulnerability mentioned in the state of the art in section II.

Line 5 displays an anomaly for the test against the value `93`. As explained for the `put data` command, in order to write object on the smart card, the communication buffer has to contain the specific DO to write. The mechanism is similar for the `get data` command. In order to retrieve a DO, b_2 (and sometimes b_3 too) has to be set to the corresponding DO code value. In this case, the value `93` corresponds to the retrieval of the signature counter during the `get data` command. As a result, it is not possible to get the signature counter from the outside. The specification precises the possibility to obtain it, as performed in the other applets of the data set.

Line 7 exposes another vulnerability. It corresponds to the test against the value `1C`. Depending on the message receive by the card, the value of b_0 has to be tested against `1C`. However, JC applet is the only one to perform such a verification. The three other applets contain this missing-check. The analyst has to notice that this vulnerability is not detected as missing-check by analyzing other `process` methods. The reason is because the majority of applets does not perform this test. As a result, not performing this test is not considered as an anomaly, even thought it is stated by the specification to do it.

b) JC `import key` command: The key importation allows an user to overwrite a private key on the applet. In order to be allowed to import a key, the administrator PIN shall be verified. In addition, b_2 and b_3 are required to be equal to respectively `3F` and `FF`. However, in the JC applet, neither a PIN code is verified, nor b_2 and b_3. As a result, any user can overwrite the private key. Moreover, the precondition required for the command to be launched is weakened.

c) OP `get data` command: The `get data` command in OpenPGP allows DO to be retrieved as requested by the communication buffer. Each DO has a corresponding code,

as defined in the specification. With the communication buffer, the user has to specify the DO. According to the specification, some DO are flagged as "Simple", and they can be retrieved directly. Some other DO are flagged as "Constructed". Most of them are not allowed to be retrieved directly. Instead, they are retrieved as encapsulated with other DO, serialized. During the analysis using *Confiance*, it has been possible to discover that there are missing-checks for specific DO in the OP `get data` command. These missing-checks are:

- DO `C1`: containing the algorithm attributes signature,
- DO `C2`: which contains the algorithm attributes decryption,
- DO `C3`: about the algorithm attributes authentication,
- DO `C5`: the fingerprints,
- DO `C6`: the list of CA-fingerprints, and
- DO `CD`: the list of dates and times generation of key pairs.

The OP applet is the only one not allowing the direct retrieve of these DO, meaning the other 3 applets are. *Confiance* is able to detect such anomaly. However, depending on the identifier used and analyzed, such anomaly might not always be detected by *ChuckyJava*. Before the renaming of identifier, these anomaly might not be discovered. However, because of the neighborhood discovery does not gather the expected neighbors, these anomaly are detected by *ChuckyJava*.

3) Confiance and ChuckyJava performance comparison:

a) Anomaly detection performances: Table III exposes the confusion matrices obtained by *Confiance* and *ChuckyJava*. Finally, based on these statistics, the precision, the recall and the F1 score are computed for both tools and they are exposed in Table IV. As we can see, both the precision and the recall are increased in *Confiance*, compared to the *ChuckyJava* ones, leading to an increased F1 score for *Confiance*.

	Actual class	
Predicted class	TP = 15	FP = 651
	FN = 39	TN = 98790

	Actual class	
Predicted class	TP = 45	FP = 147
	FN = 9	TN = 99294

TABLE III: Confusion matrices of *ChuckyJava* on top and *Confiance* on the bottom

Metric	*ChuckyJava*	*Confiance*
Precision	0.02	**0.23**
Recall	0.28	**0.83**
F1 score	0.04	**0.36**

TABLE IV: Precision, recall and F1 score comparison for both *ChuckyJava* and *Confiance*

The number of entries to analyze for a user is of 666 entries for *ChuckyJava*, against 192 for *Confiance*.

4) Scalability comparison: Table V presents the result comparison for both tools. The execution time and the number of results to analyze are exposed. The computations are performed on Ubuntu within a virtual machine where 2 threads of an Intel 7600U and 10Go of RAM are allocated. As exposed in the results of Table V, *Confiance* performs faster by reducing the executing time of an analysis of 77.6%.

	Statistics	*ChuckyJava*	*Confiance*
OpenPGP data set	Execution time (mn)	12,5	**2,8**

TABLE V: Performance comparison: *Confiance* against *ChuckyJava* over the overall data set.

The execution time of *Confiance* is faster than *ChuckyJava*. The reason is twofold. First, to mitigate the cost of finding paths from the origin of the applet to the method, the tool limits the paths to visit to 50 edges. This limit is fixed based on the TP anomalies found, which requires about 40 edges. However, the tool has been tested without any edge limit, resulting in one hour of computation, only for verifying the positive anomaly scores. The second reason is because *ChuckyJava* performs both the neighborhood discovery and a "lightweight tainting" for every identifier it analyzes. This operation is time consuming as it requires an exploration of paths in the applet graph. However, it is performed on a subset of the methods: the sensible ones. In addition, only the arguments has to be followed and not every byte of the communication's buffer.

E. Confiance's limitations

Despite the better performances of *Confiance* over *ChuckyJava*, the approach has different limitations.

- **Wrong neighbors**. The anomaly detection depends heavily on the results of JavaNeighbors. Since it sometimes does not return correct neighbors, every check performed only in the method under observation might be flagged as FP. This leads to a quick increase of the FP. There are rare exceptions where wrong neighbors enable the analyst to discover an anomaly. Anyway, this limitation is responsible for most of the FP in the approach.
- **Tests gathering**. In some cases, b_2 and b_3 are concatenated as a single `short` value. A test against `33` for b_2 and `FF` for b_3 is now a single test against `33FF`. To solve such FP, following all the fields of the communication's buffer requires to be implemented. Such taint analysis shall explore every paths of the CFG. However, the execution time overhead for this computation might be really high. As a comparison, *ChuckyJava* performs such a tainting for the identifiers under observation, and the compute time is high as shown in the results. In addition, this taint analysis is bounded to the method and not the paths leading to method. Mitigating this limitation would reduce at most 10 FP for the data set.

- **Maximum path depth**. The maximum depth has to be set. A drawback in this approach is that if this value is too low, some methods might never be reached. As a result, the anomaly detection in the call graph could not be performed, leading to FP. However, increasing this value increases quickly the computation time. The value of 60 links set in the configuration might not fit other implementations, and would require to be increased.
- **Values calculated at runtime**. The approach replaces constant values in tests. However, some tests values can be computed only at runtime. For example, if a variable's value is received from the communication buffer, then it is not possible to determine the value beforehand the applet's execution. This leads to a high false positive increase since every program of the data set handles differently information retrieved from the buffer.

VI. Conclusion

Our methodology for automatically detecting missing-checks and extra-checks is based on specification mining. The approach first needs to gather neighbor methods by using a neighborhood discovery algorithm, such as JavaNeighbors. The neighborhood discovery step, based on NLP and unsupervised machine-learning, is useful to extract a subset of methods with similar semantics in order to limit their number to analyze during the anomaly detection. Then, a simplification of the normality model is used to detect anomalies. Because the approach performs with methods as the base unit, a CFG for each applet is created, according to Java object oriented and exceptions handling features. The CFG analysis is necessary to verify if a check is performed in the method's caller, before reporting a missing-check or an extra-check. The approach is aware of limitations of *ChuckyJava*, and it mitigates some of them to improve the detection of anomalies. Our approach is implemented as *Confiance*. Compared to *ChuckyJava*, *Confiance* shows better performances for both the precision and the recall for the anomaly detection. As a result, the F1 score is improved from 0.04 to 0.36. *Confiance* performs faster than *ChuckyJava* of 77.6%.

Even if *Confiance* performs better than *ChuckyJava*, it has limitations. Future works include the improvement of the neighborhood discovery in *JavaNeighbors*, as it the main limitation of the approach. This approach currently relies on an unsupervised machine-learning technique to discover its neighbors. Many term combination have been tested (including inter-procedural term extraction) but the best performances are presented in this paper. Depending on the availability of applets sources, supervised machine-learning or deep-learning might improve the neighborhood discovery step. It requires to label methods and reduce the automation of the approach, since it is necessary to understand the specification to label methods. In order to increase the recall of the approach, tainting the byte fields of the communication's buffer can be useful. Such improvement might be costly, as it is shown for *ChuckyJava*. To resolve the tests values only available at runtime, an approach combining static analysis and constraints solvers might be interesting.

References

[1] I. O. for Standardization and I. E. Comission, "ISO 7816." [Online]. Available: https://www.iso.org/fr/search.html?q=7816

[2] A. Pietig, "Functional specification of the openpgp application on iso smart card operating systems."

[3] J. Lancia and G. Bouffard, "Fuzzing and overflows in java card smart cards," *SSTIC*, 2016.

[4] J.-L. Lanet, H. Le Bouder, M. Benattou, and A. Legay, "When time meets test," *International Journal of Information Security*, vol. 17, no. 4, pp. 395–409, Aug 2018.

[5] J. Lancia, "Un framework de fuzzing pour cartes a puce: application aux protocoles."

[6] EMVCo, "EMV integrated circuit card specifications for payment systems."

[7] A. Savary, "Détection de vulnérabilités appliquée à la vérification de code intermédiaire de java card."

[8] L. Situ, L. Wang, Y. Liu, B. Mao, and X. Li, "Vanguard: Detecting missing checks for prognosing potential vulnerabilities." ACM, 2018, p. 5.

[9] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1769–1786, 2019.

[10] L. Ouairy, H. Le-Bouder, and J.-L. Lanet, "Protection of systems against fuzzing attacks," pp. 156–172, 2018.

[11] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery." ACM, 2013, pp. 499–510.

[12] L. Ouairy, H. Le-Bouder, and J.-L. Lanet, "Normalization of java source codes," pp. 29–40, 2018.

[13] L. Ouairy, H. Le Bouder, and J.-L. Lanet, "Javaneighbors: Improving chuckyjava's neighborhood discovery algorithm," *Procedia Computer Science*, vol. 160, pp. 70–76, 2019.

[14] "A gentle introduction to the bag-of-words model," https://machinelearningmastery.com/gentle-introduction-bag-words-model/, accessed: 2019-07-09.

[15] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for idf," *Journal of documentation*, vol. 60, no. 5, pp. 503–520, 2004.

[16] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[17] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *City*, vol. 1, no. 2, p. 1, 2007.

[18] "What is the f score ?" https://deepai.org/machine-learning-glossary-and-terms/f-score.

[19] G. Cosma and M. Joy, "Evaluating the performance of lsa for source-code plagiarism detection," *Informatica*, 2013.