JavaNeighbors: Improving ChuckyJava’s neighborhood discovery algorithm
Léopold Ouairy, Hélène Le Bouder, Jean-Louis Lanet

To cite this version:

HAL Id: hal-01950822
https://inria.hal.science/hal-01950822
Submitted on 11 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
JavaNeighbors: Improving ChuckyJava’s neighborhood discovery algorithm

Léopold Ouairy\textsuperscript{a,}\textsuperscript{*}, Hélène Le Bouder\textsuperscript{b}, Jean-Louis Lanet\textsuperscript{a}

\textsuperscript{a}INRIA, Rennes, France
\textsuperscript{b}IMT-Atlantique, Rennes, France

Abstract

In this paper, a study to protect Java Card source codes against fuzzing attacks is presented. This work is based on the tool ChuckyJava. This tool aims at automatically detecting anomalies in Java source codes in a Machine Learning way, without the knowledge of their specification. First, we propose a definition of neighbor methods. Based on this same definition, this study focuses on the improvement of the neighborhood discovery step for the tool ChuckyJava. To achieve this task, we have created a new Machine Learning tool: JavaNeighbors. It is based on different Natural Language Processing techniques: both local and global weighting schemes adjustment for term extraction and Latent Semantic Analysis to mitigate the curse of dimensionality. As a result, JavaNeighbors solves four limitations of ChuckyJava and it performs faster and with a better accuracy.

© 2018 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Conference Program Chairs.

Keywords: fuzzing attacks; security; machine learning; neighborhood discovery

1. Introduction

A fuzzing attack [1] aims at exploring the most possible paths of the program’s control flow. To do this, the attacker sends parameters to a running program and he is able to detect a deviation of the expected behavior of the program. A fuzzing attack can lead to gain an illegal access to resources embedded on a program. This study focuses on the protection of Java programs against such attacks. To mitigate such an attack, one is able to perform an analysis with the Machine Learning tool ChuckyJava [2]. This tool is based on an improved version of Chucky [3]: Chucky-ng. It aims at detecting a lack of user controlled input verification which can lead to a vulnerability, known as missing-check. ChuckyJava works on Java source codes and it performs an analysis in two steps. The first called the neighborhood
discovery consists in gathering the neighbors methods together. Two methods are neighbors if they are semantically identical. The second step is the anomaly detection. It outputs to the analyst an anomaly score in order to assess if the methods are vulnerable. This study proposes to improve the ChuckyJava’s neighborhood discovery algorithm by creating a tool usable as a stand-alone: JavaNeighbors. The paper is described as follow. Different studies focusing on discovering neighbors methods are exposed in section 2. Then, each step of the design of JavaNeighbors are developed in section 3. Both ChuckyJava and JavaNeighbors are compared on the same Java Card data set as ChuckyJava. The results on the neighborhood discovery and the impact on the anomaly detection with JavaNeighbors are exposed in section 4. Finally, the conclusion is drawn in section 5.

2. State of the art

There are different use cases to discover neighbor methods. Some techniques are based on plagiarism detection because it is similar to neighbors discovery. Plagiarists may use different technique to obfuscate their code as like moving or refactoring instructions without changing the semantics of the program.

Tree or graph based approaches. In [4], authors focus on detecting objects renaming. In their tool, a Differencier overlaps both suffix trees of two different methods in order to determine if they are neighbors. From the best of our knowledge, no release is available for the tool.

In this Microsoft’s Phoenix plugin [5], authors rely on both abstract syntax trees and suffix trees to detect code clones. While the former tree is generated by the Phoenix’s compiler, the latter is generated by the authors. A drawback of the technique is that it detects only exact matches in term of names and positions for variables.

JPlag [6] is a plagiarism detection technique. The tool decomposes a source code as a list of terms. Then it assesses if there is a plagiarism by comparing substrings of these term lists pairwise of programs. We have tested this solution on the OpenPGP data set we expose in the experiment. JPlag is efficient to detect renaming in field names, but it is not able to detect correctly neighbor methods.

Machine Learning based approaches. Chucky [3] focuses on detecting anomalies in source codes. Those anomalies can be either missing-check in case of a lack of verification in the control flow, or a missing assignation, in case of missing affectation. It analyzes C/C++ source codes. Chucky first extracts functions from a source code. Next, it represents them as a graph. After a filtering step, it represents the neighbors functions in a matrix. Then, the neighborhood discovery is performed by using the k-Nearest-Neighbors classifier algorithm to gather neighbors functions. During the neighborhood discovery step, Chucky might not be able to calculate a distance between functions because of the filtering step. This problem is exposed in details in the experiment section 4.

PlaGate [7] is a Machine Learning tool which focuses on detecting Java source code plagiarism in source codes. It uses Latent Semantic Analysis (LSA) for the dimension reduction, and the cosine distance for classification. LSA is an information retrieval technique used to discover relationships between extracted terms. The terms extracted by PlaGate do not contain any hierarchy or condition information for tests, and these terms are never combined. As a result, it is possible that two different source codes are wrongly flagged as plagiarism. This situation can occur if both source codes use the same similar identifier names. At the best of our knowledge, there is no release of PlaGate.

3. JavaNeighbors

Text mining is a process of retrieving meaningful information in various texts. In our case, texts are source codes. The choice of the information to mine is crucial for JavaNeighbors. Text mining terms are defined according to our case:

- **A term** is the smallest unit used and extracted directly from the source code. A term is also known as a feature in machine learning. \( \mathbb{T} \) is a finite set of all terms and \( \text{card}(\mathbb{T}) = T \).
- **A document** is a method from the source code and it is composed of terms. Let \( m \) be a raw count vector of terms.
• A corpus is a set of documents \( D \) such that \( \text{card}(D) = D \).

JavaNeighbors performs in four steps and it takes source codes as an input. Figure 1 shows the working of the tool. The feature extraction technique aims at representing a corpus in a document-term matrix. To achieve this, the Bag of Words technique takes a source code to analyze as an input. Then, it extracts terms (or features) from it and gathers them in a numerical document-term matrix \( M \). The set \( T \) is composed of three different kind of terms from four sets \( S, E, C, A \):

- State transformation terms \( S \) and \( E \): State transformation terms are the combination of a \( S \) term with a \( E \) term: \( SE \). \( S \) terms are state such as \( S = \{ s_1 \cdots s_n \} \) corresponding to Java API types, for example KeyPair. A transformation term \( E \) is defined as \( E = \{ e_1 \cdots e_m \} \), the method called from this state term. As an example, KeyPair can be combined with doFinal resulting in a valid couple \( (s_i, e_j) \) term: \( (\text{KeyPair}, \text{doFinal}) \). Finally, \( \text{card}(SE) = E \).

- Control flow terms \( C \): They are a set of control flow terms \( C = \{ c_1 \cdots c_k \} \). JavaNeighbors gathers in a same term tests from \textit{if} and \textit{case}. \textit{If} and \textit{case} are abstracted as \textit{TEST}. In addition, a control flow term is combined with the purpose of its test: a valid \( SE \) term, a constant in a case of comparison or a generic term if none of these is used.

- Loop statements such as \textit{while}, \textit{for}, etc. are combined with their depth, if nested. Finally \( \text{card}(C) = C \).

- Java Card API method calls terms \( A \): The set \( A \) contains the allowed Java Card API method calls \( A = \{ a_1 \cdots a_l \} \). For example, selectingApplet() is a valid \( A \) term. Finally \( \text{card}(A) = A \).

With these sets, \( T \) is defined as \( T = \{ SE \} \cup \{ C \} \cup \{ A \} \).

Let \( m_1 \) and \( m_2 \) be two different raw count vectors, such that \( m_1 = \{ t_1 \cdots t_T \} \) and \( m_2 = \{ t_1 \cdots t_T \} \), where \( t \in T \). Finally, methods \( m_1 \) and \( m_2 \) are neighbors if \( m_1 \approx m_2 \). The document-term matrix \( M \) with method vectors as rows is created as in Equation (1).

\[
M = \begin{pmatrix}
se_1 \cdots se_S & c_1 \cdots c_C & a_1 \cdots a_A \\
\cdots & \cdots & \cdots \\
se_1 \cdots se_S & c_1 \cdots c_C & a_1 \cdots a_A
\end{pmatrix}
\begin{pmatrix}
m_1 \\
\cdots \\
m_D
\end{pmatrix}
\]  

(1)

Weighting Matrix. So far, the current distance-term matrix is actually a weighted matrix raw counting the occurrences of the terms. Its weighting can be adjusted by combining a local, a global weighting and a normalization scheme in order to modify the importance of terms. However, this study focuses on the local and the global weighting schemes.

- **Local weighting scheme** It adjusts the current weight of a term, based on other terms used within this same document. For example, it can be used to normalize the weight based on the current document length. JavaNeighbors uses the raw count (RC) of terms.

- **Global weighting scheme** It adjusts the current weight of a term, according to its usage in the overall corpus. In order to improve the quality of neighbors discovered, JavaNeighbors has to put in front rare terms which are more meaningful. To achieve this, JavaNeighbors uses the Inverse Document Frequency \( \text{IDF} \) \cite{8}. As a result, if a term is used in all documents of the corpus, then its weighting value is set to zero. On the contrary, its weight is increased depending its rarity.
formation becomes rare in this space. LSA [9] is a dimension reduction technique. It shows co-occurrences between terms. This relation is called a concept. LSA reduces to k concepts all the terms used in the corpus. The data set to analyse might change in the number of terms extracted. As a result, JavaNeighbors sets the value of k to 40% of the total number of different terms extracted.

Distance calculation. From modified $M$ by LSA, JavaNeighbors calculates the document-document matrix. $M$ is transformed as $M = M \cdot M^T$, where $M^T$ is the transpose of $M$. Now, each row (or column) represents a document vector. From this last matrix, JavaNeighbors calculates the distance row pairwise of this matrix. In Natural Language Processing, the cosine distance [10] is suitable since it compare the angle between two vectors. However, in source code processing, the use of the Bray-Curtis dissimilarity distance measure (sometime entitled Sørensen distance measure) [10] is more suitable because it extrapolates the quantities between dimensions of vectors.

Fig. 1. Steps of the design of JavaNeighbors. Let $m_1$, $m_2$ and $m_3$ be methods. In addition, let $t_1$, $t_2$, $t_3$ and $t_4$ be terms from $T$ used in these three methods. For the example, LSA reduces $M$ term dimension to two concepts: $k_1$ and $k_2$.

4. Neighborhood discovery performance comparison

OpenPGP [11] is a free version of the encryption standard Pretty Good Privacy. An applet is a compiled Java Card program embedded in smart cards. This data set is composed of four source codes of applets implementing the v2.0 of the OpenPGP specification [11] and they are available online. Table 1 shows both the lines of code (LOC) and the number of methods in each of these OpenPGP implementations. In [12], the authors expect the analyst to manually modify the data set’s source code in order to reduce the wrong anomaly score assignment during the ChuckyJava, known as normalization step. Since it affects ChuckyJava’s neighborhood discovery, we have manually performed these modifications on the data set.

In ChuckyJava, the neighborhood discovery of a method is performed in several steps. First, it extracts the API symbols used within the method under analysis. Those symbols are identifiers: variables, constants and method call names. For each of these symbols extracted, ChuckyJava performs the following.

1. The methods which do not use the API symbol under observation are filtered out.
2. **ChuckyJava** distances are calculated based on raw count vectors of extracted caller names and types of both parameters and local variables. The vectors are gathered in a matrix. Then the vectors of the matrix are adjusted (rewarded/punished) according to similar caller and file names. As an example, two different methods named `process` get a reduction of distance together. On the contrary, the distance is increased.

3. The neighbors for the current method for the API symbol under observation are returned. Results might differ from another API symbol analyzed within the same method.

Sometimes, **ChuckyJava** is not able to return any result. This happens if the API symbol under observation is only used in the method to analyse. This is a synonym problem. As an example, two neighbor methods can use an object, but named differently in both method (`buffer` or `myBuffer`). Those symbols are considered as different. This is the same result for method call names. The normalization step helps to mitigate this limitation, but it is not enough to completely solve this problem.

In addition, an analyst has to provide to **ChuckyJava** a specific number of expected neighbors. If this number is higher than the real number of methods using the API symbol, then **ChuckyJava** is not able to return any result. **ChuckyJava** has been instrumented to return results if at least two different methods share the same API symbol, regardless of the number of neighbors expected. However, the first limitation is a core feature of the tool’s algorithm and it cannot be mitigated. In comparison, **JavaNeighbors** does not require any normalization step and the analysis perform in one iteration.

In this data set, **ChuckyJava** does not return any result 15% of the time. Because of this, we are not able to draw its ROC curve. This curve exposes the True Positive Rate (TPR) by the False Positive Rate (FPR) for results. However, for the same FPR of 0.025, **ChuckyJava** obtains 0.35 as TPR as best tradeoff value, whereas **JavaNeighbors** obtains 0.60 as TPR.

Both of the methods `verify` and `process` mentioned during the experiments are the ones involved about the anomaly detected in the **ChuckyJava** adaptation paper [2]. The method `verify` aims at verifying one of the passwords, depending on the arguments sent. The `process` method aims at analyzing a message received by the applet, and execute a specific command according to this message’s arguments. We focus on the comparison for those two methods, and give the anomaly detection results with the new neighbors discovered by **JavaNeighbors**. This score is comprised between −1.00 and 1.00. The former is interpreted as the method is the only one in the neighborhood to perform an operation known as `missing-check` or `missing-assignation`. On the contrary, an anomaly score of 1.00 informs an analyst that the method is the only one not performing an operation in the neighborhood, known as `extra-check` or `extra-assignation`. This experiment is twofold. It shows the results of the neighborhood discovery and then check the anomaly detection based on the **JavaNeighbors** neighbors.

In the data set, each applet contains a `process` method. Table 2 exposes the results we have obtained for both neighborhood discovery tools on `OP verify` and `MP process`. For the `verify` method, only `MP` implements it within its `process` method. As a result, no `MP verify` is expected to be found. On the contrary, each applet of the data set implements a `process` method, and they have to be returned in the results. Based on our distance results, the threshold for the neighborhood discovery step of **JavaNeighbors** is set 0.30.

**Missing-check of OFFSET.** 6. **ChuckyJava** calculates an anomaly score of 0.67 based on the wrong neighbors as exposed. With such a score, an analyst has to manually check the source in order to assess if there is a vulnerability. This anomaly can even be filtered out by an anomaly score threshold. The anomaly score obtained with **ChuckyJava** is explained because, according to the **OpenPGP** specification, the `changereferencedata` has to check the first parameter byte `P1` as `verify` does. With **JavaNeighbors** we are able to compare the `verify` methods together. The

<table>
<thead>
<tr>
<th>Implementation</th>
<th>LOC</th>
<th>Number of method</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenPGPApplet (OP)</td>
<td>2143 (34%)</td>
<td>66 (48%)</td>
</tr>
<tr>
<td>JCOOpenPGPApplet (JP)</td>
<td>1449 (23%)</td>
<td>23 (17%)</td>
</tr>
<tr>
<td>MyPGPId (MP)</td>
<td>1423 (23%)</td>
<td>23 (17%)</td>
</tr>
<tr>
<td>FluffyPGP (GP)</td>
<td>1276 (20%)</td>
<td>25 (18%)</td>
</tr>
<tr>
<td>Total</td>
<td>6291 (100%)</td>
<td>137 (100%)</td>
</tr>
</tbody>
</table>

Table 1. OpenPGP implementation applets data set
Table 2. First four results for both ChuckyJava and JavaNeighbors of the neighborhood discovery for methods OP verify and MP process sorted in ascending order.

anomaly score we have obtained is 1.00, because all the verify methods actually check byte P1, but the one from OpenPGPApplet.

Wrong usage of the CLA byte. ChuckyJava compares the process method with others semantically not similar. Since MP changeresetcv and performsecurityoperation are not required to assign a value to the CLA byte, according to the specification, it detects the anomaly with a score of $-1.00$. On the contrary, process methods do not have to modify the value of the CLA byte. JavaNeighbors is able to detect this anomaly with the same anomaly score of $-1.00$, but based on the correct neighbors.

JavaNeighbors has shown better results for the neighborhood discovery than ChuckyJava. In addition, it has removed some limitations of ChuckyJava. The first one is about the filtering step. In JavaNeighbors, no method is excluded for the neighborhood discovery because of differences in variable names used. Moreover, JavaNeighbors is able to detect neighbors, regardless of the file, method and caller names. As a result, it prevents the synonym problem for names. JavaNeighbors returns the neighbors in only one iteration, instead of each API symbols used in the method as like ChuckyJava does. Finally, JavaNeighbors does not require a normalization step since it does not rely on identifier or method names, but on Java API state and transformation terms.

5. Conclusion and future work

We have formulated our definition of neighbor methods. Based on it, we have designed a neighborhood discovery tool JavaNeighbors. It is based on the Machine Learning classifier algorithm: k-Nearest-Neighbors. On the contrary of ChuckyJava, JavaNeighbors relies on a Natural Language Processing technique and LSA to adjust the terms importances and reduce the dimension. Thanks to this, JavaNeighbors solves many limitations of ChuckyJava. In addition, JavaNeighbors performs with a better accuracy of a factor 1.7 for the TPR and FRP tradeoff, based on our data set. Moreover, JavaNeighbors executes the neighborhood discovery 5 times faster than ChuckyJava on this data set. Our future work consists in improving the anomaly detection algorithm of ChuckyJava. This could lead our work to increase its ability to detect anomalies in source codes.

References