Search Based Clustering for Protecting Software with Diversif ed Updates

Mariano Ceccato[1], Paolo Falcarin[2], Alessandro Cabutto[2], Yosief Weldezghi Frezghi[1], and Cristian-Alexandru Staicu[3]

[1] Fondazione Bruno Kessler, Trento, Italy
ceccato@fbk.eu, frezghi@fbk.eu
[2] University of East London, London, United Kingdom
falcarin@uel.ac.uk, a.cabutto@uel.ac.uk
[3] Department of Computer Science, TU Darmstadt, Germany
cristian-alexandru.staicu@crisp-da.de

Abstract. Reverse engineering is usually the stepping stone of a variety of attacks aiming at identifying sensitive information (keys, credentials, data, algorithms) or vulnerabilities and f aws for broader exploitation. Software applications are usually deployed as identical binary code installed on millions of computers, enabling an adversary to develop a generic reverse-engineering strategy that, if working on one code instance, could be applied to crack all the other instances. A solution to mitigate this problem is represented by Software Diversity, which aims at creating several structurally different (but functionally equivalent) binary code versions out of the same source code, so that even if a successful attack can be elaborated for one version, it should not work on a diversif ed version. In this paper, we address the problem of maximizing software diversity from a search-based optimization point of view. The program to protect is subject to a catalogue of transformations to generate many candidate versions. The problem of selecting the subset of most diversif ed versions to be deployed is formulated as an optimisation problem, that we tackle with different search heuristics. We show the applicability of this approach on some popular Android apps.

Keywords: software diversity, clustering, obfuscation, security

1 Introduction

The latest BSA Global Software Piracy Study[4] states that 39% of software installed on computers around the world in 2015 is not properly licensed, amounting to $52 billion in losses due to unlicensed software; the same study shows that malware often spreads through unlicensed software distributed on the internet, causing a wider number of security attacks and consequent revenue losses. In particular, the 98% of mobile apps lack binary code protection and they can be easily reverse engineered and modif ed[5]. Software vendors need effective solutions to contrast Man-At-The-End attacks [11],

[4] BSA Global Software Piracy Survey: http://globalstudy.bsa.org/2016/
[5] State of Application Security: https://www.arxan.com/resources/state-of-application-security/

where the end user is the attacker, owning the device running the software, and able to reverse engineer and modify the code, in order to use and spread unlicensed copies.

Obfuscation is a common protection against reverse engineering, and it consists of semantic-preserving code transformations that make a program more difficult to understand by changing its structure, while keeping the original functionalities. A multitude of techniques to perform code obfuscation have been proposed [8]. From a security viewpoint, obfuscation can help software diversity so that an attacker can find more difficult to map critical code in one release to another one.

Diversified updates is a software protection technique that aims at mitigating the risk of such attacks. When a program is frequently updated with a different version, then an available crack can be used for a limited amount of time, until a diversified update is pushed. The deployed versions should be pairwise different from the ones previously deployed, such that an attack available for one version cannot be easily replayed on another version.

The open problem we want to tackle is how to determine whether the subsequent diversified version maximizes its own diversity with respect to the previous versions, mitigating the security risks by maximizing diversity.

In this paper, we propose a novel approach to generate diversified versions of the program to protect. These can be used in an update strategy aimed at limiting the time available to an attacker to be successful. Given the availability of a catalogue of transformations, first of all we propose a novel strategy to filter those that are not effective in achieving diversification. These transformations that remain after filtering are combined in all the possible permutations, to form the complete set of the candidate versions. Then, our second novel contribution is to formulate the identification of diversified versions as a clustering problem, to be addressed with search based optimization heuristics.

The paper is structured as follows. Section 2 presents our approach to generate diversified versions for updates. Then, in Section 3 we introduce our setting for the empirical validation, while Section 4 presents and comments the experimental results. Section 5 compares our approach to the related literature while Section 6 concludes the paper.

2 Automatic Generation of Maximally Diversified Versions

Software diversity aims at distribution of unique binaries, so that it become much less likely that a single attack will affect large numbers of targets, and as a consequence the impact of reverse engineering attacks will be reduced. The distribution of unique binaries also has the effect that attackers cannot simply analyse their own software copies to locate critical code in certain binary code sections, because such code might have been relocated in different sections due to binary code diversity.

2.1 Approach Overview

Our code protection technique based on diversified updates, consists in generating several structurally different (but functionally equivalent) binary code versions out of the

same source code such that they maximise their pairwise diversity. This protection strategy aims at reducing the exploitation of reverse engineering attacks: a successful attack on one code instance cannot be easily replayed on a diversif ed update.

Our approach is composed of the subsequent steps:

- A catalogue of code transformations are applied separately to the program to protect, so as to generate several distinct versions of the initial program;
- These versions are analysed, to f lter out transformations that do not work well on the current program;
- The remaining transformations are combined together (in all the possible combinations) to generate many versions candidate for updates;
- We measure the similarity among all the pairs of versions;
- Candidate versions are subject to clustering, to group in the same cluster all the versions that are very similar to one another;
- We select one version from each distinct cluster. Since the version selected in this way are different from one another, they can be used to support diversif ed updates.

2.2 Program Transformations

Code obfuscation aims at transforming a program such that it becomes much harder to understand and reverse engineer, while its observable behaviour remains the same.

Code obfuscation represents an available approach to generate versions with a high level of diversity, with the added value of thwarting code comprehension.

We adopted Zelix KlassMaster[6] a commercial obfuscation tool for Java and Android. Zelix KlassMaster provides several activation points for obfuscating Java classes. It also provides a way to prevent methods, classes and packages from being obfuscated, or to identify the portion of code to protect with obfuscation. The tool can be streamlined by the use of scripts, which make it very easy to automate.

Zelix KlassMaster supports 15 distinct conf guration parameters to control which transformations are activated and how they are conf gured. Among them, 8 parameters supports binary values, other 3 parameters have three possible values each, and the other two parameters allow four values each. This means that, potentially, a total of $2^8 * 3^3 * 4^2$ = 110,592 distinct obfuscated versions can be generated using this tool, just by resorting to its different conf gurations. Moreover, the number of versions can be further increased by selecting the subset of methods and/or classes on which to apply the obfuscation (instead of the whole application), but this dimension is not investigated in this study.

2.3 Similarity Metric

To quantify the similarity between two versions, we rely on the Normalized Compression Distance (NCD [14])[7]. The formula used to compute similarity is shown in Equa-

[6] `http://www.zelix.com/klassmaster/`

[7] Our approach is general, and it is compatible with any other pairwise similarity metric.

tion 1, where NCD is the Normalized Compression Distance and C_{rzip}[8] is the size of the compressed text.

$$S(v1, v2) = 1 - NCD(v1, v2) = 1 - \frac{C_{rzip}(v_1 v_2) - min(C_{rzip}(v_1), C_{rzip}(v_2))}{max(C_{rzip}(v_1), C_{rzip}(v_2))} \quad (1)$$

This metric is based on *rzip*, a lossless compression algorithm, to estimate the amount of common information shared among two documents. In fact, size reduction is achieved by removing repeated sub-sequences of bits.

If two versions v_1 and v_2 are very similar, the compression of the concatenation $v_1 v_2$ will not bring additional information and it will result in a size closer to the smaller of the two versions. Thus, the NCD distance will tend to zero and similarity (that is $1 - NCD$) will be close to 1.

Conversely, when v_1 and v_2 are different the size of the compression of the concatenation would tend to reach the sum of the sizes of v_1 and v_2, the distance will tend to one and similarity will tend to 0.

We base similarity computation on the textual representation of the Java code, obtained by executing the *javap* disassembler. We drop irrelevant information for disassembled code, such as constant headers, compilation info, comments, white lines and we replace the identif ers with labels. Eventually, we compute the similarity as specif ed in Equation 1 using *rzip* as compression algorithm. We used NCD metric implementation with *rzip* algorithm because its history buffer is wider than gzip, which is limited to 32 Kbytes [5].

2.4 Filtering Twin Obfuscations

Many versions can be generated by blindly combining all the available code obfuscation transformations. However, some of these distinct transformations in the catalogue could generate programs that are not so different, so they should be detected and excluded.

Since transformations can be combined, let's call the transformations in the catalogue the *atomic* obfuscations. If we consider m atomic obfuscations, we can elaborate $n = 2^m$ distinct combinations of atomic obfuscations to deliver n candidate *versions* for updates. Since the number of versions n is exponential in the number of atomic obfuscations m, we need to carefully select the m atomic obfuscation to keep, i.e. only the relevant ones.

When two atomic obfuscations are just small variations of the same transformation algorithm, or when they are two different algorithms that emit very similar obfuscated code (for example an atomic obfuscation only targeting and rewriting exception handling code may have little effect on an original application with few exception code blocks), it does not make sense to consider both of them for diversity. Including one of the two similar variants is enough, and the other can be considered redundant: we propose to apply a preliminary f ltering to drop some of the m atomic obfuscations from the search space, when they are not promising as a diversif er component for the application. When two atomic obfuscations a and b are very similar to each other, we call a and b *twin* obfuscations.

[8] https://rzip.samba.org/

Our approach to detect twin obfuscations and f lter them out is as follows:

- We consider only the atomic obfuscations, i.e. each version is obtained by applying only an atomic obfuscation from the catalogue: in this way, we only obtain m versions;
- We compute the pairwise similarity of these m versions. Similarity values are stored in a similarity matrix of size mxm. A value in the similarity matrix in the i-th row and j-th column represents the similarity between version i and version j;
  For each atomic obfuscation a, the a-th row in the similarity matrix represents the *signature vector* X_a. The signature vector contains the similarity values between a and all the other $m - 1$ obfuscated versions. The b-th element of this vector, namely $X_a(b)$, represents the similarity between code obfuscated with a and code obfuscated with b.
- Two atomic obfuscations are *twins* when their signature vectors are very similar, i.e. the two transformations generate code with the same values of similarity when compared with the same alternative versions. We compute the *twin value* $t_{a,b}$ between atomic obfuscation a and b as the square of the distance between their signature vectors X_a and X_b with the sum of squared residuals:

$$t_{a,b} = \sum_{i=1..n, i \neq a, i \neq b} (X_a(i) - X_b(i))^2$$

- When all the pairwise twin values $t_{x,y}$ are available (one for each obfuscation pair (x, y)), we sort them in ascending order to detect the most likely twins;
- We exclude the twins by excluding the atomic obfuscations with lowest *twin values*. Let us say that $t_{a,b}$ is the smallest value among all the twin values (f rst value in the sorted set). At this stage, we can exclude either a or b. To decide which one to exclude, we consider the next twin value $t_{x,y}$ (in the sorted twin values in ascending order). There could be three cases:
  - $(x = a) \vee (y = a)$: we make the decision to exclude a;
  - $(x = b) \vee (y = b)$: we make the decision to exclude b;
  - $(x \neq a) \wedge (y \neq a) \wedge (x \neq b) \wedge (y \neq b)$: we make no decision at this point and we iterate. We consider the next twin value $t_{w,z}$ in the sorted list, and we compare a and b with w and z.

There are multiple strategies to decide when to stop excluding twin obfuscations. A possible strategy is to set a threshold and exclude atomic obfuscations whose twin values are below the threshold. Alternatively, we can set a target size m_{max} for the number of atomic obfuscations and stop f ltering when this target is met, i.e. when $m \leq m_{max}$.

In this work, we opted for the second strategy. We set the upper limit to the number of versions n_{max} to 500. Therefore, the number of atomic obfuscations m is approximately[9] 9 ($2^9 = 512$). Eventually, the number of pairwise similarity values k to measure is 130,816, in fact the distinct pairs of n versions are $k = n(n - 1)/2$.

[9] The number of atomic obfuscations m can be actually larger, because some combinations cause and error in the obfuscation tool, or simply do not work. Thus, more atomic obfuscations are required to meet the target number of versions n.

Anyway, this f ltering strategy is required to keep the number of versions to generate and the number similarity values to measure limited to a tractable size. Anyway, the exact solution to the clustering problem is still intractable (see Section 2.5).

2.5 Clustering Based on Similarity

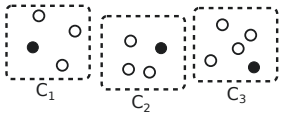

Fig. 1. Diversif ed updates based on clustering for similarity.

We formulate the problem of computing the set of maximally dissimilar versions as a clustering problem, as shown in the example in Figure 1. Clustering is used to partition the available versions into groups that contain very *similar* versions, three groups in the example. Versions from the same cluster (e.g., in C_1) are very similar to each other, so they cannot be used in the same update plan. The f nal set of versions to be used as updates is selected by taking just one element from each high-similarity group, they are the black elements in Figure 1. In this way, very similar versions are never used in the update plan. Clustering is driven by the similarity metric def ned in Equation 1.

Given a partition of all the available versions into similarity clusters, we def ne the *intra-similarity* A_i of the cluster i as the average similarity of all the pairs of elements in the cluster:

$$A_i = \frac{\sum_{v_1,v_2} S(v_1, v_2)}{|C_i|(|C_i| - 1)/2}, \quad \forall v_1, v_2 \in C_i \tag{2}$$

We def ne the *inter-similarity* between two clusters C_i and C_j as the average similarity of the versions from the two clusters:

$$E_{i,j} = \frac{\sum_{v_1,v_2} S(v_1, v_2)}{|C_i|\,|C_j|} \quad v_1 \in C_i, v_2 \in C_j \tag{3}$$

Considering that our objective is to search for a clustering conf guration whose clusters contains elements as similar as possible (high intra-similarity) and low similarity between elements from different clusters (low inter-similarity), we def ne the overall *similarity quality* among the clusters as the average intra-similarity minus the average of all the inter-similarity:

$$SQ = \frac{1}{n_c} \sum_{i=1}^{n_c} A_i - \frac{1}{\frac{n_c(n_c-1)}{2}} \sum_{i,j=1}^{k} E_{i,j} \tag{4}$$

where n_c is the number of clusters in the partition to evaluate.

At this stage, the software diversity problem can be expressed as a search problem, aiming at f nding the clustering partition that maximize the similarity quality SQ.

2.6 Search Strategies

The analytic solution of clustering is intractable [23], because the number of potential solutions to the clustering problem is exponential in the number of elements to cluster. Considering that the number of candidate versions for update are hundreds of thousands, we adopt search heuristics. They are *Greedy agglomerative clustering*, *Hill climbing* and *Single objective genetic algorithm*.

Greedy agglomerative clustering: Agglomerative clustering is a greedy algorithm to f nd a candidate good partition in the search space. This algorithm starts from an initial conf guration, where each element is assigned to a different cluster. At each step, inter-/intra-similarity are computed and the two most similar clusters (those with the highest inter-similarity) are merged to form a single cluster. This process is iterated and, at each step, the total number of cluster decreases by 1. The iteration terminates when all the clusters are merged in a single f nal big cluster.

During this process, we record the similarity quality SQ of all the visited conf gurations, and the one with the highest value represents the f nal optimal solution.

This algorithm produces candidate clustering conf gurations with decreasing number of clusters, in the interval $[0, n]$. However, solutions with too few clusters are not relevant to solve our problem, even if their similarity quality SQ would be very high, because not enough versions would be available for updates. Thus, we consider interesting only those clustering conf gurations with a number of clusters above a threshold, that we set to 10.

Hill climbing: Hill-climbing starts from an initial random conf guration of clustering. At each step, neighbour solutions are considered and one of them is randomly chosen among those that improve the f tness function SQ of the current clustering conf guration. This process is iterated until no better solution can be found in the neighbourhood. However, given the huge space of the neighbour conf gurations, only a subset of it is probed, and this subset is selected choosing 100 conf guration with uniform probability among all the neighbour cases.

Neighbour solutions consist of all the clustering conf gurations that can be obtained from the current clustering conf guration with an atomic change. An atomic change consists of applying one of these mutation operators:

- (i) Moving one element from a cluster to another cluster; and
- (ii) Removing one element from a cluster and create a brand new cluster with just this element;

The search stops when no neighbour can be found that improve the f tness function or the search budget is consumed.

Single objective genetic algorithm: Genetic algorithms are a family of optimization heuristics inspired by biological evolution. A population of solutions is evolved by giving higher probability of recombining to solutions with higher values of a *f tness function*. The aim is to push the population to evolve and explore the part of the solution space with better and better values of f tness function. In particular, we adopt a steady state genetic algorithm. In this variant, offspring replace the parents at each iteration regardless of their f tness function [2].

In our case, the population of solutions is represented by clustering conf gurations. For a clustering conf guration, the f tness function is represented by the similarity quality SQ.

The initial population is represented by 100 versions, including random clustering conf guration. At each evolution iteration, we *select* 70% of the population, using linear ranking selection with a selection pressure sp of 1.5. The selected versions are paired randomly. Each of these pairs of solutions undergoes *crossover* with rate of 0.5.

Crossover, consists in elaborating two brand new solutions (offspring), based on the two selected solutions (parents). Let's assume that the two parents, namely clustering C_1 and clustering C_2, contain respectively n_1 and n_2 clusters. Two cross points r_1 and r_2 are randomly selected, such that $r_1 < n_1$ and $r_2 < n_2$. Then, r_1 clusters are randomly selected from C_1 and r_2 clusters from C_2 to form the new C_3 offspring conf guration. The remaining $n_1 - r_1$ clusters from C_1 and $n_2 - r_2$ clusters form C_2 are used to create the new C_4 offspring conf guration.

At this stage C_3 and C_4 could be invalid clustering conf gurations, because they could contain repeated elements or they could miss elements, so they should be f xed. In case an element is repeated, one instance of the repeated element is randomly selected and removed. Conversely, if an element is missing, it is added to a random cluster.

In steady state GA, when crossover takes place, only offspring survives for the next generation while parents do not [22]. Otherwise, if there is no crossover, the parents survive for the next generation. The offspring is subject to mutation with a rate of 0.03. Mutation operators are the same operators used to visit the neighbourhood in hill climbing search strategy.

The search stops when the search budget is consumed or when a plateau is reached, i.e. no improvement in the population after 100 iterations.

3 Experimental Settings

3.1 Research Questions and Variables Selection

Our experimental investigation aims at answering the following research questions:

- **RQ$_0$:** What is the interval of validity of the normalized compression distance?
- **RQ$_1$:** What is the distribution of *Similarity* among all the version pairs?
- **RQ$_2$:** Is f ltering effective in discarding useless obfuscations?
- **RQ$_3$:** How many diversif ed versions can be identif ed by the search heuristics?

RQ$_0$ is a sanity check, to verify that we are using the metric in the correct interval of validity. RQ$_1$ aims at studying how values of Similarity are spread. Then, RQ$_2$ is intended to validate the f ltering procedure that we proposed. We adopted a f ltering procedure to control the (exponential) number of versions to consider, by excluding those obfuscations that are not effective in generating diversif ed versions. Eventually, the last research question RQ$_3$ directly compares the search strategies, to identify the most effective to solve the software diversity problem.

To answer these research questions, we measure and collect the following variables:

- *Similarity:* the similarity among version pairs based on the compression size (as def ned in Section 2.3);
- *Similarity Quality:* the f tness function (as def ned in Section 2.5) to compare clustering conf gurations; and
- *Number of Clusters:* how many clusters are in a clustering conf guration. This number corresponds to the number of diversif ed versions that can be used as diversif ed updates.

3.2 Experimental Procedure

The empirical investigation is conducted according to the following experimental procedure:

- The original version of an app (as it is distributed by the apps market) is subject to all the atomic obfuscation transformations available in Zelix KlassMaster (no combinations of obfuscations);
- Twin obfuscations are then detected and excluded for this particular app;
- The remaining atomic obfuscation transformations are applied to the app, in all the possible combinations, resulting in the versions candidate for diversif ed updates;
- Pairwise similarity is computed among all the pairs of these versions;
- The search heuristics (agglomerative clustering, hill climbing and genetic algorithm) are applied to compute optimal clustering based on similarity.

Agglomerative clustering is a deterministic algorithm and it requires a f xed number of f tness function evaluations, that is equal to the number of versions to group into the clusters. Conversely, hill climbing and genetic algorithm are non-deterministic, so we set a search budget: in particular, they are stopped after 100.000 f tness function evaluations or when a plateau (a local optimum) is detected.

3.3 Subject Apps

We apply the experimental procedure on several real world Android apps. We select 10 from the most popular apps as ranked in the off cial Android store, namely Google Play (data collected in 2013). They spread on different categories (utility, social network, games, voip, internet browser) and their popularity goes from half a million to 500 millions of downloads. Their size is between 100kB to almost 10MB. the smallest apps contain about 200 classes, while the largest apps contain about 10,000 classes.

4 Results

4.1 RQ0: Validity of the Normalized Compression Distance

As shown by Cebrián et al. [5], metrics based on the Normalized Compression Distance provide reliable results in an limited interval. In particular, NCD metrics give unreliable results when size of the f le to compress is lager than the sliding window used by the

compression algorithm. For example, Cebrián et al. reports that *gzip* can be used for f les up to 32Kb.

Here we adopt a validation procedure similar to the one used by Cebrián et al., i.e. we study the *idempotency* property of NCD based on *rzip* that requires $NCD(x, x) = 0$. We take a large text f le and we truncate it to have a shorter f le x. Then we plot $NCD(x, x)$ for increasing size of x, from 0 to 1GB with steps of 16MB.

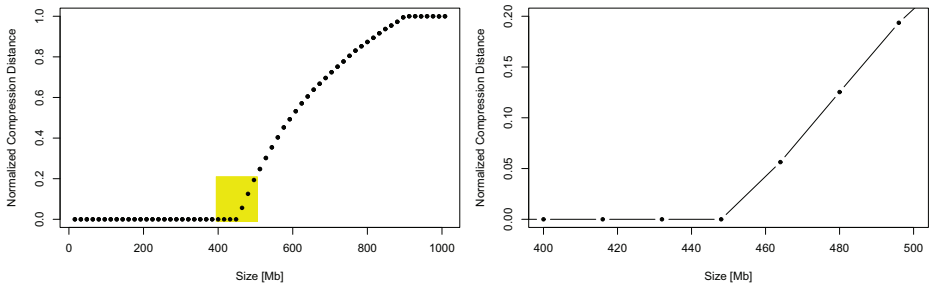

Fig. 2. Interval of validity of the Normalized Compression Distance.

Results are shown in Figure 2, left-hand side plot. The most interesting region is highlighted in yellow and detailed in the right-hand side plot. The idempotency property (zero distance between x and x) is satisf ed when the size of f les is lower than 448MB. NCD values are not reliable for larger f les.

For the subsequent experiments, the size of decompiled code will be lower than 20MB, so the NCD metric is used in its interval of validity.

4.2 RQ1: Distribution of Similarity

First of all, we examine the distribution of the values of similarity. Figure 3 show the histogram of *Similarity* for *Skype*. The histogram contains all the versions, after f ltering twin obfuscations, for approximately 130,000 pairs.

As we can see, values of similarity are clustered in two groups. A f rst group that contains quite dissimilar pairs is centred in 0.4, ranging mostly in the interval $[0.1, 0.5]$. The second group contains quite similar pairs and it is centred in 0.8. Probably, diversi- f ed updates will be selected among versions whose similarity falls in the f rst group.

4.3 RQ2: Effectiveness of Filtering

Table 1 shows which atomic obfuscations remain after applying f ltering, more precisely, which atomic obfuscations are combined to diversify the code. A check mark shows when an atomic obfuscation (column) passes f ltering and so it is used to generated candidate diversif ed versions for a case study (row). The last row summarizes on how many apps each obfuscation has been applied. As we can see, the set of obfuscations that passes f ltering is quite different among different apps. Some obfuscations

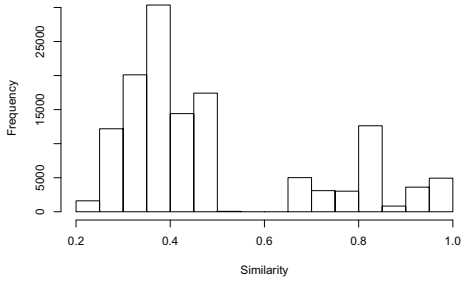

Fig. 3. Histogram of Similarity in Skype.

Table 1. Obfuscation transformations that pass f ltering.

App	Atomic obfuscations															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
airdroid	√		√		√	√	√		√	√	√	√		√	√	√
chrome	√	√	√		√		√				√	√	√		√	√
contacts	√	√	√	√			√	√			√	√	√		√	√
esx-f lexplorer	√		√	√	√					√	√	√	√		√	√
facebook	√		√		√		√		√		√	√	√		√	√
gotetris	√	√	√	√	√		√	√		√			√	√		√
opera	√	√	√	√	√					√		√	√	√		√
skype	√	√	√	√	√					√	√	√	√		√	√
twitter	√	√	√	√		√	√	√	√	√	√	√	√	√	√	√
wordfriends	√			√			√			√	√				√	√
Tolal	10	6	9	7	7	2	7	3	3	7	8	8	8	4	8	10

are applied to most of the case studies (two obfuscations are applied to all 10 apps, an obfuscation to 9 apps and four obfuscations are applied to 8 apps), while others are used less frequently (one obfuscation is applied on 2 apps and two obfuscations are applied to 3 apps).

This suggests that the f ltering step is quite app dependent, because the effectiveness of atomic obfuscation transformations in diversifying the code indeed depends on the code to transform. Thus, there is no universal rule on what atomic obfuscations to adopt in general when diversifying the code. The f ltering step shall be repeated for each app that we want to diversify.

It should be noted that this f ltering step is fully automatic, based on the algorithm presented in Section 2.4.

Due to the fact that the obfuscation tool Zelix KlassMaster (that we do not control) fails to generate certain conf gurations, the number N of the atomic obfuscations required to reach n_{max} combinations is different for different case study apps.

4.4 RQ3: Diversif ed Versions

After f ltering twin obfuscations, we applied the three search heuristics to the subject apps, to see how many diversif ed versions they are able to identify.

Table 2. Results of clustering.

App	Agglomerative clust.		Hill climbing		Genetic algorithm	
	SQ	N	SQ	N	SQ	N
airdroid	**0.3533**	13	0.3377	24	0.2093	**35**
chrome	**0.4547**	10	0.4148	28	0.2332	**35**
contacts	**0.5431**	15	0.4786	23	0.2447	**34**
esx-f lexplorer	0.1637	11	**0.3193**	27	0.2068	**107**
facebook	-0.5674	14	**0.0017**	17	-0.1105	**27**
gotetris	**0.3927**	12	0.3711	32	-0.0346	**34**
opera	0.2934	16	**0.3854**	**26**	0.2360	41
skype	**0.4351**	10	0.4287	32	0.2502	**96**
twitter	**0.4337**	13	0.4255	24	0.2562	**41**
wordfriends	-0.5792	12	**0.0011**	10	-0.1991	**15**
Average	0.1923	13	0.3164	24	0.1292	46

Table 2 compares the results of the three search heuristics on the 10 apps, relevant values are highlighted in boldface. We observe negative values of similarity quality SQ when, according to Equation 4, the inter-similarity term $E_{i,j}$ prevails on the intra-similarity term A_i.

Agglomerative Clustering was able to elaborate the most diversif ed versions for the majority of the cases (for 6 out of 10 apps), because the corresponding clustering conf gurations score the highest values of Similarity Quality. Hill climbing elaborated conf gurations that were always more diversif ed in the other four cases.

Considering the number of clusters, the Genetic Algorithm was able to identify the largest set of diversif ed versions in almost all the apps (9 out 10 apps). In two of them, the number of diversif ed versions was quite impressive (107 versions for *esx-f lexplorer* and 96 versions for *skype*) however the corresponding Similarity Quality was low, but still comparable with the values obtained with the other two approaches. Hill Climbing elaborated optimal conf gurations with many clusters for the remaining app (i.e., *opera*). Eventually, the greedy algorithm elaborated large sets of diversif ed versions for no app.

5 Related Work

The concept of software diversity has interested researchers for many years [12], but only recently software diversity has become practical due to cloud computing enabling the computational power to perform massive diversif cation [19]. In the existing literature [13, 17, 10, 1], software diversity relied on random generation of different diversif ed copies, starting from the same source code. A recent survey from Larsen et al. [20] compares the different approaches for software diversity in terms of performance and security.

Most of the past software diversity approaches have been based on some form of obfuscation [7], load-time binary transformation [18], virtualization obfuscation based on customized virtual machines [16], or operating system randomization [30]. Current software diversity approaches exploit the intrinsic randomness of compiler optimizations, extending the initial idea of Forrest et al. [12] of compiler-guided code variance. Other approaches rely on binary transformation based on a random seed [26], or multi-compilers and cloud computing [13] to create a unique diverse binary version of every program, and they apply such diversif cation for mobile apps [17]. The XIFER framework [10] randomly diversif es Android apps at load-time by means of a binary rewriter. However, such diversif er can be disabled or tampered with because it is running on the Android device and because the original app is available to the attacker before it is loaded and diversif ed by the XIFER framework.

A previous work by Anckaert et al. [1] applied regular compiler transformations (e.g., optimizations) in a stochastic manner to generate diversif ed binary code versions, with random seeds to vary compiler parameters. However, there is no guarantee that two versions generated with different random seeds will not converge to "similar" code. Anckaert et al. do not tackle the problem of measuring the diversity among the different versions, which is necessary for performing a diversity evaluation. Coppens et al. [9] apply binary diversif cation changing a random seed and they iteratively compare it with the previous one till they get a new version different enough from the previous version; however they search just one version, and not the best subset of versions like in our approach. Diversity has also been applied to improve security in different research lines: code randomization has been used to defend against code-reuse attacks [25], return-oriented programming attacks [15], code injection attacks [28].

The novelty of our approach is that we are the f rst to tackle the problem of searching the most diversif ed versions with meta-heuristics, to guarantee that the deployed versions will be effectively different from one another, basing on the similarity metric chosen. Similarity can be measured with source code metrics to detect plagiarism in text and programs [14], or binary metrics in antivirus systems [29]. Other approaches using search-based heuristics, like genetic programming, to achieve code transformation [27, 21], but with a different goal, i.e. to automatically f nd patches to f x bugs. Portions of the programs are replaced by their mutated versions that convey different semantics: mutation continues until the bugs are f xed and all test cases pass. In software diversity instead, we do not change the semantics of the program, but only its structure. Interesting developments can investigate the use of similarity metrics based on clone detection [3], which detects code shared by two software versions, or software birthmark [24], which compares intrinsic software properties rather than binary code structure. Other works [4] [6] evaluated the code complexity introduced by different obfuscation algorithms by using structural metrics, that should be instead kept low in refactoring.

6 Conclusion

In this work, we tackle the problem of maximizing software diversity by searching the best subset of diversif ed code versions to be deployed in parallel or within an up-

date plan. Many candidate diversif ed versions are generated using combinations of off-the-shelf obfuscation transformations, which can generate a huge number of possible versions; we proposed an algorithm to reduce the number of versions to generate, by discarding redundant obfuscations for the particular application code, and then we use clustering to identify the most different versions to deploy. The empirical assessment shows that our approach works in diversifying 10 popular Android apps.

As future work, we intend to investigate alternative metrics to compute similarity in a way that approximate more appropriately program difference from an attacker point of view. Moreover, we intend to conduct a user study where we measure the actual learning effect when attacking two consecutive versions. The aim of this study would be to quantify for real the effort required to adapt an attack when receiving an update.

Acknowledgement

The authors want to thank Prof. Mark Harman who was involved in the initial stages of this work, and contributed by suggesting the use of clustering for this search problem. This research has been funded by the European Union 7th Framework Programme (FP7/2007-2013), under grant agreement number 609734 - ASPIRE project (Advanced Software Protection: Integration Research and Exploitation), http://www.aspire-fp7.eu/.

References

1. Anckaert, B., De Sutter, B., De Bosschere, K.: Software piracy prevention through diversity. In: Proc. of the 4th ACM workshop on Digital rights management. pp. 63–71. ACM (2004)
2. Arcuri, A., Fraser, G.: On parameter tuning in search based software engineering. In: Search Based Software Engineering, pp. 33–47. Springer (2011)
3. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Trans. on Software Engineering 33(9), 577–591 (Sept 2007)
4. Capiluppi, A., Falcarin, P., Boldyreff, C.: Code defactoring: Evaluating the effectiveness of java obfuscations. In: Proc. of the 19th Working Conference on Reverse Engineering. pp. 71–80. WCRE ’12, IEEE (2012)
5. Cebrián, M., Alfonseca, M., Ortega, A., et al.: Common pitfalls using the normalized compression distance: What to watch out for in a compressor. Communications in Information & Systems 5(4), 367–384 (2005)
6. Ceccato, M., Capiluppi, A., Falcarin, P., Boldyreff, C.: A large study on the effect of code obfuscation on the quality of java code. Empirical Software Engineering 20(6), 1486–1524 (2015)
7. Cohen, F.B.: Operating system protection through program evolution. Computers & Security 12(6), 565–584 (1993)
8. Collberg, C., Nagra, J.: Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection. Addison-Wesley Professional (2009)
9. Coppens, B., De Sutter, B., Maebe, J.: Feedback-driven binary code diversif cation. ACM Trans. Archit. Code Optim. 9(4), 24:1–24:26 (Jan 2013)
10. Davi, L., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Xifer: A software diversity tool against code-reuse attacks. In: 4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3 2012) (Aug 2012)

11. Falcarin, P., Collberg, C., Atallah, M., Jakubowski, M.: Guest editors' introduction: Software protection. IEEE Software 28(2), 24–27 (March 2011)
12. Forrest, S., Somayaji, A., Ackley, D.H.: Building diverse computer systems. In: Operating Systems, 1997., The Sixth Workshop on Hot Topics in. pp. 67–72 (May 1997)
13. Franz, M.: E unibus pluram: massive-scale software diversity as a defense mechanism. In: Proceedings of the 2010 workshop on New security paradigms. pp. 7–16. ACM (2010)
14. Freire, M., Cebrian, M., del Rosal, E.: Uncovering plagiarism networks. arXiv preprint cs/0703136 (2007)
15. Gupta, A., Kerr, S., Kirkpatrick, M.S., Bertino, E.: Network and System Security: 7th International Conference, NSS 2013, chap. Marlin: A Fine Grained Randomization Approach to Defend against ROP Attacks, pp. 293–306. Springer (2013)
16. Holland, D.A., Lim, A.T., Seltzer, M.I.: An architecture a day keeps the hacker away. ACM SIGARCH Computer Architecture News 33(1), 34–41 (2005)
17. Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., Franz, M.: Compiler-generated software diversity. In: Moving Target Defense, pp. 77–98. Springer (2011)
18. Just, J.E., Cornwell, M.: Review and analysis of synthetic diversity for breaking monocultures. In: Proc. of the 2004 ACM workshop on Rapid malcode. pp. 23–32. ACM (2004)
19. Larsen, P., Brunthaler, S., Franz, M.: Security through diversity: Are we there yet? IEEE Security Privacy 12(2), 28–35 (Mar 2014)
20. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 276–291 (May 2014)
21. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. Software Engineering, IEEE Transactions on 38(1), 54–72 (2012)
22. McMinn, P.: Search-based software test data generation: A survey. Software Testing Verif - cation and Reliability 14(2), 105–156 (2004)
23. Michael, R.G., David, S.J.: Computers and intractability: a guide to the theory of np-completeness. WH Free. Co., San Fr (1979)
24. Myles, G., Collberg, C.: Information Security: 7th Intern. Conf. ISC 2004, USA, Proceedings, chap. Detecting Software Theft via Whole Program Path Birthmarks, pp. 404–415
25. Shioji, E., Kawakoya, Y., Iwamura, M., Hariu, T.: Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks. In: Proc. of the 28th Annual Computer Security Applications Conference. pp. 309–318. ACSAC '12, ACM (2012)
26. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: Proc. of the Fifth IEEE Int. Symposium on Signal Processing and Information Technology, 2005. pp. 7–12. IEEE (2005)
27. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically f nding patches using genetic programming. In: Proc. of 31st Int. Conf. on Software Engineering. pp. 364–374
28. Williams, D., Hu, W., Davidson, J.W., Hiser, J.D., Knight, J.C., Nguyen-Tuong, A.: Security through diversity: Leveraging virtual machine technology. IEEE Security Privacy 7(1), 26–33 (Jan 2009)
29. Wong, W., Stamp, M.: Hunting for metamorphic engines. Journal in Computer Virology 2(3), 211–229 (2006)
30. Xu, J., Kalbarczyk, Z., Iyer, R.K.: Transparent runtime randomization for security. In: Reliable Distributed Systems, 2003. Proc. 22nd Int. Symposium on. pp. 260–269. IEEE (2003)