Automatic Generation of Mobile Malwares Using Genetic Programming

Emre Aydogan and Sevil Sen

Department of Computer Engineering,
Hacettepe University, Ankara, Turkey
{emreaydogan,ssen}@cs.hacettepe.edu.tr

Abstract. The number of mobile devices has increased dramatically in the past few years. These smart devices provide many useful functionalities accessible from anywhere at anytime, such as reading and writing e-mails, surfing on the Internet, showing facilities nearby, and the like. Hence, they become an inevitable part of our daily lives. However the popularity and adoption of mobile devices also attract virus writers in order to harm our devices. So, many security companies have already proposed new solutions in order to protect our mobile devices from such malicious attempts. However developing methodologies that detect unknown malwares is a research challenge, especially on devices with limited resources. This study presents a method that evolves automatically variants of malwares from the ones in the wild by using genetic programming (GP). We aim to evaluate the efficacy of current anti-virus products, using static analysis techniques, in the market. The experimental results show the weaknesses of the static analysis tools available in the market, and the need of new detection techniques suitable for mobile devices.

Keywords: mobile malware, static analysis, obfuscation, evolutionary computation, genetic programming

1 Introduction

Mobile devices have become an inevitable part of our lives. They provide us many useful functionalities such as reading and writing e-mails, surfing on the Internet, showing facilities nearby, video conferencing, voice recognition, and the like. However the popularity and adoption of mobile devices also attract virus writers in order to harm our devices. According to Kaspersky security report [1], nearly 145.000 new malicious programs are appeared in 2013 and 98.1% of these programs target Android platform. Hence, in order to protect our devices from such threats, researchers and security companies have been working on developing effective and efficient anti-virus systems recently.

There are some techniques available for malware analysis and detection with varying strengths and weaknesses. Two common types of malware detection techniques according to how code is analyzed are static and dynamic analysis. Since dynamic analysis might not be affordable on some mobile devices due to

their strong limitations in terms of power consumption, most of the proposed approaches in the literature rely on static analysis up to date. However, these tools are known to be vulnerable to some obfuscation techniques and new attacks. How they are effective against known attacks, variants of known attacks, and unknown attacks needs investigation. This is the main aim in this study. In order to be able to assess the security solutions proposed for mobile devices, we automatically generate new attacks from existing ones.

In this study, we evolve new malwares, variants of known malwares, by using genetic programming (GP) in order to evaluate the performance of existing static analysis tools. The aim here is to generate new malwares automatically that we could strengthen our static analysis tools automatically as well. Because most of the existing static tools update their signature databases when they experience new/unknown malwares, automating this process will make our detection systems more robust against attacks. While the proposed approach here only automates the generation of new/unknown attacks here, the framework is planned to be extended with developing existing solutions automatically in the future.

GP has already been applied in some approaches in the literature [2], [3], [4], [5] in order to evolve new attacks and new malwares. However most of these approaches are not fully automatic and only proposed for a specific type of attacks. A security expert is generally needed to analyse the code and to extract parameters that changes in different variants of viruses' codes, so a representation of the problem could be constructed for GP. Here, we aim to create a fully automatic system by employing some obfuscation techniques on source codes of existing malwares by using GP. The results show that GP could generate effective attacks which are able to evade from existing eight anti-virus systems which are among the most successful mobile security solutions [6]. Furthermore, it is shown that GP shows [7], [8] better performance than solely applying obfuscation techniques as usually proposed in the literature. It is also shown that the approach could produce more evasive malwares than Zelix Klassmaster [9], which is a well-known Java bytecode obfuscator.

The rest of the paper is organized as follows: Section 2 summarizes the related approaches in the literature. In Section 3, we give some background information such as the details of Android system and the GP. In Section 4, we describe the proposed method for generating new malwares. The performance of the model is discussed in Section 5. Finally Section 6 is devoted to concluding remarks and future works.

2 Related Work

Researchers have been working on suitable security solutions for mobile devices in the last few years. Many security companies have already brought their mobile solutions out. Most of these solutions are based on static analysis due to their suitability to power-constrained mobile devices. Since many mobile anti-virus

solutions out now, how they are effective against known and unknown malwares needs investigation.

There are two main studies in order to evaluate commercial anti-virus products against obfuscation techniques on Android platform in the literature. In In [7], they proposed a system called ADAM that evaluates the effectiveness of anti-virus systems against malware samples which are generated by employing some obfuscation techniques automatically while preserving the original malicious function. Rastogi et. al. In [8] has developed a system called DroidChameleon that evaluates Android anti-malware products against obfuscation attacks that are extended form of the attacks in In [7]. They automatically mutate Android applications by using polymorphic and metamorphic techniques.

Christodorescu and Jha [10] proposed a technique based on a transformation of a source code for creating test samples for desktop malware detection systems. Their technique aims to evaluate the resilience of anti-virus systems to various obfuscation techniques. Morales et. al. [11] evaluate and test how anti-virus systems protect hand-held devices against known malwares. Their results indicate that all four anti-virus systems produce high false negative rates. They also state that high false positive rates are the results of simple signature detection algorithms employed by the products. Moser et. al. [12] proposed a malware detectors rely on semantic signatures and employed a model checking for detection. They showed that static analysis alone is not efficient to detect malware, it should be complemented with dynamic analysis techniques.

Wu L. et al. [13] proposed a computer virus evolution model. Their work based on Immune Genetic Algorithm. Noreen et. al. [14] applied genetic operators to Beagle virus in order to create new variants of it. They also apply this approach to Silvio virus threatening the ARM-Linux based smart phones and generate new variants of Silvio viruses using Markov models [15]. In [16], they proposed a methodology that aims to generate new malwares by changing the semantic of malwares. They extract abstract representation of an malware, then use GP to evolve a new malware from this representation.

Ilsun et. al. [17] analyze the obfuscation techniques commonly used by virus writers. They aim to understand how to use these techniques. Christodorescu et. al. [18] propose a malware normalizer that reverts obfuscated malwares to original one by undoing the obfuscations. Their goal is to increase the detection rate of virus detectors.

Kayacık et. al. [2] use GP to evolve buffer overflow attacks in order to obfuscate the true intent of the attacker. In [3], they also use GP to generate mimicry buffer overflow attacks with the objective of finding possible vulnerabilities before attackers exploit them. Their goal is to generate attacks that seem like benign, hence they could evade detection. They extend their work in [4] by increasing the number of detectors (5) and adding delay parameter in order to build evasion attacks. In [5], they compare two approaches, “white-box” and “black-box”, while the former uses the internal knowledge of the detectors to generate evasion attacks, the latter uses only the output of the detectors.

As far as we know, there are only two approaches proposed for mobile anti-virus evaluation against new attacks [7], [8]. They clearly present how ineffective anti-virus systems is against code obfuscation techniques. The target of our study is also to test existing anti-virus solutions. However we investigate the use of GP to achieve that. It is shown that automatically evolved malwares could be more evasive than malwares generated by solely employing obfuscation techniques as in [7], [8]. GP automatically employs the best evasion strategies by reducing the size of the search space.

3 Android Platform

Android is an operating system based on the Linux kernel and runs dalvik bytecode which is a similar form of Java bytecode. There are normally multiple *.class* files in a Java application, however an Android application has a single *.dex* file consists of all the classes. The java source codes are compiled and packaged into an *.apk* file. *.apk* file is the file format to run on a device or on an emulator for Android and contains all of the elements like *.dex* file, manifest file, source codes, resource files, and the like.

Fig. 1. Code Fragment

```
.method public A (Landroid/widget/AdapterView;Landroid/view/View;IJ)V
invoke-direct {p0}, Lcom/keepwired/utility/controls/FileChooser;->B()V
invoke-direct {p0}, Lcom/keepwired/utility/controls/FileChooser;->C()V
invoke-virtual {p0, p0, v3}, Lcom/keepwired/utility/controls/FileChooser;->D(Landroid/content/Context;Ljava/lang/
      String;)V
.end method

.method private B()V
invoke-direct {v3, p0}, Lcom/keepwired/utility/controls/FileChooser$2;->E(Lcom/keepwired/utility/controls/
      FileChooser;)V
.end method

.method private C()V
invoke-direct {v4, p0}, Lcom/keepwired/utility/controls/FileChooser$3;->F(Lcom/keepwired/utility/controls/
      FileChooser;)V
invoke-direct {v3, p0}, Lcom/keepwired/utility/controls/FileChooser$4;->G(Lcom/keepwired/utility/controls/
      FileChooser;)V
.end method

.method public D(Landroid/content/Context;Ljava/lang/String;)V
.end method
```

Since we aim to use source codes of existing malwares in order to evaluate anti-virus products by using GP, we should be able do reverse engineering on these malwares. Although we could get Java source codes from an Android application easily by using some existing tools [19], [20], we cannot convert them back to the *.apk* file due to some errors encountered during the transformation process. These errors could not be resolved automatically. Therefore, we use *.smali* codes that are sequences of assembly-like dalvik bytecode instruction sets [21]. We can access *.smali* codes by extracting them from *.dex* file inside the *.apk* file. *.smali* files extracted using apktool [22] and Smali/BakSmali [23]. Then the

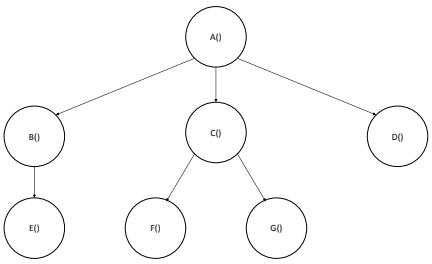

Fig. 2. CFG for code fragment in Figure 1

converted *.smali* files could be used to construct an *.apk* file. A sample smali class file is shown in Figure 1.

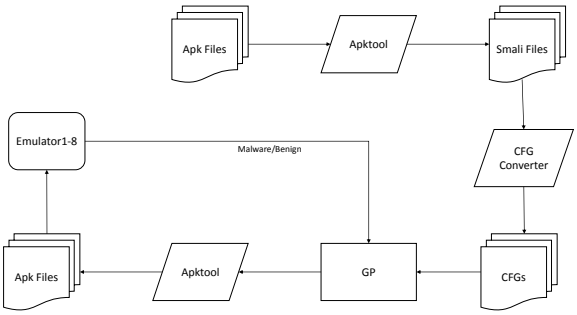

Fig. 3. Simplified Schema of Experiments

4 The Model

The conceptual schema of our experiments is shown in Figure 3. Firstly, we extract smali codes from malicious application in order to obtain CFGs (Control Flow Graph). As stated before, we use smali codes instead of Java codes which enable us to repack the application. After obtaining smali codes, we create CFGs of each malicious application in order to represent them in GP. GP is an evolutionary computation technique inspired from natural evolution. It has been one of the most employed evolutionary computation (EC) technique, since it is introduced by Koza [24]. We use the toolkit ECJ [25] for the GP implementation in the experiments. In this study, malicious applications, is represented by control flow graphs (CFG) which are given as inputs to GP.

As we mentioned, to represent an *.apk* file in GP, we extract control flow graph of each functions in the application. Applications considered here do not consist of recursive functions. An example CFG is shown in Figure 2. Each CFG corresponds to a tree, each method is represented by nodes, and the edges indicate the flow between methods. Each program is represented by an individual in GP and each individuals have different sizes of trees (CFGs) because of the fact that each *.apk* file has variable number of functions. In GP, crossover or mutation operators are applied to these CFGs. Crossover operation exchanges sub-trees of individuals. To be able to obtain executable programs, only methods with the same declarations having the same return type, the equal number of parameters with the same types are allowed to exchange. This allows us to create new malwares from existing malwares. However since the high value of crossover rate increase the number of non-executable individuals, the crossover operation is assigned to a low rate (0.1) in the experiments. And when we use crossover operator, we have to check whether newly generated programs execute properly and perform malicious activities. In order to check this, we developed a dynamic analysis tool [26]. This tool runs applications and extract some features in runtime from Android apps. The machine learning approaches are applied on these features in order to decide whether the application is malicious or not. If the application cannot be executed properly, the dynamic analysis tool does not return any value in a time interval.

Mutation operator is also applied on sub-trees which are randomly chose. It uses the five obfuscation techniques. These techniques aim to generate different variants of the malwares while preserving the original malicious function. To perform these transformations, respectively, we take *.apk* file as an input, unpack smali codes using apktool, apply obfuscation techniques on smali codes, repack smali codes into *.apk* file and finally sign it before being installed on the emulator. Rename Local Identifier (RI) changes one of the local identifiers of a method with a string generated by a random string. Two-fold Code Reordering (CR2) is based on reordering the codes of a method in the program. We reorder the codes of a method by inserting goto statement at the top and bottom parts of the method. Three-fold Code Reordering (CR3) has the same logic with the two-fold code reordering. The only difference is to insert 3 goto statement instead of 2 in this obfuscation strategy. So we change the order of the codes, however, we preserve the execution sequences of the codes at runtime. In Junk Code Insertion (JK), three instructions are inserted. Firstly, we add a local variable to the method. Primitive types such as string, int, double are used to identify the local variable. Then, PrintStream object is inserted to print out the local variable, defined one step before. Finally, the code calls the *println* function of the PrintStream object. Data Encryption (DE) encrypts all the strings that have been used in the method. Firstly, the encryption is done by using a randomly generated key. Then the string is replaced with its encrypted version ciphered with the key. The key is defined in the method as well. Finally, *decrypt* function of the SimpleCrypto class is called. If Crypto package is not already included in the code, it will be added. In every mutation operation, one obfuscation technique is selected and

applied to the node namely source codes of the function that selected randomly by GP. In order to generate more evasive attacks, we set the mutation rate to a high value, 0.9. The other GP parameters are selected as follows : 10 for population size, 100 for generation, 7 for tournament selection. The parameters not mentioned here are the default parameters of the ECJ toolkit.

We also sign each generated programs with a random key not to cause any errors while trying to install the application on the emulator. We generate random key with keytool [27]. To sign the generated programs, we use jarsigner [28] that is used to signs and verifies Java Archive (*.jar*) files.

Since the fitness function defines how the individuals solves the problem or come close to the solution, defining a well-representative fitness function is very important in any GP application. In our fitness function here, we use the output of 8 anti-virus systems which are among the most successful mobile security solutions. The anti-virus systems are selected according to their protection score given in AV-TEST [6]. Our second criteria to choose the anti-virus systems, we explore if we could get the output of the anti-virus run on a emulator. Hence we use the solutions which produce log files. Table 1 lists the anti-virus systems employed in our experiments. Each anti-virus is executed on different emulators (official Android emulator) to simulate the execution of evolved malwares and the response of anti-virus against them. The fitness function takes values between 0 and 1. It is aimed to be minimized and defined as follows :

$$fitness = \frac{\text{number of antivirus systems detecting the individual}}{\text{total number of anti-virus systems}} \quad (1)$$

Table 1. Anti-virus products used.

Vendor	Product	Version
Eset	Eset Mobile Security	3.0.882.0-16
GData	Gdata Internet Security	25.0.0
Ikarus	Ikarus mobile.security	1.7.20
Kaspersky	Kaspersky Internet Security	11.1.3.10
Avast	Avast Mobile Security	3.0.7550
Trend Micro	Trend Micro Mobile Security	5.0.0.1225
BitDefender	BitDefender Mobile Security	2.18.119
Norton	Norton Mobile Security	3.8.6.1533

5 Results

5.1 DataSet

We use the dataset given in [29]. It is the first malware dataset introduced in the literature. Many studies have already used this dataset in order to compare their result with other approaches. The dataset include 1,260 Android malwares in 49 malware families collected within more than a year, between August 2010 and

October 2011. We randomly select malwares from the dataset and their produced CFGs are given to the GP algorithm. The only criteria we used to select malwares is to be detectable by the anti-virus systems. The number of anti-virus systems detecting malwares is not taken into account. The GP algorithm is run many times with different inputs. In some runs, GP produces non-executable or non-compilable individuals due to difficulties faced with while modifying the smali files. These outputs are not discussed in the results.

5.2 Comparison with One-Level Obfuscation

Table 2. The detection ratio of obfuscated/evolved malwares employing just one obfuscation technique by the 8 anti-virus systems

	OR	*JK*	*DE*	*CR2*	*CR3*	*RI*	*MU*	*XO + MU*
NickySpy_1ce27fa92a313da39f1e31e97d3ac05a8d6ffe78	8	7	8	7	7	8	5	7
NickySpy_63e642f0d859e096342321c9e03baca7cd1210fa	8	8	8	7	7	8	6	6
Asroot_0c059ad62b9dbccf8943fe4697f2a6b0cb917548	7	7	6	7	7	7	6	6
GPSSMSSpy_0eb4b7737df1b8b52213599e405d71c9be8a68ac	6	5	5	6	6	6	5	5
GPSSMSSpy_4d43d7771e480de34dbf748867152406b91a0de8	6	5	5	6	6	6	5	5
HippoSMS_bd7e85f5a0c39a9aeecc05dbc99a9e5c52150ba6	8	7	5	8	7	7	4	7
FakeNetflix_0936b366cbc39a9a60e254a05671088c84bd847e	6	6	4	6	6	6	3	3
DroidKungFu2_8bb6106b7c1160e8812788bbd16b563f5a00080a	7	7	7	7	7	7	6	6
GPSSMSSpy_af727f5e23e69bfe2321f5d556c63f741dae8283	6	5	5	6	6	6	4	5
GPSSMSSpy_73c1657ddf52cc82b57c2db80554c59927e7970a	6	5	5	6	6	6	4	5
GPSSMSSpy_94b56252ff610126135c568b1cc7b92405b9e608	6	5	5	6	6	6	4	5
GPSSMSSpy_5900250af412b7147764706847cf1dbc54cd6e0e	6	5	5	6	6	6	4	5
DroidKungFu1_02d2e109d16d160f77a645f44314fedcdbcd6e18	8	8	8	8	8	8	7	7
RogueLemon_08a21de6b70f584ceddbe803ae12d79a33d33b50	6	5	4	6	6	6	4	5

Table 2 demonstrates the results by employing just one obfuscation technique to malwares. The column MU shows the performance of malwares generated by GP using only mutation operator and the column MU+XO shows the performance of malwares generated by GP using mutation and crossover operators together. For example, the original version of the malware given in the first row (OR) is detected by all anti-virus systems. If we apply different obfuscation techniques solely to the malware as many approaches in the literature, it could avoid being identified from some anti-virus systems. However, if we apply GP to the malware, it shows the lowest detection rate. However, Table 2 shows that if we use GP using mutation and crossover operator together, the detection results increase comparing to GP using only mutation operator due to that crossover operator is very error prone because of exchanging codes between apps are very hard to deal with. There are also some results that GP is not able to outperform some obfuscation techniques. However it never produces worse results than them as expected. We obtain more malwares producing the same result with the output of GP. Since we want to emphasize the best results, we do not list all malwares evolved in the table.

As shown in Table 2, Rename local identifier transformation (RI) is not very effective against anti-virus systems. It is only effective against HippoSMS

malware. Because RI just changes the name of the constant strings with an arbitrary string. Two-fold code reordering (CR2) does also only affect the NickySpy malware and able to evade from only one anti-virus. The best improvement on results is performed by data encryption (DE). Data Encryption hides the value of the constant strings so that if attacker wants to send message to premium-rate service, attack the websites or read secret information from mobile devices, he could easily hide their intent. The junk code insertion transformation (JK) also produces good results, even though it only adds a simple function which display a random value on the screen. Finally three-fold code reordering (CR3) leads to better results than RI and CR2, but not as good as DE and JK.

5.3 Comparison with Two-Level Obfuscation

Table 3. The detection ratio of obfuscated/evolved malwares employing two obfuscation techniques by the 8 anti-virus systems

	JK-DE	*CR2-DE*	*DE-RI*	*CR3-DE*	*JK-CR3*	*JK-RI*	*CR2-RI*	*JK-CR2*	*CR3-RI*	*MU*	*XO + MU*	*ZEL*
NickySpy_1ce27fa92a313da39f1e31e97d3ac05a8d6ffe78	7	7	8	7	7	7	7	7	7	5	7	6
NickySpy_63e642f0d859e096342321c9e03baca7cd1210fa	7	7	8	7	8	7	7	8	7	6	6	6
Asroot_0c059ad62b9dbccf8943fe4697f2a6b0cb917548	6	6	6	6	6	7	7	7	6	6	6	6
GPSSMSSpy_0eb4b7737df1b8b52213599e405d71c9be8a68ac	5	5	5	5	5	6	6	6	6	5	5	4
GPSSMSSpy_4d43d7771e480de34dbf748867152406b91a0de8	5	5	5	5	5	5	6	5	5	5	5	4
HippoSMS_bd7e85f5a0c39a9aeecc05dbc99a9e5c52150ba6	5	5	5	5	6	7	7	7	6	4	7	5
FakeNetflix_0936b366cbc39a9a60e254a0567108c84bd847e	4	4	4	4	6	6	6	6	6	3	3	5
DroidKungFu2_8bb6106b7c1160e8812788bbd16b563f5a00080a	7	7	7	-	-	7	7	7	-	6	6	6
GPSSMSSpy_af727f5e23e69bfe2321f5d556c63f741dae8283	5	5	5	5	5	5	6	5	5	4	5	4
GPSSMSSpy_73c1657ddf52cc82b57c2db80554c59927e7970a	5	5	5	5	5	5	6	5	5	4	5	4
GPSSMSSpy_94b56252ff6110126135c568b1cc7b92405b9e608	5	5	5	5	5	5	6	6	6	4	5	4
GPSSMSSpy_5900250af412b7147764706847cf1dbc54cd6e0e	5	5	5	5	6	5	6	6	5	4	5	4
DroidKungFu1_02d2e109d16d160f77a645f44314fedcdbcd6e18	8	8	8	-	-	8	8	8	-	7	7	8
RogueLemon_08a21de6b70f584ceddbe803ae12d79a33d33b50	4	4	4	-	-	6	6	6	-	4	5	4

There are several studies that employing obfuscation techniques to malwares one by one [7] or two obfuscation techniques together [8]. GP could apply multiple obfuscation techniques over nodes. Therefore, we compare our GP with technique that applies two obfuscation techniques together over malwares. The results can be shown in Table 3. When we apply Two-fold and Three-fold code reordering together, we fail to produce compiled apps so that we exclude their results in Table 3. We also apply obfuscation techniques in groups of triple and four, but due to that obfuscation techniques affect each other, the rate of the compiled apps has been very low and results not shown in Table 3. Evolutionary computation can prevent such problems, with respect to its nature ability that eliminate non-compiled and non-executable malwares. "-" in Table 3 denotes malwares that can be compiled but cannot be installed on emulators properly. Table 3 shows that when we apply obfuscation techniques in groups of two, there is no further improvement comparing to one by one.

We also evaluate the resilience of the anti-virus systems by using a well-known obfuscator, Zelik KlassMaster [9]. The column ZEL in Table 3 shows the detection rate of the anti-virus systems against malwares obfuscated by the obfuscator. The results show that GP (MU) produces comparable results with the obfuscator. It produces more evasive malwares for 4 applications. Only two

malwares obfuscated by Zelix KlassMaster [9] produces better results than our approach. In the future, we aim to compare our approach with more obfuscators in the literature.

5.4 Evalution of Static Analysis Tools

To sum up, we could say that our approach produces many malwares in one run, and it generally evolves more evasive attacks than five obfuscation techniques automatically. It allow us to test the anti-virus systems against unknown attacks as shown in Table 4. As shown, Avast Mobile Security is the most robust system against new attacks. It detects all obfuscated malwares in this study except one if we apply mutation and crossover operators together. BitDefender only misses one malware evolved by GP. Kaspersky and TrendMicro are also successful enough to be able to detect all obfuscation techniques. But three evolved malwares achieve to be undetected by Kaspersky and TrendMicro. Although the detection performance of GData is very low (6/14), it is very effective against all obfuscation techniques, except one evolved malware by GP. Norton Mobile Security is the most ineffective solution against evolved malware. Although it detects 13 malwares out of 14 per obfuscation techniques, we achieve to evolve 8 malwares that could avoid being recognized by Norton using GP (MU) and to evolve 2 evasive malwares by using GP (MU+XO). Our evolved malwares are also very effective against Eset. We also manage to escape from Ikarus Mobile Security. However, DE outperforms our method on one malware.

Table 4. The performance of anti-virus systems against obfuscated and evolved malwares

	OR	***JK***	***DE***	***CR2***	***CR3***	***RI***	***MU***	***MU + XO***
Eset	8	8	6	8	8	8	4	6
GData	6	6	6	6	6	6	5	5
Ikarus	13	8	6	12	11	13	7	12
Kaspersky	13	12	12	13	13	13	10	10
Avast	14	14	14	14	14	14	14	13
TrendMicro	13	12	11	12	12	12	9	12
BitDefender	14	14	14	14	14	14	13	13
Norton	13	11	11	13	13	13	5	11
Toplam	94	85	80	92	91	93	67	82

There are three main important advantages of using GP to create new, unseen malwares. Firstly, GP reduces the search space and helps to find the best effective obfuscation techniques upon malwares rapidly. It eliminates non-compilable or non-executable solutions naturally. More evasive malwares could be evolved by increasing the number of obfuscation techniques employed in GP. Secondly, GP could allow to co-evolve both malwares and anti-malware systems automatically. We aim to automatically improve anti-malware systems in the future. The last advantage of using GP is bloating. Generally bloating is not desired in GP applications. However bloating has a positive effect in our method that causes to generate more complex programs, hence more evasive malwares.

6 Conclusion

We propose a method that evaluates current anti-virus systems for Android platform. We apply GP to Android malwares in order to generate the variants of malwares with the objective of evading from anti-virus systems. We compare our results with the obfuscation techniques and the Zelix KlassMaster obfuscator [9]. We employ obfuscation techniques either one by one or in groups of two on malwares. Our results show that all the anti-virus systems have a weakness against obfuscation techniques and we can further decrease their detection ratio against evolved malwares by using GP.

To the best of our knowledge, this is the first work that generates evolved mobile malwares using GP in mobile platforms. One of the main characteristics of the proposed system is being fully automatic. In the future, more obfuscation techniques are planned to be added to the algorithm. Moreover, it is aimed to be compared with more obfuscators in the literature. We also aim to evolve both malwares and anti-malwares simultaneously in the future.

Acknowledgement

This study is supported by the Scientific and Technological Research Council of Turkey (TUBITAK-112E354). We would like to thank TUBITAK for its support.

References

1. Kaspersky Lab., Mobile malware evolution: 3 infection attempts per user in 2013, http://www.kaspersky.com/about/news/virus/2014/Mobile-malware-evolution-3-infection-attempts-per-user-in-2013
2. Kayacık, H.G., Heywood, M.I., Zincir-Heywood, A.N.: On Evolving Buffer Overflow Attacks Using Genetic Programming. In Proc. of the 8th Annual Conference on Genetic and Evolutionary Computation. pp. 1667–1674. ACM (2006),
3. Kayacık, H.G., Zincir-Heywood, A.N., Heywood, M.I., Burschka, S.: Generating Mimicry Attacks using Genetic Programming: A Benchmarking Study. In Proc. of IEEE Symposium on Computational Intelligence in Cyber Security. pp. 136–143 (2009)
4. Kayacık, H.G., Zincir-Heywood, A.N., Heywood, M.I.: Can a good offense be a good defense? Vulnerability testing of anomaly detectors through an artificial arms race. Appl. Soft Comput. 11(7), 4366–4383 (2011)
5. Kayacık, H.G., Zincir-Heywood, A.N., Heywood, M.I.: Evolutionary computation as an artificial attacker: generating evasion attacks for detector vulnerability testing. Evolutionary Intelligence 4(4), 243–266 (2011),
6. AV-TEST: The independent it-security institute, http://www.av-test.org/en/home/
7. Zheng, M. and Lee, Patrick P.C. and Lui, John C.S.: ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems. In Proc. of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science, vol. 7591, pp. 82–101. Springer (2013)

8. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks. In Proc. of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. pp. 329–334. ACM (2013)
9. Zelix KlassMaster: Java obfuscator - zelix klassmaster, http://www.zelix.com/
10. Christodorescu, M., Jha, S.: Testing Malware Detectors. In Proc. of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 34–44.
11. Morales, J., Clarke, P., Deng, Y., Golam Kibria, B.: Testing and evaluating virus detectors for handheld devices. Journal in Computer Virology 2(2), 135–147 (2006)
12. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection. In Proc. of Computer Security Applications Conference, pp. 421–430 (2007)
13. Wu, L., Zhang, Y.: Research of the Computer Virus Evolution Model Based on Immune Genetic Algorithm. In Proc. of the 10th IEEE/ACIS International Conference on Computer and Information Science. pp. 9–13. IEEE Computer Society (2011)
14. Sadia, N., Shafaq, M., Zubair, S.M., Muddassar, F.: Evolvable Malware. In Proc. of the 11th Annual Conference on Genetic and Evolutionary Computation. pp. 1569–1576. ACM (2009)
15. Shahzad, F., Saleem, M., Farooq, M.: A Hybrid Framework for Malware Detection on Smartphones Using ELF Structural & Pcb Runtime Traces. Tech. rep., TR-58 FAST-National University, Pakistan (2012)
16. Sadia, N., Shafaq, M., Zubair, S.M., Muddassar, F.: Using Formal Grammar and Genetic Operators to Evolve Malware. In Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection. pp. 374–375. Springer (2009)
17. You, I., Yim, K.: Malware Obfuscation Techniques: A Brief Survey. In Proc. of the International Conference on Broadband, Wireless Computing, Communication and Applications. pp. 297–300 (2010)
18. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H., Munchen, T.U.: Malware Normalization. Tech. rep.,1539, University of Wisconsin (2005)
19. JAD: Java decompiler download mirror, http://varaneckas.com/jad/
20. JEB: The interactive android decompiler, http://www.android-decompiler. com/
21. Android: Bytecode for the dalvik vm, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html
22. Apktool: A tool for reverse engineering android apk files, https://code.google.com/p/android-apktool/
23. Smali: An assembler/disassembler for androids dex format, https://code.google.com/p/smali/
24. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
25. ECJ: A java-based evolutionary computation research system, http://cs.gmu.edu/ eclab/projects/ecj/
26. Ozkan, H.B., Aydogan, E., Sen, S.: An Ensemble Learning Approach to Mobile Malware Detection. Tech. rep., Hacettepe University, Department of Computer Engineering (2014).
27. Oracle: keytool - key and certificate management tool, http://docs.oracle. com/-javase/7/docs/technotes/tools/solaris/keytool.html
28. Oracle: jarsigner, http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html
29. Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. In Proc. of the 2012 IEEE Symposium on Security and Privacy. pp. 95–109, IEEE Computer Society (2012)