Efficient Evolutionary Fuzzing for Android Application Installation Process

Veysel Hatas
Wise Lab., Dept. of Computer Engineering
Hacettepe University
Ankara, Turkey
vhatas@gmail.com

Sevil Sen
Wise Lab., Dept. of Computer Engineering
Hacettepe University
Ankara, Turkey
ssen@cs.hacettepe.edu.tr

John A. Clark
Department of Computer Science
University of Sheffield
Sheffield, UK
john.clark@sheffield.ac.uk

***Abstract*—Source code analysis techniques used for automated software testing are insufficient to find security flaws in programs. Therefore, security researchers have been employing also fuzzing techniques for finding bugs and vulnerabilities in target programs. With the proliferation of mobile devices, researchers have started to explore the use of fuzz tests on mobile platforms. While most of these studies are GUI-based and implemented at the application level, the detection of vulnerabilities in lower levels is very critical due to affecting a broader range of Android users. Therefore, in this study, a new approach is proposed to fuzz testing for Android application installation process. The use of a search heuristic namely genetic algorithms is investigated for efficient fuzz testing on DEX (Dalvik EXecutable) files. The proposed black box fuzzing tool called GFuzz is shown to be able to produce more unique crashes in Android in a shorter time than recently proposed similar approaches and to detect new and existing bugs.**

***Index Terms*—Android, security, fuzzing, search-based software testing, genetic algorithms**

I. Introduction

Technology has evolved towards mobile devices with the increased demands for instant communication such as video calling, messaging and surfing on the Internet. According to a recent study conducted by *Hootsuite* and *We Are Social* [1], the number of mobile users in the world has exceeded 5 billions in 2018. With the advancing technology, the use of mobile devices such as smart phones, smart watches, tablets, ad hoc devices has increased rapidly and the security of these devices and their users has become very important.

With mobile devices having become an integral part of modern lifes, attackers focus more and more on harming mobile devices and stealing private information from them. Security weaknesses caused by coding errors could be exploited by malicious software that results in mobile devices to be compromised and mobile data to be leaked for various purposes. According to the Android security report 2017 [2], since 2014 the Android Security Improvements (ASI) program has fixed nearly 383,000 vulnerable apps in Google Play, which is caused by 27 different types of vulnerabilities. Only in 2016 [3], 655 vulnerabilities, where 133 of them are critical are addressed by Android. Therefore, security researchers, mobile operating system and application developers have been exploring on new methods for detecting and closing these software bugs and vulnerabilities before hackers exploit them and affect billions of users. Fuzzing is a software testing technique that has been extensively used for this purpose since it was introduced by Barton P. Miller in 1998 [4].

Fuzzing (semi-)automatically tests the robustness of a program by providing invalid or semi-valid, unexpected, or random data as inputs to the program. It has been applied to many platforms from embedded devices to desktop computers in order to find software bugs in these platforms. Since 2014, studies on the automatic testing of Android applications has accelerated with the increasing usage of mobile devices [5]. However most of these studies are at the application level, there are only few studies on fuzzing for Android application installation process [6] and compiler fuzzing [7] [8] that helps to find vulnerabilities affecting many devices independent of applications running on them. Some of the bugs found at this stage could cause the compiler to crash, some others cause more serious errors such as memory read/write, null pointer that is occurred at runtime during the execution of the generated native code [9].

In this study, a new evolutionary fuzzing approach based on genetic algorithms for Android application installation process is introduced. Such meta-heuristics have been proven to be useful for generating test data and be effective for optimizing the testing process [10]. Moreover, algorithms employing heuristics are shown to perform better than fully random sampling [11]. Therefore, while random exploration strategies are surprisingly shown to be quite effective in testing in Android [12], the use of genetic algorithms are explored for effective and efficient fuzz testing in Android in this study. A new Generic algorithm-based Fuzzing (GFuzz) is introduced for finding effective test cases from an infinite search space within limited time.

GFuzz has been executed on an Android emulator for two weeks and has produced 20,452 crashes in total in this time, where 305 of them are unique. Among these crashes, the ones produced critical error signals (SIGSEGV, SIGABRT, SIGFPE and SIGILL) are manually analyzed. One of these crashes is found to be a zero-day vulnerability triggered in a native library [13]. So, in this study, by fuzzing DEX files, a vulnerability that affects many Android users regardless of the applications they have been using is found. The proposed approach is also compared with other DEX fuzzing techniques in the literature, namely Droid-FF [6], DexFuzz [8] and zzuf [14]. While Droid-FF produces the biggest number of crashes in two weeks, GFuzz is shown to be more efficient by means of producing more unique crashes in the same period of time by giving higher fitness values for unique crashes in the evolution process.

The remainder of this paper is organized as follows. Section 2 discusses the related works in the literature, and outlines existing testing tools of Android applications. An overview of genetic algorithms is given in Section 3. Section 4 introduces GFuzz and gives the implementation details. The experimental results are discussed in Section 5. Finally, the study is concluded in Section 6.

TABLE I
SUMMARY OF RELATED STUDIES

Fuzzers	Box	Approach	Target
Monkey [15]	black	random-based	apps
Dynodroid [16]	black	random-based	apps
EvoDroid [30]	white	search-based	apps
Sapienz [24]	gray	search-based	apps
Stoat [25]	black	model-based	apps
AIMDROID [26]	black	model-based	apps
DexFuzz [8]	black	mutation-based	installation process
GFuzz	**black**	**search-based**	**installation process**

II. RELATED WORK

The studies on automatic testing of Android applications has accelerated since 2014 [5]. However, most of these studies perform their tests at the application level and target GUI/events, intercomponent/interapplication communications [5].

The existing test input generation tools for Android has been compared based on four metrics: code coverage, ability to detect faults, ability to work on different platforms, and ease of use [12]. Surprisingly, tools based on random exploration strategies such as Monkey [15] and Dynodroid [16] show the best performance. It is shown that most of the behavior can be exercised by generating only UI events. Therefore, the random-based approaches are generally effective enough to cover the application code [12]. However, mobile malware could also be triggered by systems events. Therefore, input generation tools that could generate system events such as Dynodroid [16] and ACTEve [17] stand out in terms of triggering malicious activities. Targeted input generation tools such as SmartDroid [18], IntelliDroid [19], DroidBot [20] could also be referred for triggering such malicious applications.

Monkey as being the most widely used tools is also analyzed on an industrial app and shown to be very ineffective [21]-[22]. Moreover, the same authors has proposed some improvements over Monkey [21]- [22]. Since real-world industrial applications tend to have more complex implementations, six tools are compared on industrial apps [23]. While Monkey [15] performs the highest activity coverage and Sapienz [24] achieves the highest method coverage, Stoat [25] is the best tool at triggering unique crashes. However, a recent model-based study called AIMDROID [26] is claimed to outperform Monkey [15] and Sapienz [24] in activity, method and instruction coverage by focusing on exploring new activities with controlling the length of test cases.

In the literature, there have been few efforts in order to identify vulnerabilities at the byte-code level that are critical and affect a broader audience. With the fully replacement of Dalvik VM with a new runtime environment called ART in 2015, few studies have been proposed to fuzzing ART. The first and only academic study called DexFuzz [8] combines domain-aware binary fuzzing with differential testing to find bugs in ART. DexFuzz applies mutation-based fuzzing and analyze the effects of different DEX mutations. Another study targets to fuzz dex2oat (ART) and dexopt (KitKat) methods for the application install process in Android [27] using the Radamsa fuzzer [28] and finds one critical vulnerability affecting the Lollilop version. Another mutation-based approach targets also dex2oat and claims to have found important crashes by applying mutations to almost all sections of DEX files [7]. Droid-FF targets also DEX files and proposes a mostly automated approach by using dumb and intelligent fuzzing techniques [6]. Droid-FF claims to find a lot of crashes by applying two techniques namely dumb fuzzing using the Radamsa [28] and pyzuff tools and generation-based fuzzing using the PEACH fuzzer [29].

Search-based software testing, which formulates testing as an optimization problem, has been extensively used in the literature [10]. In the last couple of years, search-based techniques have also appeared for the Android platform [5]. The first evolutionary testing framework for Android called EvoDroid [30] is a white-box approach. EvoDroid firstly extracts call graph model (CGM) of the application, then breaks each path in the CGM into segments. It aims to cover as many unique CGM paths, hence maximize the code coverage by searching each segment separately. A recent study called Sapienz [24] has proposed the first multi-objective search-based testing approach that aims to maximize code coverage and fault revelation, while minimizing the length of test sequences. Another multi-objective approach seeks to minimize the whole test suite for re-usability besides to maximize code coverage and fault detection [31].

In Table I, the most effective studies in the literature and the recent studies closest to our approach are summarized. Please note that only academic studies are included in the table. The main contribution of the current study is to propose an efficient fuzzing tool by using genetic algorithms for the application installation process of Android. By fuzzing DEX files, some critical bugs are found in Android. Moreover, such bugs and unique crashes are discovered in a shorter time than similar approaches in the literature. According to the taxonomy of Android testing [5], the areas in which this study takes place are: security (objectives), general (targets), system (levels), black-box (types), emulator (environments) and search-based (methods).

III. Genetic Algorithm

Genetic algorithm is a population-based optimization algorithm, which is inspired from Darwinian's evolutionary theory. The algorithm starts with generating a population of individuals that are representative of candidate solutions for the problem. Each solution is represented as strings (usually string of 1s and 0s) in GA [32]. The first population is in general randomly initialized. Each individual in the population is evaluated by a fitness function that shows how well this individual has solved the problem or how close it is to the solution. Generation-by-generation, the population is transformed into a new, hopefully better, population of individuals by using genetic operators. The general steps of genetic algorithms are given below.

```
initialize a population
while termination criterion not satisfied do
    execute and evaluate fitness value of each individual
    apply genetic operators to the individuals
    create new population
end
return best-of-run individual
```

Algorithm 1: General steps of GA

Genetic operators are used to create new individuals using the selected individuals from the old population based on their fitness values. It aims to create better individuals in the next generations by using/modifying the better individuals in the current populations as in natural evolution. Application of these operators differentiate the evolutionary computation from random search. The main variation operators are crossover and mutation. Crossover mimics the exchange of DNA under sexual production to generate new individuals. This binary operator generates two child individuals by swapping some part of two selected parent individuals. Mutation is an unary operator that mimics natural mutation by changing selected individuals to introduce diversity into the population. New populations are generated iteratively until the termination condition is satisfied. For termination, the algorithm is run until the maximum number of generations is reached or a solution of sufficient quality is obtained.

IV. The Method

In this study, a fuzzer has been proposed to detect security vulnerabilities in the disassembler used during the installation of Android applications. Therefore, automatically generated DEX files using the genetic algorithm are sent to the dexdump disassembler tool in the emulator, which is located under /system/xbin/. Dexdump creates Dalvik Virtual Machine bytecode from dex files during the application installation process. The conceptual schema of the proposed fuzzer is presented in Fig. 1.

Firstly a group of DEX files obtained from random applications are given as the initial population to the genetic algorithm. In each generation of the evolution process, such files are executed on the emulator and assigned to a fitness value according to the flow graph proposed in the subsequent subsection. The new DEX files are generated based on these values. All generated files in time are also logged for differentiating unique crashes. The details of the evolution is presented below.

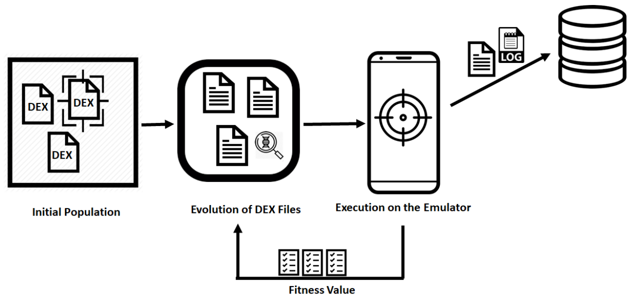


Fig. 1. Conceptual schema of GFuzz

A. Representation and Genetic Operators

As shown in Fig. 1, DEX files (classes.dex) are inputs of the genetic algorithm. Android application files are compiled into dex files which are used to hold a set of class definitions and their associated adjunct data [33]. By default, each application has a single *classes.dex* file [34]. In this study, *classes.dex* files of randomly collected applications are taken as the first population in genetic algorithm.

The DEX file format is given in Table IV. New populations are generated at each generation by applying genetic operators to the *string_ids* fields of the DEX files, which are identifiers for all the strings used by the file. The main reason of using the *string_ids field* in the evolution is to be able to produce vulnerabilities such as buffer overflow vulnerabilities that is very critical and exploitable. However, other fields such as *field_ids*, *method_ids* could be easily added to the evolution process in the future. In the crossover operator, two new individuals are generated by exchanging the fields of two selected parent individuals. In the mutation operator, random number of bits at random position in the *string_ids* field of a selected individual is changed.

When new individuals are generated through generations, the header of the newly generated DEX file should be changed. The following fields of a DEX header file are static and cannot be modified : magic, endian tag and header size. If one of these fields are modified, the file is rejected by dexdump. While such fields cannot be changed, the file size, checksum and SHA-1 signature fields should be recalculated and updated respectively when a modification has been made to the DEX file. Hence, these modified files by GA are repaired by using the open source dexRepair tool [35]. Then, newly generated DEX files are sent to the Android emulator and evaluated. The outputs of dexdump is analyzed for calculating fitness value of each individual. The better the fitness value of an individual, the more likely it is to be selected as a parent for being applied in genetic operators.

TABLE II
DALVIX EXECUTABLE FORMAT

header	the header
string_ids	string identifiers
type_ids	type identifiers list
proto_ids	method prototype identifiers list
field_ids	field identifiers list
method_ids	method identifiers
class_defs	class definitions list
data	data area

TABLE III
GA PARAMETERS

Population Size	20
Generations	≈5000 per day run for 2 weeks
Crossover Probability	0.2
Mutation Probability	0.5
Selection Mechanism	Tournament (size:7)

The major GA parameters are given in Table III. *Population size*, which is the number of DEX files given as inputs in the study, is the number of individuals in a population in any generation. *Generations* defines when (at which generation) the evolution process stops. *Crossover probability* shows how likely individuals selected for mating will exchange elements. The parameter *mutation probability* shows how likely each part of an individual's genotype elements will be altered. Since the mutation operator increases the diversity of the population, the mutation probability is chosen to be bigger than the crossover probability. The *selection mechanism* provides a great opportunity for fitter individuals to survive by picking out individuals from the current population based on their fitness values. Tournament selection is employed in our experiments. In tournament selection, a group of individuals is chosen randomly from the population and the best individual from this group is selected as parent. *Tournament size* defines the number of the individuals in this group. The parameters not listed here are the default parameters of the GA implementation [36].

B. Fitness Function

The individuals are evaluated based on the output logs of dexdump. Since the main aim of the proposed fuzzer is to generate unique crashes, the higher fitness values are assigned to the individuals who produce such crashes. In this study, the flow graph given in Fig. 2 is proposed to evaluate the fitness of an individual based on dexdump's outputs. As it is seen in the figure, the sum of the event weights in each level of the graph is equal to 1.

In the first level of the graph, it is checked whether there exists a crash or not. If there is no crash, the minimum fitness value is assigned to the individual. In the second level of the graph, the following triggered signals are checked : *SIGSEGV, SIGABRT, SIGFPE, SIGILL*. If one of these signals are triggered, it could be a sign of an expoitable

TABLE IV
THE FORMAT OF DEX HEADER

magic	magic value
checksum	alder32 checksum of rest of file
signature	SHA-1 signature of rest of file
file_size	file size in bytes
header_size	header size in bytes
endian_tag	endianness tag
link_size	size of link section
link_off	file offset of link section
map_off	file offset of map list
string_ids_size	count of strings in the string ID list
string_ids_off	file offset of string ID list
type_ids_size	count of types in the type ID list
type_ids_off	file offset of type ID list
proto_ids_size	count of items in the method prototype ID list
proto_ids_off	file offset of method prototype ID list
fields_ids_size	count of items in the field ID list
fields_ids_off	file offset of field ID list
method_ids_size	count of items in the method ID list
method_ids_off	file offset of method ID list
class_defs_size	count of items in the class definitions list
class_defs_off	file offset of class definitions list
data_size	size of data section in bytes
data_off	file offset of data

security vulnerability. The SIGSEGV error is caused by an invalid memory reference or a segmentation fault. Since a buffer overflow attack can cause a SIGSEGV error signal, the weight of this signal is considered to be slightly higher than other signals. While the SIGABRT signal is caused by a fatal error, the SIGFPE signal is caused by an incorrect arithmetic operation. When an illegal instruction is used, the SIGILL signal is produced. In the third level of the graph, the target library of the crash is checked. Because crashes in native libraries may be a sign of a more critical vulnerability, crashes that are encountered in native libraries are assigned higher fitness values than Java libraries. Finally, it is taken into account whether the collision is observed for the first time during the evolution process. The fitness value is calculated according to the weights (w) of the events in each level as given in the following equation:

$$\text{Fitness} = 1 \ / \ \prod_{each\ level\ l} w_l$$

The red state transitions in Fig. 2 shows the output of an example individual. Since the individual has caused a unique crash in a native library with the SIGSEGV signal, it is assigned to the highest fitness value as shown below:

$$\text{Fitness} = 1 \ / \ (0.9 * 0.3 * 0.7 * 0.8) = 6.613$$

The general steps of GFuzz are summarized in the Algorithm 2. The evolution process continues for two weeks. For each generated individual during this time is sent to the dexdump application on the Android Emulator for execution. Then, each individual is subjected to individual fuzz testing for calculating the fitness value. Based on this value, the evolution process continues until the termination criteria

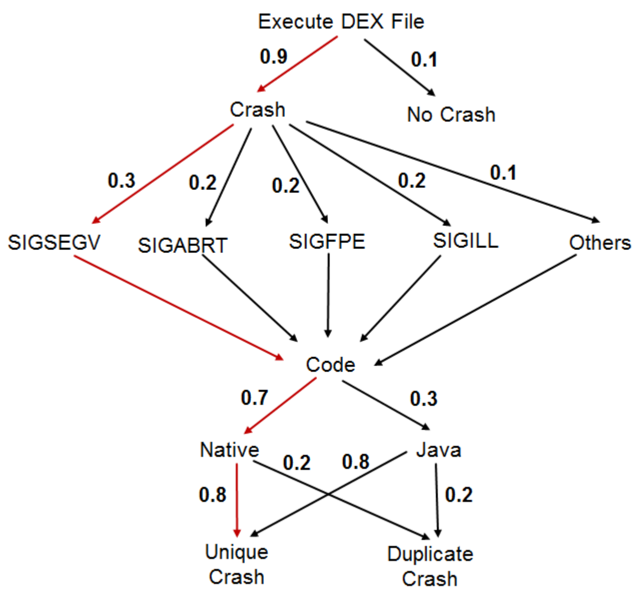


Fig. 2. Flow graph used for the fitness evaluation

(running for two weeks) is satisfied.

```
initialize a population from existing applications
for two weeks do
    for each individual (DEX file) in the population do
        execute the DEX file on the emulator
        analyze logs and evaluate fitness values
        apply genetic operators
        repair the DEX file
    end
end
```

Algorithm 2: General steps of GFuzz

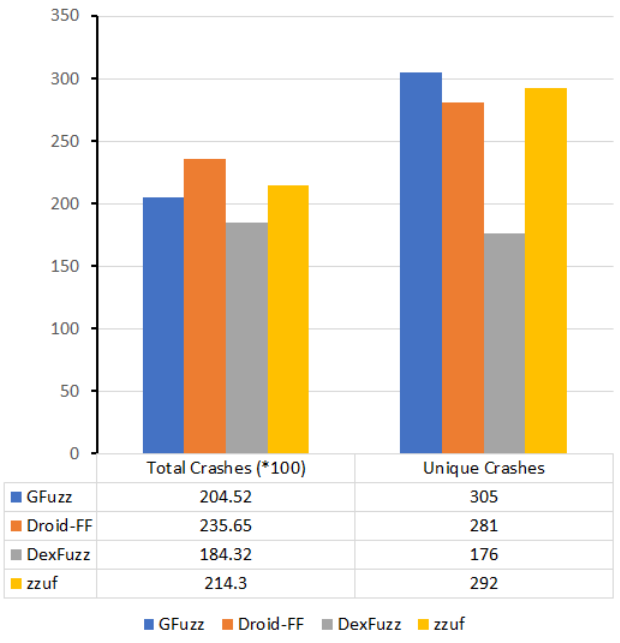
V. EXPERIMENTAL RESULTS

As a test platform, the Android emulator which runs Android 5.1.1 on Nexus 4 with 2 GB RAM is used. Android Lollilop had been one of the most popular Android versions at the time we started this study. Even though Android has released new versions and has modified its architecture to improve security since then, but that is only beneficial to users who download the latest version of Android, which is rarely the case [37]. The numbers also supports that Android Lollilop (5.0 and 5.1) with approximately 18% of the market share is still among the most used Android versions today [38].

The tests have been carried out for two weeks. In that time, 20,452 crashes have been encountered by GFuzz. 301 of these crashes are unique, since they are found in different parts of the code. 11 bugs among these crashes are manually analyzed, since they crashed with one of the following signals: SIGSEGC, SIGABRT, SIGFPE and SIGILL. One of these crashes are determined to be a new bug [13], which is discovered in the file /system/lib/libz.so. The generated signal SIGSEGV is the sign of a stack overflow vulnerability for the Android version under test. It is a critical vulnerability, since it affects all Android devices and versions that using this native library independent of the Android application. The output associated with this crash is given in Fig. 5. Furthermore, three exploitable crashes found by Droid-FF [6] is also produced by GFuzz and DexFuzz [8] in the experiments.

The proposed fuzzer is compared with Droid-FF [6], Dex-Fuzz [8] and zzuf [14]. Even though there are two evolutionary approaches for Android testing namely EvoDroid [30] and Sapienz [24] in the literature, they are not directly comparable with GFuzz. Because, these approaches apply white and gray box testing for Android applications. On the other hand, since the Droid-FF [6] and DexFuzz [8] tools employ fuzz testing over DEX files like the proposed approach, they are especially preferred for comparison in this study. The proposed study cannot be compared with d'ART [7], another fuzzer that targets the installation process of Android, since we could not access its code. The zzuf fuzzer [14] has been used in many fuzzing frameworks including Droid-FF [6], therefore it is also employed in the comparisons. All fuzzers have been run for two weeks on the same testing platform.

The results are presented in Fig. 3 and Fig. 4. Fig. 3 demonstrates the number of total and unique crashes obtained by each fuzzing test tool in two weeks. As it is seen in the results, the highest number of crashes is obtained by using the Droid-FF fuzzer. However, the proposed fuzzer has produced more unique crashes than Droid-FF in the same time. By performing the highest ratio of the number of unique crashes to the number of total crashes, the proposed fuzzer performs better than others.



	Total Crashes (*100)	Unique Crashes
■ GFuzz	204.52	305
■ Droid-FF	235.65	281
■ DexFuzz	184.32	176
■ zzuf	214.3	292

Fig. 3. Number of total and unique crashes

Since discovering vulnerabilities before attackers exploit them is very important, the efficiency of the proposed fuzzer tools should be evaluated. Therefore, the number of obtained unique crashes by each fuzzing tool over time is analyzed and presented in Fig. 4. While the proposed fuzzer has obtained 150 unique crashes in around 3 days, the DexFuzz and Droid-FF has obtained the same number of unique crashes in around 4 days. The zzuf tool has reached the same number of crashes in 5 days. It is easily seen in Fig. 4 that the proposed tool outperforms other tools by producing unique crashes in a shorter time. At any time, the number of unique crashes produced by GFuzz is observed to be bigger than the other tools' outputs.

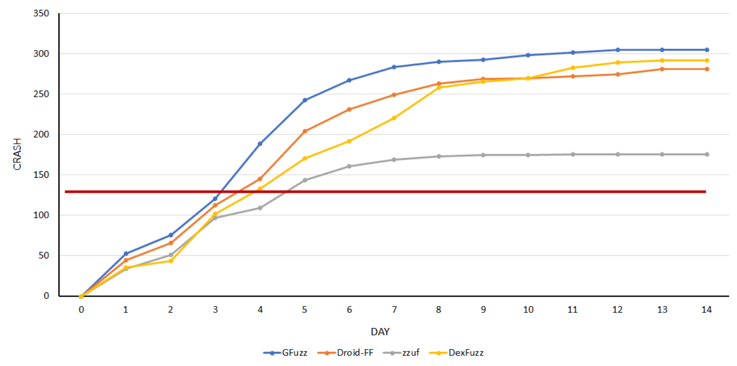
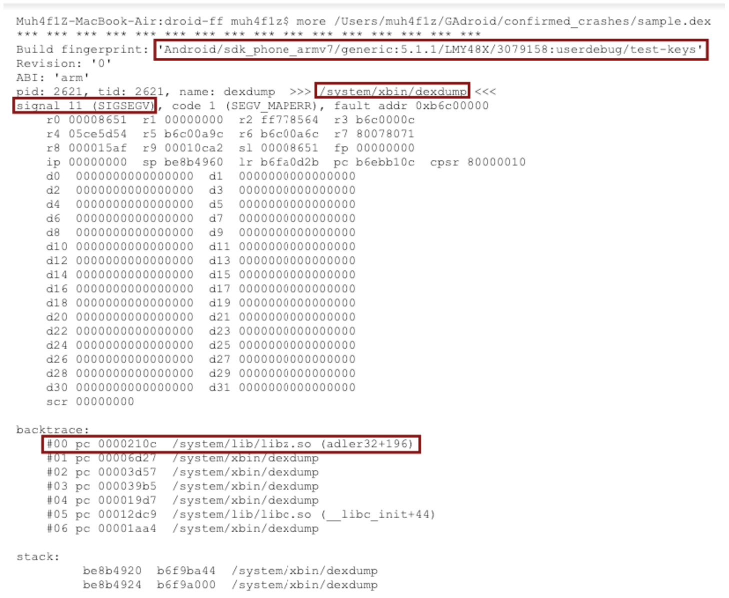


Fig. 4. Number of unique crashes over time



```
Muh4f1Z-MacBook-Air:droid-ff muh4f1z$ more /Users/muh4f1z/GAdroid/confirmed_crashes/sample.dex
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'Android/sdk_phone_armv7/generic:5.1.1/LMY48X/3079158:userdebug/test-keys'
Revision: '0'
ABI: 'arm'
pid: 2621, tid: 2621, name: dexdump  >>> /system/xbin/dexdump <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0xb6c00000
    r0 00008651  r1 00000000  r2 ff778564  r3 b6c0000c
    r4 05ce5d54  r5 b6c00a9c  r6 b6c00a6c  r7 80078071
    r8 000015af  r9 00010ca2  sl 00008651  fp 00000000
    ip 00000000  sp be8b4960  lr b6fa0d2b  pc b6ebb10c  cpsr 80000010
    d0  0000000000000000  d1  0000000000000000
    d2  0000000000000000  d3  0000000000000000
    d4  0000000000000000  d5  0000000000000000
    d6  0000000000000000  d7  0000000000000000
    d8  0000000000000000  d9  0000000000000000
    d10 0000000000000000  d11 0000000000000000
    d12 0000000000000000  d13 0000000000000000
    d14 0000000000000000  d15 0000000000000000
    d16 0000000000000000  d17 0000000000000000
    d18 0000000000000000  d19 0000000000000000
    d20 0000000000000000  d21 0000000000000000
    d22 0000000000000000  d23 0000000000000000
    d24 0000000000000000  d25 0000000000000000
    d26 0000000000000000  d27 0000000000000000
    d28 0000000000000000  d29 0000000000000000
    d30 0000000000000000  d31 0000000000000000
    scr 00000000

backtrace:
    #00 pc 0000210c  /system/lib/libz.so (adler32+196)
    #01 pc 00006d27  /system/xbin/dexdump
    #02 pc 00003d57  /system/xbin/dexdump
    #03 pc 000039b5  /system/xbin/dexdump
    #04 pc 000019d7  /system/xbin/dexdump
    #05 pc 00012dc9  /system/lib/libc.so (__libc_init+44)
    #06 pc 00001aa4  /system/xbin/dexdump

stack:
         be8b4920  b6f9ba44  /system/xbin/dexdump
         be8b4924  b6f9a000  /system/xbin/dexdump
```

Fig. 5. Output of the crash associated with the discovered zero-day vulnerability

VI. CONCLUSION

Android is among the most target platforms by attackers. Therefore, it is important to find security vulnerabilities of Android and its applications before exploited by attackers. Although studies on automatic testing of Android applications has been accelerated in the last few years, only ≈ 28% of these studies target security in their tests [5]. Furthermore, most of these studies are at the application level. However, finding vulnerabilities affecting a broader range of users other than the users of particular applications is very critical. Therefore, in order to achieve that a DEX file fuzzing approach based on genetic algorithms is introduced in this study.

This study investigates the use of genetic algorithms, known to be effective for software testing, to fuzz testing in Android. The proposed approach called GFuzz aims to find critical vulnerabilities with the proposed fitness evaluation flow. By giving more weights to unique crashes in the fitness evaluation, GFuzz is shown to be able to find more unique crashes than other similar fuzzers in the literature, namely Droid-FF [6], DexFuzz [8] and zzuf [14] in the same period of time. Moreover, it finds a new vulnerability in a native library besides some existing vulnerabilities produced by Droid-FF [6]. GFuzz has been shown to be an effective and efficient fuzzer with experiments conducted over two weeks. In the future, some part of the exploitable analysis is aimed to be automated and integrated into the fuzzer.

ACKNOWLEDGMENT

This study is partly supported by DAASE (Dynamic Adaptive Automated Software Engineering) and TUBITAK (Project No: 115E150) projects. We would like to thank them for their support.

REFERENCES

[1] W. A. Social and Hootsuite. (2018) 2018 digital yearbook. https://www.slideshare.net/wearesocial/2019-digital-yearbook-86862930.

[2] Android. (2017) Android Security 2017 Year in Review. (Visited January 2019) [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2017_Report_Fir

[3] Android. (2016) Android Security 2016 Year in Review. (Visited January 2019) [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2016_Report_Fir

[4] Barton P. Miller. (1988) Original fuzz project assignment. (Visited January 2019) [Online]. Available: http://pages.cs.wisc.edu/ bart/fuzz/CS736- Projects-f1988.pdf.

[5] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, no. 99, pp. 1–22, 2018.

[6] A. Joseph, "Droid-ff: The first android fuzzing framework," (Visited January 2019) [Online]. Available: https://conference.hitb.org/hitbsecconf2016ams/sessions/hitb-lab-droid-ff-the-first-android-fuzzing-framework/.

[7] A. Bechtsoudis, "Fuzzing objects dárt: Digging into the new android l runtime internals.(2015)," (Visited January 2019) [Online]. Available: https://conference.hitb.org/hitbsecconf2015ams/sessions/fuzzing-objects-d-art-runtime-internals/.

[8] S. Kyle, H. Leather, B. Franke, D. Butcher, and S. Monteith, "Application of domain-aware binary fuzzing to aid android virtual machine testing," in *ACM SIGPLAN Notices*, vol. 50, no. 7. ACM, 2015, pp. 121–132.

[9] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian *et al.*, "Toward engineering a secure android ecosystem: A survey of existing techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 38, 2016.

[10] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–12.

[11] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[12] S. R. Choudhary, A. Gorla, and A. Orso, “Automated test input generation for android: Are we there yet?(e),” in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 429–440.

[13] V. Hatas. (2018) Android dexdump buffer overflow vulnerability. WISE Lab., Hacettepe University. https://seclists.org/fulldisclosure/2018/Sep/2.

[14] (2019) zzuf: multi-purpose fuzzer. (Visited January 2019) [Online]. Available: http://caca.zoy.org/wiki/zzuf.

[15] (2019) UI/Application Exerciser Monkey. (Visited January 2019) [Online]. Available: https://developer.android.com/studio/test/monkey.html.

[16] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An input generation system for android apps,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[17] S. Anand, M. Naik, M. J. Harrold, and H. Yang, “Automated concolic testing of smartphone apps,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 59.

[18] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, “Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications,” in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93–104.

[19] M. Y. Wong and D. Lie, “Intellidroid: A targeted input generator for the dynamic analysis of android malware,” in *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[20] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Droidbot: a lightweight ui-guided test input generator for android,” in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 23–26.

[21] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, “Automated test input generation for android: Are we really there yet in an industrial case?” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 987–992.

[22] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, “Automated test input generation for android: towards getting there in an industrial case,” in *Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 253–262.

[23] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, “An empirical study of android test generation tools in industrial cases,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 738–748.

[24] K. Mao, M. Harman, and Y. Jia, “Sapienz: multi-objective automated testing for android applications,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[25] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based gui testing of android apps,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 245–256.

[26] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, “Aimdroid: Activity-insulated multi-level automated testing for android applications,” in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 103–114.

[27] A. Blanda, “Fuzzing android: a recipe for uncovering vulnerabilities inside system components in android,” *BlackHat EU*, vol. 2015, 2015.

[28] (2019) Radamsa: a general-purpose fuzzer. (Visited January 2019) [Online]. Available: https://gitlab.com/akihe/radamsa.

[29] (2019) PEACH Fuzzer Community Edition. (Visited January 2019) [Online]. http://www.peach.tech/resources/peachcommunity/.

[30] R. Mahmood, N. Mirzaei, and S. Malek, “Evodroid: Segmented evolutionary testing of android apps,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.

[31] A. Rohella and S. Takada, “Testing android applications using multi-objective evolutionary algorithms with a stopping criteria,” in *30th International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*. Knowledge Systems Institute Graduate School, 2018, pp. 308–313.

[32] D. E. Goldberg, *Genetic algorithms*. Pearson Education India, 2006.

[33] Android. (March 2019) Dalvik executable format. https://source.android.com/devices/tech/dalvik/dex-format.html.

[34] (2019) Enable multidex for apps with over 64K methods. (Visited January 2019) [Online]. Available: https://developer.android.com/studio/build/multidex.

[35] Anestis Bechtsoudis. dexRepair. (Visited January 2019) [Online]. Available: https://github.com/anestisb/dexRepair.

[36] C. Sheppard, “Genetic algorithms with phyton,” (Visited January 2019) [Online]. Available: https://github.com/handcraftsman/GeneticAlgorithmsWithPython/blob/master/ch01/gene

[37] Symantec. (April 2017) Internet security threat report, volume 22. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf.

[38] A. Developers. (2019) Distribution dashboard. https://developer.android.com/about/dashboards.