Code Coverage Measurement for Android Dynamic Analysis Tools

Chun-Ying Huang
Dept. of
Computer Science and Engineering
National Taiwan Ocean Univsersity
Keelung, Taiwan
Email: chuang@ntou.edu.tw

Ching-Hsiang Chiu
Dept. of
Computer Science and Engineering
National Taiwan Ocean Univsersity
Keelung, Taiwan
Email: chchiu@snsl.cs.ntou.edu.tw

Chih-Hung Lin
CyberTrust Technology Institute
Institute for Information Industry
Taipei, Taiwan
Email: chlin@iii.org.tw

Han-Wei Tzeng
Dept. of
Computer Science and Engineering
National Taiwan Ocean Univsersity
Keelung, Taiwan
Email: hwtzeng@snsl.cs.ntou.edu.tw

Abstract*—It is common to inspect an Android application using static or dynamic analysis techniques. Most traditional tools adopt static analysis techniques due to its low cost and high performance properties. However, since an inspected target could be obfuscated, it is also common to work with dynamic analysis techniques so that complete runtime information can be obtained to provide in-depth application behavior. Although there are already a lot of tools based on dynamic analysis techniques, the capability of such a tool is unknown. It is straightforward to understand the capability of a dynamic analysis tool by measuring its code coverage. However, to our knowledge, there is not a universal approach for measuring code coverage for all dynamic analysis tools, especially when a tool is only accessible remotely. In this paper, we propose an approach to measure code coverage for dynamic analysis tools. We design and implement the approach to measure code coverage for both online and off-line dynamic analysis tools. We then pick online tools including ABM, Anubis, CopperDroid, Tracedroid, as well as off-line tools including standard Android emulator, DroidBox, and DroidScope. Our measurement results show that the average coverage rate for each tool lies between 20% and 60%. We believe that our approach can provide more information for researchers and developers to better understand and improve the capability of dynamic analysis techniques.

***Keywords*-Android, code coverage, dynamic analysis, mobile security, software inspection**

I. Introduction

Mobile devices might be the most closest devices to humans' daily life. The current mobile device market is shared by Android and iOS operating system. According to reports form IDC [1], the top two players (Android and iOS) dominates the market with 76.6% and 19.7% share in Q4 2014, respectively. In comparison to iOS, Android adopts a relatively open strategy for developers and users. Developers are able to develop Android applications on most mainstream operating systems, and then distribute their applications through official Google Play market, third-party marketplace, or any (untrusted) storage spaces. Users can install applications from any sources and customize their device with their preferred interfaces and styles. Although these flexibilities have successfully attracted a large number of users, the open design also provides more chances for malicious users to approach Android users. As a result, users often feel that Android is more risky[1] than iOS, even if an application is installed from the official market. The risks brought by an application may include attaching massive advertisements, degrading system performance, occupying system resources, breaking system programs, requesting over-privileged permissions, leaking personal information, and even launching network attacks.

There are a lot of approaches to prevent an Android device from being compromised by a risky application. The first-line defense is that Android OS by default disallows a user to install an application from an untrusted source [3]. When it happens, a warning message is popped up to show that an application could be harmful because its installation source cannot be validated. However, a user can turn off this feature at any time. There are alternative on-device approaches that protect users from risky applications. Anti-virus software fits in this category. Anti-virus companies like McAfee and Kaspersky offer their approaches for Android users to do real-time protection on file accessing. Due to performance considerations, most anti-virus software is based on pattern matching techniques. There are also off-device approaches that help users to check whether an application is malicious. Most of these approaches are online tools. VirusTotal [4] is an online tool that attempts to scan an uploaded software using tens of different anti-virus scanners. Similarly, AndroTotal [5] is another online tool that designed specifically for Android applications. These tools scan files submitted by a user and return scanning reports generated by one or more virus scanners. A user is able to justify whether an application is malicious based on the reports.

It is insufficient if a scanner only determines whether an application is malicious. To further understand the behavior and characteristics of an application, we need better analysis tools. Androguard [6] is a static analysis tool that is able to do reverse engineering for an Android application package file (apk file in short). It also provides a set of tools to manage and detect malicious Android applications using control flow

[1]The use of the word "risky" is more precise then "vulnerable." The open nature of Android does actually give more chances for malicious attackers. However, as to vulnerabilities, recent reports [2] show that the number of vulnerabilities found on iOS is not less than that found on Android. The two platforms could be equally vulnerable.

graph based signatures. Andrubis [7] (later integrated into Anubis) is an online tool that is able to monitor and report detailed Android application behavior using dynamic analysis techniques. Alternatively, behavioral analytical reports may be also obtained by using an off-line tool, for example, Tracedroid [8] and DroidBox [9]. These behavior analysis tools can be implemented using static or dynamic analysis techniques. A static analysis tool does not need to launch an application. The reports are generated by making in-depth analysis to source codes (either in a high- or low-level language) and other resource files in a package. In contrast, a dynamic analysis tool has to launch an application, and then analyzes application behavior based on collected runtime information including system call traces, file accesses, network activities, and system logs. Although dynamic analysis techniques require more resources, they are able to defend against code obfuscation techniques and are better for hybrid execution environment adopted by Android. As a result, a lot of researches adopt reports from dynamic analysis tools to make further detection.

Although dynamic analysis techniques have been widely used, the capability of such a tool is unclear, and there is not a general approach to measure the capability of the tools. One major issue for dynamic analysis tools is that *they may not inspect an inspected application thoroughly*. Therefore, a fundamental metric to evaluate the capability of a dynamic analysis tool is measuring the code coverage for the tool. The code coverage is the ratio of codes that has been executed in a dynamic analysis environment. Code coverage can be measured in diverse granularities including instruction-level, line-level, block-level, function-level, or class-level. However, measuring code coverage is usually not a general and trivial task, especially when a dynamic analysis tool is deployed only online. In this work, we attempt to propose a general approach to measure code coverage for Android dynamic analysis tools. The proposed approach can be applied to both online and off-line tools. We also conduct a preliminary code coverage measurement for several dynamic analysis tools including online tools (ABM, Anubis, CopperDroid, and Tracedroid) and off-line tools (Android emulator, DroidBox, and DroidScope). We believe the proposed approach can better reveal the capability of a measured tool and provide in-depth information for future improvement.

The rest of this paper is organized as follows. We introduce related works in Section II. The proposed approach is introduced and discussed in Section III. Section IV presents our measurement results for several selected online and off-line tools. Finally, a concluding remark is given in Section V.

II. Related Work

A number of research groups have developed their own tools to understand Android application behaviors. Enck et al. [10] proposed TaintDroid, which is able to monitor the information flow inside an Android device. TaintDroid is neither a dynamic analysis tool nor a sandbox. It is a custom-built firmware with features to track how apps use sensitive information. As a result, it is not possible to implement TaintDroid as a standalone application and must be a complete system firmware. Lantz et al. [9] proposed DroidBox, which extends TaintDroid and provides additional static pre-check and API monitoring features. DroidBox detects data leaks by tainting sensitive data and placing taint sinks throughout the API. In addition, by logging relevant API function parameters and return values, a potential malware can be discovered and reported for further analysis. Anubis is an online dynamic analysis tool originally designed for inspecting malware targeted on personal computers. The core component was developed by Bayer et al. [11]. In 2012, Anubis also implemented a sandbox environment for inspecting Android applications (codename: Andrubis). In addition to dynamic analysis in sandboxes, Andrubis also performs static analysis, yielding information including the app's activities, services, required external libraries and actually required permissions. But the detailed design of Andrubis is unknown to the public. Huang et al. [12] proposed android behavior monitor (ABM), which integrates open source components and is built upon standard Android emulator. In addition to its open design, ABM adopts several strategies to improve code coverage including emulation of random user inputs, sending short messages, and making phone calls. Yan and Yin [13] proposed DroidScope to analyze Android application behavior. DroidScope is built on top of QEMU emulator and is able to reconstruct the OS-level and Java-level semantic views completely from the outside. In addition, a number of tools including API tracer, native instruction tracer, Dalvik instruction tracer, and taint tracker are developed to conduct further analysis. Reina et al. [14] proposed CopperDroid, which is also a dynamic analysis tool built upon QEMU emulator. CopperDroid has similar designs and implementations to DroidScope. CopperDroid monitors low-level system calls so it is able to monitor malware behavior whether it is initiated from Java, JNI or native code execution.

In addition to the tools themselves, a number of researches attempt to detect malicious applications based on dynamic behavior. Xie et al. [15] proposed pBMDS to observe unique behaviors of the mobile phone applications and the operating users on input and output constrained devices, and leverage a hidden Markov model to learn application and user behaviors from process state transitions and user operational patterns. Based on these information, pBMDS can identify behavioral differences between malware and human users Bläsing et al. [16] proposed AASandbox that performs both static analysis and dynamic analysis on android programs to automatically detect suspicious applications. Static analysis scans the software for malicious patterns at the source code level. Dynamic analysis executes the application in a sandbox that monitors and logs low-level accesses to the system for further analysis. It is unfortunately that the authors did not show the performance on analyzing malware. Burguera et al. [17] also proposed to detect malware using behavior dynamics. They developed a client named Crowdroid that is able to monitor Linux kernel system calls and report them to a centralized server. Based on the collected dataset, they cluster each dataset using a partition clustering algorithm and hence differentiate between benign and malicious applications. Zhou et al. [18] proposed DroidRanger to detect malicious applications based on both static and dynamic features. They first propose a permission-based behavioral foot printing scheme to detect new samples of known Android malware families. They then applied a heuristics-based filtering scheme to identify certain inherent behaviors of unknown malicious families. Evaluations show that their system can detect known malware as well as certain zero-day malware.

There are a lot of dynamic analysis tools, but seldom of them discussed about their code coverage rates and approaches to measure code coverage rates. Veen and Rossow [8, 19] claim that the average code coverage rate is about 33%. However, the author may measure the number using their custom approaches, which could be not applicable to other platforms. Although the code coverage is not measured, literatures have recommended several strategies to improve code coverage. The simplest approach is to work with the Android Monkey tool [20] to generate random input events. Mahmood et al. [21] proposed a systematic approach to generate a large number of test cases for fuzzing an application, as well as a test bed that given the generated test cases, executes them in parallel on numerous emulated Androids running on the cloud. The cases are generated by reverse engineering an application and create relationships between UIs and codes. The generated test cases would have better code coverage than random input events, but it is not clear how the coverage rate is measured. Machiry et al. [22] proposed DynoDroid, which is also an input generating system that attempts to improve the code coverage for testing Android applications. The code coverage of DynoDroid is measured using Emma [23]. However, there are several restrictions when using Emma with Android. The details are discussed later in Section III-A

III. The Proposed Approach

Android applications are major developed with the Java programming language. Developing with Java is not new, and there are already a number of tools that can do coverage analysis for Java codes. However, these tools do not work with Android's development environment directly. As a result, the official Android software development kit (SDK) has incorporated the well-known Emma tool to its development environment. Although Android SDK has incorporated Emma, an inspected application has to be modified and meets the requirements of running Emma on Android. In this section, we give a brief description on using Emma on Android first and then introduce how our approach works.

A. Emma on Android

Emma is a code coverage measurement tool originally designed for Java, and Android has incorporated Emma to be parts of its SDK. There are two major benefits of using Emma. First, most Android applications with source codes can be measured. Second, Emma produces a detailed and comprehensive report for a developer, and code coverage rates are presented in various granularities including class-level, method-level, block-level, and line-level. Although Emma is a feature-rich tool, there are some restrictions when a developer attempts to run Emma on Android.

- The "on-the-fly" mode is not supported. This means that we cannot measure coverage rates for pre-compiled binaries.
- Source codes must be modified to load Emma.
- Support only Java bytecode, not Dalvik bytecode [24].
- Must enable the instrumentation mode by using the *am* script to launch the inspecting application under the shell prompt. Alternatively, readers may work with the plug-in integrated with Eclipse or Android Studio IDE.

Due to the above restrictions, Emma only works with a local application having complete source codes. It is also clear that Emma could not be a good choice from the perspective of large-scale evaluation of code coverage for dynamic analysis tools.

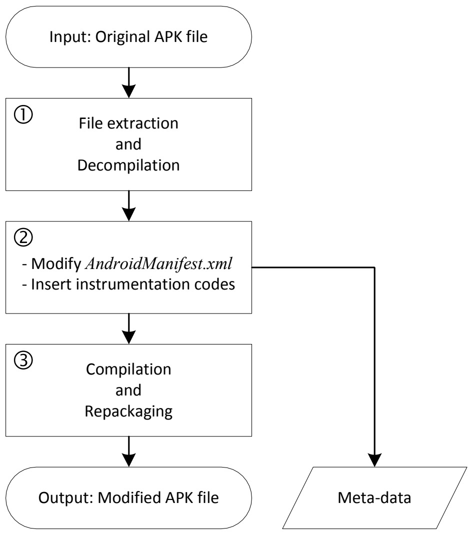

Fig. 1. The process of the proposed approach.

B. Our Approach

Our approach has three design objectives. First, we expect to measure the code coverage without Java source codes. Second, we expect to support both local measurement and remote measurement. Third, we expect to provide similar code coverage granularities to Emma. The basic idea of our approach is simple. We decompile an apk file back to its source code in assembly language. We then insert measurement codes into the decompiled codes, recompile, repackage, and finally use the patched binary file to test code coverage rates. The process is shown in Figure 1. We discuss the details of the three major steps in the figure in later subsections.

1) File Extraction and Decompilation: In the first step, we extract files inside an Android application package (apk) file and do the decompilation. Although an apk file is actually a zip archive, we use *apktool* [25] to extract files because

TABLE I
LIST OF SMALI BRANCH INSTRUCTIONS.

GOTO	IF		OTHER
goto	if-eq	if-eqz	packed-switch
goto/16	if-ne	if-nez	sparse-switch
goto/32	if-lt	if-ltz	try ... catch ...
	if-ge	if-gez	
	if-gt	if-gtz	
	if-le	if-lez	

the tool also converts files like *AndroidManifest.xml* into a human readable format. The decompilation is done by using the *baksmali* tool [26]. *baksmali* reads the *classes.dex* file in an apk file and converts each identified bytecode into assembly source codes, i.e., the *.smali* files. The assembly language syntax is loosely based on Jasmin's/dedexer's syntax, and supports the full functionality of the dex format including annotations, debug info, and line info.

2) Modification and Code Instrumentation: In the second step, we have to patch the extracted files obtained in the first step. Similar to Emma, required permissions for writing measurement results have to be declared in the *AndroidManifest.xml* file. To handle the decompiled source codes, we implement a compiler to parse smali codes and insert our customized instrumentation codes. To achieve different levels of granularities, the compiler has to identify correct places to insert instrumentation codes. We discuss how instrumentation codes are inserted to provide different levels of coverage granularities. For class-level granularity, it is straightforward because smali use a single file to store a single class. We can simply find the constructor of each class and insert instrumentation codes at the entry point of a constructor. For method-level granularity, it is also straightforward because smali use a *.method* keyword to indicate the beginning of a function. Similarly, instrumentation codes can be inserted at the entry point of a method. Handling block-level granularity is a little bit complicated because the compiler has to identify the entry point of each block. To identify a block, we check instructions listed in Table I. A branch instruction is used as a separator to split a code block into two blocks. Based on the logic, we identify block entry points and insert instrumentation codes for each identified block. Finally, for line-level granularity, the compiler simply reads comments in the generated smali files, which have indicated the line numbers in the code.

We assign a *ClassId* to each identified class. *ClassId* is a unique and non-negative integer in an application package. Similarly, we also assign *MethodId* and *BlockId* to each identified method and block, respectively. With these identities, a unique identity string can be generated for each instrumented point. The identity string is in the format of "*ClassId # MethodId # BlockId*." The instrumentation code inserted into smali codes is simply a function call in the following form of *doCount("ClassId # MethodId # BlockId")*, where *doCount* is a static function exported from a top-level class. A sample implementation of *doCount* function is given in Figure 2, which uses a *HashMap* to count the number of appearance of each identity string.

```
public class Coverage {
  static HashMap<String, Integer> rec =
      new HashMap<String, Integer>();
  public static void doCount(String id) {
    if(rec.containsKey(id)) {
      rec.put(id, rec.get(id)+1);
    } else {
      rec.put(id, 1);
    }
  }
};
```

Fig. 2. A sample implementation of the *doCount* function.

TABLE II
A SAMPLE OF DALVIK REGISTER ARRANGEMENT IN A METHOD.

Name	Alias	Register purpose
v0		Store the 1st local variable.
v1		Store the 2nd local variable.
v2	p0	Store the 1st parameter passed to the method.
v3	p1	Store the 2nd parameter passed to the method.
v4	p2	Store the 3rd parameter passed to the method.

Although it looks straightforward to insert instrumentation codes at the identified locations, the implementation is nontrivial due to Dalvik virtual machine's function calling convention. Dalvik virtual machine uses registers to pass function call parameters instead of stacks. As a result, when the instrumentation code attempts to make a function call, the additional register used to passing string identity could affect the regular use of other registers originally used in the patched codes, and even crash the application. To prevent patched codes from being crashed, we discuss two possible strategies, *direct-call* and *indirect-call*, to insert instrumentation codes without destroying the patched application.

The direct-call strategy is to call the *doCount* function directly at the instrumented point. Since we need to pass an identity string to the counting function, the simplest way to patch the codes is to add an additional register to store the identity string. However, it does not always work due to Dalvik's instruction restriction on using the registers. An example of register arrangement in a method is shown in Table II. In the example, the method has two local variables and three parameters. Dalvik virtual machine uses registers of smaller index numbers to store local variable and registers of higher index numbers to store function parameters. There are aliases created for function parameters. In the example, the $p0$, $p1$, and $p2$ are aliases for register $v2$, $v3$, and $v4$, respectively. If an additional register is added for storing a local variable, the variable would be $v2$ and all register index number for function parameters are increased by one as well. Note that the aliases are still referenced to the correct registers, i.e., $p0$, $p1$, and $p2$ become aliases for $v3$, $v4$, and $v5$, respectively. Adding an additional local variable register may cause an issue

```
const-string v1, "32#0#0"
invoke-static {v1}, Lorg/test/Coverage;->doCount(Ljava/lang/String;)V

const-string v17, "32#23#0"
invoke-static/range {v17 .. v17}, Lorg/test/Coverage;->doCount(Ljava/lang/String;)V
```

Fig. 3. Samples of code instrumentation using the direct-call strategy.

```
.method public static count_32_23_0()V
   .locals 1
   const-string v0, "32#23#0"
   invoke-static {v0}, Lorg/test/Coverage;->doCount(Ljava/lang/String;)V
   return-void
.end method
```

Fig. 4. A sample of code instrumentation using indirect-call strategy.

because not all registers are accessible for all instructions. Some instructions like `move` and `invoke` use a 4-bit field to store register index number. As a result, these instructions can only access register $v0$—$v15$. If a parameter register moves from $v15$ to $v16$, instructions originally used to access $v15$ may not work for $v16$. We summarize possible cases when patching a method. Suppose originally we have l local variable registers and p parameter registers.

a) $l + p \leq 15$: This case is safe. When adding an additional local variable register, the total number of registers is still less than or equal to 16.
b) $l \geq 16$: This case is also safe because all parameter registers are greater than 15. These registers must be accessed using compatible instructions.
c) $l < 16$ and $l + p \geq 16$: This case is not safe. In this case, there must be one parameter register moves from $v15$ to $v16$. Incompatible instructions access to the $v16$ register trigger exceptions and crash the program.

Samples using the direct-call strategy can be found in Figure 3.

In contrast to the direct-call strategy, the indirect-call strategy wraps the calling of *doCount* into a newly created wrapper function and then calls the wrapper function at the instrumented location. An example of function wrapping is given in Figure 4. In the example, the calling of *doCount("32#23#0")* is implemented in the *count_32_23_0* function. The corresponded instrumented location can simply call the *count_32_23_0* function, which requires no parameter and therefore no additional registers are required. One limitation of the indirect-call strategy is that, a single *.dex* file can only contain a maximum number of 65536 functions. In case the number of functions exceeded the limitation, we can consider to either use only class-level and method-level granularities or split a single *.dex* file into multiple *.dex* files. Although implementing with the direct-call strategy is more concise, due to its complexity on handling register dynamics, we consider the *indirect-call* strategy to implement the proposed approach. When all code instrumentations are done, a meta-file is generated to store the mapping from *ClassId* and *MethodId* to class name and method name, respectively.

```
1#8#0      1
1#8#1      1
1#12#24      1
1#12#25      1
...
Visited classes: 2
Visited methods: 7
Visited blocks: 40
```

Fig. 5. A sample result generated based on the measurement results.

3) Compilation and Repackaging: The final step is to compile assembly source codes into the *.dex* files and repackage all the files into an Android package. The compilation can be done by using the *smali* tool and the repackaging can be done using the *apktool*. Once a patched package is available, we can submit it to an online or off-line dynamic analysis tool and receive the measurement results. In addition to store the results in a local storage, when working remotely, our measurement codes are configured to send measurement results periodically back to our own server. A sample measurement result is shown in Figure 5. By matching the identity strings against the meta-data generated in the second step, we can restore the string identities back to the original class names, method names, and block IDs. The coverage rate can be also computed based on the measurement results and the meta-data. To create a diverse application packages for evaluating the real capability of a dynamic analysis tool, the proposed process can be done automatically without human interaction. A number of apk files are prepared, and are fed to the proposed approach for generating test packages. The test packages are then used to evaluate a dynamic analysis tool.

IV. Evaluation

We evaluate the proposed approach from three aspects. We first evaluate the ratio of packages that can be successfully reverse engineered, patched, and repackaged. We then compare the measured coverage rate for our approach against Emma. Finally, we present the measurement results for selected Android dynamic analysis tools.

A. Repackaging Success Rate

We downloaded top three application package files from 26 categories available on Google Play market plus 12 randomly selected applications (total 90) and test if a package can be properly handled by our proposed approach. Among all the 90 packages, 76 of them can be correctly extracted using the *apktool* and decompiled by *baksmali*. However, after patching the codes, only 36% of them (27 packages) can be recompiled and repackaging to apk files. The failure on extracting a package is because *apktool* is not able to correctly handle file names of images in 9-patch format. The failure on repackaging a package is due to the number of newly created methods exceeds the maximum limit of a single *.dex* file. Although the success rate is not high, we can still generate sufficient large number of testing packages at the current stage. Improving the repackaging success rate is one of our future works.

B. Comparison with Emma

We then compare the measurement result of code coverage rate reported from Emma and our proposed approach. Since Emma requires Java source codes, we downloaded six application projects from the F-droid website [27], which is an online repository for hosting open source Android applications. Each project is compiled and tested using official Android emulator for five times. The compared result is shown in Table III. The result shows that our approach has measurement results similar to Emma.

TABLE III
Compare our approach against Emma.

Package: `net.sourceforge.opencamera`				
	Class	Method	Block	Line
Emma	50.30%	46.36%	33.19%	44.29%
Ours	53.94%	49.83%	39.30%	42.17%
Package: `com.uberspot.a2048`				
	Class	Method	Block	Line
Emma	100.00%	64.71%	70.48%	68.04%
Ours	100.00%	61.11%	65.17%	61.40%
Package: `com.ruesga.android.wallpapers.photophase`				
	Class	Method	Block	Line
Emma	54.62%	37.62%	31.79%	32.75%
Ours	37.58%	43.62%	36.85%	38.95%
Package: `com.monead.games.android.sequence`				
	Class	Method	Block	Line
Emma	57.69%	41.62%	36.20%	36.81%
Ours	50.00%	41.08%	35.07%	35.56%
Package: `com.notriddle.budget`				
	Class	Method	Block	Line
Emma	54.55%	35.90%	23.35%	25.74%
Ours	50.47%	33.90%	21.59%	26.54%
Package: `home.jmstudios.calc`				
	Class	Method	Block	Line
Emma	93.48%	46.59%	46.95%	40.95
Ours	89.24%	45.98%	41.75%	38.59

C. Measuring Android Dynamic Analysis Tool

We finally use the proposed approach to measure the code coverage rate of Android dynamic analysis tools. We selected both online and off-line tools. Online tools include ABM, Anubis, CopperDroid, and Tracedroid. Off-line tools include official Android emulator, DroidBox, and DroidScope. We tried to setup all the off-line tools with the same configurations. The official emulator and DroidBox both emulate Android 4.3.1 (API level 18). DroidScope emulates Android 2.3.3 (API level 10). The packages used to conduct benchmarks are those packages described in Section IV-A. We evaluate each selected tool five times using the packages, and the method-level and block-level results are shown in Figure 6, 7 and Figure 8, 9, respectively. Due to space limitations, we only showed 11 results from all the results. We summarize our findings based on the measurement results. First, we found that online tools have performances similar to off-line tools. This might be because most online tools are simply integration works and have setup similar to these off-line tools. Second, we found that some measurement results show a coverage rate close to zero. Based on our experiences with off-line analysis tools, we think the phenomenon is caused by the following two major reasons. First, due to limited API levels implemented on dynamic analysis tools, some of these zero coverage rates are because of incompatible API levels. In this case, an application package is not launched by the tool. On the other hand, some of the zero rates are because an application package crashes or stops in a very short time. Most tools still report that the execution is successful. Finally, we found that the measured coverage rates for all the tools are not good enough, which lies between 20% and 60%. An application package could have a lower coverage rate because it is blocked by UI operations like login forms, license agreements, and even pop-up advertisements. These findings show that there is still a large room for researchers to improve the performance of dynamic analysis tools.

V. Conclusion

Modern Android behavioral analysis tools often implement dynamic analysis techniques for understanding run-time behavior of applications. However, the capabilities of various dynamic analysis tools are not clear to their users. In this paper, we proposed a universal approach to systematically measure the code coverage rate of Android dynamic analysis tools, and it is applicable to various implementations including online and off-line tools. We evaluated several well-known online and off-line tools include ABM, Anubis, CopperDroid, Tracedroid, DroidBox, DroidScope, and official Android emulator. Our measurement results show that: 1) Online tools have similar performance to off-line tools; 2) A good tool has to provide diverse API levels so that it is able to inspect as many packages as possible; 3) The code coverage rate is still not good enough, which lies between 20% and 60%. These

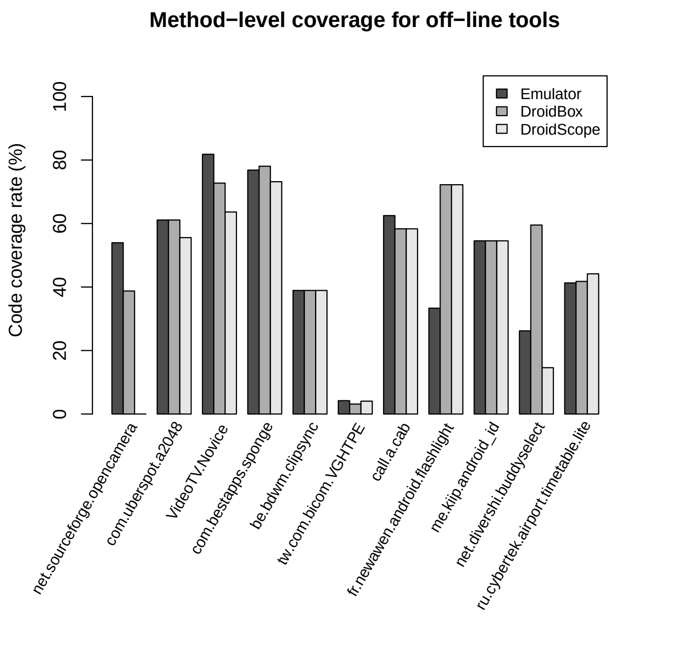

Fig. 6. Code coverage rate measurement results: Method-level results for off-line tools.

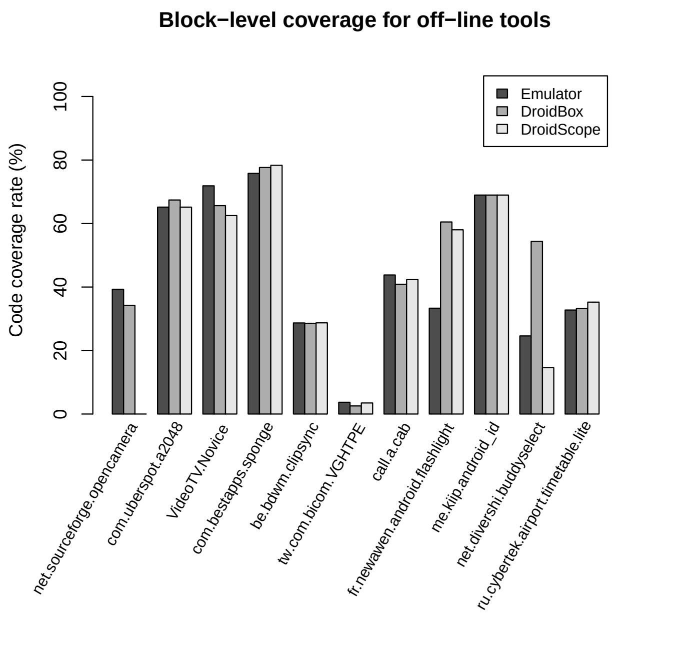

Fig. 8. Code coverage rate measurement results: Block-level results for off-line tools.

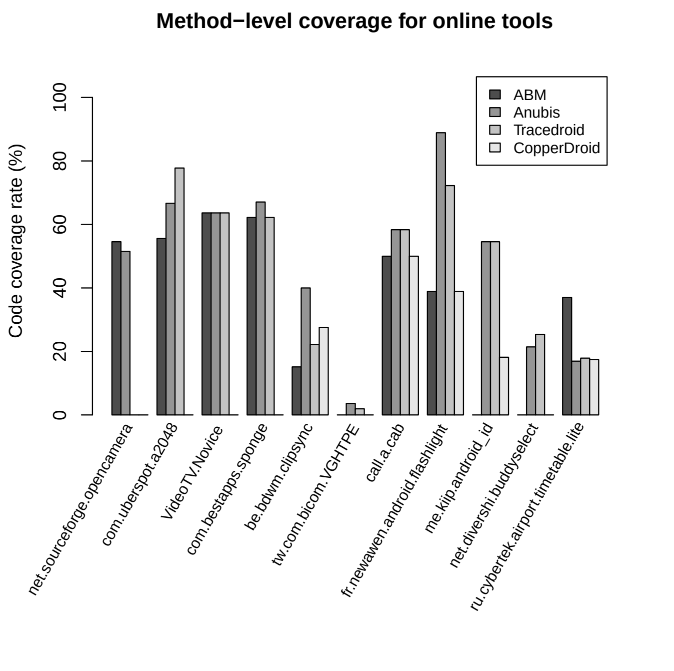

Fig. 7. Code coverage rate measurement results: Method-level results for online tools.

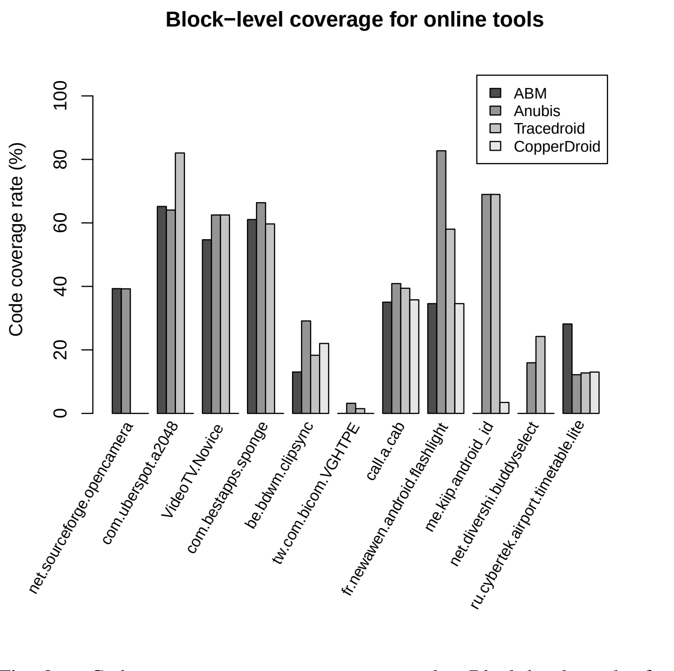

Fig. 9. Code coverage rate measurement results: Block-level results for online tools.

findings show that there is still a large room for improving the performance of dynamic analysis tools. We believe that a good dynamic analysis tools is the key to build up a success security analysis and software quality assessment framework. To better understanding and improving the capabilities of dynamic analysis tools, our future works include conducting automated large-scale evaluations and improving code coverage rates of dynamic analysis tools.

References

[1] International Data Corporation, "Smartphone OS market share, Q4 2014." [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] C. Florian, "Most vulnerable operating systems

and applications in 2014," February 2015. [Online]. Available: http://www.gfi.com/blog/most-vulnerable-operating-systems-and-applications-in-2014/

[3] Google, Inc., "Google settings (android): Protect against harmful apps." [Online]. Available: https://support.google.com/accounts/answer/2812853

[4] "VirusTotal - free online virus, malware and URL scanner." [Online]. Available: https://www.virustotal.com/

[5] "Scan Android application - AndroTotal." [Online]. Available: http://andrototal.org/

[6] "Reverse engineering, malware and goodware analysis of android applications ... and more (ninja !)." [Online]. Available: https://github.com/androguard/androguard

[7] "Anubis: Malware analysis for unknown binaries." [Online]. Available: http://anubis.iseclab.org/

[8] V. van der Veen and C. Rossow, "Tracedroid - dynamic Android app analysis." [Online]. Available: http://tracedroid.few.vu.nl/

[9] P. Lantz and A. Desnos, "DroidBox: An android application sandbox for dynamic analysis." [Online]. Available: https://code.google.com/p/droidbox/

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[11] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A tool for analyzing malware," in *Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference*, ser. EICAR, 2006.

[12] C.-Y. Huang, S.-P. Ma, M.-L. Chang, C.-H. Chiu, and T.-C. Huang, "An open and automated android behavior monitor in cloud," *Journal of Internet Technology*, vol. 18, no. 2, pp. 297–305, Mar 2014.

[13] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29.

[14] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," in *Proceedings of the* 6^{th} *European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.

[15] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "pBMDS: A behavior-based malware detection system for cellphone devices," in *Proceedings of the Third ACM Conference on Wireless Network Security*, ser. WiSec '10. New York, NY, USA: ACM, 2010, pp. 37–48.

[16] T. Blsing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, Oct 2010, pp. 55–62.

[17] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM, 2011, pp. 15–26.

[18] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*. The Internet Society, 2012.

[19] V. van der Veen, "Dynamic analysis of android malware," Master's thesis, VU University Amsterdam, August, 2013.

[20] "UI/Application exerciser Monkey." [Online]. Available: http://developer.android.com/tools/help/monkey.html

[21] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of android applications on the cloud," in *Proceedings of the 7th International Workshop on Automation of Software Test*, ser. AST '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 22–28.

[22] 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*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234.

[23] "EMMA: a free Java code coverage tool." [Online]. Available: http://emma.sourceforge.net/

[24] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Conference on Security*, 2011, pp. 21–21.

[25] "A tool for reverse engineering Android apk files." [Online]. Available: http://ibotpeaches.github.io/Apktool/

[26] "smali: An assembler/disassembler for Android's dex format." [Online]. Available: https://code.google.com/p/smali/

[27] "F-Droid: Free and open source app repository." [Online]. Available: https://f-droid.org/