Comparing Capability of Static Analysis Tools to Detect Security Weaknesses in Mobile Applications

Tosin Daniel Oyetoyan[1] and Marcos Lordello Chaim[2]

[1] Department of Software Engineering, Safety and Security
SINTEF Digital, Trondheim
Norway
tosin.oyetoyan@sintef.no
[2] Software Analysis and Experimentation Group (SAEG)
School of Arts, Sciences and Humanities
University of Sao Paulo
chaim@usp.br

Abstract. Smartphones are prevalent today and store sensitive and private data. Malicious applications are constant threats to user data on smartphones as they could sniff or manipulate them by exploiting software weaknesses in legitimate mobile applications. Static analysis tools can be used to reduce these risks during development. However, it is important to know the capability of these tools in order to make informed decisions and avoid false-sense of security. In this preliminary study we investigate the detection capability of mainstream vs. Android-specific tools to guide decision-making during tools' selection.

Keywords: Security, Android, Static analysis tools, Mobile, CWE, OWASP

1 Introduction

Smartphone devices are very popular today. These devices aggregate personal data related to our lifestyle, relationships, finances, professions, locations, recordings, conversations, preferences, videos and photos [21]. These are very sensitive and private data. A breach as a result of vulnerabilities in the mobile software could have devastating impact on the user. Malicious mobile applications could sniff and manipulate sensitive user data [5] or even launch a denial-of-service attacks [19]. Despite these challenges, developers often do not code with a mindset

In: M.G. Jaatun, D.S. Cruzes (eds.): Proceedings of the International Workshop on Secure Software Engineering in DevOps and Agile Development (SecSE 2017), published at http://ceur-ws.org

of attackers because they care more about functionalities. As a result, common and inadvertent mistakes become exploitable vulnerabilities [5].

Static analysis of the application's source or object code has been advocated as a strategy to detect weaknesses [4] during implementation. The goal is to detect part of the code that could become vulnerable. Static analysis tools (SATs) are utilized to support developers to identify security risks in their code. The goal of this research is to assess tools that detect security-related weaknesses in Android applications. We choose Android because of its open platform and market dominance. Data from the third quarter of 2016 show Android with 86.8% of marketshare followed by Apple's iOS with 12.5% and others (e.g., Windows phone, Symbian) with 0.7% [6]. In addition, other smartphone platforms have similar security model, however, Android is claimed to have the most sophisticated application communication system [5].

In Android, user-installed applications are sandboxed, each runs in a dedicated process, each has its own private data directory, and employs the least privilege principle [9]. Android defines four types of components: **Activity** (user interface), **Service** that executes processes in the background, **Content Provider** for data sharing, and **Broadcast Receiver** that responds asynchronously to system-wide messages. Communication between applications are achieved through a message passing mechanism (Intent messages). Configuration of application components are done in the mandatory manifest file. In order to protect applications, Android defines four types of permissions: Normal, Dangerous, Signature, and SignatureOrSystem.

Specific challenges in Androids make static analysis different from regular Java applications [18]. Android apps run in a special virtual machine named Dalvik that generate bytecodes differently from regular Java virtual machine. As a result, static analysis tools must be able to analyze the Dalvik bytecode when Java source code is not provided. Further, Android apps could have many entry (Main) points which make them different from regular Java applications. Additionally, in Android apps, different components have their own lifecycle. Because these lifecycle methods are not directly linked to the execution flow, they limit the soundness of some analysis scenarios.

Organizations develop both standard desktop and mobile applications, and also manage them in a similar Software Configuration Management environment. Moreover, in agile development and DevOp environments, tools are success factors that ensure continuous deployment and fast delivery [8]. The tendency is to run one type of SAT across the code base during a build operation. In our experience, a common question that practioners have asked us is whether mainstream SATs are good enough for scanning mobile applications. We are thus interested to compare *non-specific* and Android-specific SATs in their capability in terms of strengths and limitations to detect relevant mobile-related weaknesses. This is relevant to allow users make informed decisions about what tools to use, how to use them, and what results to expect. Our mainstream tools are chosen from the open source community based on availability and accessibility. In this preliminary study, we concern ourselves with the scope of weaknesses that can be found

by Android-specific SATs and mainstream SATs. The following two research questions summarize the problems we partly address in this paper:

RQ1. What are the similarities and differences between mainstream SATs and Android SATs in the type of weaknesses they detect?
RQ2. What are the runtime costs of executing SATs in mobile apps?

We have used the combination of common weaknesses and enumeration (CWE) dictionary by MITRE [20] and OWASP top 10 2016 data for our assessment.

The remainder of the paper is organized as follows: In Section 2, we discuss the approach we have used in this study. Section 3 presents our preliminary results and provides some discussions of the results. In Section 4, we present an overview of related studies. Section 5 discusses the limitations and threat to the validity of our work. Finally, we conclude the paper in Section 6.

2 Approach

2.1 Common Weaknesses in Android Applications and CWE Selection

Based on OWASP top 10 2016, the most common security risks in mobile applications are: (1) Improper platform use, (2) Insecure data storage, (3) Insecure communications, (4) Insecure authentication, (5) Insuficient cryptography, (6) Insecure authorization, (7) Client code quality issues, (8) Code tampering, (9) Reverse engineering, and (10) Extraneous functionality [23]. Many empirical studies have as well validated the existent of these risks in many real-world Android applications. (see [5,11,15,19])

In this preliminary assessment, we have used 8 weaknesses [20] categories to assess the selected static analysis tools. Three categories are specific to Android applications. The rest are general quality weaknesses applicable to all applications. The rationale behind this choice is to investigate how the tools could detect weaknesses in the different categories. Additionally, we mapped the selected CWEs to the OWASP's top security risk categories.

CWE-927: Use of Implicit Intent for Sensitive Communication (#3) An implicit intent can be used to transmit data without specifying the receiver. It is possible for any application to process the intent by using an Intent Filter for the intent.

CWE-926: Improper Export of Android Application Components (#1) Android application components (Activity, Service, or Content Provider) are exported through the manifest file. Exporting components without proper restriction as to which applications can launch or access the data could result into integrity, confidentiality and availability issues.

CWE-319: Unencrypted Socket (#3) The study by Enck et al. [11] shows that certain Android applications include code that use the Socket class

OWASP – Open Web Application Security Project (`https://www.owasp.org`).

directly. Java sockets are potential attack surface as they represent an open interface to external services.

CWE-921: Storage of Sensititve Data in a Mechanism without Access Control (#2) This weakness occurs when applications store sensitive information in file systems or devices that are not protected. Examples include memory cards or USB devices.

CWE-359: Exposure of Private Information ('Privacy Violation') (#6) Accessing private data such as passwords or credit card numbers need explicit authorization. Privacy violation could occur when unauthorized entities have access to data.

CWE-478: Missing Default Case in Switch Statement (#7) This weakness occurs when code that uses switch statement omit the default case. Execution logic may be altered when the system encounters variable value not handled in the logic. Security issues may happen, if switch logic is used to handle security decision or is linked to other aspects of code where security decision happens.

CWE-611: Improper Restriction of XML External Entity Reference ('XXE') (#7) Applications that process XML documents could be vulnerable to XXE-attacks if proper validations and sanitations are not put in place. An example is the CVE-2016-6256 XML External Entity(XXE) attack in the SAP Business One Android Application.

Debug Mode Activated (DMA) (#10) There are cases where production code is shipped with developer's configuration. An example is when debug option is enabled which can lead to disclosure of confidential and senstitive data.

2.2 Selection of tools and applications

Our tool selection was guided by the tools' availability and ease of use. Both Emanuelsson and Nilsson [10] and Hofer [14] report on installation as a seemingly important metric when choosing a static analysis tool. Practitioners can be wary of tools that are very complicated to set up and use. As a result, the selected tools are open-source or those available for use without cost and are also easy to install and use.

We selected FindBugs and FindSecBugs as mainstream tools as they are widely available and used to assess code weaknesses at industrial settings. We selected 4 Android SATs that have pre-built libraries and can be easily configured and executed. Table 1 lists the Android SATs with their URLs. The techniques utilized by the tools to scrutinize a mobile app are listed in column *"Technique"*. The idea of selecting tools using different techniques is to assess their ability to identify the CWEs related to OWASP's top risk categories and also evaluate the runtime costs of each technique.

We choose 7 open-source real mobile applications for assessment. In Table 2, we present the apps, a short description, the size of the object code, and the volume of downloads. We selected apps from different domains (e.g., secure communication, content management, graphics manipulation), and with fairly large size (0.3M to 6.8M), to expose the tools to a variety of contexts. Moreover,

Table 1. Android SATs

SAT	URL	Technique
AndroidLint	http://tools.android.com/tips/lint	Source code scanner
Amandroid	http://pag.arguslab.org/argus-saf	Taint analysis
JAADS	https://github.com/flankerhqd/JAADAS	Taint analysis
Androbugs	https://www.androbugs.com/	Object code scanner

they are largely used apps: two of them have more than 1M, one more than 600K, and two more than 100M downloads. Thus, they are real-world apps which are being used by users.

Table 2. List of assessed apps

App	Description	Size (Mb)	# Down.
AntennaPod	open-source podcast manager for Android http://antennapod.org/	6.2	>100K
ConnectBot	SSH, telnet, and terminal emulator https://connectbot.org/	3.8	>1M
Conversations	XMPP-based instant messaging client https://conversations.im/	6.8	>10K
iFixit	Online repair guides for consumer electronics and gadgets https://www.ifixit.com/	6.0	< 5K
KeePassDroid	Password manager http://www.keepassdroid.com/	3.7	>1M
RingDroid	Ringtone maker https://github.com/google/ringdroid	0.344	>100K
Zxing	Barcodereader https://github.com/zxing/zxing	0.75	>600K

2.3 Analysis

We run each tool against the selected Android applications. The results of the tools are generated in different formats. This presents enormous challenge for tools' comparison. In addition, there is no pre-CWE mappings for the Android-specific tools. As a result, we manually inspect the tools' messages and map them to an appropriate CWE wherever applicable. We did not check whether the result is false positive or not in this study as we are concerned only with the identification of weakness types identified by each tool. Lastly, we manually search for the occurence of each weakness categories in the tools' result for each application.

3 Preliminary Results and Discussion

We summarise the initial results of our assessments in Table 3. The first column describes the CWE that is investigated. The second column (merged) lists the

Table 3. CWE Detected by Tools

CWE	Tools						Apps
	FindSecBugs	FindBugs	AndroidLint	Amandroid	AndroBugs	JAADS	
CWE-927: Use of Implicit Intent for Sensitive Communication	✓	×	×	×	×	✓	iFixit, AntennaPod, Conversations
CWE-926: Improper Export of Android Application Components	×	×	✓	✓	✓	×	AntennaPod, iFixit, Zxing
CWE-319: Unencrypted Socket	✓	×	×	×	×	×	iFixit
CWE-921: Storage of Sensititve Data in a Mechanism without Access Control	✓	×	×	✓	✓	✓	All apps
CWE-359: Exposure of Private Information ('Privacy Violation')	✓	×	×	×	×	×	Conversations
CWE-478: Missing Default Case in Switch Statement	✓	✓	×	×	×	×	Zxing
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')	✓	×	×	×	×	×	Keepassdroid, Zxing
Debug Mode Activated (DMA)	×	×	×	×	✓	✓	All apps

Table 4. Execution time of mainstream and Android SATs

Apps	FindSecBugs	FindBugs	AndroidLint	Amandroid	AndroBugs	JAADS
AntennaPod	5min30sec	4min1sec	1min7sec	5h52min39sec	46sec	11min59sec
ConnectBot	3min3sec	2min15sec	0min51sec	1h45min28sec	21sec	6min35sec
Conversations	3min28sec	4min12sec	0min54sec	4h59min22sec	46sec	13min57sec
iFixit	6min54sec	4min46sec	0min22sec	3h15min27sec	22sec	7min1sec
KeePassDroid	6min52sec	5min8sec	15min6sec	0h55min33sec	17sec	5min8sec
RingDroid	3min12sec	1min43sec	0min4sec	0h6min39sec	$<$ 1sec	1min43sec
Zxing	4min56sec	3min28sec	1min24sec	0h18min43sec	2sec	2min18sec

tools and indicate whether the tool finds the stated CWE. The third column lists the applications where the stated CWE is found. For example, only FindSecBugs found CWE-319 in iFixit whereas none of the other tools found this weakness in iFixit. In addition, the weakness was not spotted in the rest of the apps.

From the results, we make the following observations: AndroBugs checks inter-component communication-based, configuration and deployment weaknesses. JAADS checks inter-component, communication-based and configuration weaknesses. Amandroid analyses inter-component communication; however, users have to reason about the results. FindSecBugs is tailored for security audit in general with limited extension to Android applications. For example, it does not analyse the AndroidManifest.xml file. AndroidLint reports on many quality issues from its report but miss many specific security issues.

RQ1. What are the similarities and differences between mainstream SATs and Android SATs in the type of weaknesses they detect?

In this preliminary study, we found FindSecBugs to cover a wide range of the weakeness categories but missed the topmost important risk (CWE-926) and the OWASP top #10 (Debug Mode Activated). Mainstream SATs are therefore useful and necessary to uncover relevant mobile-specific weaknesses but they are not sufficient. Furthermore, general quality issues can sometimes be very important when they occur where security decision is being taken (e.g. Missing Default in Switch). Android-specific tools could not detect the above weakness. In addition, the Android SATs did not detect CWE-319 (Unencrypted Socket), CWE-359 (Exposure of Private Information), and CWE-611 (Improper Restriction of XML External Entity Reference). Both OWASP top #1 and #10 are not detected by any of the mainstream tools but are detected by some Android-specific tools. In addition, to check the 8 weakness categories requires at least 3 combination of tools from the mainstream and Android SATs, as a result, we conclude that one tool is not enough to catch the whole range of weaknesses.

Nevertheless, it would be possible for FindSecBugs and FindBugs to detect some Android-specific CWEs if the manifest file were analyzed and patterns particular to Android applications were supported. These relatively simple modifications would have a beneficial impact on the development of more secure Android mobile apps because FindSecBugs and FindBugs are widely known and used at industrial settings.

RQ2.What are the costs of running SATs in mobile apps? Tools' performance depends on the technique utilized. Taint analysis is more costly than code scanning for bug patterns. The time to run the selected SATs are presented in Table 4. We have used a computer running Ubuntu 16.04 LTS equipped with Intel Core i7-4510, 2GHz CPU, and 15.6 GBytes of RAM. All tools were run three times and the average time are reported in Table 4. The data for Amandroid represents the time to run the five different taint analysis provided by the tool.

We report the `user` value of the Linux `time` command for all SATs, which represents the user CPU time. The exception is AndroindLint for which we used a stop watch. For AntennaPod (in row one of Table 4), on the average, FindSecBugs took five minutes and 30 seconds, FindBugs took four minutes and

one second, AndroidLint, one minute and seven seconds, Amandroid, five hours, 52 minutes and 39 seconds, AndroBugs, 46 seconds, and JAADS, 11 minutes and 59 seconds.

The mainstream tools (FindBugs and FindSecBugs) and AndroidLint take at most tens of minutes to analyze the code because they scan it for patterns of possible vulnerabilities. AndroBugs scans the apk for particular patterns, but it does not scan the whole code. As a result, it requires few seconds to obtain its report. The most costly tools are those that utilize taint analysis. Amandroid provides a thorough analysis, but it demands a high runtime cost to obtain the data. JAADS taint analysis is much faster than Amandroid's, but its report is not as comprehensive.

This preliminary data suggest that tools that scan the code for bug patterns and perform light taint analysis can be utilized during development time. On the other hand, thorough taint analysis is only fitting in a continuous integration environment, especially, during overnight builds.

4 Related work

Empirical studies have been conducted to compare the strengths and shortcomings of SATs [14, 16, 17, 25, 27]. In general, they run SAT against a set of programs with known vulnerabilities. Most of the studies assess performance such as the precision, recall, true negative rate and accuracy of tools [17, 27]; others assess also the cost of running the tools, e.g. [14, 27]. There are also efforts that have quantitatively evaluated static analysis tools with regards to their performances to detect security weaknesses in benchmark synthetic code. The Center for Assured Software (CAS) [22] developed a benchmark test cases with "good code" and "flawed code" across different languages to evaluate the performance of static analysis tools and assessed 5 commercial tools. Goseva-Popstojanova and Perhinschi [12] investigated the capabilities of 3 commercial tools. Their findings show that the capability of the tools to detect vulnerabilities was close to or worse than random guessing. Díaz and Bermejo [7] compares the performance of nine tools mostly commercial tools using the SAMATE security benchmark test suites. They found an average recall of 0.527 and average precision of 0.7. They found also that the tools detected different kinds of weaknesses. Charest [3] compared 4 tools against 4 out of the 112 CWEs in the SAMATE Juliet test case. The best average performance in terms of recall is 0.46 for CWE89 with 0.21 average precision. All these studies have used real or synthetic code with known vulnerabilities to detect the performance of the tools. In this study, we have only investigated whether the tools can detect certain weaknesses with mappings to MITRE CWEs in the mobile apps.

Android apps have been empirical studied [1, 2, 13, 24] and various program analysis techniques for security assessment in Android have been investigated [26]. To the best of our knowledge, there are not studies that investigate similarities and differences between mainstream and Android-specific SATs. We present the first step of study to assess how mainstream vs. mobile-specific tools compare in

detecting top security risks in mobile apps. Currently, we are not focusing on tools' performance such as the recall or precision of tools but rather on whether they are able to detect specific top risks vulnerabilities relevant to Android apps. Additionally, we are interested in investigating their runtime costs.

5 Limitations and Threats to Validity

Our assessment could not cover the whole spectrum of CWEs that could map to the OWASP top-10. We have used a selected set of CWEs from MITRE dictionary and map them to the OWASP top-10 lists. Possibilities exist that other CWEs not in the list we assessed could map to any of the OWASP top-10.

This phase of our study did not focus on identifying false positives from the results of the tools. In addition, information about performance metrics such as recall or precision are not addressed in this study. In our future study, we plan to identify real weaknesses and also seed artificial weaknesses in the apps to be able to compute the performance metrics.

We have performed only manual assessment of the tools. This limit the precision and the scope of analysis we could perform. Our plan includes automatic and statistic analysis in the next phase.

The CWE we selected did not cover the entire spectrum of weaknesses relevant for mobile applications beyond the OWASP top-10. Our future work plans to expand the scope of the CWE for our analysis.

Finally, our preliminary result does not offer a strong conclusion regarding any of the tools we have assessed. This is a limitation but also a cautious one because we have not provided the actual performance of the tools but rather their detection capabilities. However, the result does provide useful advice regarding the possibilities of Android SATs and mainstream SATs for detecting weaknesses in mobile applications.

6 Conclusions and Future Work

We report the initial assessment of the SATs capability to detect top security risks in mobile applications. The verification of the CWEs detected by the tools were carried out manually which constitutes a threat to the internal validity of the results. Although we have selected apps with different characteristics, we caution the reader not to expand the conclusions beyond the set of the selected apps. In our future work, we plan to automate the collection and analysis of the data from the apps to reduce the risks to internal and external validity. Additionally, we intend to conduct statistical analysis of the results to support the conclusions.

We presented the first step of a research on the capability of mainstream and Android-specific static analysis tools to detect security weaknesses in mobile apps. The results of a preliminary assessment of two mainstream tools (FindBugs and FindSecBugs) and 4 Android-specific tools (Amandroid, AndroBugs, AndroidLint, and JAADS) are presented. These tools were run against 7 real-world mobile apps.

In this preliminary study, we found that mainstream tools can cover a wide range of the weakeness categories; however, important risks may go undetected if the practitioner rely only on these tools. On the other hand, Android-specific tools were able to detect top risk weakeness but also miss some general security and quality issues. The runtime cost of the tools is dependent on the analysis technique. As expected, data-flow based techniques (e.g., taint analysis) are more costly than scanning for bug patterns. Our initial assessment indicates that practitioner cannot prescind from the mainstream tools when developing mobile apps. Nevertheless, she or he should consider adding Android-specific tools to cover significant risk categories. In our future work, we aim to conduct a large scale study of many Android applications and many static analysis tools. We are also interested in assessing the quality of the tools' results. For example, what percentage of the detected OWASP Top 10 risks are false positives.

Acknowledgments This research was carried out within the project "SoS-Agile: Science of Security in Agile Software Development", funded by the Research Council of Norway, under the grant 247678/O70. Marcos L. Chaim's was on a research stay in Norway and was funded by a personal guest researcher scholarship from the IKTPLUSS program.

References

1. Avancini, A., Ceccato, M.: Security testing of the communication among android applications. In: 2013 8th International Workshop on Automation of Software Test (AST). pp. 57–63 (May 2013)
2. Chan, P.P., Hui, L.C., Yiu, S.M.: Droidchecker: Analyzing android applications for capability leak. In: Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks. pp. 125–136. WISEC '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2185448.2185466
3. Charest, N.R.T., Wu, Y.: Comparison of static analysis tools for java using the juliet test suite. In: 11th International Conference on Cyber Warfare and Security. pp. 431–438 (2016)
4. Chess, B., McGraw, G.: Static analysis for security. IEEE Security & Privacy 2(6), 76–79 (2004)
5. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. pp. 239–252. ACM (2011)
6. Corporation, I.D.: Smartphone os market share, 2016 q3 (2017), http://www.idc.com/promo/smartphone-market-share/os, visited on June, 13 2017
7. Díaz, G., Bermejo, J.R.: Static analysis of source code security: Assessment of tools against samate tests. Information and software technology 55(8), 1462–1476 (2013)
8. Dybå, T., Dingsøyr, T.: Empirical studies of agile software development: A systematic review. Information and software technology 50(9), 833–859 (2008)
9. Elenkov, N.: Android security internals: An in-depth guide to Android's security architecture. No Starch Press (2014)
10. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. Electronic notes in theoretical computer science 217, 5–21 (2008)

11. Enck, W., Octeau, D., McDaniel, P.D., Chaudhuri, S.: A study of android application security. In: USENIX security symposium. vol. 2, p. 2 (2011)
12. Goseva-Popstojanova, K., Perhinschi, A.: On the capability of static code analysis to detect security vulnerabilities. Information and Software Technology 68, 18–33 (2015)
13. Guo, C., Xu, J., Yang, H., Zeng, Y., Xing, S.: An automated testing approach for inter-application security in android. In: Proceedings of the 9th International Workshop on Automation of Software Test. pp. 8–14. AST 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2593501.2593503
14. Hofer, T.: Evaluating static source code analysis tools. Tech. rep. (2010)
15. Jiang, Y.Z.X., Xuxian, Z.: Detecting passive content leaks and pollution in android applications. In: Proceedings of the 20th Network and Distributed System Security Symposium (NDSS) (2013)
16. Kannavara, R.: Securing opensource code via static analysis. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 429–436 (April 2012)
17. Kratkiewicz, K., Lippmann, R.: Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. In: In Proc. 2005 Workshop on the Evaluation of Software Defect Detection Tools (BUGS'05) (2005)
18. Li, L., Bissyande, T.F.D.A., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Le Traon, Y.: Static analysis of android apps: A systematic literature review. Tech. rep., SnT (2016)
19. Martin, T., Hsiao, M., Ha, D., Krishnaswami, J.: Denial-of-service attacks on battery-powered mobile computers. In: Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on. pp. 309–318. IEEE (2004)
20. MITRE: Common Weakness Enumeration (CWE)—a community-developed list of software weakness type (2017), https://cwe.mitre.org/, visited on June, 14 2017
21. Mueller, B.: OWASP Mobile Application Security Verification Standard v0.9.3: Foreword. Tech. rep., OWASP – Open Web Applications Security Project (2017), https://github.com/OWASP/owasp-masvs/releases/download/0.9.3/OWASP_Mobile_AppSec_Verification_Standard_v0.9.3.pdf, visited on June, 12 2017
22. Okun, V., Delaitre, A., Black, P., SAMATE, N.: Static analysis tool exposition (sate) iv.[online], mar. 2012. See https://samate.nist.gov/SATE.html
23. OWASP: Mobile Top 10 2016 (2017), https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10
24. Payet, E., Spoto, F.: Static analysis of android programs. In: Proceedings of the 23rd International Conference on Automated Deduction. pp. 439–445. CADE'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2032266.2032299
25. Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for java. In: Proceedings of the 15th International Symposium on Software Reliability Engineering. pp. 245–256. ISSRE '04, IEEE Computer Society, Washington, DC, USA (2004), http://dx.doi.org/10.1109/ISSRE.2004.1
26. Sadeghi, A., Bagheri, H., Garcia, J., Malek, S.: A taxonomy and qualitative comparison of program analysis techniques for security assessment of android apps. IEEE Transaction on Software Engineering (2017), to appear
27. Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. SIGSOFT Softw. Eng. Notes 29(6), 97–106 (Oct 2004), http://doi.acm.org/10.1145/1041685.1029911