Meizodon: Security Benchmarking Framework for Static Android Malware Detectors

Sebastiaan Alvarez Rodriguez
Leiden University
s.f.alvarez.rodriguez@umail.leidenuniv.nl

Erik van der Kouwe
Leiden University
e.van.der.kouwe@liacs.leidenuniv.nl

ABSTRACT

Many Android applications are uploaded to app stores every day. A relatively small fraction of these applications, or apps, is malware. Several research teams developed tools which automate malware detection for apps, to keep up with the never-ending stream of uploaded apks (Android PacKages). Every tool seemed better than the last, some even claiming accuracy scores well over 90%. However, all of these designs were tested against test sets containing only self-written apks, synthetic malicious apks, or otherwise statistically unsound samples. Many of these tools are open source. We propose Meizodon, a novel framework to install Android static security analysis tools and run them efficiently in a distributed fashion, in equal environments and against a suitable dataset. This allows us to make a fair and statistically sound comparison of the most recent and best known tools, on real, 'practical' malware: malware created by malware creators, not by researchers, and found in the wild. From the results, we conclude that Android static security analysis tools do show great promise to classify apks in practice, but are not quite there yet. We demonstrate that Meizodon allows us to efficiently test analysis tools, and find that the accuracy of tested analysis tools is low (F1 scores are just over 58%), and analysis fails for many apks. Additionally, we investigate why accuracy is low, and why so many analyses result in errors.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

security, malware detection, android, static analysis

ACM Reference Format:
Sebastiaan Alvarez Rodriguez and Erik van der Kouwe. 2019. Meizodon: Security Benchmarking Framework for Static Android Malware Detectors. In *Central European Cybersecurity Conference (CECC 2019), November 14–15, 2019, Munich, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3360664.3360672

1 INTRODUCTION

Mobile phone technology and high-speed wireless communication techniques have rapidly developed in recent years. Many people use mobile phones for various purposes, such as accessing the web, performing financial transactions, and sending mails. Statistics [15, 16] show that about 75% of the estimated 4, 680, 000, 000 mobile phones are running a version of Google's Android OS. Moreover, many Internet of Things (IoT) devices of other types also run Android.

Phones often contain abundant sensitive information, for example about their owner's social life, as well as banking details, contacts, e-mails, and location history. Security breaches in these devices yield a variety of very high-value data. The large popularity of Android attracts malware creators, who try to infect these devices in order to steal confidential information, ransom devices, or otherwise harm or abuse devices. To protect our devices, we need to be able to determine which applications are benign and which are malware.

There are many different tools available to detect malicious Android applications [3, 5, 7, 9, 10, 18, 19]. While each paper contains an evaluation to determine whether the tool is effective, those numbers are hard to compare whenever there are even small differences in the methodology. Finding out which approach performs best is critical to maximize Android security, to minimize the amount of (CPU) power needed to do so, and to decide which designs are the best basis for further research. To do so meaningfully requires an independent and realistic test that can be applied to all systems. So far, there have been very few [11, 13] papers about comparing these tools. Unfortunately, these papers do not focus on **practical** results, statistics, and usability. To address these issues and make Android malware analysis comparable, we built a framework that supports the most prominent and recent Android static security analysis tools, and we use this framework to compare the tools' outputs and statistics.

Contributions. The main contributions of this work are:

- A novel approach to realistically measure the effectiveness of Android static security analysis methods;
- A framework that implements this approach, which has been made available open source to facilitate future research in Android static security analysis; and
- An independent overview of each tool's performance in terms of accuracy and speed on a *real-world*, practical dataset.

2 BACKGROUND

In this section, we provide necessary background information to better understand what kinds of Android security analysis tools exist, how Android static security analysis tools work in general, which tools we chose to implement for Meizodon, and how the tools are related.

CECC 2019, November 14–15, 2019, Munich, Germany

ACM ISBN 978-1-4503-7296-1/19/11…$15.00
https://doi.org/10.1145/3360664.3360672

2.1 Android Malware Analysis

Many different ways to counter the increasing amount of malware have been proposed. These approaches can be categorized as static analysis, dynamic analysis, or a hybrid between the two.

Static analysis focuses on the Android Package. It does not execute the program, but rather analyzes the application's code, manifest files, and API calls. Typically, methods in this analysis category are relatively fast on apps with a small amount of code, and slower on apps with more code. Static analysis tools are less effective when app behavior is not evident from the code. Examples where static analysis often fails include obfuscated malware, external payloads, and inter-process communication, as shown by Rastogi et al. [14].

Dynamic analysis focuses on run-time behavior. The app is installed and executed in a safe and monitored environment to see if any suspicious behavior occurs. Dynamic analysis performs the same simulations, regardless of the amount of code in the app. Malicious apps are getting smarter, and check whether they are being analyzed with dynamic analysis, as shown by Petsas et al. [12].

2.2 Static Malware Analysis in Android

Many malicious Android apps aim to take some information from a source (e.g. a contacts list) to a sink (e.g. an *HTTPRequest*). This behavior is not always obvious, as there can be a long, obfuscated data flow path in between. Tracking information from sources to sinks is called *taint analysis*. Taint analysis includes four types of analysis: (1) analysis within each app component; (2) Inter-Component Communication (ICC) analysis between components; (3) Inter-App Communication (IAC) analysis between apps; and (4) native analysis [1], which tracks communication through binary (Java Native) shared objects, originally written in non-Java languages, such as C or C++.

Android static security analysis tools need to be able to follow some basic patterns and understand some specific behavior patterns. First of all, a tool needs to have different kinds of awarenesses, or sensitivities, in order to successfully analyze Android code. Some degree of object, field and context awareness are useful, as well as flow and path awareness, to maintain order of appearance of statements through the program. Object awareness is used to be able to follow taint paths through specific object instances. Field awareness keeps track of taint paths through specific objects through abstract containers. Context awareness takes caller contexts into account when performing method calls. Flow awareness [6] takes calling order into account when analyzing. Lastly, path awareness remembers branch conditions for variables when propagating through branching paths in code.

A static security analysis tool for Android also needs to be able to handle Java-specific constructs correctly, such as Java reflection [8], methods, and dynamic error handling techniques. Additionally, such tools need to be able to handle Android-specific constructs correctly, such as implicit and explicit intent communication, broadcasters, and shared context services.

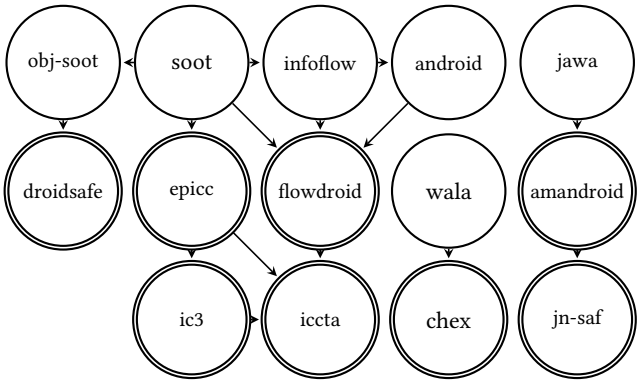
2.3 Our Selection of Tools

For this work, we selected different kinds of static malware analysis tools, to properly test the robustness of our framework and determine whether it is able to handle most tools. We incorporated Amandroid [19], DroidSafe [5], IccTa with IC3 [9], and JN-SAF [18] into our framework to run comparable analyses with. Although we could have chosen to implement support for any tool, we chose these tools because they generally have a high amount of citations, as shown in Table 1. JN-SAF has been included despite fewer citations, as it is newer and has support for precise native flow tracking. Moreover, it is made by the same research group as Amandroid, allowing a comparison between the 'old' (2015) vs new (end 2018) tools, native flow approximations/ignorance vs precise native flow awareness, and we could check if JN-SAF provided better accuracy and/or performance than its predecessor. The framework also provides alpha support for IccTa with Epicc [10] and FlowDroid [3]. Most of these tools are related to each other, as is shown in Figure 1.

Tool	Amandroid	DroidSafe	IccTa+IC3	JN-SAF
Citations	291	270	338	3
Year	2014	2015	2015	2018

Citations reported by Google Scholar, visited on 2019-05-22.

Table 1: Android Static Security Analysis tools, citations, and publication years



Doubly circled are all static security analysis methods. $A \rightarrow B$ means 'A used by B'

Figure 1: Android Static Security Analysis family

3 RELATED WORK

Previous work to compare static malware analysis tools has focused mostly on relative tool accuracy. They used mainly *synthetic malware apps* to make comparisons. Apps from this class of malware have been created by researchers and contain minimal examples of malicious behavioural patterns. Our work uses only *real-world malware apps* to acquire practical output statistics. This class of apps is produced by malware creators, with real, harmful intentions.

BREW [11] runs a set of analyzers on a shared dataset, much like Meizodon does, but it does not provide any means to actually install the analysis tool. They tested on existing benchmarks known to contain only *synthetic malware*. The tests they performed on a dataset of real-world malware were too small to make statistically sound claims regarding practical accuracy and usability. Our work

measures real-life usability in addition to real-world detection rates on a much larger dataset.

Qui et al. [13] also analyzed malware analysis tools. They compared the tools by running each of them on a dataset, which was built by combining each tool's provided example dataset, written by the tool's authors. They inspected each unexpected result (which is laudable, as this generally costs quite some time and effort). However, they risk overlap between the training set and testing set, because they evaluated the tools on a dataset written by tool's authors. This is a major threat to the validity of the comparison. Moreover, they tested on *synthetic* rather than *real-world malware*. In contrast, our work uses independent, real-world apps, resulting in better validity and representativeness of the comparison.

Our work distinguishes itself from previous work by performing experiments on real-world malware, generating practical rather than theoretical results. Also, this work gives more detailed results than previous work, including comprehensive information about analysis errors and analysis speed for each tool.

4 OVERVIEW

Meizodon is designed as a no requirement, easy to setup framework, which handles installation, execution, and result analysis of Android static security analysis tools. The framework automatically handles resource regulations and halts running tools when their respective times run out.

In order to test a tool, our framework goes through the steps described below. Meizodon first installs the tool, also handling installation requirements. It then does a first dynamic code call to perform installer setup. Next, the tool is executed on a dataset, with user-specified parameters. During execution, dynamic calls are made to execute tool instances over multiple cores. After executing an instance, potentially interesting parameters are stored in a .csv file. When all instances have finished, Meizodon is able to analyze the set of execution results through a final dynamic call to analysis code. A comprehensive .csv file is generated, which can be analyzed further by statistics programs. We describe the individual components in greater detail in Section 5.

5 DESIGN&IMPLEMENTATION

In this section we describe how the Meizodon framework works. Within this framework, we implemented support for four open-source tools: Amandroid [19], DroidSafe [5], IccTa with IC3 [9] and JN-SAF [18]. Others can be added with little effort, as described in Section 5.5.

5.1 Configuring the Tools

Our aim is to determine the practical usability of tools. Therefore, we use the default options for each tool where possible. If a tool had more precise or newer features, we activated these features. While setting configurations, we discovered that some tools, such as JN-SAF, have some faulty behavior for certain combinations of configuration settings and apks. We notified respective tool authors of these findings, and changed configuration options to working alternatives for the experiments.

5.2 Selection of Samples

For our experiments, we needed a test set containing real malware, preferably one containing no samples which have been used to develop any tool we test. By using a test set which was used during development for some tool, we would get a bias for this tool. To tackle this problem, we created a tool to randomly select apks from the AndroZoo [2] Android application data set. We took 96 apks from AndroZoo, as were available at week 19 of 2019. Our test set includes 48 benign and 48 malicious apks, both groups being a random sample representative of the population. AndroZoo is a data set containing millions of malicious and benign apks. AndroZoo determines whether apks are malware by running them through a platform, VirusTotal, against a list of known dynamic antivirus applications (none of which is among the tools we test). It counts the number of antivirus applications flagging a given apk and stores this number, as well as the scan date.

For our benign data set, we specified that at most zero antivirus applications may register our apks as malware. For the malicious data set, we enforced that at least eight different antivirus applications registered our apks as malware. This way, we have a high chance of only including malware, while keeping our dataset to draw random samples from large enough, to ensure there is a negligible chance we take a sample which was used for testing of one of our tested tools.

Ideally, one would prefer to have a ground truth available, stating with certainty whether each app is benign or malicious. When experimenting with real-world malware, however, it is not possible to know for sure. For a very small number of apps, one could manually inspect the apps to make an approximation of a ground truth, given enough time. Such manual work is not viable for data sets that are sufficiently large to draw statistical conclusions, and even with manual inspection it is possible to overlook malicious behavior. Automatically constructing ground truth approximations with VirusTotal, as explained above, is a reasonable alternative. While a single analysis tool could give false detections for a given app, we only use apps where multiple scanners are in agreement, and the scanners are independent of the static analysis tools we test. This way, the chance of incorrectly classified apps being in the test set decreases considerably. It should be noted that, even though the chance of mistakes is small, it could happen that, for some apps, many VirusTotal scanners are wrong. This would mean the results for those particular apps are misclassified, and the quality of the static analysis over- or underestimated. Even in the worst case, however, the difference with the correct value would be no larger than the fraction of apps misclassified. Moreover, in our evaluation we manually investigated the main sources of misclassification, and found no cases where VirusTotal was incorrect.

5.3 Timeouts

During early testing, we noted that some tools, especially JN-SAF, tend to loop forever on certain apks, filling up available RAM and only stopping when physical RAM limits have been exceeded. Also, we noticed multiple severe outliers in execution times on some analyses for this tool. We let JN-SAF run on a random test set without setting upper limits, and we provided 312 GB RAM. The results are shown in Figure 2. Note that the greatest outlier took

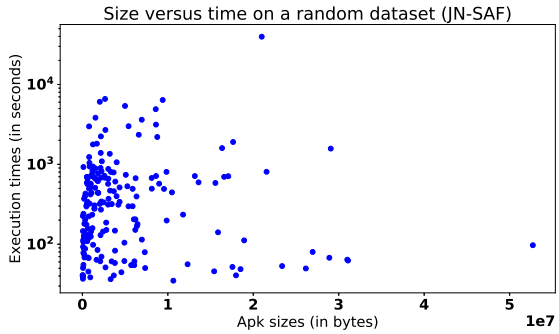


Figure 2: Unbound run with JN-SAF

about 40,000 seconds, or 11 hours, to finish with an error, because it consumed all 312 GB RAM. Also, note that about 4% of the apks in this test consume 45.7% of the total run time. For this reason, we consider a timeout to be justifiable, and we added a timeout system in our framework.

5.4 Result Analysis

Evaluating results by hand would be a very inefficient and time consuming task. Therefore, we implemented automated result analysis in our framework and for all tested tools. We studied the output files for each tool and use regular expressions to match conclusions (benign or malicious), errors, and warnings. Our framework gathers the matches and uses these to generate reports that are consistent between different tools.

5.5 Effort to Support an Analysis Tool

Our framework has been designed specifically to make it easy to add a new static analysis tool. To achieve this, one needs to: (1) provide a configuration file with download links for all dependencies (if any); (2) provide a configuration file with download links for the tool and all libraries; (3) write a few lines of Python code if special actions need to be performed after downloading (e.g. execute an installer after downloading it); (4) write some Python code to perform an execution call to the tool; and (5) write some Python code to extract errors, warnings, and the classification from the output file(s). Integrating all four tools we used took a total of 110 minutes (an average of 28 minutes per tool), with IccTa+IC3 taking most time (50 minutes). How much time is needed for a specific tool mostly depends on on how much external input the tool in question needs, and what it requires from its environment.

6 EVALUATION

In order to evaluate our framework, we implemented support for the tools Amandroid [19], DroidSafe [5], IccTa-IC3[9], and JN-SAF [18]. We analyzed 96 apks for our experiment, of which 48 are benign and 48 malicious, for each of our 4 tested tools. Benign and malicious apks were randomly selected from AndroZoo [2]. The number of virus detection hits for our malicious test set, as reported by AndroZoo, can be found in Figure 3. Additionally, we tested our framework on apks from the AMD [17] data set. However, we did not use this data set for our analysis, as it is maintained by the same research group that created Amandroid and JN-SAF. Using this data

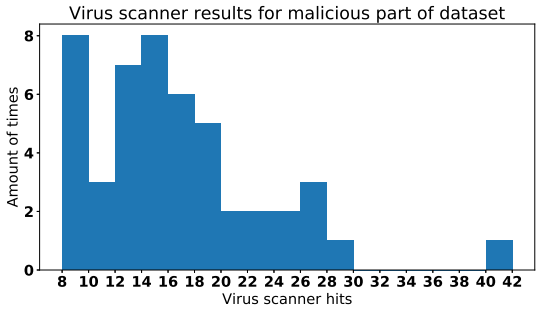


Figure 3: Virus detection hits for malicious apks reported by AndroZoo

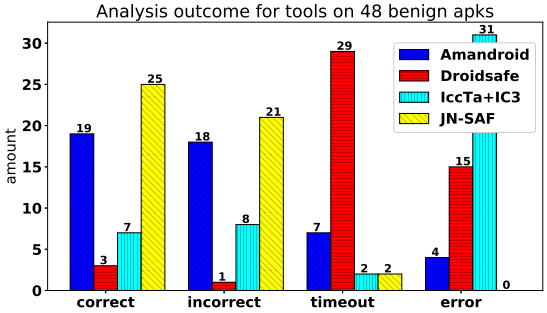


Figure 4: Analysis outcome for benign apks

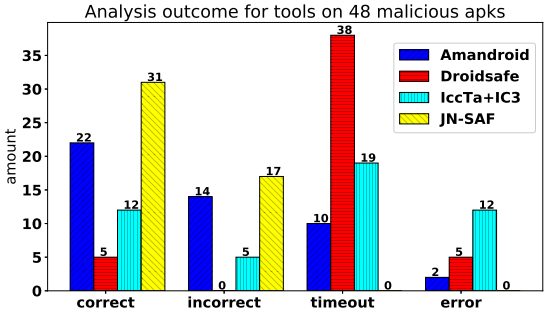


Figure 5: Analysis outcome for tools on 48 malicious apks

set would introduce a risk of bias, if the creators of Amandroid and JN-SAF used their data set as a training set for their tools.

We used the DAS-4 compute cluster [4] to run the 384 apk analyses in parallel. During our experiment, we gave every tool analysis one Intel Xeon E5620 CPU (16 cores at 2.40GHz) and 40 GB RAM. Furthermore, each analysis had a timeout of 2 hours. For every tool, we left all configurations and options on the default as much as possible, such that we are able to run the tools and as many available hardware resources were used, within their respective bounds.

6.1 Classification Accuracy of the Tools on a Practical Data Set

The precision, recall and F1 score of tested tools are shown in Table 2. We define precision as true positives divided by total positives, recall

Name	Precision	Recall	F1 score
Amandroid	0.61	0.55	0.58
Droidsafe	1.00	0.83	0.91
Iccta+IC3	0.71	0.60	0.65
JN-SAF	0.65	0.60	0.62

Table 2: Classification accuracy for all tested tools

as true positives divided by the total number of malicious apps, and the F1 score as $2\frac{precision \cdot recall}{precision+recall}$.

Figure 4 shows classification results for benign apps. Amandroid and JN-SAF produce correct results as often as incorrect ones. After investigating this remarkable behavior, we discovered that about 60% of all incorrect malicious classifications were caused by log calls in input apks. Log calls output given string data to several log channels. These calls could be used to communicate between apps. One app could leak information to a log channel, while another channel scans it and sends it to external servers. However, an application may only scan its own logs. While logging cannot be used for malicious purposes, release-version Android apps should not have any log calls within them. Within the other 40%, writing data to SharedPreferences (SPs) and sending PendingIntents (PIs) were often considered malicious. SPs are used to store user preferences for the app that writes them. Which apps can read stored preferences of some app depends on the write parameters of the writing app. PIs are used to request other apps to handle certain actions, such as taking a picture. Most of these methods are related to storing data or uniquely identifying a device. More than half of the incorrect malicious classifications could be removed by not considering log calls as sinks. Even more could be removed if they would not consider PI and SP write calls as malicious. Finally, we found that HttpUrlConnections were relatively often the source of incorrect taint paths. If analysis tools were to stop seeing incoming HttpUrlConnections as malicious, many incorrect taint paths would disappear.

We found that DroidSafe is very slow, having a timeout in more than half our tests. Sadly, this means we cannot make any statistical claims regarding classification accuracy for this tool. We will look closer at Droidsafe's timeouts in the next section.

Finally, IccTa+IC3 has a very large error rate. There is not enough output of IccTa+IC3 to make any claims about classification accuracy for this tool.

Classification results for malicious apps are shown in Figure 5. We see better results for Amandroid and JN-SAF, although the number of timeouts in Amandroid is slightly higher. DroidSafe has fewer errors on the malicious part of our data set than on the benign apks. However, it has a large number of timeouts. When DroidSafe does not crash and does not run for too long, the accuracy is 87.5 percent. However, the number of correctly classified apks is too small to make any claims about accuracy and real-life usability, when comparing correctly classified instances with the number of problems (timeouts and errors) for this tool. IccTa+IC3 has fewer errors, but most of these errors were replaced by timeouts.

We performed a χ^2 test to determine whether there is a significant difference in classification accuracy between Amandroid and JN-SAF, the only tools that succeed on a large number of samples. We find a χ^2 value of 0.20 with one degree of freedom, which yields

Benign	Amandroid	DroidSafe	IccTa+IC3	JN-SAF
mean	2,136	4,704	2,224	1,450
std. dev.	2,654	3,228	2,542	2,184
Malicious	**Amandroid**	**DroidSafe**	**IccTa+IC3**	**JN-SAF**
mean	3,124	6,115	3,839	711
std. dev.	2,676	2,260	3,332	1,010

Table 3: Analysis times (sec) for tools on 96 apks

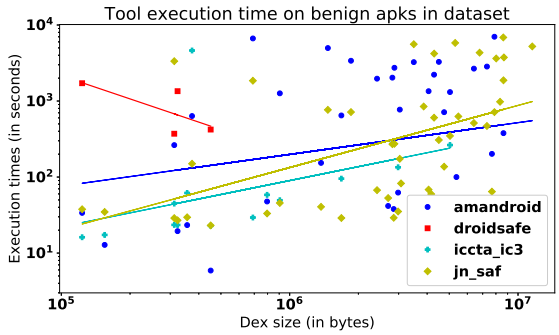


Figure 6: Dex size versus time on benign dataset

$p = 0.65$. There is clearly no significant difference in classification accuracy on our practical dataset.

6.2 Performance of the Tools on our Practical Data Set

During our experiment, we also reviewed analysis time. One can see these results in Table 3. Execution times are measured with Python's *time* module.

We find large differences in the total analysis time required for benign apks. JN-SAF is the fastest, followed by Amandroid, IccTa+IC3, and finally DroidSafe. Amandroid and IccTa+IC3 are on a similar level, while DroidSafe is extremely slow. We believe DroidSafe is an order of magnitude slower than our other tested tools, since almost all individual analysis times of DroidSafe seem much more time consuming, when comparing to others. This would explain the many timeouts and a very high total and average analysis time.

For the malicious apks in Table 3, DroidSafe and IccTa+IC3 take considerably longer to analyze this part of our data set. Increased analysis execution times of these tools is explained by the much increased timeout share for both IccTa+IC3 and DroidSafe, when comparing Figure 4 with Figure 5. Amandroid is also notable for increased analysis times. JN-SAF is doing even better on this set of apks when compared to its performance on the benign apk data set, being more than twice as fast on the malicious apk part of our data set.

JN-SAF is clearly the fastest of the tested tools, having relatively very low analysis execution times for both the benign and the malicious apks in our data set.

We plotted dex size (compiled code size) of apks against analysis time, to check whether there exists a strong correlation between these variables. This is not the case, as shown in Figure 6. Explained variance (R^2) ranges from 4.5% (Amandroid) to 51.7% (Droidsafe). Although small dex sizes generally take less time to be analyzed, there is no strong correlation with performance for larger dex sizes. This is expected, as code analysis times also depend on factors such

benign	Amandroid	Droidsafe	IccTa+IC3	unique
Amandroid	-	3 (75%)	4 (100%)	0 (0%)
Droidsafe	3 (20%)	-	11 (73%)	4 (27%)
IccTa+IC3	4 (13%)	11 (35%)	-	19 (61%)
malicious	**Amandroid**	**Droidsafe**	**IccTa+IC3**	**unique**
Amandroid	-	0 (0%)	0 (0%)	2 (100%)
Droidsafe	0 (0%)	-	1 (20%)	4 (80%)
IccTa+IC3	0 (0%)	1 (8%)	-	11 (92%)

Table 4: Shared errors for benign dataset

Error cause/Tool	Amandroid	Droidsafe	IccTa+IC3
Memory	0	4	25
API not found	0	10	1
support lib not found	3	0	0
Race conditions	0	0	4
decompile error	2	3	1
class annotations	0	0	9
other	0	3	3

Table 5: Error causes for tested tools on test set

as the number of global variables, object creations, source accesses, and inter component communications. These create exponentially larger analysis graphs, given that these code properties are part of possible taint paths. As all paths from sources to sinks need to be traversed, these factors result in much larger impact on analysis times than a large amount of code could do.

6.3 Errors

The analysis results show large numbers of errors. To further investigate this, we determined recurring error occurrence statistics of apks. The results are shown in Table 4. Although some apks produce errors on multiple tools, JN-SAF always succeeds, showing us that all apks can be analyzed without fatal errors. There is no strong connection between errors.

Next, we manually checked the output logs for each error. Results are found in Table 5. We conclude IccTa+IC3 often fails with *OutOfMemoryException*, because it consumes exceptionally high amounts of memory compared to other tested tools. It is also likely to fail if class annotations are in one of the taint paths of analyzed apk. Class annotations are used to give compiler directives for or provide metadata information to classes they annotate.

For Droidsafe, we found out that there were decompile problems in some cases, because Droidsafe was unable to find all classes given in the manifest of analyzed apks. However, most of Droidsafe's errors are related to analyzing code parts of a standard type Android API. Examples of APIs of this type are the Android Camera v2 API, Firebase AI API, gms Drive API, and Media API. Droidsafe was unable to analyze certain parts of these APIs, such as, in many cases, the gms Drive's DataBufferAdapter widget.

Amandroid seems to fail to find Android support v4 implementation classes, even though analyzed apks specify target android SDK versions which contain the support v4 library and Amandroid has access to these SDK versions.

6.4 Timeouts

There is large number of timeouts, coming mostly from Droidsafe. After looking into the reasons for this, we firstly conclude that Droidsafe's multithreading does not work well. Its core usage spikes every few seconds, after which it drops to one or two cores. Moreover, the way Droidsafe analyzes apks is inefficient. It uses separate programs to decompile apks and analyze decompilation result, and is more disk-intensive than the other tested tools. Another important factor for timeouts is infinite recursion. Sadly, some, if not all tools seem to loop infinitely on certain input apks.

The number of timeouts could be lowered by giving each analysis more time, but even when including all outliers, some analyses will still get timeouts due to infinite recursion, or get *OutOfMemoryExceptions*. Furthermore, we would have to set timeouts very high, drastically lowering throughput. This is shown by Figure 2. Note that the greatest outlier took about 40, 000 seconds, or 11 hours, to finish with an error, because it consumed all available RAM.

7 CONCLUSION

We built a framework to test Android static security analysis tools in a way that can compare their effectiveness and efficiency on a realistic data set. We used it to perform 384 analyses, and we conclude that the versions of tested systems do not perform as well on real world apps as they do on the theoretical test sets often used to evaluate them in the original papers.

Mainly Amandroid and JN-SAF show promise in classifying real world apps. However, their prediction accuracy should be increased before these tools could be used reasonably, by ignoring log calls and outgoing Pending Intents for example.

Although great progress has been made in the field of Android static security analysis research, this paper shows we are not there yet, when reviewing practical purposes.

It is good practice to independently compare analysis tools with each other, in order to find out each one's relative value. It is also good practice to compare tools with malware detection purposes on *practical*, real life applications, to see what tools can really do to protect the billions of Android devices on this world.

8 AVAILABILITY

Our framework, Meizodon, is available open source at https://github.com/Sebastiaan-Alvarez-Rodriguez/Meizodon. This repository includes a list of all apks in our dataset and analysis results of each apk for every tool.

REFERENCES

[1] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65:230–246, 2017.

[2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[4] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[5] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[6] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[7] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[8] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 756–761. IEEE, 2016.

[9] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. Composite constant propagation and its application to android program analysis. *IEEE Transactions on Software Engineering*, 42(11):999–1014, 2016.

[10] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 543–558, 2013.

[11] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341. ACM, 2018.

[12] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[13] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186. ACM, 2018.

[14] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[15] Statcounter. Mobile operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed:2019-02-08.

[16] Statista. Number of mobile phone users worldwide from 2015 to 2020 (in billions). https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/. Accessed:2019-02-08.

[17] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*, pages 252–276, Bonn, Germany, 2017. Springer.

[18] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150. ACM, 2018.

[19] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.