Universiteit Leiden
Opleiding Informatica

Static Security Analysis Methods in Android Systems: a Comparison

Name: Sebastiaan Alvarez Rodriguez

Date: 28/06/2019

1st supervisor: Dr. Erik van der Kouwe
2nd supervisor: Dr. Todor Stefanov

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

1 Abstract

Many Android applications are uploaded to app stores every day. A relatively small part of these applications, or apps, is malware. Several research teams developed tools to make automatic malware detection for apps possible, 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, or synthetic malicious apks, or otherwise statistically unsound samples. Many of these tools are open source. We propose a novel framework, named Paellego, to install Android static security analysis tools, and run them efficiently distributed, in equal environments, against a suitable dataset. This allows us to make a fair and statistically sound comparison of the best known and recent tools, on real 'practical' malware: Malware apps created not by researchers, but by actual malware creators, which were distributed by app stores. From the results, we conclude that Android static security analysis tools do show great promise to classify apks, but are not quite there yet.

Contents

2 Introduction

Development of mobile phone technology and high-speed wireless communication techniques have been going on for many years now. The products of the mobile phone industry have spread like wildfire, and are used by many people as a device to access the internet with, and perform financial transactions, send mails, etcetera. Statistics [1, 2] show that about 75% of estimated $4,680,000,000$ mobile phones are running a version of Google's Android OS. Then, we do not even take all existing Internet of Things (IoT) devices into account, for which Google has also produced versions of the widely used OS.
Phones of many people contain their owner's complete social life, as well as banking details, contacts and emails, location history, home address and work address, etcetera. Many people use their phone to perform financial transactions, browse the internet, read their mail, send messages and so forth. Security breaches in these devices yield a variety of very high-value data. Protection of these devices is crucial, to prevent enormous problems in terms of privacy, identity theft and data theft. The large popularity of Android attracts malware creators, trying to infect these devices, in order to steal confidential information, ransom devices or otherwise harm or abuse devices.
The Android operating system is focused on apps, like many other mobile operating systems. People and organisations which want to harm Android devices often, if not always, create apps to infect devices.
Developers release $6,140$ new apps [3] to the Google Play Store on an average day, or 4.26 new apps every minute, or 1 new app every 14 seconds. In order to analyse this large number of apps without building an ever growing wait queue over time, and without flagging benign apps as malicious or vice-versa, analysis has to be done with very high speed and accuracy, and has to be cheap, preferably.
There are many different open-source tools available for performing Android static security analysis [4, 5, 6, 7, 8, 9, 10]. Since the role of protecting Android devices is so important, an in-depth comparison of these methods is vital to protect these devices from malware and the devices' owners from harm. There have been very few [11, 12] papers about comparing these tools. Existing papers make unsound statistical claims, or do not compare tools on test sets consisting of actual malware, created by malware creators, with truly malicious intentions (*practical* malware). Instead, these papers use test sets consisting of malware created by researchers (*theoretical* malware), and therefore give no insights in practical usability, statistics and performance.
In this paper, we aim to answer the following research questions:

RQ1 What is the classification accuracy of recent, well known Android static security analysis tools on a *practical* dataset?

RQ2 What is the performance of recent, well known Android static security analysis tools on a *practical* dataset?

We created a framework, named Paellego, to help us find answers to the research questions, by running the most prominent and recent Android static security analysis tools to compare the tools' outputs and statistics on actual, practical malware.

Contributions The main contributions of this work are:

- A new framework to facilitate future research in Android static security analysis methods, and
- An independent overview of each method's performance in terms of accuracy and speed on a *real-world*, practical dataset.

3 Background

In this section, we provide necessary background information to better understand what different kinds of Android security analysis tools exist, what they need to handle in order to function properly and how they work in general.

3.1 Android Malware Analysis

Many different methods to counter the increasing amount of malware have been proposed. These approaches can be categorized as static analysis, dynamic analysis, or a hybrid.
Static analysis focuses on the Android Package. It does not execute the program, but rather analyses the application's code, manifest files, and API calls. Typically, methods in this analysis category are relatively fast on apps with small code amount, and relatively slow on apps with very large amounts of code. Static analysis tools are less effective when app behavior is not evident from the code. Examples where static analysis often fails include advanced obfuscated malware, external payloads, and inter-process communication [13].
Dynamic analysis focuses on run-time behavior. The app is installed and executed in a monitored environment to see if any suspicious behavior occurs. Commonly, methods in this analysis category are relatively slow on apps with a small amount of code, and fast apps with a large amount of code. This is because these analysers perform behavior analysis with the same simulations, regardless of app code amount. Furthermore, dynamic analysers have a fundamental weakness: When their simulations are unable to trigger malicious behaviour through simulation, they fail. This results in huge accuracy problems when more advanced, or simulation-aware malware [14] is analysed.

3.2 Android Static Malware Analysis Requirements

Nearly all 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. The tracking of information from sources to sinks is called *taint analysis*.
Taint analysis extends to four types of analysis: The basic analysis, within each app component, Inter-Component Communication (ICC) analysis, analysing between components, and Inter-App Communication (IAC) analysis, for analysis between apps. A relatively new kind of analysis is Native analysis [15], which tries to keep track of all sources and sinks through binary 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 analyse 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 [16] takes calling order into account when analysing. 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 and Android-specific constructs correctly, such as Java reflection [17], and Android's implicit intent communication.

3.3 Android Static Malware Analysis General Functionality

In general, Android static malware analysis tools analyse apks by performing the following four steps on a given apk file.
The *first* step is to extract the apk file. These files contain all app code, resources, assets, certificates, and manifest files. Especially the manifest files need to be extracted, as they contain important metadata for analysis.
In the *second* step, tools read the extracted manifest files. Mainly, they read the required app permissions and a list of components and their Intent communication filters. The tools use the required app permissions to check later whether illegal resource accesses occur, such as writing to external storage without permission. Also, the first indications of maliciousness appear here, since certain legacy permission combinations occur almost solely in malware apps [18]. The tools store all component information from the manifest, as each component must be found in code and analysed later. Intent filters are important for Inter App Communication (IAC) analysis, since inter app communication is handled mostly through PendingIntents, which components of apps may receive (filter enabled) or ignore (filter disabled). For this reason, analysis tools performing IAC analysis store this information.
In the *third* step, static analysis tools generate a flow graph, which is a process involving multiple sub steps, depending on the implementation of the tool.
In the *first* sub step, static analysis tools analyse the individual components with basic analysis. The tools search for sources, or information accesses, such as reading a phone contact list, within a component.
In the *second* sub step, tools search sinks. A sink is a stream moving out of an app to some external object. A standard example of this is a HttpUrlConnection object, which is used in Java to establish connections to remote servers.
In the *third* sub step, the tools generate partial taint paths. Generation is initiated by marking, or tainting, all sources. These will be the initial taint nodes. Next, each program statement reading from a taint node is tainted. A basic example is `String lookup = contactslist[0]`, where variable `lookup` would be tainted. Whenever a function returns one or more taint nodes, the function itself is tainted. After generating all nodes, the tools generate each possible path through these taint nodes, which form all partial taint paths.
What happens after partial taint path generation is dependent on what analysis strategies are implemented in a given static analysis tool. If basic component analysis is implemented, each partial taint path within a component is expanded by following function calls within that component, until a sink, a call to another component, or a PendingIntent is found. If Inter-Component Communication (ICC) analysis is implemented, also calls to other components are followed to further generate taint nodes and eventually taint paths. If Inter App Communication (IAC) analysis is implemented, then a graph is constructed, which consists of all possible communication paths between multiple analysed apps. All existing taint paths leading to a PendingIntent, are then followed through this graph to the components of all apps able to receive this PendingIntent. Taint paths are then expanded by investigating these components. If Native analysis is implemented, taint paths leading to a shared binary object are expanded by applying essentially the same methods as with basic component analysis on the binary.

Depending on what awarenesses are implemented, generated taint paths may be annotated with e.g. branch path information.
In the *fourth* step, after generating a flow graph, all possible paths from a source to a sink are computed and stored or displayed.

4 Related Work

As with most research, this research is not the first in its kind. Others have looked into analysers and relations between them in terms of accuracy.
In the sections below, we will have two notions of apps:

- *theoretical app*: An app which has been written to test vulnerability detection in analysers, and which is not found in the real malware world. These apps are produced by scientists and commonly contain minimal examples of malicious behavioral patterns. Commonly, these apps do not perform malicious actions as a whole.
- *'real world' app*: An app found in the real world, which infects or has infected real devices with real, harmful intentions. This class of apps is produced by malware creators, and is this paper's primary interest.

4.1 Analysing Analysers

One group of researchers has made config files for a framework of BREW and AQL [11] to run a set of analysers on a shared dataset, much like Paellego does. However, they did not provide any means to actually install the analysers (or any useful information about installing). They also tested on existing benchmarks known to contain only theoretical apps, specifically made for testing analysers. Lastly, their framework appears to only consist of config files.
Finally, one research team also analysed analysers available at the time of their publication [12]. They compared each Android static security analysis tool in their test by running them on a dataset consisting of each implementation's test set written by the implementation's authors. They inspected each unexpected result (which is laudable, as this generally costs quite some time and effort). However, running implementations on a dataset written by implementation's authors, which was made purely for the sake of showing how well some implementation works, does not provide practical, 'real life' information and statistics. This team tested on theoretical apps instead of real world apps.

4.2 Dynamic Analysers

Dynamic analysers fall outside the scope of this research, but represent a different, interesting way to perform Android malware analysis. Android dynamic security analysis is another field of Android security analysis, which is commonly implemented by antivirus programs, because analysis times for analysing larger apps are generally shorter with dynamic analysis tools than with static analysis tools. However, this speed comes at a loss of classification accuracy. Examples of dynamic security analysis tools include DroidDolphin [19], CopperDroid [20] and DroidScope [21].

4.3 Proprietary Analysers

Proprietary Analysers fall outside the scope of this research, as they commonly do not provide very exact information about found threats and do not provide information about what analysis techniques are used. Most proprietary security analysis tools use dynamic analysis, because analysis times for analysing larger apps are generally shorter with dynamic analysis tools than with static analysis tools. Examples of proprietary malware analysis tools are Eset Mobile Security [22] and Avast Mobile Security [23].

4.4 Android malware datasets

Several testbenches of theoretical malware exist in order to test theoretical performance of Android static security analysis tools. DroidBench [10] and ICC-Bench [24] are examples of test benches containing theoretical apps. All of our tested tools used DroidBench to test their own performance.
There exist a few Android malware datasets with practical malware. The most important datasets of this kind are the AndroZoo [25] and the AMD dataset [26]. We used these datasets to build our framework and generate random test sets.

4.5 Other

VirusTotal [27] is an online service which scans uploaded files for malicious behaviours, by scanning uploaded files with many antivirus tools. Among others, apk files are supported on this platform by 61 proprietary antivirus tools. In fact, the proprietary analysers named in Section 4.3 are used by VirusTotal. AndroZoo uses VirusTotal: For each apk in the AndroZoo dataset, there is information available which indicates how many antivirus tools of VirusTotal flag a given apk as malicious.

4.6 Our Work

This work is different in the sense that it does not use theoretical apps, and therefore does not fall for theoretical usability statistics, but independently tests each Android static security analysis tool against a dataset of practical, real Android malware, with true malicious purposes, found in the dangerous corners of the internet. In this work, we provide true statistics over a test set of real world apps.

5 Overview

Paellego is designed as a framework with as few requirements as possible, 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 timers run out.
In order to test a tool, our framework needs to run through all of the following steps shown in Figure 1. At the start of it all is tool installation. Paellego handles installation requirements. After this, the framework performs a first dynamic code call to perform installer setup. Secondly, 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 execution of an instance, potentially interesting parameters are stored in a .csv file. When all instances have finished, Paellego is able to analyse the set of execution results through a final dynamic call to analyse code. A comprehensive .csv file is generated, which can be analysed further by researchers. We describe the individual components in greater detail in Section 6.

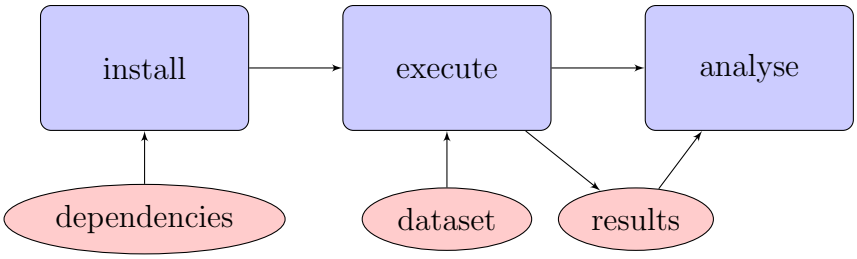


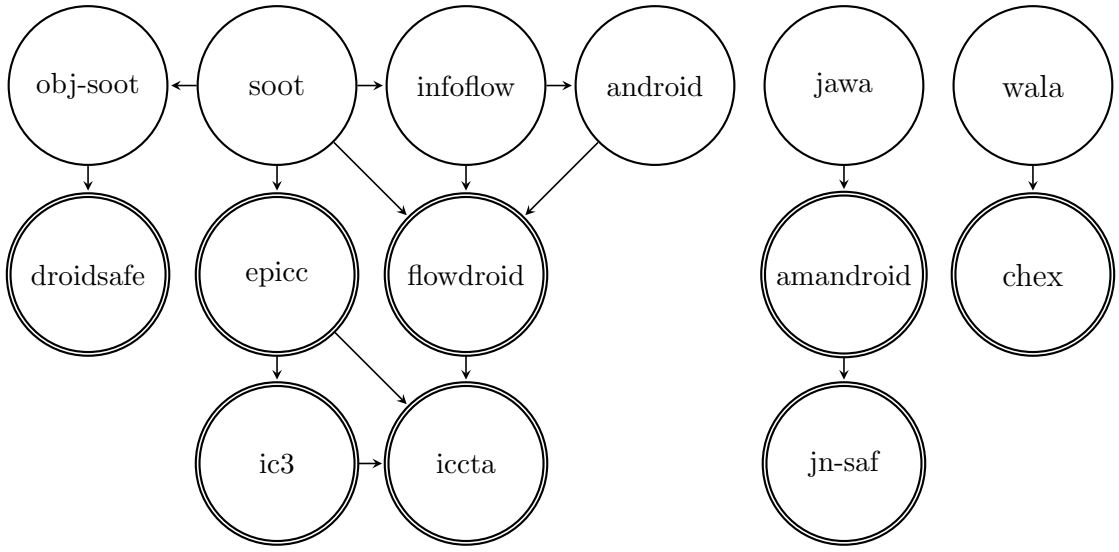
Figure 1: Standard flow of running a tool with Paellego

6 Design

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 [5], DroidSafe [6], IccTa with IC3 [8] and JN-SAF [4] into our framework to be able to perform our experiments with these tools. We will provide more detailed information about our experiments in Section 8. The framework also provides alpha support for IccTa with Epicc [7] and FlowDroid [10]. Most of these tools are related to each other, as shown in Figure 2.

For our framework, we implemented support for four open-source tools to experiment with: Amandroid [5], DroidSafe [6], IccTa with IC3 [8] and JN-SAF [4]. Although we could have chosen to implement support for any tool, we chose for these tools, because they generally have a high amount of citations, as shown in Figure 3. Only JN-SAF does not have so many, as it is relatively new. However, JN-SAF appeared to be interesting to include into the analysis, because it is new, has support for precise native flow tracking and is made by the same research group as Amandroid. This way, we could experiment with 'old' (2015) versus new (end 2018), native flow approximations/ignorance vs precise native flow awareness, and we could check if JN-SAF provided better accuracy and/or performance over its predecessor, Amandroid.

However, not everything went well: At first, we also wanted to implement support for FlowDroid, as FlowDroid is actively developed by the same authors as IccTa+IC3. In fact, FlowDroid contains the source code of IccTa+IC3 as part of its inner framework. We created installation code, and execution code, as with the other tools. Despite this fact, FlowDroid never seemed to be able to detect malicious apks as malware, where IccTa+IC3 did detect them. Contacting authors of FlowDroid (and IccTa+IC3) did not help. As time was running out and we could not find a reason for FlowDroid to refuse functioning as intended, we decided to discontinue working on FlowDroid.



Doubly circled are all static security analysis tools. Single circled are relevant dependencies of static security analysis tools. $A \rightarrow B$ means 'A used by B'

Figure 2: Android static security analysis family

Tool	Amandroid	DroidSafe	IccTa+IC3	JN-SAF
Citations	291	270	338	3
Publication	2014-11-03	2015-02-08	2015-05-16	2018-10-08

Dates follow yyyy-mm-dd. Citations are as counted by Google scholar, visited on 2019-05-22.

Figure 3: Android Static Security Analysis tools, citations, publication dates

6.1 Configuration Settings

As time was limited, we decided to not spend time on determining the optimal equality settings for configurations. We kept as much to default options as possible. After all, we are interested most in practical usability of tools. If a tool had more precise or newer features, we activated these features. During determining configuration settings, we discovered that some tools, such as JN-SAF, have infinite recursion problems under some settings, with certain apks as input. We notified the respective tool authors of these findings and changed configuration options to working alternatives for the experiments.

6.2 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. If we were to use a test set which was used by some tool, we would get a bias for this tool, resulting in unfair and statistically unsound results. To tackle this problem, we created a tool to randomly select apks from the AndroZoo [25] Android application dataset, containing approximately $9,000,000$ apks, at the time of visiting. For this experiment, we created one set containing only benign apks, and another, containing only malicious apks. AndroZoo has no ground truth. However, there is a parameter for each apk, representing how many different antivirus applications rate a given apk as malicious. This rating is extracted from VirusTotal [27], a platform to analyse (in this case) apks with 61 independent antivirus scanners. Luckily, none of our tested tools is included in the 61 (proprietary) scanners applied by VirusTotal. Thus, we do not have to fear a bias to one or more of our tested tools.
For our experiments, we consider an apk benign if none of the different antivirus applications flag a given apk as malware. We consider an apk malicious, if at least eight different antivirus scanners flag a given apk as malicious. This way, we have a high chance of only including malware, while keeping our dataset large enough to assure that there is a negligible chance we take a sample which was used for testing of one of our tested tools.
We randomly selected 48 benign apks, and 48 malicious apks, to have a statistically significant sample size.

6.3 Result Analysis

An execution of an Android static security analysis tool produces execution output for each apk it is executed on. The execution output of an apk contains a classification (malicious/benign) for that apk, or no classification at all because of a crash. For each executed tool, for each analysed apk, classification and error responses have to be found in execution output, because we are interested in classification accuracy for this research.

Evaluating the execution output by hand would be a very inefficient and time consuming task. Therefore, we implemented a method of result analysis in our framework for all tested Android static security analysis tools. This way, we can determine analysis outputs for all tools and for all apks, and represent the data in a uniform way. We studied where classification output and errors are located in the execution output of each tested tool. After executing a set of our tools on a given (set of) apks, we executed the result analysis on the execution output to gather classifications and errors of the used tools. Our results are based on this gathering strategy of our framework.
Most Android static analysis tools write Java stack traces to their standard outputs on error, while others catch these errors and produce a single error line. Every tool presents classification in a different way. If tools change their output mechanisms, then also the result analysis code of these tools in Paellego must change.

7 Implementation

Very early in the project, our framework consisted of only bash functions, to experiment with a few analysis implementations. This soon turned out to be a language suitable for running executables, but not very useful to implement all other features we had in mind, e.g. multi-processing. We migrated the whole project to Python3, because of its greater programming ease, equally good program-call subroutines, and its outstanding thread-safety, multithreading and multiprocessing subroutines. In this section, we will provide detailed information about our framework, its requirements, and encountered problems and solutions.

7.1 Requirements

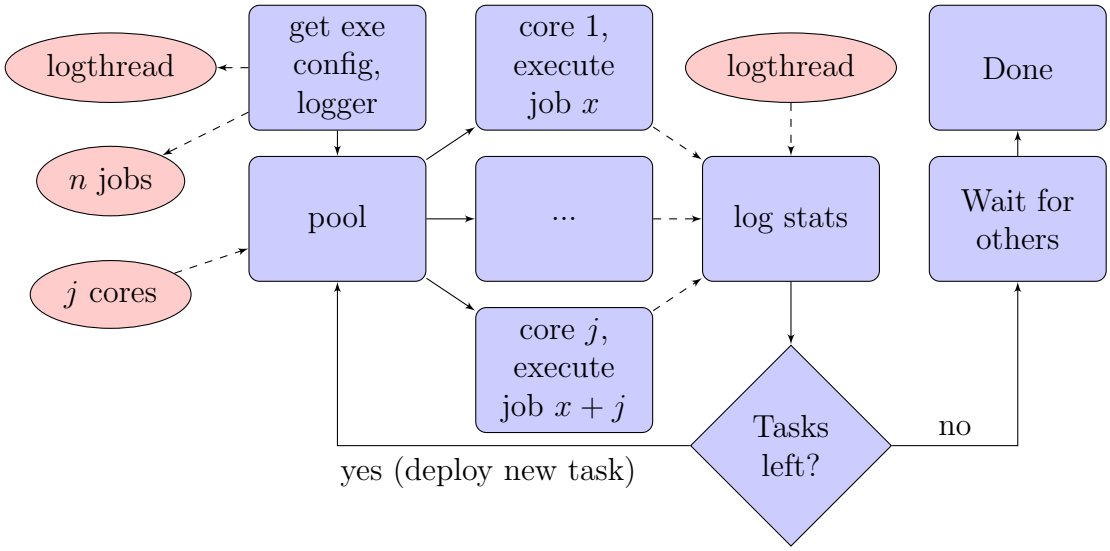
We wanted to create a light, powerful framework, to facilitate installation, execution, and result analysis of Android static security analysis tools. The framework needed to be easy to extend by researchers, in a way they can implement support for their own tools. The framework should be very easy to install, because this kind of multiprocessing oriented software is commonly run on cluster computers, which may not provide root access to any user. Lastly, the framework must be resource aware, as taint analysis is very time and RAM consuming. Paellego satisfies all of these requirements.

7.2 The Framework

In order to fulfill our requirements, we chose to use the language Python [28], because of its extensive documentation, general availability on different operating systems, and relatively good subprocess-call modules, which can execute programs that are not written in Python. This last ability of Python proved very useful, as some security analysis implementations required users to build the tool as part of the installation, and required users to install all kinds of dependencies, making it impossible to avoid non-Python execution calls. Python3 was chosen over Python2 for its additional functionalities, speed improvements, and to ensure compatibility in the future (since Python2 and older are legacy and are not forward compatible). Lastly, Python3 is often available on server computers, meaning our framework can be used directly.

Paellego dynamically imports code for each found tool to install it, to execute it, and to analyse results. A new Android static security analysis tool can be implemented for Paellego with ease, without requiring comprehensive knowledge of the framework. Only the dependency download locations have to be specified, as well as where downloaded artifacts have to be stored, what should be done before downloading, after downloading, before running, to run, after running, before analysing results, and to analyse results. A complete wiki is available, to show how the process of extending works exactly.

Most, if not all Android static security analysis tools use taint analysis to analyse, which is quite demanding in both time and hardware resources. To better utilize hardware and reduce required analysis time on systems with multiple cores, our framework supports *multiprocessing*. This way, we are able to run multiple analyses in parallel on a machine, to achieve higher performance on multi-core systems. This is shown in the high-level framework execution flowchart in Figure 4. In Figure 4 is shown, on an abstract level, how multiple analyses run in parallel. Here, j represents the amount of cores allocated by the user for execution. n represents the



A 'job' is a pair of one Android static security analysis tool and one apk to analyse

Figure 4: High-level Paellego tool execution flowchart

amount of jobs, where a job is a single apk analysis by one tool. Job x represents the job executed after $x - 1$ others.
We found out that some frameworks, especially IccTa-IC3 and JN-SAF, tend to hang on some apks until they crash because of out-of-memory errors. These facts required the framework to support timeouts, to assure analyses are done in reasonable time and to prevent infinite loops and thus infinite analyses.

7.3 Problems

During each development stage of Paellego, we encountered a set of problems. One of the biggest, most occurring problems we encountered was: Lack of *documentation* of analysis methods. It was extremely difficult to figure out how to install the tools exactly, what dependencies are needed, which versions of those dependencies, and how to satisfy all runtime conditions. Most researchers apparently did not believe comprehensive, understandable documentation was necessary for their tools. We had to figure out how to install and run every method ourselves. This took many hours, fixing errors until installation/running succeeded. Sometimes, we did have a few lines of clues from documentation, which proved generally useful, be it incomplete, in discovering what had to be done to install analysis method implementations.
The second biggest issue was: Non-uniform analysis output. Every researcher makes their implementation different. There is no convention on the analysis output for this kind of tools, sadly. In order to make the results comparable and presentable, we added an 'analyse' function to Paellego, which makes all specified results uniform to others.
A last issue was: Design mistakes of researchers. Researchers for some tools made quite some design mistakes in their code. Here is a subset of what was encountered: Hundreds of build warnings, path hardcodings, well-hidden configuration files, code working from only certain current working directories, calls in proprietary Java API's, and so on. This issue was solved by providing work-around solutions to get past these design mistakes, generating required configs,

accessing shell subroutines with correctly setup current working directory etcetera. Of course not every problem has been solved: Some problems lie outside the scope of this research, such as crashes of analysis tools on certain input apk files. These crashes are logged to execution output csvs. We investigate encountered crashes in Section 8.

8 Evaluation

In order to evaluate our framework, we implemented support for the tools Amandroid [5], DroidSafe [6], IccTa-IC3[8], and JN-SAF [4].
Furthermore, we tested our framework on apks from the AMD [26] dataset. We did not use this dataset for our analysis, as it is maintained by the same research group that created Amandroid and JN-SAF. Using the AMD dataset might introduce bias, as the creators of Amandroid and JN-SAF most likely used this dataset as a training set of their tools, and we should therefore not use it as validation set.

8.1 Setup

We analyse 96 apks for our experiment, of which 48 are benign and 48 malicious, for each of our 4 tested tools. Thus, we ran a total of 384 apk analyses in parallel, on the DAS-4 server cluster [29].
As discussed in Section 6, we use the AndroZoo dataset to generate a sample of size 96, with 48 benign and 48 malicious apks. AndroZoo provides information per apk about the amount of VirusTotal proprietary antivirus tools which flagged a given apk as malicious. For our malicious samples, we specified a given apk must be flagged by at least 8 of those antivirus tools as malicious. The actual amount, for all apks combined, is shown in Figure 5, where each bin contains the amount of apks which were flagged by VirusTotal antivirus programs for x times, with x within bin limits.
During our experiment, we gave every tool analysis exactly one Intel Xeon E5620 CPU (16 cores at 2.40GHz and 40 GB RAM). Furthermore, each analysis had a *timeout* of exactly 2 hours. For every tool, we left all configurations and options to default as much as possible, so we are able to run the tools and maximum available hardware resources were allocated for the tools, within their respective bounds.

8.2 Classification Accuracy of the Tools on our Practical dataset

We computed precision, recall, and F1 score ('F-measure') to give a quick overview of the classification accuracy results. The computed values are shown in Table 1. Figure 6 shows the distribution of benign apks of our test set in all possible categories (correct, incorrect,

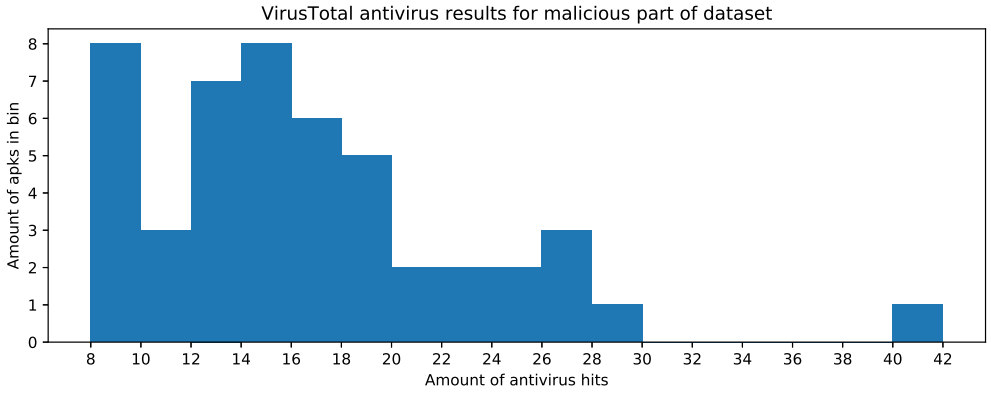


Figure 5: Virus detection hits for malicious dataset

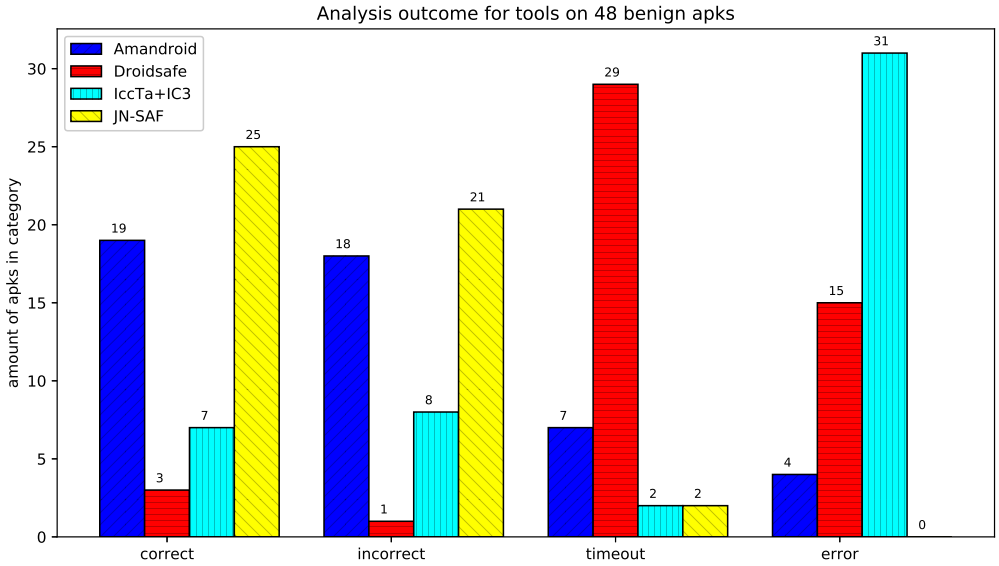


Figure 6: Analysis outcome for tools on 48 benign apks

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 1: Classification accuracy for all tested tools

timeout, error), for each tested Android static security analysis tool. Here, we see Amandroid and JN-SAF produce correct as often as incorrect results on average. After investigating this remarkable behaviour, we discovered that about 60% of all incorrect malicious classifications were caused by log calls in input apks, which output given string data to several log channels. Log calls could only 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. Despite the fact that logging can not 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 and sending PendingIntents were often considered malicious. SharedPreferences are used to store user preferences for the app that writes them. They can be read by only the writing app, or all apps, depending on code. PendingIntents are used to request other apps to carry out 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 manipulating the analysis tools such that they do not consider log calls as sinks. Even more could be removed if they would not consider PendingIntent and SharedPreferences write calls as malicious. A last thing we noticed: 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.

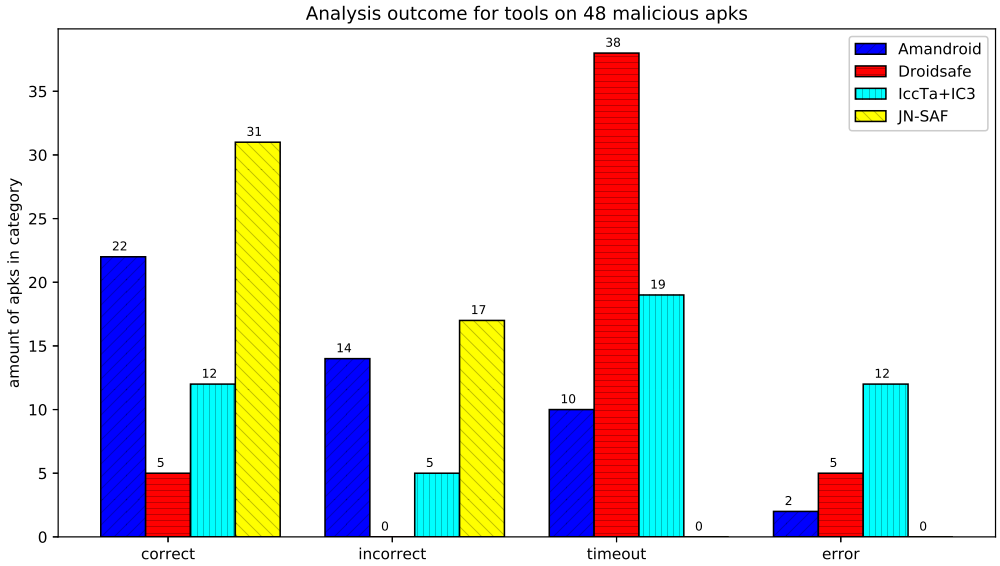


Figure 7: Analysis outcome for tools on 48 malicious apks

We believe we could have 80% less incorrect malicious classifications, if we would implement the suggested changes above.
In Figure 6, we also see that DroidSafe is very slow, having a timeout in more than 50% of our tests. However, this means we cannot make any statistical claims regarding the classification accuracy for this tool. We will look closer at Droidsafe's timeouts in the following sections.
Lastly, we see IccTa+IC3 has a very large error rate. There is not enough output of IccTa+IC3 to make any claims about the classification accuracy for this tool.
Figure 7 shows the distribution of malicious apks of our test set in all possible categories (correct, incorrect, timeout, error), for each tested Android static security analysis tool. Here, we see better results for Amandroid and JN-SAF. Both tools are more often correct, although the number of timeouts in Amandroid is slightly higher. DroidSafe has fewer errors on the malicious part of our dataset 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%. 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.
In order to check whether there exists dependence between classification and tools, we performed a chi-squared test, with $\alpha = 0.05$, on tools that gave a statistically significant amount of classification responses. The required information is in Table 2.
H_0: Tool and classification correctness are independent.
H_a: There exists a dependence between tool and classification correctness.

$$\chi^2 = \frac{(41-42.40)^2}{42.40} + \frac{(32-30.60)^2}{30.60} + \frac{(56-54.60)^2}{54.60} + \frac{(38-39.40)^2}{39.40} \approx 0.20$$

For $df = (2-1) \times (2-1) = 1$, the right-tail chi-squared distribution value probability $= 3.84$, for $df = 1$ and $\alpha = 0.05$. As our $\chi^2 < 3.84$, we conclude $P_{H_0} \geq \alpha$. There is no strong

Tool	Correct	Incorrect	total
Amandroid	41 (42.40)	32 (30.60)	73
JN-SAF	56 (54.60)	38 (39.40)	94
total	97	70	167

Table 2: Tool overall classification distributions. Information: distr. (expected distr.)

Benign	Amandroid	DroidSafe	IccTa+IC3	JN-SAF
mean	2136	4704	2224	1450
Std. dev.	2654	3228	2542	2184
Malicious	**Amandroid**	**DroidSafe**	**IccTa+IC3**	**JN-SAF**
mean	3124	6115	3839	711
Std. dev.	2676	2260	3332	1010

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

enough evidence to reject H_0. There is no statistically significant dependence between our currently tested analysis tools and classification correctness. This means there is no statistically significant overall accuracy for Amandroid and JN-SAF on our practical dataset.

8.3 Performance of the Tools on our Practical dataset

During our experiment, we also reviewed analysis time. One can see these results in Table 3. Execution times are measured with the Python *time* module, which produces a negligible error on measured times.

In Table 3, we see quite large differences in the average analysis times required per tool. JN-SAF is the fastest, followed by Amandroid, IccTa+IC3, and finally DroidSafe. Amandroid and IccTa+IC3 are on an equal level, while DroidSafe is extremely slow. When looking at the maximal analysis time (without considering timeouts), we see that the slowest classification response from DroidSafe was 1716 seconds. We believe DroidSafe is an order of magnitude slower than our other tested tools, because almost all individual analysis times of DroidSafe seem much higher, when compared to other tools. This would explain the many timeouts and a very high average analysis time.

When reviewing speed tests for the malicious apks in Table 3, one thing we immediately notice is: DroidSafe and IccTa+IC3 take considerably more time to analyse this part of our dataset than the benign part. Increased analysis execution times of these tools is explained by the much increased timeout share for both IccTa+IC3 and DroidSafe, when comparing Figure 6 with Figure 7. Amandroid is also conspicuous for increased analysis times. JN-SAF is even faster on this set of apks when compared to its performance on the benign apk dataset, being more than two times faster on the malicious apk part of our dataset. We can say with large certainty that JN-SAF is the fastest of our tested tools, having relatively very low analysis execution times for both the benign and the malicious apks in our dataset.

We plotted the dex size of the analysed apks, i.e., the size of compiled code translated to a Dalvik executable, against the required analysis time with all tools, to check whether there exists a strong correlation between these variables.

This is not the case, as shown in Figure 8, where we plotted dex size (bytes, x-axis) against analysis time (seconds, y-axis). Although small dex sizes generally take less time to be analysed,

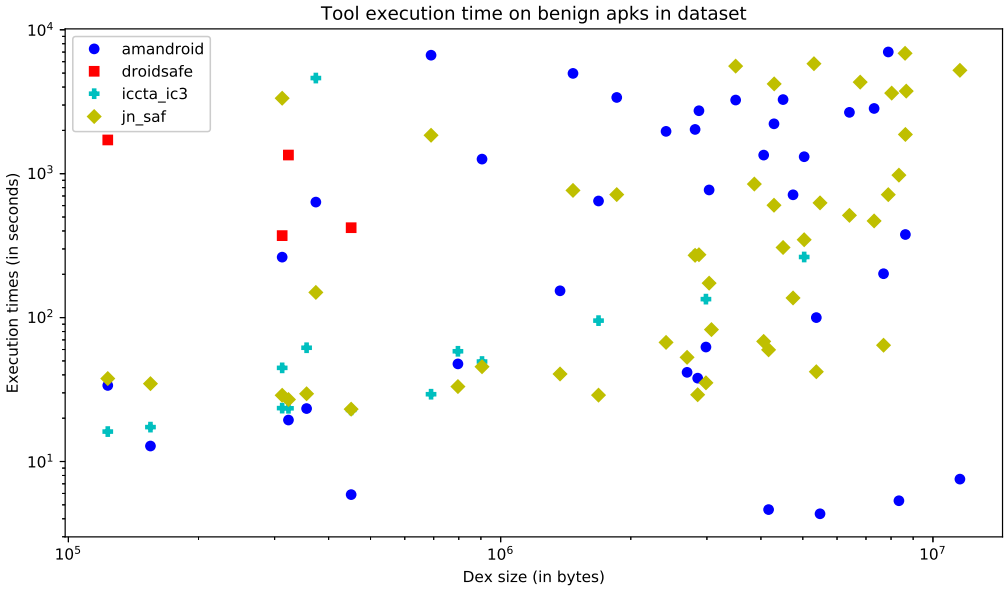


Figure 8: Size versus time on benign dataset

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

there is no strong correlation with the performance for larger dex sizes. This makes sense, as code analysis times are based on global variable amount, object creation amount, source accesses, inter component communication etcetera, which create exponentially larger analysis graphs, given that these code properties are part of possible taint paths. These factors result in much larger analysis times than simply creating larger components, as all paths from sources to sinks need to be traversed. Both methods increase dex size, however.

8.4 Errors

Benign and malicious dataset results show very high error amounts. To further investigate this, we determined recurring error occurrence statistics of apks. The results are shown in Table 4. Here we see that there is no large amount of apks causing recurring error occurrence statistics. On the benign part of our dataset the percentage of unique crashes is lower than on the malicious part of our dataset. This is because there are very many error responses on apks analysed with DroidSafe and IccTa+IC3, resulting in a smaller chance for errors to be unique. JN-SAF is not displayed in this figure, as this tool never ended analysis with an error,

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

even on those apks that produced errors with all other tools. There is no strong connection between errors on the dataset.
Next, we manually checked the output logs for each error. The results are shown in Table 5. We conclude that IccTa+IC3 is having many errors due to the following factors: It consumes exceptionally high amounts of memory, when compared to other tested tools, causing it to crash much more frequently on *OutOfMemoryExceptions*. Also it is likely to fail if there are classes with class annotations in one of the taint paths of analysed apk. Class annotations are used to give compiler directives for or provide metadata information to classes they annotate. For Droidsafe, we found out that in some cases, there were decompile problems, as Droidsafe could not find all classes given in the manifest of analysed apks. However, most of Droidsafe's errors are related to analysing code which is part of a standard type Android API. Examples of API's of this type are the Android Camera v2 API, Firebase AI API, gms Drive API, and Media API. Droidsafe was unable to analyse certain parts of these API's, such as, in many cases, the gms Drive's DataBufferAdapter widget. This API, created in 2014, is currently deprecated and will shutdown in December 2019. Apparently, the researchers have not tested their tools on apps containing these API's, in the period of 2014-2016, when it was still maintained.
Amandroid seems to have great trouble finding Android support v4 implementation classes, even though analysed apks specify target android SDK versions which contain the support v4 library and Amandroid has access to these SDK versions.

8.5 Timeouts

There is an unreasonably high amount of timeouts, coming mostly from Droidsafe. After looking into what the reason is, we firstly conclude that Droidsafe's multithreading is not sound. Its core usage spikes every few seconds, after which it drops to one or two cores. Secondly, the way Droidsafe analyses apks is inefficient. It uses separate programs to decompile apks and analyse decompiled result and is more disk-intensive than the other tested tools. Another important factor for timeouts is infinite recursion. Unfortunately, some, if not all tools seem to loop infinitely on certain input apks.
The amount of timeouts could be lowered by giving each analysis tool more time per analysis. However, even after apparently catching 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 in Figure 9, where we let JN-SAF run on a random test set without setting upper limits, and providing 312 GB RAM. Note that the greatest outlier took about $40,000$ seconds, or 11 hours, to finish with an error, because it consumed all 312 GB RAM. Also, one should notice that about 4% of the apks in this test

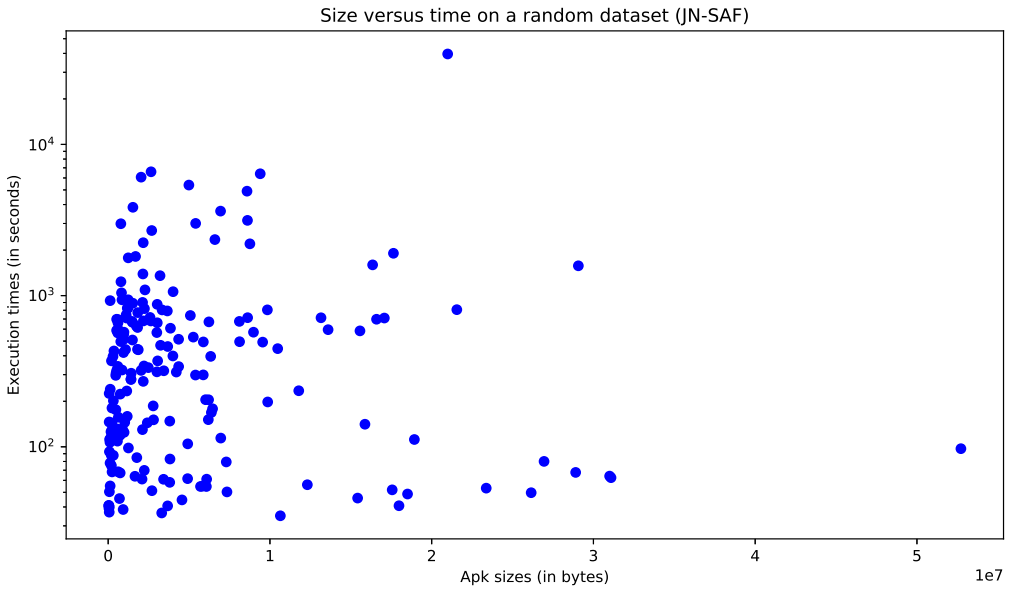


Figure 9: JN-SAF running on 200 randomly sampled apks, without timeouts

consume 45.68% of the total run time.

9 Limitations

There exist a few limitations in our work.
Due to the limited amount of time available for each apk, we most likely got timeouts for some apks which would have given a classification response, eventually. This varies greatly per tested tool. During our experiment, we did not expect such a high percentage of timeouts for Droidsafe and IccTa+IC3. It is an interesting finding in itself, as no other study disclosed statistics for non-classification outputs. However, it completely disables our ability to predict classification quality and performance for these tools, and is therefore a limitation of this work.
Furthermore, Amandroid and JN-SAF have used the AndroZoo dataset for testing, which we used for our experiments. This could lead to a statistical bias towards these tools. However, the probability that any apk we selected from the AndroZoo dataset is actually used by any tool developer we tested is very small, as the AndroZoo dataset contains approximately $9,000,000$ apks, from which we took random samples, as discussed in Section 6.2.
Lastly, as stated in Section 6.2: We use the AndroZoo dataset, which has no solid ground truth. Instead, it only counts the number of virus scanners of VirusTotal which flag given apk as malicious. Because of this, we only have an indication of maliciousness for each apk.

10 Conclusion

We built a framework to test Android static security analysis tools in a way that compares their effectiveness and efficiency on a realistic dataset. We used it to perform 384 analyses and conclude that the versions of the systems, that are available as open source, do not perform as well on real world apps as they do on the theoretical testbenches often used to evaluate them in the original papers.

Using a Chi-squared test, we found there exists no dependence between classifying accuracy on a practical dataset and tested Android static security analysis tools. Therefore, we conclude that it is better to use these static security tools on theoretical testbenches instead of running them on real world apps, in their current state. With enough improvements, however, we believe the amount of incorrect classifications on practical, real world apps could be drastically lowered. Then, static analysis tools could replace existing dynamic analysis tools for practical purposes, such as vetting uploaded apps for app stores, before making them available to the public. Such a replacement is favorable, as dynamic analysers have a fundamental weakness: they fail when their simulations are unable to trigger malicious behaviour through simulation. Using static analysers would remove this weakness.

Most of the tested tools claimed very high accuracy and scalability in their papers. In our experiments, these scanners did not perform as well as the papers claimed.

Although great progress has been made in the field of Android static security analysis research, this paper shows we are not yet at the point where we can use this kind of tools for practical purposes.

It is a good practice to independently compare analysis tools with each other, in order to find out each one's relative value. It is also a 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.

References

[1] Statcounter. *Mobile Operating System Market Share Worldwide*. http://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed:2019-02-08. Statcounter.

[2] 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. Statista.

[3] Statista. *Average number of new Android app releases per day from 3rd quarter 2016 to 1st quarter 2018*. https://www.statista.com/statistics/276703/android-app-releases-worldwide/. Accessed:2019-02-08. Statista.

[4] 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*. ACM. 2018, pp. 1137–1150.

[5] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. "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*. ACM. 2014, pp. 1329–1341.

[6] 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*. Vol. 15. 2015, p. 110.

[7] 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)*. 2013, pp. 543–558.

[8] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. "Composite constant propagation and its application to android program analysis". In: *IEEE Transactions on Software Engineering* 42.11 (2016), pp. 999–1014.

[9] 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*. IEEE Press. 2015, pp. 280–291.

[10] 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". In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269.

[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*. ACM. 2018, pp. 331–341.

[12] 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*. ACM. 2018, pp. 176–186.

[13] 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*. ACM. 2013, pp. 329–334.

[14] 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*. ACM. 2014, p. 5.

[15] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. "DroidNative: Automating and optimizing detection of Android native code malware variants". In: *computers & security* 65 (2017), pp. 230–246.

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

[17] 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)*. IEEE. 2016, pp. 756–761.

[18] Veelasha Moonsamy, Jia Rong, and Shaowu Liu. "Mining permission patterns for contrasting clean and malicious android applications". In: *Future Generation Computer Systems* 36 (2014), pp. 122–132.

[19] Wen-Chieh Wu and Shih-Hao Hung. "DroidDolphin: a dynamic Android malware detection framework using big data and machine learning". In: *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*. ACM. 2014, pp. 247–252.

[20] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors." In: *Ndss*. 2015.

[21] Lok Kwong Yan and Heng Yin. "DroidScope: Seamlessly Reconstructing the {OS} and Dalvik Semantic Views for Dynamic Android Malware Analysis". In: *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 2012, pp. 569–584.

[22] Miroslav Trnka and Peter Paško. *Android malware Protection and Internet Security*. https://www.eset.com/. ESET, 2019.

[23] Pavel Baudiš and Eduard Kučera. *Android Mobile Security*. https://www.avast.com/. Avast Software s.r.o., 2019.

[24] Fengguo Wei. *ICC-Bench benchmark suite*. https://github.com/fgwei/ICC-Bench. 2017.

[25] 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)*. IEEE. 2016, pp. 468–471.

[26] 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)*. Bonn, Germany: Springer, 2017, pp. 252–276.

[27] Bernardo Quintero, Emiliano Martínez, Víctor Manuel Álvarez, Karl Hiramoto, Julio Canto, Alejandro Bermúdez, and Juan A. Infantes. *VirusTotal*. https://www.virustotal.com/. Chronicle, 2004.

[28] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, Netherlands, 1995.

[29] 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". In: *Computer* 49.5 (2016), pp. 54–63.