Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification[☆]

Francisco Handrick da Costa [a,*], Ismael Medeiros [a], Thales Menezes [a], João Victor da Silva [a], Ingrid Lorraine da Silva [a], Rodrigo Bonifácio [a], Krishna Narasimhan [b], Márcio Ribeiro [c]

[a] Computer Science Department, University of Brasília, Brazil
[b] Software Technology Group, TU Darmstadt, Germany
[c] Institute of Computing, Federal University of Alagoas, Brazil

ARTICLE INFO

Article history:
Received 5 March 2021
Received in revised form 2 July 2021
Accepted 10 September 2021
Available online 30 September 2021

Keywords:
Malware detection
Mining sandboxes
Android platform
Static analysis and dynamic analysis

ABSTRACT

The popularization of the Android platform and the growing number of Android applications (apps) that manage sensitive data turned the Android ecosystem into an attractive target for malicious software. For this reason, researchers and practitioners have investigated new approaches to address Android's security issues, including techniques that leverage dynamic analysis to *mine Android sandboxes*. The mining sandbox approach consists in running dynamic analysis tools on a benign version of an Android app. This exploratory phase records all calls to sensitive APIs. Later, we can use this information to (a) prevent calls to other sensitive APIs (those not recorded in the exploratory phase) or (b) run the dynamic analysis tools again in a different version of the app. During this second execution of the fuzzing tools, a warning of possible malicious behavior is raised whenever the new version of the app calls a sensitive API not recorded in the exploratory phase.

The use of a mining sandbox approach is an effective technique for Android malware analysis, as previous research works revealed. Particularly, existing reports present an accuracy of almost 70% in the identification of malicious behavior using dynamic analysis tools to mine android sandboxes. However, although the use of dynamic analysis for mining Android sandboxes has been investigated before, little is known about the potential benefits of combining static analysis with a mining sandbox approach for identifying malicious behavior. Accordingly, in this paper we present the results of two studies that investigate the impact of using static analysis to complement the performance of existing dynamic analysis tools tailored for mining Android sandboxes, in the task of identifying malicious behavior.

In the first study we conduct a non-exact replication of a previous study (hereafter BLL-Study) that compares the performance of test case generation tools for mining Android sandboxes. Differently from the original work, here we isolate the effect of an independent static analysis component (DroidFax) they used to instrument the Android apps in their experiments. This decision was motivated by the fact that DroidFax could have influenced the efficacy of the dynamic analyses tools positively—through the execution of specific static analysis algorithms DroidFax also implements. In our second study, we carried out a new experiment to investigate the efficacy of taint analysis algorithms to complement the mining sandbox approach previously used to identify malicious behavior. To this end, we executed the FlowDroid tool to mine the source–sink flows from benign/malign pairs of Android apps used in a previous research work.

Our study brings several findings. For instance, the first study reveals that DroidFax alone (static analysis) can detect 43.75% of the malwares in the BLL-Study dataset, contributing substantially in the performance of the dynamic analysis tools in the BLL-Study. The results of the second study show that taint analysis is also practical to complement the mining sandboxes approach, with a performance similar to that reached by dynamic analysis tools.

[☆] Editor: Raffaela Mirandola.

[*] Corresponding author.
E-mail address: 180040723@aluno.unb.br (F.H. da Costa).

https://doi.org/10.1016/j.jss.2021.111092
0164-1212/© 2021 Elsevier Inc. All rights reserved.

1. Introduction

Almost two-thirds of the world use mobile technologies (Comscore, 0000), and the Android Operating System has dominated the market of smartphones, tablets, and others electronic devices (Anon, 2021). Due to this growing popularity, the number of incidents related to Android malicious software (malware) has significantly increased. In only three years, researchers reported a substantial increase in the population of Android malwares: from just three families and a hundred samples in 2010 to more than a hundred families with thousands of samples in 2013 (Faruki et al., 2015; Sufatrio et al., 2015). Security issues in Android software applications[1] have become a relevant research topic, and many techniques have been developed to identify vulnerabilities in Android apps (Arzt et al., 2014b), including the use of static analysis algorithms either to identify privacy leaks or to reveal the misuse of cryptographic primitives (Krüger et al., 2018; Rahaman et al., 2019), for instance.

Another alternative for protecting users from Android malicious behavior consists in the use of dynamic analysis to mine Android sandboxes (Jamrozik et al., 2016). The mine sandbox approach starts with an exploratory phase, in which a practitioner takes advantage of automatic test case generator tools that explores an Android application while recording the set of sensitive APIs the app calls. . This set of sensitive calls comprises a sandbox infrastructure. After the exploratory phase, the sandbox might then monitor any call to sensitive APIs while a user is running the app, blocking the calls that have not been identified during the exploratory phase—thereby protecting Android users from additional malicious behavior (Jamrozik et al., 2016). Jamrozik et al. argue in favor of dynamic analysis for mining sandboxes, instead of using static analysis—mostly because of the overapproximation problem: "static analysis often assume that more behaviors are possible than actually would be" (Jamrozik et al., 2016). In addition, code that uses dynamic features (such as reflection) poses additional challenges to static analysis algorithms—even though *dynamic features* of programming languages are often used to introduce malicious behavior. Even though these claims are reasonable, previous research results do not present empirical assessments about the limitations of static analysis to mine sandboxes. Consequently, it is not clear whether and how both approaches (dynamic and static analysis) could complement each other in the process of mining Android sandboxes.

The lack of understanding about static and dynamic analysis complementing each other also appears in the work of Bao et al. (2018) (hereafter BLL-Study), which presents an empirical study that explores the performance of dynamic analysis for identifying malicious behavior using the mining sandbox approach. Their study leverages DroidFax (Cai and Ryder, 2017) to instrument 102 pairs of Android apps (each pair comprising a benign and a malicious version of an App) and to collect the information needed to mine sandboxes (that is, the calls to sensitive APIs). Although the authors report a precision of at most 70% of dynamic analysis tools to differentiate the benign and malicious versions of the apps, the authors ignore the fact that DroidFax statically analyzes the Android apps and also records calls to sensitive APIs (besides instrumenting the apps). As we discuss in this paper, this DroidFax static analysis component leads to an overestimation of the performance of the dynamic analysis tools for mining sandboxes and might have introduced a possible threat to the conclusions of that work. In the security domain, overestimating the performance of a technique for malware identification brings serious risks, and we show here that DroidFax inflated significantly the performance of the dynamic analysis tools for mining sandboxes, as reported in the BLL-Study.

The goal of this paper is two fold. First we present the results of an external, non-exact replication (Shull et al., 2008) of the BLL-Study. To this end, we take advantage of DroidXP, a tool suite that helps researchers (including ourselves) to integrate test case generation tools and compare their performance on mining Android sandboxes. We discussed the design and implementation of DroidXP in a conference paper (da Costa et al., 2020), which also includes an initial evaluation of DroidXP. As a matter of fact, the results of the first DroidXP evaluation revealed a possible overestimation in the performance of dynamic analysis tools as reported in the BLL-Study—which in the end motivated us to conduct the non-exact replication of that study. Here we extend our previous work with a couple of customizations of DroidXP, which allowed us to reproduce the BLL-Study by means of a series of new experiments that reveal the actual performance of the dynamic analysis tools. Section 3.1 revisit the DroidXP design, while Section 3.2 discuss the setup of our replication study.

Second, in this paper we also explore how a static analysis approach (based on taint analysis) compares and complements the mining sandbox technique for identifying malicious behavior that infects benign applications. The idea here is to compare the dataflows from *source* to *sink* statements computed using two executions of the FlowDroid infrastructure (Arzt et al., 2014a): one execution that analyzes a benign version of an Android app and one execution that analyzes a malicious version. We consider that the taint analysis approach is able to identify a malware whenever we find a dataflow from a source to a sink in the second execution that does not appear in the first one. We detail the settings of this taint analysis study in Section 3.3

Altogether, this paper brings the following contributions:

- A replication of the BLL-Study that better clarifies the performance of dynamic analysis tools for mining Android sandboxes. The results of our replication (Section 4.1) give evidence that the previous work overestimated the performance of the dynamic analysis tools—that is, without DroidFax (an independent component used for running the BLL-Study experiment), the performance of the tools drop between 16.44% to 58%.
- A broad comprehension about the role of static analysis tools for mining sandboxes, showing that we can benefit from using both static and dynamic analysis for detecting malicious Android apps. In addition, we give evidence that a well known static analysis approach, based on taint analysis, leads to a performance similar to the dynamic analysis approach for differentiating benign and malicious versions of the same app (Section 4.2).
- A reproduction package of our study that is available online, including scripts for statistic analysis[2] and tooling for reproducing and extending our study. The repository for DroidXP is available at GitHub.[3]

2. Background and related work

In this section, we introduce the concepts and terminology that are necessary to understand the remainder of this paper. First, Section 2.1 introduces some background information about the use of *sandboxes* to protect invalid access to sensitive resources. After that, in Section 2.2 we review the mining sandbox approach for detecting malicious behavior in Android apps. Finally, Section 2.3 presents some background information about taint analysis.

[1] In this paper, we will use the terms Android Applications, Android Apps and Apps interchangeably, to represent Android software applications.

[2] https://htmlpreview.github.io/?https://github.com/droidxp/paper-replication-package/blob/master/replication.html.

[3] https://github.com/droidxp/benchmark.

2.1. The sandbox approach for protecting resources

A *sandbox* is an isolated environment on an electronic device within which applications cannot affect other programs outside its boundaries, like the file system, the network, or other device data (Maass et al., 2016). Sandboxes enable testing and execution of unsafe or untested code, possible malware, without worrying about the integrity of the electronic device that runs the application (Bordoni et al., 2017). This need might arise in a variety of situations, such as when executing software input by untrusted users, in malware analysis, or even as a security mechanism in case a trusted system gets compromised (Maass et al., 2016). A sandbox environment must be able to shield the host machine or operating system from any damages caused by third-party software. Thus, a sandbox environment should have the minimum requirements to run programs (make sure the program will not impact resources outside the sandbox), and make sure it will never assign the program greater privileges than it should have, working with the principle of *least privilege*, giving permissions to users according to their needs, i.e., giving them no more power than needed to successfully perform their task. This principle prevents escalating privileges and unauthorized access to resources, thereby improving the system's overall reliability.

Within the Android ecosystem, least privilege is realized through sandboxing process, where apps never access the data of other apps, and an app just accesses user resources, like contacts and location, through specific APIs (Application Programming Interface), which are in-turn guarded by permissions. Google Play Store is the primary market source for Android apps, and has a flexible policy regarding the apps' publishment process. Therefore, every month administrators remove several Android apps from the Play Store because of issues related to spyware and other types of malware (Wang et al., 2018). For security reasons, Google Play lists each app with its requested permissions. However, many malicious apps usually ask for more permissions than their APIs normally would require (Felt et al., 2011). Those permissions are presented to the user during a new app's installation, since Android version 6, but most users are careless since they are only interested in the end product (Felt et al., 2012).

Nowadays, malware becomes more stealthy and hackers learn how to avoid anti-virus signature checks, for instance by obfuscating calls to native code that is allowed to make system calls (Glanz et al., 2020) or conducting side attacks to make system calls from a benign app.

2.2. Mining Android sandbox

The mining Android sandbox approach (Jamrozik et al., 2016) relies on test generator tools to explore an Android app's dynamic behavior, and thus mine a set of sensitive resources the app needs. The sandbox uses this set of sensitive APIs to ensure the app execution's security by restricting the resources that are allowed to use. The mining sandbox approach works in two phases. In the first, named exploratory phase, a practitioner uses test generator tools to execute a benign version of an app and record the set of sensitive APIs the app calls. In the second phase, named execution, the sandbox constraints the app to access only the sensitive APIs mined in the first phase. Accordingly, the sandbox ensures that a malicious app could not call any sensitive API, besides those calls to APIs recorded in the first phase.

The idea of automatically mining software resources or components to infer behavior is not new, and has been discussed before. For instance, Whaley et al. (2002) combine dynamic and static analysis for API mining and so infer program behavior based on an usage example of Java classes. Ammons et al. (2002) propose a machine learning approach, called specification mining, to discover temporal and data-dependence relationships that a program follows when interacting with an API or abstract data types.

The main purpose of a test generator tool is to program crashes or bugs in general. Nonetheless, it is also possible to use test generator tools to explore program behavior (dynamic analysis), and thus assist in the task of building sandboxes. Regarding test generator tools used for mining Android sandboxes, Jamrozik and Zeller (2016) proposed DroidMate, a test generation tool that implements a pseudo-random graphical user interface (GUI) exploration strategy, and was the first approach to leverage test generation to extract sandbox rules from apps. Li et al. (2017b) proposed DroidBot, a test generator tool that explores sensitive resources access from Android apps, following a model-based exploration strategy. In their work, the authors present a comparison between DroidBot and Monkey (Monkey, 2020b) regarding malware analysis and show that DroidBot can trigger several sensitive calls more often than Monkey. Sensitive calls in the Android context occurs when an Android app functionality can result in accessing or leaking of Android users' sensitive data. Examples of sensitive calls access user location or network information. Humanoid is another test generator tool for Android (Li et al., 2019)—actually a DroidBot evolution. It is also a GUI test generator that learning how humans interact with Android apps. In contrary to random input generators, Humanoid uses a learned model to generate human-like test inputs, and prioritize the possible interactions on a GUI, according to their importance.

Bao et al. (2018) present a comparative study test generator tools to identify malicious behavior using the mining sandboxes approach. Their study indicates that the tools were efficient in identifying at most 70% of the malware in a specific dataset and also reports that after, combining all test generator tools, it was possible to detect 75.49% of malicious behavior explored (77 among 102). However, this study did not focus on the possible interference of static analysis in the final result, since this study used DroidFax (Cai and Ryder, 2017) to instrument the apps, though, as we discuss in this paper, DroidFax also performs a static analysis of the apps whose results complement the dynamic analysis approach for mining sandboxes.

2.3. Taint analysis

Taint analysis is a special type of static or dynamic analysis that aims to track data flows within programs (Pauck et al., 2018). Typically, taint analysis is used to identify sensitive information leakage by detecting taint flow between "sources" and "sinks". In the context of Android apps, a data leak occurs when sensitive data, such as contact, or device ID, flows from a sensitive resource to a method that might *sink* information to other peers, like sending a message. Taint analysis can present possible malicious data flow to malware detection tools or even for a human check, which can decide if the "source–sink" relationship is or is not an unwanted behavior. Thereby, taint analysis monitors sensitive sources "tainted" through the app by starting at a pre-defined point.

In the Android context, sources are the APIs in which apps access sensitive information, called sensitive APIs. The analysis follows the data flow until it reaches a sink, like a method that sends SMS. It brings precise information about which data will be leaked (Arzt et al., 2014b). The Android SDK provides APIs that allow apps to send private data to other apps on the same device, or remote devices. As these APIs may lead to sensitive data leakage, they are security-critical and require special attention and control (Enck et al., 2010). (Listing 1) presents a simple data leakage example. In this example, the device information is captured at line 4 (source) and then leaked at line 9 (sink), by SMS transmission.

Listing 1: Simple Data Leakage

```
localObject2 = (TelephonyManager)getSystemService("phone");
if (localObject2 != null)
{
 this.imei = ((TelephonyManager)localObject2).getDeviceId();//source
}
if ("".equals(this.destMobile)) {
 getDestMobile();
}
sendSMS(this.destMobile, "imei:" + this.imei)//sink
```

Huang et al. (2015) propose a scalable taint analysis for Android apps that applies traditional taint analysis techniques with targeted optimizations specific to Android OS. FlowDroid (Arzt et al., 2014b) improves the precision of traditional approaches by including context and flow sensitivity. A significant issue with taint analysis is the cost of the tool itself hampering the performance. FastDroid (Zhang et al., 2021) mitigates this issue by introducing an intermediate light-weight abstraction to perform the analysis, called taint value graph (TVG). To improve efficiency and precision, FastDroid focuses on exploring the propagation of taint values, rather than the traditional data flow analysis. FastDroid constructs taint value graphs (TVGs) exploring taint values, then it extracts a subset of potential taint flows (PTFs) from it. FastDroid improves the analysis process by performing analysis only on (PTFs). In this paper we investigate the use of taint analysis to identify malicious behavior, by mining the source an sink pairs from distinct versions of an app.

3. Study settings

Our research work aims to better understand the use of static analysis to mine Android sandboxes and explore the benefits of combining taint analysis with the mine sandbox approach for identifying malicious behavior. On the one hand, Jamrozik et al. suggest that a static analysis approach for mining sandboxes might be ineffective—due to *overapproximation problem* (Jamrozik et al., 2016). However, to the best of our knowledge, there is no empirical study comparing static and dynamic analysis for mining sandboxes. On the other hand, the BLL-Study explored the mining sandbox approach by comparing the performance of five **dynamic analysis tools** (DroidMate, DroidBot, PUMA, GUIRipper, and Monkey) for identifying malicious behavior. Nonetheless, their research also involved an external static analysis component (DroidFax) whose impact on the results was not measured—in terms of malware identification. This lack of understanding about the implications of static analysis for mining sandboxes motivates our research, which investigates the following research questions.

(RQ1) What is the impact of the DroidFax static analysis algorithms on the results of the BLL-Study? We estimate the impact in terms of the number of detected malwares.

(RQ2) What is the effective performance of each sandbox, in terms of the number of detected malware, when we discard the contributions of the DroidFax static analysis algorithms?

(RQ3) What are the benefits of using taint analysis algorithms to complement the dynamic analysis approach for mining sandboxes, in terms of additional malwares identified?

Answering the research questions RQ1 and RQ2 allows us to better understand the relevance of combining static and dynamic analysis for mining Android sandboxes. Moreover, exploring RQ1 and RQ2 can reveal a possible overestimation of the performance of the dynamic analysis tools in the BLL-Study. Answering research question RQ3 allows us to open up the possibility of finding new strategies for malware detection, complementing the performance of dynamic analysis through the use of static analysis algorithms. We conducted two empirical studies to answer the research questions above. We address the research questions RQ1 and RQ2 in the first empirical study, whose goal is to conduct a non-exact replication of the BLL-Study. We conduct the first empirical study using DroidXP (da Costa et al., 2020) (Section 3.1), a tool that simplifies the execution of experiments that compare the performance of dynamic analysis tools in the task of identifying malwares, using a mining sandbox approach. We present the study settings of the first empirical study in Section 3.2. In the second empirical study we use FlowDroid (Arzt et al., 2014b) to investigate the suitability of taint analysis algorithms to complement the mining sandbox approach for identifying malwares, and thus it targets our third research question (RQ3). We present the settings of the second empirical study in Section 3.3.

3.1. The DroidXP benchmark

We designed and implemented DroidXP to systematically assess and compare the performance of test generation tools for mining android sandboxes. It allows the integration and comparison of test case generation tools for mining sandboxes, and simplifies the reproduction of the studies. DroidXP relies on a simple *Command Line Interface* (CLI) that simplifies the integration of different test generation tools and favors the setup and execution of the experiments. DroidXP also relies on DroidFax, which instruments Android apps and collects relevant information about their execution, including the set of sensitive APIs a given app calls during a test execution. DroidFax also collects inter-component communication (ICC) using static program analysis.

The DroidXP CLI provides commands for listing all test case generation tools (executing the project with the option "list-tools") that had been integrated into the tool and commands for executing the experiments. An *experiment run* can be configured according to several parameters, including:

- `-tools`: Specifies the test tools used in the experiment
- `-t`: Specifies the threshold (in seconds) for the execution time in the experiment
- `-r`: Specifies the number of repetitions used in the experiment
- `-output-format`: Specifies the output format
- `--debug`: Specifies to run in DEBUG mode (default: false)
- `--disable-static-analysis`: Disable DroidFax static analysis phase (default: false)

Fig. 1 shows the DroidXP architecture, based on the pipes-and-filters architectural style (Buschmann et al., 1996). The architecture includes three main components; where each component is responsible for a specific phase of the experiments execution (instrumentation, execution, and result analysis).

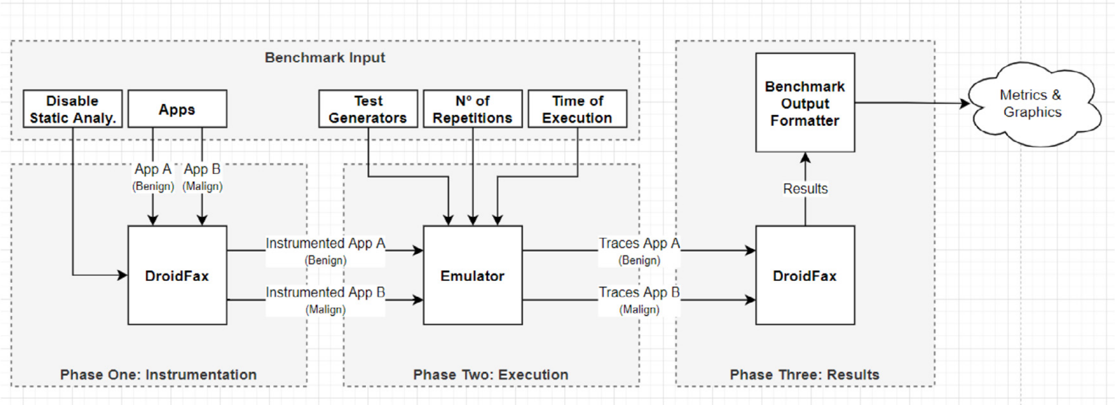

Fig. 1. DroidXP architecture.

3.1.1. Phase 1: Instrumentation

In the first phase, a researcher must define the corpus of APK files DroidXP should consider during an experiment execution. After that, DroidXP starts the DroidFax service that instruments each APK file, so that DroidXP would be able to collect data (e.g., calls to sensitive APIs) about each execution. To improve the performance of DroidXP, the instrumentation phase runs only once for each APK. In this phase, the DroidFax tool also runs some static analysis procedures—when the option `--disable-static-analysis` is not set.

3.1.2. Phase 2: Execution

In this phase, DroidXP installs an (already instrumented) APK file into an Android emulator, and then executes a test case generation tool during a period of time. This process repeats for every test case generation tool and APK files. To provide repeatability of the experiment, DroidXP removes all data stored in the emulator before starting a new execution. That is, every execution uses a *fresh* emulator, without any information that might have been kept during previous executions. It is relatively easy to add new test case generation tools into DroidXP. Indeed, every new tool must override two methods of a `Tool` abstract class (according to the Strategy Design pattern (Gamma et al., 1995).

3.1.3. Phase 3: Result analysis

During the execution of the instrumented apps, all data that is relevant to our research is collected by Logcat (Anon, 2020a), one of the Android SDK's native tools. Logcat dumps a log from the Android emulator while the already instrumented app is in execution. The part of the log we analyze in this phase comprises the data sent by the methods within the Android app that were instrumented on the first phase using the DroidFax tool.

This data includes method coverage from the execution of each test generator tool and the set of sensitive APIs the app calls during its execution. This set of calls to sensitive APIs is necessary to estimate the test generator performance in identifying malicious apps—by spotting differences between the sensitive API accessed by each version of an app (benign or malign). In the end, DroidXP outputs the results of the experiment, which gives the performance of one or more testing generator tools in mining sandboxes.

We used the DroidXP infrastructure to conduct our first empirical study, whose settings we present in the following section.

3.2. First study: A replication of the BLL-Study

The BLL-Study reports the results of an empirical study that compares the performance of test generation tools to mine Android sandboxes (Bao et al., 2018). Since the BLL-Study does not compute the possible impact of DroidFax into the performance of the test generation tools, here we replicate their work to understand the impact of the DroidFax static analysis algorithms into the BLL-Study results.

Our replication differs from the original work in a few decisions. First, here we isolate the effect of the DroidFax static analysis algorithms, in the task to identify malicious apps. In addition, although we use the same dataset of 102 pairs of Android apps used in the BLL-Study, here we discarded 6 pairs for which we were not able to instrument—out of the 102 pairs used in the original work, originally shared in the AndroZoo repository (Allix et al., 2016). We also introduced a recent test generator tool (Humanoid Li et al., 2019), which has not been considered in the previous work. Finally, we extended the execution time of each test generation tool, executing each app from the test generation tool for three minutes (instead of one minute in the original work), and built the sandboxes after executing each test generation tool three times—the original work executed each test generation tool only once. It is important to note that our goal here is not to conduct an exact replication of the BLL-Study, but instead understand the role of the DroidFax static analysis algorithms in the performance of test case generation tools for mining sandboxes.

Besides Humanoid, our study considers three test generation tools used in the BLL-Study: DroidBot (Li et al., 2017b), DroidMate (Jamrozik and Zeller, 2016), and Monkey (2020b). We selected DroidBot and DroiMate because they achieved the best performance on detecting malicious behavior—when considering the 102 pairs of Android apps (B/M) in the BLL-Study. It is important to note that here we used a new version of DroidMate (DroidMate-2), since it presents several enhancements in comparison to the previous version. We also considered the Google's Monkey open source tool, mostly because it is the most widely used test generation tool for Android (Zeng et al., 2016). Monkey is part of the Android SDK and does not require any additional installation effort. We included Humanoid in our study because it is a recent tool that emulates realistic users, creating human-like test inputs using deep learning techniques.

3.2.1. Data collection

Similarly to the BLL-Study, besides method coverage information, our experiments record every call to *sensitive methods* of the Android platforms, while a given test case generation tool is running. We consider the same set of 97 sensitive methods from the AppGuard privacy-control framework uses (Backes et al., 2013).

We executed DroidXP using two configurations. In the first (named WOS), we executed DroidXP using the dataset of 96

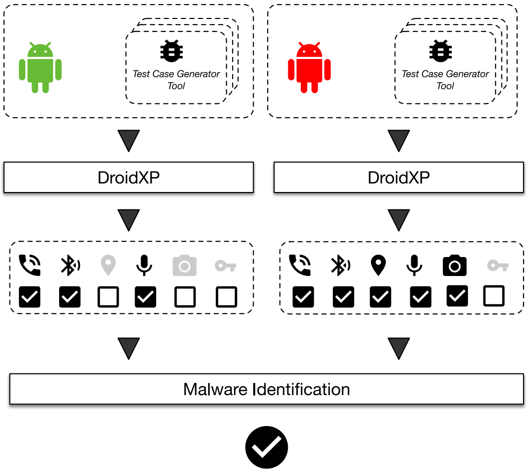

Fig. 2. Overview of our approach for malware identification in the first study.

pairs of Android apps—each pair including a benign and a malign version, the four test case generation tools (DroidBot, DroidMate, Monkey, and Humanoid), and the `--disable-static-analysis` option of DroidFax, which disables the execution of the DroidFax static analysis component from the experiment. The WOS configuration runs the test case generation tools for three times, using a time limit of three minutes. In the second configuration (named WS), we executed DroidXP using the same dataset of 96 pairs of Android apps, though also executing a fake test case generator tool (named Joker) **without** the `--disable-static-analysis` option. Joker simulates a test tool that does not run the Android apps during an experiment execution, and its usage allow us to estimate the actual performance of the DroidFax static analysis component.

Using the WS configuration, the Execution Phase of DroidXP does not collect any call to sensitive APIs, and thus we can estimate the performance of the static analysis component of DroidFax (answering RQ1). Differently, the WOS configuration disables the static analysis component of DroidFax and we could better estimate the true performance of the test case generation tools for mining android sandboxes (answering RQ2). For comparison purpose, we also executed the four test case generation tools using the DroidFax static analysis component.

3.2.2. Data analysis

DroidXP produces a dataset with the sensitive APIs that the benign/malign versions of an app call, during the execution of each test case generation tool. We estimate the performance of a test case generation tool by considering the percentage of malwares in our dataset its resulting sandbox is able to identify .

Recall that we build a sandbox during the exploratory phase of the mining sandbox approach. This exploratory phase records the set of sensitive APIs a benign version of an app calls—during the execution of a test case generation tool. Similarly to the BLL-Study, we consider that a sandbox of an app identifies a malware whenever the malicious version makes a call to a sensitive API that has not been recorded during the exploratory phase (see Fig. 2).

To sum up, in order to analyze the performance of the test case generation tools (including Joker), we just have to compare the calls to sensitive APIs made by the benign and malign versions of the apps, during the execution of the tools. In the end, we generate a set of observations, where each observation contains the tool name, the number of the repetition (in the range [1..3]), a boolean value reporting the use of the DroidFax static analysis component, and a boolean value indicating whether or not the malware has been identified. We use descriptive statistics and plots to compare the performance of the tools and answer RQ1 and RQ2. We also use *Logistic Regression* (James et al., 2014, Chapter 4) to understand the statistical relevance and the contribution of each feature (tool, repetition, DroidFax static analysis component) to malware identification. Our hypothesis here is that the DroidFax static analysis component has a positive effect on the performance of the sandboxes to identify malwares.

3.3. Second study: Use of taint analysis for malware identification

In the second empirical study we investigate whether or not a taint-based static analysis approach is also promising for identifying malwares, given a version of an app that we can assume to be secure (goal of research question RQ3). To this end, we leverage the FlowDroid taint analysis algorithms for Android apps (version 2.8), in order to identify dataflows that might lead to the leakage of sensitive information. Our goal here is to investigate if it is possible to detect malicious behavior by means of the *divergent* source–sink paths that FlowDroid reveals after analyzing a benign and a malign versions of an Android app.

3.3.1. Data collection

FlowDroid takes as input an Android Application Package (APK file) and a set of API methods marked either as **source** or **sink** (or both). Source methods are those that access *sensitive information* (e.g., a method that access the user location), while sink methods are those that *might share information with external peers* (e.g., a method that sends messages to a recipient). We rely on the source–sink definitions of the FlowDroid implementation (Arzt et al., 2014a; Rasthofer et al., 2014), which involves a curate list of source and sink methods (including callbacks and other Android API methods of interest). FlowDroid then uses a *context, flow, and field sensitive analysis* to identify dataflow paths from sources to sinks (Arzt et al., 2014a).

Our data collection approach involves three steps (see Fig. 3). In the first, we execute FlowDroid to mine the source–sink paths from a benign version of an app, and then enumerate a set (S1) with the possible dataflows between sources and sinks. All paths in S1 are considered secure in our analysis. In the second step we repeat the FlowDroid execution, though considering the malicious APK version of the app. This leads to a second set (S2) of source–sink paths.

It is important to note that not all source–sink paths are malign, and then we follow a specific methodology to identify malwares using taint analysis. That is, we only report a malware when FlowDroid finds an additional source–sink path in the malicious version of an app, which has not been identified when analyzing the benign version. Therefore, in the third step we compute the difference (S3) between the sets S2 and S1 (i.e., $S3 = S2 \setminus S1$). If the set S3 is not empty, we assume that FlowDroid has identified the malware.

In this second study we use the same dataset of 96 pairs of Android apps (B/M) used in the first empirical study.

3.3.2. Data analysis

We use two metrics in this second study: the total number of malicious apps FlowDroid is able to find and the execution time for running the taint analysis algorithm for each app. Similarly to the first empirical study, we use descriptive statistics and plots to compare the performance of the taint analysis and mining sandbox approaches. We also use *Logistic Regression* (James et al., 2014, Chapter 4) to better understand the statistical significance of the benefits of using FlowDroid (in comparison to the DroidFax static analysis component only). Our hypothesis here is that FlowDroid outperforms, in terms of the number of detected malware, the sandbox generated by the DroidFax static analysis component.

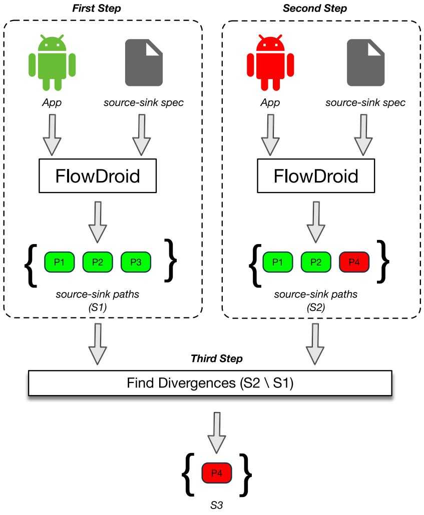

Fig. 3. Overview of our approach in the second study.

4. Results and discussion

In this section we detail the findings of our study. We present the results of the first and second studies in Sections 4.1 and 4.2, respectively. In Section 5 we summarize the implications of our study.

4.1. Result of the first study: A BLL-Study replication

Our first study is a replication of the BLL-Study. As discussed in the previous section, we first executed the analysis using the DroidXP benchmark with its default configuration. Then we repeated the process, however this time we isolate the effect of the static analysis component of DroidFax. In this way, we could better estimate the performance of the dynamic analysis tools for mining Android sandboxes. Table 1 summarizes the results of the executions. The columns Exec. (WS) and Exec. (WOS) show the number of malwares identified when executing each tool with the support of the DroidFax static analysis algorithms (WS) and without the support of DroidFax static analysis algorithms (WOS). The Impact column shows (in percentage) to what extent the DroidFax static analysis algorithms influences the performance of the sandboxes created after executing the test generation tools. We calculate the impact using Eq. (1).

$$Impact = \frac{(Exec.\,(WS) - \; Exec.\,(WOS)) \times 100}{Exec.\,(WS)} \tag{1}$$

Table 1 shows that the impact of DroidFax in the results is significant, ranging from 16.44% (DroidBot) to 51.79% (Humanoid). Note that, in the BLL-Study, the authors do not present a discussion about the influence of DroidFax in the performance of the test generation tools, even though this influence is not negligible. Considering the Joker tool, our fake test generation tool that does not execute the apps during the benchmark execution, DroidFax improves the performance in 100%. This result is expected, since the Joker tool does not execute any dynamic analysis. Next we discuss the result of each individual test generation tool.

Table 1
Summary of the results of the first study.

Tool	Exec. (WS)	Exec. (WOS)	Impact (%)
DroidBot	73	61	16.44
Monkey	71	56	21.13
DroidMate	68	52	23.53
Humanoid	56	27	51.79
Joker	42	0	100.00

DroidBot in the first execution (Exec. WS) led to a sandbox that detected a total of 73 malware among 96 pairs present in our dataset (76.04%), detecting more apps with malicious behavior than any other tool. Similar to the BLL-Study, DroidBot is the test case generation tool whose resulting sandbox detected the largest number of malicious apps. Moreover, in our second execution (Exec. (WOS)), removing the DroidFax static analysis support reduced the DroidBot performance in 16.44%, the smaller impact we observed among the tools.

Monkey in the first execution (Exec. (WS)) produced a sandbox that detected 71 out of the 96 pairs of Android apps. Contrasting, in the original study, the Monkey's sandbox detected 48 malwares within the 102 pairs (47.05%). This difference might be due to the fact that Monkey uses a random strategy for test case generation and here we considered the outcomes of three executions—while in the BLL-Study, the authors consider the outcomes of one execution. Considering our second execution (Exec. (WOS)), there is a reduction of 21.13% in the Monkey's performance, leading to a sandbox that was able to detect 56 malwares.

DroidMate in the first execution (Exec. (WS)) led to a sandbox that detected 68 apps with malicious behavior (70.83%). In the BLL-Study study, DroidMate also detected 68 malwares, though considering the 102 pairs of apps used in the original study. In the second execution (Exec. (WOS)), without the DroidFax static analysis algorithms, the resulting sandbox's performance drops by 23.53%, being able to detect 52 out of the 96 pairs of Android apps.

Humanoid showed the worst performance, even though a previous work (Li et al., 2019) presented that it leads to the highest number of lines coverage in comparison to Monkey, DroidBot, and DroidMate. This might suggest that, since Humanoid learn how humans interact with apps, and use the learned model to guide test generation, at simulate environment, this method to generate test inputs are less effective to build Android sandbox, in comparison with techniques that rely on random testing (such as Monkey). In the first execution (Exec. (WS)), the resulting Humanoid sandbox identified 56 malwares in our dataset (58.33%). Humanoid was the most affected in the second execution (Exec. (WOS)), whose resulting sandbox presents a reduction of 51.79% in the number of detected malwares. Since the BLL-Study did not explore Humanoid, we do not have a baseline for comparison with the previous work.

Joker is our fake test case generation tool that help us understand the performance of the DroidFax static analysis algorithm for mining sandboxes. We integrated Joker into the DroidXP benchmark as an additional test case generation tool that does not run the Android apps. As a result, the analysis using Joker reveals the performance of DroidFax static analysis algorithms alone. For the first execution, with the DroidFax static algorithms enabled, even though Joker does not execute the Android apps, its resulting sandbox detected 43.75% of the malwares. For the second execution, that is, disabling the DroidFax static analysis algorithm, the resulting Joker sandbox was not able to detect any malware. Therefore, our results show that DroidFax alone is able to detect more than 40% of the malicious version of the apps.

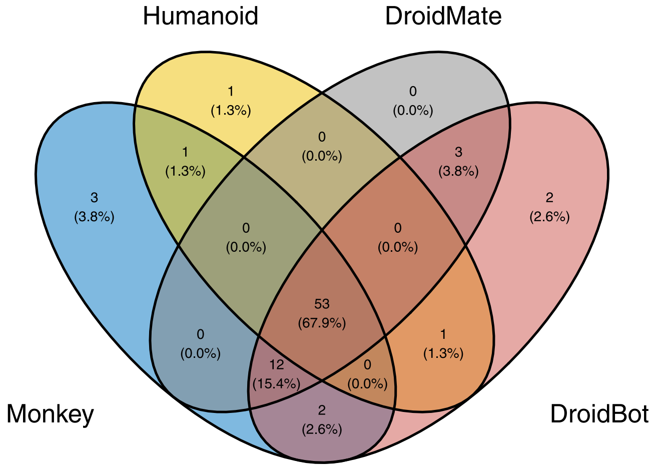

Fig. 4. Venn Diagram highlighting how the sandboxes from the tools can complement each other.

Finding 1. *Integrating the dynamic analysis tools with the DroidFax static analysis algorithms improves substantially the performance of the resulting Android sandboxes for detecting malicious behavior.*

The Venn-diagram of Fig. 4 summarizes how the tools can complement each other. Note in the diagram that 53 malwares have been detected by all sandboxes generated in the first execution (with the DroidFax static analysis algorithms), out of the 78 malwares identified by at least one sandbox. In addition, the DroidMate sandbox did not detect any malware that had not been detected by the other tools. Differently, the Monkey sandbox detected three malwares that had not been detected by any other sandbox, the DroidBot sandbox detected two malwares that had not been detected by any other sandbox, and the Humanoid sandbox detected one malware that had not been detected by any other sandbox. Contrasting with the BLL-Study, our results suggest that using DroidMate in combination with Monkey, DroidBot, and Humanoid does not improve the general performance of an integrated environment for mining Android sandboxes.

Finding 2. *Our results suggest that one might benefit from using an integrated environment that combines Monkey, DroidMate, and Humanoid to mine Android sandboxes. Contrasting with the BLL-Study, introducing the DroidMate tool does not improve the overall performance for detecting malwares using a mining sandbox approach.*

Altogether, ignoring Joker, our study reveals that from 58.33% (Humanoid) to 76.04% (DroidBot) of the malicious apps investigated in our study can be detected using the sandboxes generated after running the test case tools with the support of the DroidFax static analysis algorithms. We also investigate if the use of the DroidFax static analysis component leads to a statistically significant benefit on malware identification. To this end, we build a logistic regression model in the form $Malware \sim Tool + StaticAnalysis + Repetition$. Table 2 shows the results of the logistic regression analysis, highlighting that (a) Humanoid has a negative, though significant impact on malware identification; and (b) the use of DroidFax static analysis has a positive and significant impact on malware identification.

Table 2
Results of the Logistic Regression (first study)

	Estimate	*p-value*	*C.I.*
Tool [DroidBot]	0.1034	0.4718	(−0.133,0.340)
Tool [DroidMate]	−0.0561	0.6955	(−0.292,0.180)
Tool [Humanoid]	−0.8910	0.0000 ***	(−1.131,-0.651)
Tool [Monkey]	−0.0110	0.9390	(−0.247,0.225)
DroidFax static analysis	0.8867	0.0000 ***	(0.743,1.031)
Repetition	−0.0171	0.7487	(−0.105,0.071)
AIC	3001.07		
Num. obs.	2304		

*** *p-value* < 0.001.

Besides that, in the first execution (WS), none of the resulting sandboxes could detect 18 malwares in our dataset (18.75%). According to the Euphony tool (Hurier et al., 2017), 12 of these 18 malwares are *adwares*, 3 are *trojans*, 2 are PUPs (*Potentially Unwanted Program*), and one is an *exploit*. At this point, an additional question arises: what are the characteristics of the malwares that have (not) been identified using the mining sandbox approach? To explore this question, we take advantage of the `dex2jar` tool to reverse-engineer all 96 malwares considered in our analysis and computed the *diffs* of the benign/malicious versions of the APPs. The results of this activity are available in our replication package.[4] In what follows we dissect a few examples of malwares that at least one of the resulting was able to identify. After that, we present the characteristics of a malware that none of the sandboxes was able to detect. Our goal here is to provide a lower-level intuition about the classes of malware the mining sandbox approach is able to detect. A reader that is not interested in these details could skip to Section 4.2.

To start with, consider the malicious version of the app `com.andoop.flyracing`—which both DroidBot and Humanoid sandboxes could detect in our analysis. In this particular case, the malicious version changes the Android Manifest file, adding permissions to receive and send SMS messages (Listing 2). Adding these permissions, a malicious app may get money fraudulently by sending messages without user confirmation, for instance. The pair `L:M` indicates a code segment that appears in line `L` of the malicious (`M`) version of an app.

After decompiling this malware, we also observed that the malicious version of the `MainService` class introduces a behavior that collects sensitive information (the International Mobile Equipment Identity, IMEI) and sends it using an SMS message (Listing 3).

The malicious version of the app `com.happymaau.MathRef` also changes the Manifest file to require additional permissions as well as change the behavior of the app (with malicious code). All sandboxes were able to detect this malware. In this case, the malicious version of the app changes the Android Manifest file, requiring permissions to access the network and WiFi states (Listing 4). These changes allow an app to view the status of all networks and make changes to configured WiFi networks.

The malicious version also introduces a method `a`, that actually collects network and WiFi information, like Mac address and the network state (see Listing 5). This information is then shared using an HTTP request.

All resulting sandboxes also detected the malicious version of the app `ru.qixi.android.smartrabbits`. This particular malware also changes the Android Manifest file, requesting permission to access the location service (Listing 6). This permission allows access to location features, such as the Global Positioning System (GPS) on the phone, if it is enabled. Malicious applications

[4] https://github.com/droidxp/paper-replication-package/blob/master/diff/.

Listing 2: Diffs in the `com.gau.screenguru.finger` AndroidManifest file of the malicious version

```
67:M >    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
68:M >    <uses-permission android:name="android.permission.SEND_SMS"/>
```

Listing 3: Diffs in the malicious version of the class `com.android.main.MainService` (app `com.gau.screenguru.finger`)

```
492:M > localObject2 = (TelephonyManager)getSystemService("phone");
493:M > if (localObject2 != null)
494:M > {
495:M >  this.imei = ((TelephonyManager)localObject2).getDeviceId();
496:M >  this.imsi = ((TelephonyManager)localObject2).getSubscriberId();
497:M >  this.iccid = ((TelephonyManager)localObject2).getSimSerialNumber();
498:M > }
// [...]
519:M > if ("".equals(this.destMobile)) {
520:M >  getDestMobile();
521:M > }
522:M > sendSMS(this.destMobile, "imei:" + this.imei)
```

Listing 4: Diffs in the `com.happymaau.MathRef` AndroidManifest file of the malicious version.

```
165:M >    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
166:M >    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
```

Listing 5: Diffs in the malicious version of the class `com.mn.vymq.b.d` (app `com.happymaau.MathRef`)

```
105:M > private String a(Context paramContext)
106:M > {
107:M > String str = ((TelephonyManager)paramContext.getSystemService("phone")).getDeviceId();
108:M > StringBuilder localStringBuilder = new StringBuilder();
109:M > localStringBuilder.append(str);
110:M > paramContext = (WifiManager)paramContext.getSystemService("wifi");
111:M > if (paramContext == null) {}
112:M >  for (paramContext = null;; paramContext = paramContext.getConnectionInfo())
113:M >  {
114:M >   if (paramContext != null)
115:M >    {
116:M >      paramContext = paramContext.getMacAddress();
117:M >      if (paramContext != null) {
118:M >       localStringBuilder.append(paramContext);
119:M >      }
120:M >    }
121:M >    return a(localStringBuilder.toString());
122:M >  }
123:M > }
```

Listing 6: Diffs in the `com.happymaau.MathRef` AndroidManifest file of the malicious version.

```
8:M >    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
9:M >    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

can use these features to determine where the phone owner is, which is a classic and well-documented privacy threat.

In addition, the malicious app clandestinely monitors the geographic location of the user and sink this information to a web server. Listing 7 shows how the method `c`, from the class named `q`, collects this sensitive information.

This pattern of changing the Android Manifest file and including new method calls characterizes the classes of malwares for which the mining sandbox approach excels. In a different vein, the malicious version of the app `com.andoop.flyracing` is among the apps that none of the sandboxes could detect. Indeed, the malicious version only changes the Android Manifest file, modifying the meta-data `ADMOB_PUBLISHER_ID`. The AdMob is a monetizing service provided by Google, and changing the AdMob *publisher identifier* account redirects the advertisement's revenue to another destination. Based on this observation, we envision integrating a different approach that reasons about modifications to the Android Manifest file and that might complement the mining sandbox approach into the task for detecting malwares; since the mining sandbox approach is not able to detect malicious packages that do not introduce new method calls for sensitive APIs.

Listing 7: Diffs in the malicious version of the class `net.crazymedia.iad.d.q` (app `ru.qixi.android.smartrabbits`)

```
65:M > private Location c(Context paramContext)
66:M > {
67:M > try
68:M >  {
69:M >  if (Arrays.asList(paramContext.getPackageManager().getPackageInfo
              (paramContext.getPackageName(),4096).requestedPermissions).contains
             ("android.permission.ACCESS_FINE_LOCATION"))

70:M >   {
71:M >    paramContext = (LocationManager)paramContext.getSystemService("location");
72:M >    Criteria localCriteria = new Criteria();
73:M >    localCriteria.setAccuracy(1);
74:M >    localCriteria.setAltitudeRequired(false);
75:M >    localCriteria.setBearingRequired(false);
76:M >    localCriteria.setCostAllowed(true);
77:M >    localCriteria.setPowerRequirement(1);
78:M >    paramContext = paramContext.getLastKnownLocation
              (paramContext.getBestProvider(localCriteria, true));

79:M >    return paramContext;
80:M >   }
81:M >  }
82:M >  catch (PackageManager.NameNotFoundException paramContext)
83:M >  {
84:M >   paramContext.printStackTrace();
85:M >   return null;
86:M >  }
87:M >  catch (Exception paramContext)
88:M >  {
89:M >   paramContext.printStackTrace();
90:M >  }
91:M >  return null;
92:M > }
```

Listing 8: Diff in the file `com.andoop.flyracing` AndroidManifest file of the malicious version. `B` stands for the benign version, while `M` stands for the malicious version.

```
1:B < <meta-data android:name="ADMOB_\PUBLISHER_\ID"
                    android:value="a14cf7346295891"/>
---
1:M > <meta-data android:name="ADMOB_\PUBLISHER_\ID"
                    android:value="a14f099bfbf3c61"/>
```

4.2. Results of the second study: Use of taint analysis for malware identification

In this second study we used a taint analysis approach to mine differences between the benign and malicious versions of the 96 Android apps in our dataset. To this end we leverage the FlowDroid tool, which tracks how sensitive information flows through the apps using taint analysis algorithms. Regarding accuracy, the taint analysis approach detected 58 out of the 96 pairs in our dataset (60,42%). That is, using the taint analysis implementation of FlowDroid alone outperforms the Monkey, DroidMate, and Humanoid sandboxes computed in the second execution (without the DroidFax static analysis algorithms). This result shows that static analysis algorithms are promising to complement the mining sandbox approach.

Finding 3. *The performance of FlowDroid to identify malicious behavior is equivalent to the performance of the mining sandbox approach supported by dynamic analysis only—i.e., without the DroidFax static analysis algorithms.*

Additionally, we investigate if we could benefit from combining the static analysis strategies from FlowDroid and DroidFax. Fig. 5 shows a Venn-diagram summarizing the results. So, when combining the results from FlowDroid and DroidFax, we were able to detect 67 of the malicious apps (69.79%), a result compatible to the performance we found as response to the first execution of the test case generation tools—which also considers the DroidFax static analysis algorithms. More interesting, from those 67 malicious apps identified, 33 malwares had been found by both FlowDroid and DroidFax, even though they follow a completely different static analysis approach. Furthermore, FlowDroid shows to be more effective than DroidFax alone, detecting 25 malicious apps that had not been detected by DroidFax (while DroidFax detected 9 malicious apps that had not been detected by FlowDroid). The results of a logistic regression analysis, considering the model $Malware \sim Tool$, where Malware is a response variable indicating if the malware has been detected or not and

Table 3
Results of the Logistic Regression (second study)

	Estimate	*p-value*	*C.I.*
Tool [FlowDroid]	0.4229	0.0428 **	(0.080,0.766)
Tool [DroidFax static analysis component]	−1.2730	0.0000 ***	(−1.560,-0.986)
AIC	334.61		
Num. obs.	288		

*** *p-value* < 0.001; *** *p-value* < 0.05.

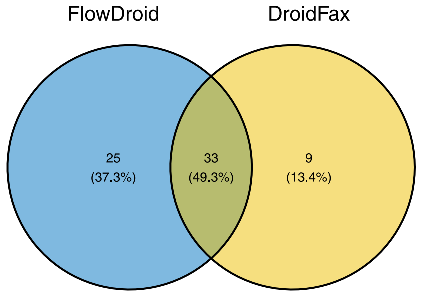

Fig. 5. Venn Diagram highlighting the possible benefits of integrating FlowDroid and DroidFax.

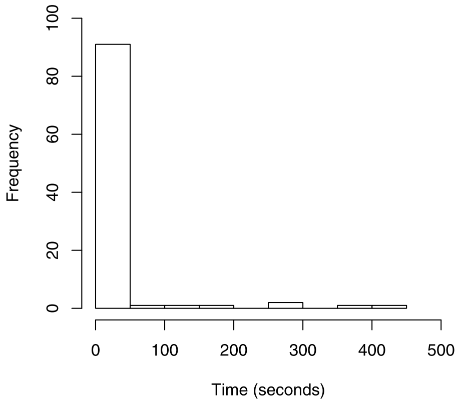

Fig. 6. Histogram summarizing the time to execute FlowDroid.

Tool is either FlowDroid or the sandbox DroidFax static analysis component generates, reveals the existence of a significant difference between the performance of both tools (see Table 3).

> **Finding 4.** *Integrating the results of static analysis tools (such as FlowDroid and DroidFax) seems promising, leading to a performance similar to that achieved when combining test case generation tools with the DroidFax static analysis algorithms.*

The execution of FlowDroid is also feasible: the analysis takes only 32.08 s per app on average, totaling a processing time of 52 min to analyze all 96 pairs of Android apps. Even though the time to execute the FlowDroid analysis depends on the size of the app, the longest run took only 437 s. Fig. 6 summarizes the FlowDroid execution time—which most often concludes the execution in less than 50 s (32.11 s on average, with a standard deviation of 70.04).

Finally, we highlight that FlowDroid was able to detect 4 malwares among the 18 malicious Android apps that had not been detected by the sandboxes constructed in the first study. Among these four malwares, 2 are *trojans*, 1 is an *exploit*, and 1 is an *adware*.

Table 4
Malwares detected in 96 pair (B/M) increased by the taint analysis approach.

Test generation tool	FlowDroid increase	Total	%
DroidBot	6	79	82.29
Monkey	7	78	81.25
DroidMate	7	75	78.12
Humanoid	16	72	75.00
Joker	25	67	69.79

> **Finding 5.** *Although FlowDroid presents a performance similar to that of using the dynamic analysis approach for mining sandboxes, it was able to detect four additional malwares (out of the 18) that had not been detected in the first study.*

5. Implications

The results discussed so far bring evidence that the BLL-Study study overestimated the performance of the dynamic analysis tools in malware identification using the mining sandboxes. This finding has implications for both researchers and practitioners. First, we revisit the literature showing that DroidFax alone is also effective for mining sandboxes, being able to identify 43.75% of the malwares in our dataset. Moreover, DroidFax identifies malwares that none of the generated sandboxes were able to find, increasing the performance of the sandbox in at most 51.79% (in the case of Humanoid).

Table 1 in the previous section summarizes this finding: when executing the mining sandbox approach without the support of DroidFax static analysis, Humanoid's sandbox could identify only 27 malwares (28.12% of the malwares in our dataset). Conversely, the DroidBot sandbox achieved the best performance in terms of the number of detected malware without the DroidFax support for static analysis, being able to identify 63.54% of the malwares. The message here is that researchers and practitioners should explore the use of DroidFax (or a similar tool) in conjunction with dynamic analysis techniques for mining sandboxes—reviewing the findings of the BLL-Study (Bao et al., 2018) and enriching the discussion about the limitations of static analysis for mining sandboxes (Jamrozik et al., 2016).

In the second study we used FlowDroid to explore a novel approach for malware identification, which aims to compare the source–sink paths of two versions of an app (one known to be secure and another that might have been repackage or that might have an injected malicious behavior). Contrasting with the static analysis limitations discussed in (Jamrozik et al., 2016), our findings indicate that this approach is also effective for malware identification. Indeed, our taint analysis approach using FlowDroid detects several malwares that none of the sandboxes generated with the dynamic analysis tools (plus the DroidFax static analysis component) could identify (see Table 4). These result has also implications for both academia and industry. First, this it reinforces the benefits of integrating both static and dynamic analysis for malware identification. Second, this finding suggests that practitioners can benefit from using an integrated approach that combines the mining sandbox approach with taint analysis for malware identification.

6. Threats

As any empirical work, this work also has limitations and threats to its validity. We organize this section using the taxonomy of Wohlin et al. (2012, Chapter 8).

Conclusion Validity is concerned with the issues that might compromise the correct conclusion about the causal relation between the treatment and the outputs of an experiment. The use of inadequate statistical methods and low statistical significance are examples of threats to the conclusion validity. Besides using descriptive statistics and plots, we also leverage binomial logistic regression to support our conclusions in our two empirical studies. Indeed, the results of our logistic regression analysis give evidence about the existence of a true pattern in the data, indicating that the DroidFax static analysis component increases the performance of the sandboxes we built from the execution of the dynamic analysis tools (first study) and that FlowDroid outperforms the DroidFax static analysis component in the task of identifying malwares (second study).

Internal Validity relates to external factors that might impact the independent variables without the researchers' knowledge. Our two empirical studies are *technology-oriented* (Wohlin et al., 2012; Silva et al., 2020), which are not subject to learning effect threats. Nonetheless, due to the random behavior of the test case generation tools, we should not validate the results of this experiment without considering the presence of random events in the execution. To mitigate this threat, we have used a configuration of DroidXP that runs multiple times each tool and computes the average result from those executions. So, we could adequately compare the results of our experiment with the results of the BLL-Study. Beyond that, we tested only 96 of the original 102 pairs of apps in this experiment because the we could not execute those six pairs of apps due to crashes in the Android emulator. However, our goal here is not to conduct an exact replication of the previous work, but actually to better understand how static analysis supports and complements the mining sandbox approach for malware identification.

Construct Validity concerns possible issues that might prevent a researcher to draw a conclusion from the experimental results. The design of our first study involves one treatment (a two-level factor indicating the use or not of the DroidFax static analysis component) and three independent variables: **app id** (96 level factor), the test case generation tool (4-level factor, including DroidBot, DroidMate, Monkey, and Humanoid), and the 3-level factor repetition (we executed every tool three times for all apps, with and without the DroidFax static analysis component). The dependent variable indicates if a malware has been identified by the sandbox of a given test case generation tool built with (or without) the DroidFax static analysis component (in a particular repetition). This design leads to a total of 2304 observations, which is in conformance with the recommendations of Arcuri and Briand (2011) for this kind of experiment. Our second study presents a more straightforward design, comprising a two factor treatment (FlowDroid x the DroidFax static analysis) and the same set of 96 apps of the first study. The dependent variable indicates if a malware has been identified by FlowDroid or by the sandbox the DroidFax static analysis component generates. This design leads to a smaller number of runs, but we still believe that it is sufficient to draw our conclusions (as the results of the logistic regression indicate).

External Validity concerns whether or not the researchers can generalize the results for different scenarios. Our study shares some of the threats the BLL-Study had presented. In particular, here we used the same set of pairs of apps from a *piggy-backed* dataset released by Li et al. (2017a). That is, using this dataset, we could not cover all categories of Android malware. Besides that, we only used a small number of four test case generation tools in this study. To mitigate these threats and enrich the generalization of our research, we make available DroidXP, which does allow future experiments to evaluate other test case generation tools in different malware datasets.

7. Conclusions

In this paper we reported the results of two empirical studies that explore techniques for Android malware identification. The first study is a non-exact replication of a previous research work (Bao et al., 2018), which investigates the Android mining sandbox approach for malware identification. There, Bao et al. report that more than 70% of the malwares in their dataset can be detected by the sandboxes built from the execution of five test case generation tools (such as Monkey and DroidMate). Our replication study revealed that this performance is only achieved if we enable a static analysis component from DroidFax that was supposed to only instrument the Android `apk` files, though that independently contributes to building the sandboxes statically. As such, the use of DroidFax leads to an overestimation of the performance of the mining sandbox approach supported by dynamic analysis. Indeed, the execution of DroidFax alone enabled us to generate a sandbox that can identify 43.75% of the malwares from their dataset.

In the second study we investigated a new approach based on taint analysis for malware identification, which leads to promising results. First, the taint based static analysis approach detected 60.42% of the malwares in the dataset. When combining taint analysis with the mining sandbox approach, we were able to identify 82.29% of the malwares in the dataset. These results have implications for both researchers and practitioners. First, we review the literature showing, for the first time, empirical evidence that the mining sandbox approach benefits from using both dynamic and static analysis. Second, practitioners can improve malware identification using a combination of the mining sandbox approach with taint analysis. Nonetheless, both the mining sandbox approach and taint analysis present limitations. In particular, we are not able to identify a malware that uses the same set of calls to sensitive APIs of the benign version of an app, using the mining sandbox approach. Similarly, we are not able to identify a malware that presents the same paths from sources to sinks of the corresponding benign version of an app, using the taint analysis approach. To mitigate these limitations, we envision the use of other approaches—such as machine learning algorithms to classify changes in non-code assets (e.g., Android manifest files) and symbolic execution to differentiate malicious calls or source–sink paths.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Allix, K., Bissyandé, T.F., Klein, J., Traon, Y.L., 2016. AndroZoo: collecting millions of Android apps for the research community. In: Kim, M., Robbes, R., Bird, C. (Eds.), Proceedings of the 13th International Conference on Mining Software Repositories. MSR 2016, Austin, TX, USA, May 14–22, 2016, ACM, pp. 468–471. http://dx.doi.org/10.1145/2901739.2903508.

Ammons, G., Bodík, R., Larus, J.R., 2002. Mining specifications. In: Launchbury, J., Mitchell, J.C. (Eds.), Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland, OR, USA, January 16–18, 2002, ACM, pp. 4–16. http://dx.doi.org/10.1145/503272.503275.

Anon, 2020a. Logcat. https://developer.android.com/tools/help/logcat.html. (Accessed 15 March 2020).

Anon, 2021. statcounter. https://gs.statcounter.com/os-market-share/mobile/worldwide. (Accessed 10 February 2021).

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, Association for Computing Machinery, New York, NY, USA, pp. 1–10. http://dx.doi.org/10.1145/1985793.1985795.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le, Y., Octeau, D., McDaniel, P., 2014a. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14, Association for Computing Machinery, New York, NY, USA, pp. 259–269. http://dx.doi.org/10.1145/2594291.2594299.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.D., 2014b. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: O'Boyle, M.F.P., Pingali, K. (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14, Edinburgh, United Kingdom - June 09–11, 2014, ACM, pp. 259–269. http://dx.doi.org/10.1145/2594291.2594299.

Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P., 2013. AppGuard - fine-grained policy enforcement for untrusted Android applications. In: García-Alfaro, J., Lioudakis, G.V., Cuppens-Boulahia, N., Foley, S.N., Fitzgerald, W.M. (Eds.), Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM 2013, and 6th International Workshop. SETOP 2013, Egham, UK, September 12–13, 2013, Revised Selected Papers, In: Lecture Notes in Computer Science, vol. 8247, Springer, pp. 213–231. http://dx.doi.org/10.1007/978-3-642-54568-9_14.

Bao, L., Le, T.B., Lo, D., 2018. Mining sandboxes: Are we there yet? In: Oliveto, R., Penta, M.D., Shepherd, D.C. (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering. SANER 2018, Campobasso, Italy, March 20–23, 2018, IEEE Computer Society, pp. 445–455. http://dx.doi.org/10.1109/SANER.2018.8330231.

Bordoni, L., Conti, M., Spolaor, R., 2017. Mirage: Toward a stealthier and modular malware analysis sandbox for Android. In: Foley, S.N., Gollmann, D., Snekkenes, E. (Eds.), Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11–15, 2017, Proceedings, Part I. In: Lecture Notes in Computer Science, vol. 10492, Springer, pp. 278–296. http://dx.doi.org/10.1007/978-3-319-66402-6_17.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture - Volume 1: A System of Patterns. Wiley Publishing.

Cai, H., Ryder, B.G., 2017. DroidFax: A toolkit for systematic characterization of Android applications. In: 2017 IEEE International Conference on Software Maintenance and Evolution. ICSME 2017, Shanghai, China, September 17–22, 2017, IEEE Computer Society, pp. 643–647. http://dx.doi.org/10.1109/ICSME.2017.35.

Comscore, I., 0000. Comscore, https://www.comscore.com/Insights/Presentations-and-Whitepapers/2018/Global-Digital-Future-in-Focus-2018.

da Costa, F.H., Medeiros, I., Costa, P., Menezes, T., Vinícius, M., Bonifácio, R., Canedo, E.D., 2020. DroidXP: A benchmark for supporting the research on mining Android sandboxes. In: 20th IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM 2020, Adelaide, Australia, September 28–October 2, 2020, IEEE, pp. 143–148. http://dx.doi.org/10.1109/SCAM51674.2020.00021.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P.D., Sheth, A., 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Arpaci-Dusseau, R.H., Chen, B. (Eds.), 9th USENIX Symposium on Operating Systems Design and Implementation. OSDI 2010, October 4–6, 2010, Vancouver, BC, Canada, Proceedings, USENIX Association, pp. 393–407, URL http://www.usenix.org/events/osdi10/tech/full_papers/Enck.pdf.

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M., 2015. Android security: A survey of issues, malware penetration, and defenses. IEEE Commun. Surv. Tutor. 17 (2), 998–1022. http://dx.doi.org/10.1109/COMST.2014.2386139.

Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.A., 2011. Android permissions demystified. In: Chen, Y., Danezis, G., Shmatikov, V. (Eds.), Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS 2011, Chicago, Illinois, USA, October 17–21, 2011, ACM, pp. 627–638. http://dx.doi.org/10.1145/2046707.2046779.

Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.A., 2012. Android permissions: user attention, comprehension, and behavior. In: Cranor, L.F. (Ed.), Symposium on Usable Privacy and Security. SOUPS '12, Washington, DC, USA - July 11–13, 2012, ACM, p. 3. http://dx.doi.org/10.1145/2335356.2335360.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA.

Glanz, L., Müller, P., Baumgärtner, L., Reif, M., Amann, S., Anthonysamy, P., Mezini, M., 2020. Hidden in plain sight: Obfuscated strings threatening your privacy. CoRR, arXiv:2002.04540.

Huang, W., Dong, Y., Milanova, A., Dolby, J., 2015. Scalable and precise taint analysis for Android. In: Young, M., Xie, T. (Eds.), Proceedings of the 2015 International Symposium on Software Testing and Analysis. ISSTA 2015, Baltimore, MD, USA, July 12–17, 2015, ACM, pp. 106–117. http://dx.doi.org/10.1145/2771783.2771803.

Hurier, M., Suarez-Tangil, G., Dash, S.K., Bissyandé, T.F., Traon, Y.L., Klein, J., Cavallaro, L., 2017. Euphony: harmonious unification of cacophonous anti-virus vendor labels for Android malware. In: Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, pp. 425–435.

James, G., Witten, D., Hastie, T., Tibshirani, R., 2014. An introduction to statistical learning: With applications in R. Springer Publishing Company, Incorporated.

Jamrozik, K., von Styp-Rekowsky, P., Zeller, A., 2016. Mining sandboxes. In: Dillon, L.K., Visser, W., Williams, L.A. (Eds.), Proceedings of the 38th International Conference on Software Engineering. ICSE 2016, Austin, TX, USA, May 14–22, 2016, ACM, pp. 37–48. http://dx.doi.org/10.1145/2884781.2884782.

Jamrozik, K., Zeller, A., 2016. DroidMate: a robust and extensible test generator for Android. In: Proceedings of the International Conference on Mobile Software Engineering and Systems. MOBILESoft '16, Austin, Texas, USA, May 14–22, 2016, ACM, pp. 293–294. http://dx.doi.org/10.1145/2897073.2897716.

Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M., 2018. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. In: Millstein, T.D. (Ed.), 32nd European Conference on Object-Oriented Programming. ECOOP 2018, July 16–21, 2018, Amsterdam, the Netherlands, In: LIPIcs, vol. 109, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 10:1–10:27. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2018.10.

Li, L., Li, D., Bissyandé, T., Klein, J., Traon, Y.L., Lo, D., Cavallaro, L., 2017a. Understanding Android app piggybacking: A systematic study of malicious code grafting. IEEE Trans. Inf. Forensics Secur. 12 (6), 1269–1284. http://dx.doi.org/10.1109/TIFS.2017.2656460.

Li, Y., Yang, Z., Guo, Y., Chen, X., 2017b. DroidBot: a lightweight UI-guided test input generator for Android. In: Uchitel, S., Orso, A., Robillard, M.P. (Eds.), Proceedings of the 39th International Conference on Software Engineering. ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume, IEEE Computer Society, pp. 23–26. http://dx.doi.org/10.1109/ICSE-C.2017.8.

Li, Y., Yang, Z., Guo, Y., Chen, X., 2019. Humanoid: A deep learning-based approach to automated black-box Android app testing. In: 34th IEEE/ACM International Conference on Automated Software Engineering. ASE 2019, San Diego, CA, USA, November 11–15, 2019, IEEE, pp. 1070–1073. http://dx.doi.org/10.1109/ASE.2019.00104.

Maass, M., Sales, A., Chung, B., Sunshine, J., 2016. A systematic analysis of the science of sandboxing. PeerJ Comput. Sci. 2, e43. http://dx.doi.org/10.7717/peerj-cs.43.

Monkey, 2020b. https://developer.android.com/studio/test/monkey. (Accessed 10 February 2020).

Pauck, F., Bodden, E., Wehrheim, H., 2018. Do Android taint analysis tools keep their promises? In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (Eds.), Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04–09, 2018, ACM, pp. 331–341. http://dx.doi.org/10.1145/3236024.3236029.

Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.D., 2019. CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. CCS '19, Association for Computing Machinery, New York, NY, USA, pp. 2455–2472. http://dx.doi.org/10.1145/3319535.3345659.

Rasthofer, S., Arzt, S., Bodden, E., 2014. A machine-learning approach for classifying and categorizing Android sources and sinks. In: 21st Annual Network and Distributed System Security Symposium. NDSS 2014, San Diego, California, USA, February 23–26, 2014, The Internet Society, URL https://www.ndss-symposium.org/ndss2014/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks.

Shull, F., Carver, J.C., Vegas, S., Juzgado, N.J., 2008. The role of replications in empirical software engineering. Empir. Softw. Eng. 13 (2), 211–218. http://dx.doi.org/10.1007/s10664-008-9060-1.

Silva, E., Leite, A.F., Alves, V., Apel, S., 2020. ExpRunA: a domain-specific approach for technology-oriented experiments. Softw. Syst. Model. 19 (2), 493–526. http://dx.doi.org/10.1007/s10270-019-00749-6.

Sufatrio, Tan, D.J.J., Chua, T., Thing, V.L.L., 2015. Securing Android: A survey, taxonomy, and challenges. ACM Comput. Surv. 47 (4), 58:1–58:45. http://dx.doi.org/10.1145/2733306.

Wang, H., Li, H., Li, L., Guo, Y., Xu, G., 2018. Why are Android apps removed from Google Play?: a large-scale empirical study. In: Zaidman, A., Kamei, Y., Hill, E. (Eds.), Proceedings of the 15th International Conference on Mining Software Repositories. MSR 2018, Gothenburg, Sweden, May 28–29, 2018, ACM, pp. 231–242. http://dx.doi.org/10.1145/3196398.3196412.

Whaley, J., Martin, M.C., Lam, M.S., 2002. Automatic extraction of object-oriented component interfaces. In: Frankl, P.G. (Ed.), Proceedings of the International Symposium on Software Testing and Analysis. ISSTA 2002, Roma, Italy, July 22–24, 2002, ACM, pp. 218–228. http://dx.doi.org/10.1145/566172.566212.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A., 2012. Experimentation in software engineering. Springer Publishing Company, Incorporated.

Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., Xie, T., 2016. Automated test input generation for Android: are we really there yet in an industrial case? In: Zimmermann, T., Clel-Huang, J., Su, Z. (Eds.), Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, Seattle, WA, USA, November 13–18, 2016, ACM, pp. 987–992. http://dx.doi.org/10.1145/2950290.2983958.

Zhang, J., Tian, C., Duan, Z., 2021. An efficient approach for taint analysis of android applications. Comput. Secur. 104, 102161. http://dx.doi.org/10.1016/j.cose.2020.102161.