Noname manuscript No.
(will be inserted by the editor)

TaintBench: Automatic Real-World Malware Benchmarking of Android Taint Analyses

Linghui Luo [16] · Felix Pauck [1] · Goran Piskachev[2] · Manuel Benz[1] · Ivan Pashchenko[3] · Martin Mory [1] · Eric Bodden [12] · Ben Hermann [4] · Fabio Massacci [35]

Received: date / Accepted: date

Abstract Due to the lack of established real-world benchmark suites for static taint analyses of Android applications, evaluations of these analyses are often restricted and hard to compare. Even in evaluations that do use real-world apps, details about the ground truth in those apps are rarely documented, which makes it difficult to compare and reproduce the results. To push Android taint analysis research forward, this paper thus recommends criteria for constructing real-world benchmark suites for this specific domain, and presents TaintBench, the first real-world *malware* benchmark suite with documented taint flows. TaintBench benchmark apps include taint flows with complex structures, and addresses static challenges that are commonly agreed on by the community. Together with the TaintBench suite, we introduce the TaintBench framework, whose goal is to simplify real-world benchmarking of Android taint analyses. First, a usability test shows that the framework improves experts' performance and perceived usability when documenting and inspecting taint flows. Second, experiments using TaintBench reveal new insights for the taint analysis tools Amandroid and FlowDroid: (i) They are less effective on real-world malware apps than on synthetic benchmark apps. (ii) Predefined lists of sources and sinks heavily impact the tools' accuracy. (iii)

The first two authors contributed equally to this research.

Linghui Luo
E-mail: linghui.luo@upb.de

Felix Pauck
E-mail: fpauck@mail.upb.de

[1]Department of Computer Science, Paderborn University, Paderborn, Germany
[2]Fraunhofer IEM, Paderborn, Germany
[3]Department of Information Sciences and Engineering, University of Trento, Trento, Italy
[4]Department of Computer Science, Technical University of Dortmund, Dortmund, Germany
[5]Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
[6]Corresponding author

Surprisingly, up-to-date versions of both tools are less accurate than their predecessors.

Keywords Taint Analysis · benchmark · real-world benchmark · Android Malware

Acknowledgements We would like to thank Christian Brüggeman and Markus Schmidt for their assistance in the manual inspection and Lisa Nguyen Quang Do for discussions in the early stage of this project. This research was supported by the research training group Human Centered Systems Security (NERD.NRW) sponsored by the state of North Rhine-Westphalia in Germany. It was also partly supported by the European Union H2020 programme under grant agreement 952647 (AssureMOSS) and grant agreement 830929 (CyberSec4Europe).

1 Introduction

Mobile devices store and process sensitive data such as contact lists or banking information, which require protection against misuse. In case of Android, the most-used mobile operating system (statcounter, 2019), and its app marketplaces such as Google Play Store, it is crucial to protect users' security and privacy. Hence, in case of most marketplaces, any app trying to enter is automatically reviewed. Numerous malware detection mechanisms have been developed to do so (Wei et al., 2014; Arzt et al., 2014; Enck et al., 2014; Gordon et al., 2015; Grech and Smaragdakis, 2017; Youssef and Shosha, 2017). Nonetheless, frequent news reporting malware apps bypassing such mechanisms and lurking into marketplaces show that this process sometimes fails (Soni, 2020; Micro, 2020).

Static taint analysis, in particular, is able to detect security threats, e.g., data leaks (as in spyware which is a subset of malware), before they are actually exploited. It tracks data flows from sensitive *sources* (e.g., API which reads the contact list) to sensitive *sinks* (e.g., API which posts data to the Internet). Such data flows between sources and sinks are called *taint flows*. Note, multiple intentionally, accidentally or maliciously programmed data-flow paths might result in the same taint flow as the example in Listing 1 shows, hence, a taint flow is usually counted as detected once a connection consisting of a single or multiple data-flow paths between the associated source and sink is found.

```
1 void onCreate(){
2     A a = new A(); B b = new B();
3     a.f = source();//source
4     if (a.f.startsWith("0"))
5         b.f1 = a.f;//on data-flow path 1
6     else
7         b.f2 = a.f;//on data-flow path 2
8     leak(b);//sink
9 }
```

Listing 1 Two data-flow paths result in one taint flow from the source to the sink.

```
1 void onCreate() {
2     String[ ] arr = new String[3];
3     arr[0] = "Hello";
4     arr[1] = source();//source
5     leak(arr[1]);//expected taint flow
6     leak(arr[0]);//unexpected taint flow
7
8 }
```

Listing 2 Expected taint flow (true leak at line 5) vs. unexpected taint flow (no leak at line 6).

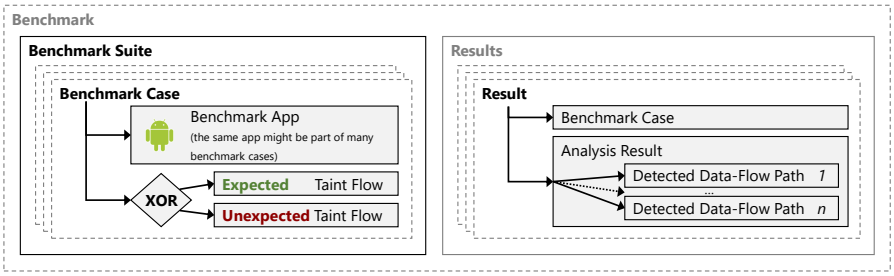


Fig. 1 Left: a benchmark suite consists of a set of benchmark cases. Each benchmark case is a combination of a benchmark app and a specified expected/unexpected taint flow. Right: a tool's analysis result on a benchmark app is a set of data-flow paths connecting sources and sinks. These data-flow paths are compared to flows specified in the benchmark cases.

To be accurate, taint analyses must evolve along with the Android operating system without losing accuracy when analyzing apps designed for older systems. Hence, each year novel tools, or new versions of existing tools that realize taint analysis implementations become available. To show the relative effectiveness of each new analysis prototype, its authors are expected to evaluate it empirically. Fortunately, there exist a few well-established benchmark suites for this purpose, e.g., DroidBench (Arzt et al., 2014), SecuriBench (Livshits and Lam, 2005) and ICC-Bench (Wei et al., 2014).

The terms used in the context of such benchmark suites as well as their structure is visualized in Figure 1. The typical usage of these benchmark suites and the issues coming along with it are described in the following. First, all the benchmark suites enumerated before are sets of micro *benchmark apps* — small programs which are artificially constructed for benchmarking purposes only.

Multiple *expected* or *unexpected* taint flows are defined for and implemented in each benchmark app. In Listing 2, an expected flow from line 4 to line 5 is defined, as well as an unexpected flow from line 4 to line 6. The expected flow in this example specifies a true data leak, while the unexpected flow specifies a false-positive case on which an imprecise tool might still report (e.g., a tool overapproximates for arrays and taints the whole array once a tainted value is written into an array.). Established benchmark suite often uses unexpected cases to assess the precision of a tool (e.g., `BenchmarkTest00009` in the OWASP Benchmark (OWASP, 2021) and `ArrayAccess1.apk` in DroidBench (DroidBench 3-0, 2016)). Once an Android taint analysis tool finds an expected taint flow while benchmarking, it is counted as a *true positive* (TP). A missed expected taint flow is counted as a *false negative* (FN). Consequently, finding and missing unexpected taint flows are counted as *false positives* (FP) and *true negatives* (TN) respectively. We call the combination of one (un-)expected taint flow and one benchmark app a *benchmark case* (see Figure 1).

Evaluations of Android taint analyses frequently use micro benchmark apps as representatives of real-world apps (Cam et al., 2016; Bohluli and Shahriari,

2018; Pauck and Zhang, 2019; Zhang et al., 2019; Wei et al., 2014; Arzt et al., 2014). To evaluate, the *analysis result* — computed for each benchmark case — is compared against the associated taint flow which was expected or not expected to be found. While doing so, TPs, TNs, FPs and FNs are counted. For example, when the actual analysis result contains an expected taint flow it is counted as a TP. Based on these countings, the benchmark outcome is then usually evaluated with respect to accuracy in terms of the metrics precision, recall, and F-measure. Additionally, the analysis time required per benchmark case or app is recorded in most evaluations to argue about an implementation's efficiency and scalability. This way analysis tools are steadily adapted to achieve better accuracy scores and run more efficiently for established micro benchmark suites. Multiple studies indicate that the scores achieved only hold for micro benchmarks apps and cannot be achieved when analyzing real-world apps (Qiu et al., 2018; Pauck et al., 2018; Luo et al., 2019a). Thus, many tools show an *over-adaption* to micro benchmark suites.

In contrast to evaluations on micro benchmark suites, evaluations on real-world apps are uncommon and — due to missing or undocumented details — usually not reproducible. For instance, the authors of DroidSafe (Gordon et al., 2015) evaluated both DroidSafe and FlowDroid (Arzt et al., 2014) on 24 real-world Android malware apps. However, information about malicious taint flows is only documented in form of types of sources and sinks (e.g., source type is location and sink type is network). Missing details about code locations related to these flows makes it impossible to reproduce the results, which hinders the measurement of research progress. Additionally, a documentation of *all* expected and unexpected taint flows (*ground truth*) for a set of real-world apps rarely exists since the only tools which could be used as oracles to determine these flows are the tools to be evaluated. Thus, the associated evaluation results often come unchecked and only comprise an enumeration of countable findings (e.g., data-flow paths found) without guarantees that the findings actually represent feasible taint flows (Avdiienko et al., 2015; Arzt et al., 2014).

The lack of publicly available real-world benchmark suites with a well-documented ground truth hinders progress in Android taint analysis research. This paper fills this gap. It first defines a set of sensible construction criteria for such a benchmark suite. It further proposes the TaintBench benchmark suite designed to fulfill these construction criteria. The suite comes with a set of real-world malware apps and precisely hand-labeled expected and unexpected taint flows, the so-called *baseline definition*. During a rigorous and long-lasting construction process three field-experts agreed that this baseline definition forms a valid subset of the ground truth. Along with the suite, this paper introduces the TaintBench framework, which allows a faster benchmark-suite construction, a reproducible evaluation of analysis tools on this suite, and easier inspection of analysis results. Through a usability test, we were able to confirm that the framework effectively assists experts in documenting and inspecting taint flows. Last but not least, we compared TaintBench to DroidBench and then used both benchmark suites to evaluate current and previously eval-

uated versions of Amandroid (Wei et al., 2014) and FlowDroid (Arzt et al., 2014). While we were able to reproduce results previously obtained on DroidBench, using TaintBench we find that (1) Amandroid and FlowDroid have shortcomings — especially low recall — on real-world malware apps, (2) predefined lists of sources and sinks heavily impact the tools' performance and there is no perfect list for TaintBench, (3) surprisingly, up-to-date versions of both tools are less accurate than their predecessors. Particularly they find fewer actual taint flows. Regarding FlowDroid this seems to happen due to a bug causing sources and sinks that are actually irrelevant for specific taint flows to have a shadow effect on FlowDroid's taint computation.

To summarize, this paper makes the following contributions:

- the first real-world malware benchmark suite with a documented baseline: TaintBench (39 malware apps with 203 expected and 46 unexpected taint flows documented in a machine-readable format),
- the TaintBench framework, which allows tool-assisted benchmark suite construction, evaluation and inspection,
- a usability test for evaluating the efficiency and usability of tools from the TaintBench framework,
- a comparison of TaintBench and DroidBench, and
- an evaluation of current and previously evaluated versions of FlowDroid and Amandroid using both DroidBench and TaintBench.

All artifacts contributed with this paper are publicly available:

`https://TaintBench.github.io`

The rest of the paper is organized as follows. We first discuss related work in Section 2. We propose the criteria for constructing real-world benchmarks for Android taint analysis and explain the concrete construction process in Section 3. We introduce the TaintBench framework which assists real-world benchmarking of Android taint analyses and show its effectiveness by presenting a usability test with experts in Section 4. The constructed TaintBench suite, its evaluation and results of benchmarking Android taint analysis tools with both DroidBench and TaintBench are presented in Section 6. Threats to validity and a conclusion are given in Section 8 and Section 9.

2 Related Work

In the area of Android taint analysis there exist many static (Arzt et al., 2014; Wei et al., 2014; Gordon et al., 2015; Li et al., 2015; Bosu et al., 2017), dynamic (Enck et al., 2014), and hybrid (Benz et al., 2020; Pauck and Wehrheim, 2019) analysis tools as well as a couple of benchmark suites (Arzt et al., 2014; Wei et al., 2014; Mitra and Ranganath, 2017). We highlight the most prominent static analysis tools and benchmark suites with respect to taint analyses.

FlowDroid (Arzt et al., 2014), Amandroid (Wei et al., 2014), IccTA (Li et al., 2015) and DroidSafe (Gordon et al., 2015) are the most cited static Android taint analysis tools. Amandroid, FlowDroid and IccTA use configurable lists of sources and sinks to be considered during analysis. SuSi (Rasthofer

et al., 2014) is a machine-learning approach developed to automatically create such lists by inspecting the Android APIs. More comprehensive or precise lists were produced in more recent research (Piskachev et al., 2019). DroidSafe's list of sources and sinks is hard-coded in its source code, which makes it hard to adapt for real-world apps. While DroidSafe and IccTA are not maintained anymore, FlowDroid and Amandroid still appear to receive frequent updates (Amandroid*, 2018; FlowDroid*, 2019). Furthermore, all tools support different features and sensitivities that influence the precision and soundness. FlowDroid and Amandroid, for example, are context-, flow-, field-, object-sensitive and lifecycle-aware. Only IccTA and Amandroid support the analysis of inter-component communication (ICC). None of the tools is path-sensitive due scalability drawbacks and their static nature. Table 1 shows an overview of the main characteristics of these tools. Evaluations of the abilities of each tool can be found in previous studies (Qiu et al., 2018; Pauck et al., 2018).

The most used (cited) and hence most established benchmark suite in this field of research is DroidBench (Arzt et al., 2014). DroidBench is a collection of artificial apps that forms a micro benchmark suite. Its ground-truth description can be found in code comments in the source code associated with each benchmark app. The up-to-date version 3.0 (DroidBench 3-0, 2016) comprises 190 apps with benchmark cases in 18 different categories related to the features and sensitivities exploited. Subsets, variants and extensions of DroidBench have been used to evaluate certain features or more specialized taint analysis tools (Wei et al., 2014; Bosu et al., 2017). For example, ICC-Bench (Wei et al., 2014) comprises benchmark cases to evaluate the abilities of analyses to handle inter-component communication. A recent suite is DroidMacroBench (Benz et al., 2020) — a collection of 12 real-world commercial Android apps with annotated taint flows reported by FlowDroid. However, the authors only labeled the taint flows as feasible (i.e., it is possible for data to flow from a given source to a given sink) or infeasible without characterizing or giving details about the flows due to the high complexity of commercial apps. For example, it remains unclear whether the tainted data is sensitive. Thus,

Table 1 Overview of the Main Characteristics of Relevant Static Taint Analysis Tools for Android Applications

Tool	*configurable sources and sinks*	*actively maintained*	*ICC*	*context-sensitive*	*flow-sensitive*	*field-sensitive*	*object-sensitive*	*path-sensitive*	*lifecycle-aware*
FlowDroid	✔	✔	✘	✔	✔	✔	✔	✘	✔
Amandroid	✔	✔	✔	✔	✔	✔	✔	✘	✔
IccTA	✔	✘	✔	✔	✔	✔	✔	✘	✔
DroidSafe	✘	✘	✘	✔*	✘	✔	✔	✘	✔

*: static method only

Table 2 Overview of the Main Characteristics of Relevant Android Benchmark Suites

Benchmark Suite	Real-world apps	Number of apps	Open source	Ground Truth
DroidBench	✘	190	✔	Comments in source code
DroidMacroBench	✔	12	✘	Jimple code labeled as in-/feasible
Ghera	✘	180	✔	README file

regarding security aspects, many feasible labeled taint flows might not be real security threats. Moreover, because DroidMacroBench comprises closed-source apps, the authors could not legally make the suite publicly available as open source. Due to these two limitations, it cannot be used as publicly accessible and comparable proving ground truth for taint analyses.

Ghera (Mitra and Ranganath, 2017) is a repository of micro benchmark apps sorted into different categories of Android vulnerabilities partially including taint flows. As of January 2021, Ghera contains 8 categories with 60 vulnerabilities where each one contains three apps, a benign, a malicious and a secure one.[1] The ground truth is documented in a text file that hold a natural language description of the vulnerability within the specific app. Table 2 summarizes DroidBench, DroidMacroBench, and Ghera. The Ghera benchmark apps were used in a recent study (Ranganath and Mitra, 2020) that evaluated the effectiveness of existing security analysis tools in detection of security vulnerabilities. They found out that the evaluated tools could only detected a small number of vulnerabilities in the Ghera benchmark apps. To that effect their findings support some findings provided by us during evaluation (see Section 6). However, not all of their benchmark apps are suitable for benchmarking taint analysis tools. For example, the apps in the Crypto catalog of Ghera contains cryptographic misuses that require typestate analysis rather than taint analysis. Furthermore, our study answers questions raised in their study such as whether these taint analysis tools are actually effective in detecting real-world issues without extra configuration (see Open Questions 3&4 in (Ranganath and Mitra, 2020)).

Pauck et al. (Pauck et al., 2018) proposed ReproDroid to refine, execute, and evaluate on benchmark suites. Among other suites they refined DroidBench such that each benchmark app now comes with a precisely defined, machine-readable ground truth in *AQL* (Android App Analysis Query Language) format. We include and extend ReproDroid for enabling automatic evaluation of analysis tools in our TaintBench framework as described in Section 4.2.

3 Construction Criteria

This section describes how we constructed a real-world malware benchmark suite for Android taint analysis. Before doing so and since there exists no widely accepted real-world benchmark suite for this purpose, we propose the following three criteria for benchmark suite construction.

[1] `https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks`

I) Ground-Truth Documentation: Mitra et al. proposed the *Well Documented* benchmark characteristic — benchmarks should be accompanied by relevant documentation (Mitra and Ranganath, 2017). In context of our work, each benchmark app comes with a documentation of the expected and unexpected taint flows for benchmarking. Such documentation was provided by DroidBench, however, only in form of code comments that mostly hold natural language descriptions. It lacks information about the exact code locations of the taint flows. Pauck et al. (Pauck et al., 2018) pointed out that such documentation could lead to incorrect evaluation of analysis results. Thus, on top of the "Well Documented" characteristic, the taint flows of each benchmark app should be documented in a standard machine-readable format such as XML or JSON. It should contain both high-level textual information which describes the purpose of each taint flow (as in DroidBench's comments) and exact code locations of the source, the sink, and the intermediate statements of each flow. Furthermore, since different taint analysis tools may use different intermediate representations (IRs) the format must support an encoding of code locations that can be converted into arbitrary IR code locations.

However, to create such a ground truth is difficult in case of real-world apps. This is particularly true for taint flows that even tools cannot detect. Thus, *our goal* here is to create a *baseline definition* (i.e., a subset of all expected and unexpected taint flows) for each benchmark app. Regarding expected taint flows, we focus on those flows which are not only *feasible* (i.e., it is possible for data to flow from a given source to a given sink) but also *critical* under security aspects, e.g., leaking sensitive information. Our work aims to serve as a starting point towards a solid real-world benchmark suite for Android taint analysis. The facilities we put in place should allow (and possibly foster) extension and improvement of the suite.

II) Representativeness: Nguyen Quang Do et al. recommended representativeness with respect to the target domain of the evaluated tool or analysis as an important aspect for benchmark selection (Do et al., 2016). In this paper, we choose to focus on Android apps *identified as malware*. There are several reasons for this decision. First, some available Android malware datasets come with descriptions of the malicious behaviors, which are unavailable for open-source datasets (e.g., F-Droid (F-Droid, 2020)) or commercial applications. Such descriptions accelerate the manual inspection process, for example, when faced with the manual task of separating true from false findings produced by automated tools, since the descriptions provide hints of the malicious behavior. Labels such as malware families or types are insufficient to drive this process. Second, well-known Android malware datasets have often been used in evaluations in scientific papers (Wong and Lie, 2016; Zheng et al., 2012; Rastogi et al., 2013; Huang et al., 2014) including Android taint analysis approaches (Huang et al., 2015; Yang et al., 2016). Last but not least, Android malware is less likely to cause licensing issues. A benchmark suite must be open-source and publicly available, and thus cannot legally include commercial apps. Because of including closed-source, commercial apps in the DroidMacroBench suite,

the authors could not make the suite publicly available (Benz et al., 2020). The meaning of representativeness is twofold in our paper:

- the expected taint flows in the benchmark suite should be *representative* of taint flows that address static-analysis challenges. These challenges are commonly agreed on by the community such as field-sensitivity or the necessity to model implicit control flows through the application's lifecycle. A good example is DroidBench, which groups its benchmark cases into 18 categories based on such challenges, e.g., aliasing, callbacks and reflection.
- the benchmark apps should be *representative* of the dataset it is sampled from. Reif et al. provide a tool to generate metrics for Java programs to assess representativeness during benchmark creation (Reif et al., 2017). Similarly, we define a set of metrics which are relevant for Android taint analysis benchmarking in this paper. Details are introduced in Section 6.1.

III) Human-understandable Source Code: Source code availability has been widely used in previous benchmark works (Blackburn et al., 2006; Prokopec et al., 2019; Mitra and Ranganath, 2017). Whenever possible, the benchmark suite should provide human-understandable source code (either directly or by decompilation) in addition to compiled executables. This criterion is important for the following three reasons. First, it can help users of the benchmark suite to understand the documented taint flows. Second, it allows the inspection of potential false positives produced by automated tools. Lastly, it enables the community to do source-code level analysis such that the baseline definition can be checked, improved and extended. Considering our focus on malware, source code is naturally hard to come by. In the following we will elaborate how we address this challenge.

3.1 Concrete Construction of the TaintBench suite

For the suite's construction, we decided to use available Android malware datasets, since they are very likely to contain malicious taint flows that can and should be detected by Android taint analysis tools. Note that malicious taint flows are not equal to taint-style vulnerabilities. Malicious taint flows are intentionally built into the apps by malware writers, while vulnerabilities are weaknesses in benign apps that are unintentionally caused by design flaws or implementation bugs. To obtain suitable malware samples to be included in TaintBench, we compared well-known Android malware datasets as shown

Table 3 Comparison of Android Malware Datasets

Dataset	# App	Malware Info	Last Update
Contagio (Contagio Mobile, 2012)*	344	Behavior descriptions	2018
AMD (Wei et al., 2017)	24,533	Behavior descriptions	2016**
VirusShare (VirusShare, 2014)	34,265,389	Labels	2019
Drebin (Arp et al., 2014)	5,560	Labels	2012

*: Online source (Contagio Mobile Malware, 2018) (accessed 02/18/2021),
**: Currently unavailable (02/18/2021)

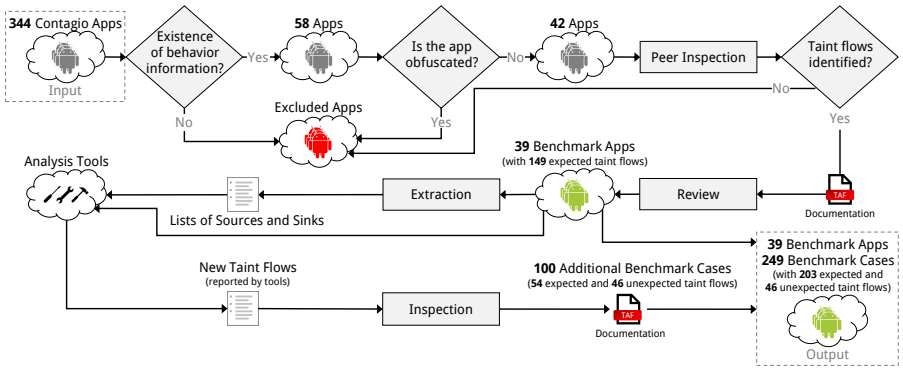


Fig. 2 Benchmark Suite Construction Process

in Table 3. Considering manual inspection required for identifying the taint flows, we prefer datasets which have more detailed information such as behavior description, i.e., Contagio (Contagio Mobile, 2012) and AMD (Wei et al., 2017). From these two, we then chose the Contagio dataset, since it was updated more recently and its size allows us to qualitatively study all samples.

Because original source code is not available for the apps in Contagio, we opted to decompile the Android malware apps. Modern Android decompilation technology has been improved such that high-level source code files can be reconstructed successfully in most cases. Decompilation is widely used in reverse engineering and validation of software analysis results for closed-source applications (Luo et al., 2019a; Benz et al., 2020). Another issue was that some applications in the dataset were obfuscated (e.g., class/method/parameter names were renamed to “a”, “bbb” etc.) such that the decompiled code was very difficult for humans to understand . Considering the difficulty of formulating high-level descriptions for discovered taint flows (as we stated in the documentation criterion) in obfuscated applications and to ease the future validation of the baseline definition by other researchers, we excluded obfuscated applications from our selection. Nonetheless we argue that our selection is not biased, as later shown in Section 6, our selection is a representative subset of the Contagio dataset.

Figure 2 shows our benchmark suite construction process. The Contagio dataset contains 344 apps, only 58 of them have references to behavior information. From these 58 apps, 42 apps that are not obfuscated became candidates for taint-flow inspection. Initially, we planned to apply existing Android taint analysis tools to the apps and manually check the analysis results, but we quickly gave up on this plan due to the following reasons:

- Too many flows to be checked. The three tools (Amandroid, DroidSafe and FlowDroid) we initially tried already reported 21,623 flows.
- False negatives remain undetected by tools. We manually inspected a few malware apps. As our inspection reveals, the tools frequently miss critical taint flows which are part of the actual malicious intentions of the malware

apps (e.g., leaking banking information), i.e., yield false negatives. Often the sources and sinks that appeared in critical taint flows are not in the tools' configuration. These false negatives were described in the behavior information written by security experts. Thus, they could be identified manually.
- The tools also frequently report false positives that prolong code inspection. For the Android malware *fakebank_android_samp*, for instance, FlowDroid reported 23 taint flows in its default configuration, but only 10 of them are true positives. Moreover, 8 of these 10 only concern the logging of sensitive data using the Android Logging Service, something that is considered secure since Android version 4.1, which protects such logs from being read without authorization (Rasthofer, 2013). All remaining 13 flows are false positives.

Consequently, we opted for an alternative approach starting with manual inspection. We manually inspected the 42 candidate apps along with their behavior information to identify a set of expected taint flows. For example, if an app's behavior information like "monitor incoming SMS messages" is included, our inspection starts at sources which read incoming messages.

The inspection for each app was performed by two people, both with background in Android taint analysis research, working together as a pair in front of the same computer. Whenever a taint flow was discovered and confirmed by both inspectors, it was added to the documentation. After each inspection, a third inspector (a different person) reviewed the documented taint flows. Only the taint flows confirmed by all three inspectors were retained in the final suite. The percentage of agreement between the first two and the third inspector was 96.82%. This resulted in 39 benchmark apps with 149 expected taint flows.

Next, we also used an automated tool (see TB-Mapper in Section 4.2.1) to extract sources and sinks from these 149 expected taint flows and used them to configure selected Android taint analysis tools — those which we use during evaluation as well: FlowDroid and Amandroid. We then applied the taint analysis tools under this configuration to all benchmark apps. This way 100 taint flows were revealed which have not been documented during our initial manual inspection. We manually checked and rated these newly discovered taint flows. Each flow was rated independently by two authors as expected or unexpected. The results were compared and a consensus was established. This resulted in further 100 *additional* benchmark cases — 54 expected and 46 unexpected taint flows. For each expected taint flow, we also documented the intermediate steps of one data-flow path as witness. At the end, the TaintBench suite consists of 39 benchmark apps with 203 expected and 46 unexpected taint flows. We developed a few tools, introduced in the next section, to support this process. More details about the selected Android taint analysis tools and their configurations are given in Section 6.

4 The TaintBench Framework

Along with the TaintBench suite, we contribute the TaintBench framework, which (as our usability test in Section 5 shows) simplifies and speeds up real-world-app benchmark suite construction (Part ❶ – Section 4.1) for Android taint analysis, allows automatic evaluation (Part ❷ – Section 4.2) of analysis tools, and supports source code inspection (Part ❸ – Section 4.3) of taint flows. Figure 3 gives an overview of this framework, which is structured into three parts. Every box in the figure refers to a tool extended or built for this framework. All elements contributed along with this study are marked by a ☆-symbol.

We describe the framework and all tools it comprises using a running example consisting of an artificial app (`example.apk`) as depicted in Figure 4. The first class is an Activity component (`MainActivity`) which comprises one source (s_1) and one sink (s_5) in its `onCreate` lifecycle method. The source extracts the device's id (`getIMEI`) which is considered as sensitive data. Once it reaches the sink (`sendTextMessage`), it is leaked via an SMS. The flow from

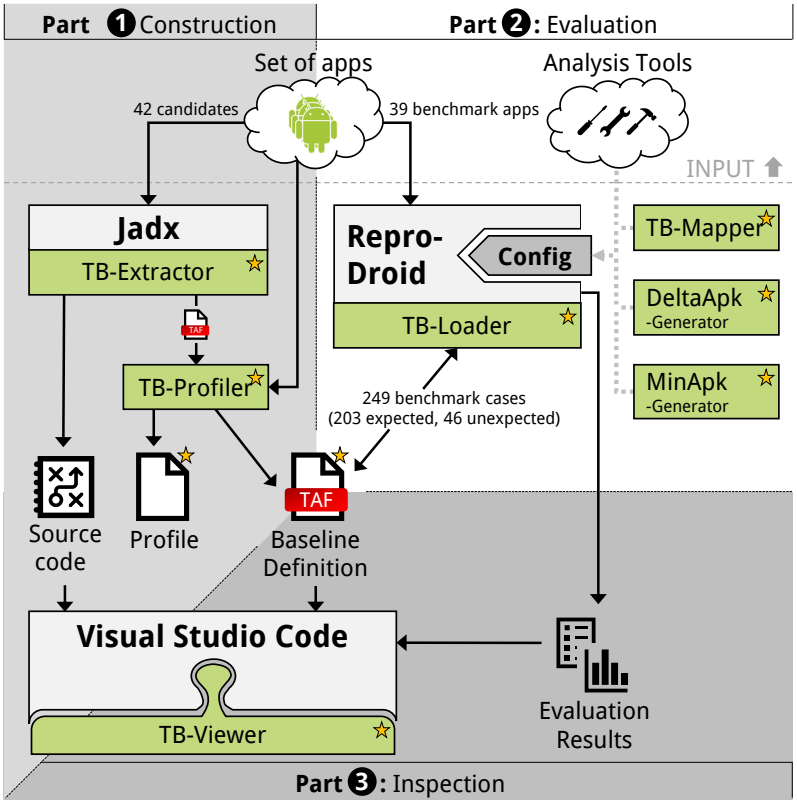


Fig. 3 Overview of the TaintBench Framework

s_1 to the logging statement (s_7) should not be recognized as a leak, since only the value of the not-null check is logged. Class `Foo` contains only one method (`bar`) which comprises a second taint flow from s_8 (source) to s_9 (sink). Details are omitted to maintain the legibility of the example. In summary, the example contains two expected taint flows (solid green edges) and one undocumented taint flow which could mistakenly be identified as a third one (blue dashed edges):

$$(s_1 \rightarrow s_5), (s_8 \rightarrow s_9), (s_1 \dashrightarrow s_7)$$

4.1 Part ❶ – Construction

In Part ❶, two tools come into play. The first tool, the JADX decompiler (JADX, 2020), allows us to extract source code from Android application package (APK) files. We extended JADX's GUI by adding the TB-EXTRACTOR. It enables us to manually specify source, sink, and intermediate assignments of taint flows by selecting the relevant source code statements directly in the extended GUI. The extension also allows inspectors to add a high-level description and *attributes* (i.e., special language or framework features) to each taint flow. Once the taint flow specification is done, TB-EXTRACTOR outputs a JSON file which stores the information logged for each taint flow.

The second tool used here is the TB-PROFILER. It takes both the APK and the JSON file generated by TB-EXTRACTOR and outputs automatically

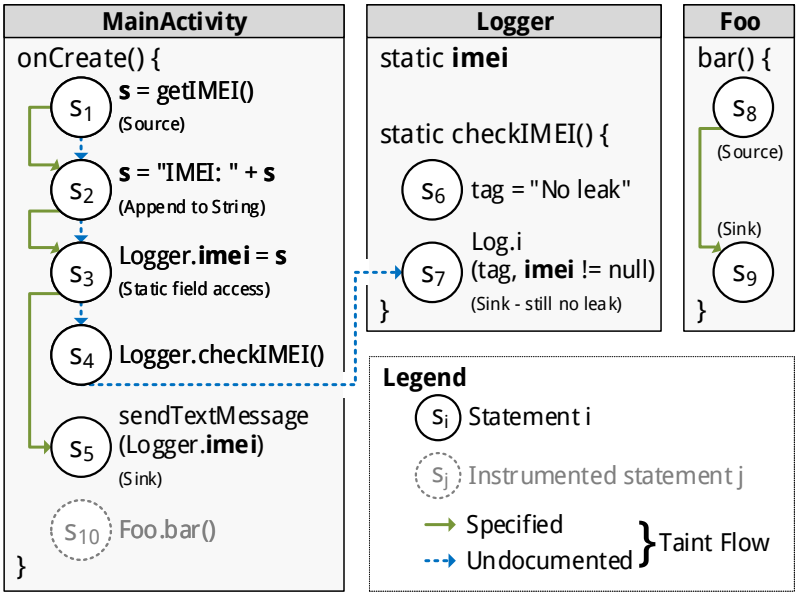


Fig. 4 Running Example (`example.apk`)

```
{ "findings": [{
    "ID": 1,
    "isUnexpected": false,
    "description": "This malicious flow sends IMEI in a SMS."
    "source": {
          "statement": "String s = getIMEI();",
          "methodName": "onCreate",
          "className": "MainActivity",
          "lineNo": 1,
          "targetName": "getIMEI",
          "targetNo": 1,
          "IRs": [{"type": "Jimple",
                   "IRstatement": "$r2 = virtualinvoke ..."}]},
    "sink": {
          "statement": "sendTextMessage(Logger.imei);", ...},
    "intermediateFlows": [{
          "ID": 1, "statement": "s = "IMEI: " + s;", ...},{
          "ID": 2, "statement": "Logger.imei = s;", ...}],
    "attributes": {
          "staticField": true,
          "appendToString": true},
}, { ... } ]...}
```

Listing 3 The TAF-File for the Running Example

detectable attributes which were missed or incorrectly assigned by the inspectors. For example, if there is a statement on a documented taint flow which involves reflection and the human inspectors forgot to assign the attribute `reflection`, the TB-Profiler will detect it automatically by checking the API signatures and language features involved in the respective statements.[2] To avoid false attributes produced by TB-Extractor being documented, the human inspectors make the final decision if the detected attributes should be assigned or not. The JSON file derived this way can be stored as the *baseline definition* that specifies the taint flows for the associated benchmark app. The TB-Profiler extracts other static information from the APK, such as the target platform version, a list of used permissions, sensitive API calls etc., and stores them in a profile file.

We introduce our documentation of the baseline definition with the running example. To that effect the baseline definition holds the taint flows illustrated as solid green edges in Figure 4. To store the baseline definition, a JSON file conforming our Taint Analysis Benchmark Format (TAF) is generated (output format of TB-Extractor). A shortened version of the TAF-File for the running example is provided in Listing 3. We did not use the Static Analysis Results Interchange Format (SARIF) (OASIS, 2019) or similar formats (e.g. AQL (AQL, 2020)), since most of them are either too general or contain too many properties (e.g., the rule of a tool used to produce a finding) which are irrelevant in our domain. We wanted to document the taint flows we manually specified with the Jadx decompiler. Thus, we did not have the information

[2] A list of all attributes considered is given in Table 7 in Section 6.1.

of the rule of a specific tool or access path of a taint. Furthermore, SARIF does not support differentiating expected and unexpected taint flows, which is important for benchmarking. If we would use the AQL to encode our baseline information, we could only differentiate expected and not expected taint flows by attaching a generic key-value pair (attribute). Additionally, encoding intermediate steps in AQL format would make our baseline unnecessarily lengthy and hence impede human-readability and consequently the manual documentation of taint flows. In contrast, TAF allows to encode all relevant information in a clearly and precisely structured way. Thereby information given in TAF can easily be converted into other formats such as the AQL.

Each element of the array `findings` describes one taint flow. It can be either expected or unexpected, indicated by the attribute `isUnexpected`. When the value is equal to `true`, the respective flow is unexpected and it means there is no data flow between the source and the sink. Otherwise, it is an expected taint flow. In the listing, the expected taint flow ($s_1 \rightarrow s_5$) is visible. The second flow ($s_8 \rightarrow s_9$) is hidden in Line 22. An example, that shows how source, sink, and intermediate flows are described with code locations, is given in Lines 5-18. If a statement contains multiple function calls, `targetName` and `targetNo` specify which function call precisely is meant. Intermediate flows are assigned with `ID`s which indicate the order of their appearances in the taint flow. The attributes `staticField` and `appendToString` indicate that the tainted data flows through a statically declared variable (s_3) and is appended to a String (s_2). The `IRs`-array holds the intermediate representations (IR) associated to the statement, such as Jimple (Vallée-Rai et al., 1999; Lam et al., 2011). Jimple is the IR of the analysis framework Soot on which FlowDroid is based, and it is supported by ReproDroid. We included Jimple in the baseline, but one could certainly fill this array with IRs from other frameworks. Jimple statements are automatically added by TB-Loader in Part ❷.

4.2 Part ❷ – Evaluation

The harness we provide to evaluate Android taint analysis tools on the TaintBench suite is an extension to ReproDroid (Pauck et al., 2018). ReproDroid is a configurable open-source benchmark reproduction framework to (i) refine, (ii) execute, and (iii) evaluate analysis tools on benchmark suites. Considering the first step (i), ReproDroid allows to create benchmark cases via a GUI. First, a set of benchmark apps can be imported. Second, sources and sinks contained in these apps can be selected — manually or automatically by comparison to a given list of source and sink APIs (e.g., the SuSi (Rasthofer et al., 2014) list). By specifying sources and sinks, taint flows are implicitly specified too. Lastly, ReproDroid allows to categorize these implicitly defined taint flows as expected or unexpected. We adapted ReproDroid to accept our baseline definition as additional input (see TB-Loader in Figure 3). Thereby, the information of the baseline definition are used to automatically select sources and sinks and categorize taint flows as expected or unexpected.

Once the benchmark suite is fully setup in ReproDroid it can be stored. Stored benchmarks can then be loaded to be executed with or without using ReproDroid's GUI. Our extension can also be used to export tool-specific lists of the defined sources and sinks. Currently, the format of Amandroid and FlowDroid is supported.

For the running example, assume the expected (green arrow) and unexpected (red arrow) taint flows in the baseline are:

$$(s_1 \rightarrow s_5), (s_8 \rightarrow s_9), (s_1 \not\rightarrow s_9), (s_8 \not\rightarrow s_5) \qquad \textbf{*}$$

All taint flows are automatically converted into benchmark cases in the AQL format used in ReproDroid. This format allows one to compose queries (AQL-Queries), to run arbitrary analysis tools, and also standardizes the tools' results (AQL-Answers). Hence, in case of taint analysis tools such an AQL-Answer primarily contains a collection of data flow paths that represent taint flows.

When executing an analysis tool on a benchmark suite (ii), ReproDroid creates one AQL-Query per benchmark case. If the same query with respect to the same benchmark app is asked in two or more cases, the analysis result is not computed again but loaded and reused. The configuration of ReproDroid allows us to specify which set of tools should be used to answer which type of query. By adapting the configuration or transforming the query according to configurable strategies, various queries can be constructed. In our comprehensive experiments we configured ReproDroid to use four analysis tools and six different strategies (see Section 6.2). Once a tool is applied to a benchmark case, one AQL-Answer is computed and stored as result for this specific case.

To evaluate a tool on a benchmark suite (iii), for each benchmark case ReproDroid compares the expected and unexpected taint flow (constructed on the basis of the baseline definition) with the actual AQL-Answer computed for the respective case. Expected cases allow one to assess the recall of a taint-analysis tool, while unexpected cases allow one to judge the tool's precision. The total number of identified (resp. missed) expected taint flows across all benchmark cases is used to determine the number of true positives (resp. false negatives). Flows which match the unexpected taint flows are false positives. Flows that neither match expected nor unexpected taint flows are not counted. Due to the construction process of the TaintBench suite (see Section 3.1) this was never the case considering the experiments conducted in our evaluation (see Section 6).

To evaluate an analysis tool on our running example, let us assume that ReproDroid is configured to solely employ one analysis tool. Thus, the following AQL-Query is posed:

```
Flows IN App('example.apk') ?
```

Assume the actual AQL-Answer contains four flows:

$$(s_1 \rightarrow s_5), (s_8 \rightarrow s_9), (s1 \not\rightarrow s_9), (s_1 \dashrightarrow s_7)$$

ReproDroid's evaluation only considers the first three flows: the first two are true positives and the third one is a false positive. A flow is evaluated as true positive (resp. false positive) only if it matches a defined expected case (resp. unexpected case) (see ✱ above). The last flow (s_1->s_7) is not specified as expected or unexpected case — it is not documented, yet. Whenever facing such an undocumented taint flow, manual inspections is required to decide if it should be documented as an expected or unexpected case. This way the 100 additional taint flows were added to TaintBench's baseline definition (see Section 3.1). To support this inevitable manual inspection, the tool TB-Viewer was created, which will be introduced later in Section 4.3.

4.2.1 Evaluation-Support Tools

To further support empirical evaluations in the context of the TaintBench framework, ReproDroid was configured with three additional novel tools.

1) To reduce the complexity of TaintBench apps with respect to each benchmark case, we introduce the MinApk-Generator. As explained below, the MinApk-Generator allows one to infer insights about the reason why an actual taint flow may remain undetected by a tool (false negative). MinApk-Generator can be used through AQL's slicer interface although it is no classic slicer:

```
Slice IN App('example.apk') !
```

The MinApk-Generator prunes the original APK and generates a *minified APK* for each taint flow defined in the baseline. Considering a taint flow, any part in the code that is not connected to the source, sink, or intermediate flows is removed. This task can be performed more efficiently than slicing the app from source to sink, since the information about intermediate flows is given. Considering the running example in Figure 4, the `checkIMEI()` method of class `Logger` is removed because it does not appear in the baseline definition. However, this method would be kept by an ideal forward slicing algorithm starting from s_1, since the static field `Logger.imei` is used in the method. MinApk-Generator only keeps the static field of this class. Since lifecycle methods might be removed this way, a new analysis entry point is created. To do so, the component that is launched on app start gets selected. Calls to all methods holding sources are added to one of its lifecycle methods, e.g., a call to `Foo.bar()` (s_{10}) is added to the `onCreate()`-method of `MainActivity` in Figure 4. In consequence, it is ensured that the taint flow is reachable in the call graph of the minified APK.

As such, MinApk-Generator creates a minified version of the app that still contains the original taint flow. The reduction may be unsound, causing an analysis to also show false negatives on the minified version. However, whenever a taint flow — manually labeled as expected taint flow but undetected in the original app — *is detected* in the minified app, one can consider

an incomplete call graph to be the reason why the analysis tool misses this flow in the original app.

2) The DeltaApk-Generator automatically generates variants of an input app in which a single predefined taint flow, specified in the baseline definition, is killed. DeltaApk-Generator can be used to check if the evaluated analysis tool has over-approximated to detect a taint flow. It is used as a preprocessor in ReproDroid. The following query asks for flows in our example app after killing ($s_1 \rightarrow s_5$) with `ID=1` (see Line 2 in Listing 3):

```
Flows IN App('example.apk' | 'DELTA') WITH ID = 1 ?
```

DeltaApk-Generator kills the flow from the source `getIMEI` by instrumenting overriding assign statement. It inserts a new assign statement `s = null` directly after the statement s_1. Hence, the tainted variable `s` is immediately sanitized in the generated *delta APK*. For a tainted variable which has primitive type, DeltaApk-Generator inserts a statement that assigns a constant value to it. This way all flows are killed from the source. A precise taint analysis tool should report the taint flow ($s_1 \rightarrow s_5$) for `example.apk`, but not for its preprocessed version created by DeltaApk-Generator. If the taint flow is still detected in the delta APK, it is a false positive.

3) TB-Mapper answers AQL-Queries as the following one:

```
TOTS [ SourceAndSinks IN App('example.apk') ? ] !
```

This query asks for an AQL-Answer which lists all the sources and sinks in the baseline definition of `example.apk` and converts the detected sources and sinks into a tool specific format (`TOTS`), e.g., a file that comprises a list of sources and sinks used by FlowDroid.

4.3 Part ❸ – Inspection

The TB-Viewer, which is the main component of the third part (❸), comes in form of a Visual Studio Code (VSC) (Microsoft, 2020) extension using the MagpieBridge framework (Luo et al., 2019b). It is used whenever manual inspection is needed. This tool displays specified taint flows directly on the benchmark app's source code in VSC. It allows us to interactively inspect and compare the baseline definition (Part ❶) with the findings of an evaluated analysis tool (Part ❷).

To do so, TB-Viewer provides four lists in a tree view as shown for the running example in Figure 5: (`A`) a list of expected and unexpected taint flows with data-flow paths that are specified in the baseline definition, (`B`) a list of flows which are reported by an analysis tool during evaluation, (`C`) a list of matched and (`D`) a list of unmatched flows. List `C` contains all those taint flows of List `A` that are detected during evaluation. The contrary holds for List `D`: it comprises all flows that are reported during evaluation, but do not match any flow in the baseline. These unmatched flows in list `D` cannot be evaluated automatically with ReproDroid by Part ❷, which is why TB-Viewer supports their manual inspection. TB-Viewer enables the expert

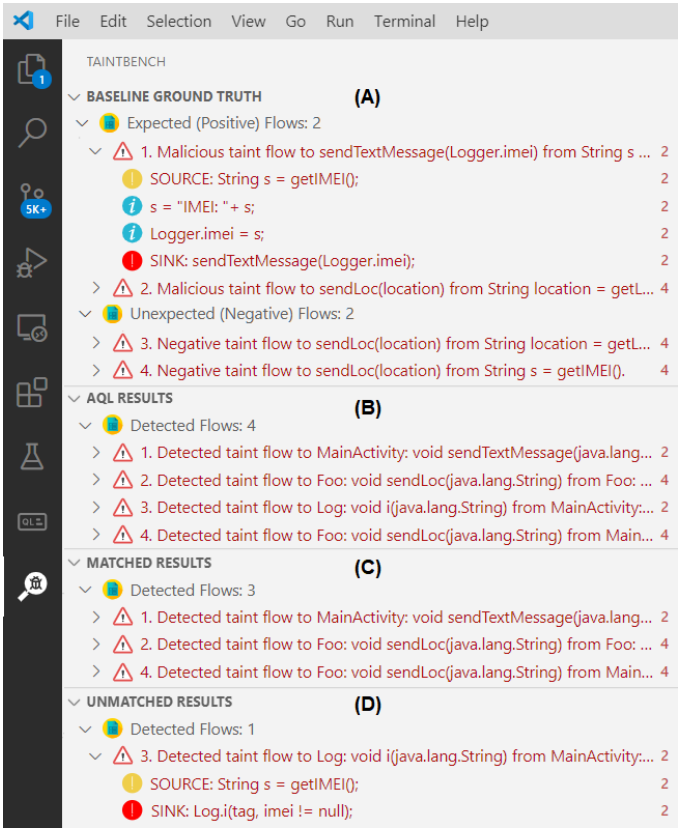


Fig. 5 Screenshot of TB-Viewer w.r.t. Running Example

to navigate through an application's source code along visually highlighted taint flows defined in these lists, more precisely, to navigate step-wise from the source to the sink of each taint flow. Considering the running example and the four flows reported by the configured tool, List `A` and `C` would contain the two expected taint flows depicted by solid green edges in Figure 4 and an unexpected flow ($s_1 \dashrightarrow s_9$). List `D` holds one flow: ($s_1 \dashrightarrow s_7$) — dashed blue edges. Once this latter flow is added to the baseline definition, list `D` will be empty.

We have installed TB-Viewer in the Gitpod online IDE (Gitpod, 2019) for GitHub, thus, all benchmark cases of the TaintBench suite can be viewed in a web browser.[3]

[3] Find the access at `https://taintbench.github.io/taintbenchSuite`

5 Evaluation of the TaintBench Framework via a Usability Test

We conducted a controlled experiment with users to evaluate the effectiveness of the two GUI-based tools in the TaintBench framework, namely jadx with TB-Extractor and Visual Studio Code (VSC) with TB-Viewer. Thereby we wanted to answer the following research questions:

RQ1 Do users spend less time to inspect and document taint flows using TB-Extractor and TB-Viewer than using plain jadx and Visual Studio Code?

RQ2 Do users perceive TB-Extractor and TB-Viewer to be more usable than plain jadx and Visual Studio Code?

5.1 Participants

The TaintBench framework is designed for experts and it is hard to find suitable users. We sent emails to researchers who work in area of program analysis and developers who have experience in developing static analysis tools. We were able to recruit five experts to participate in our study. Four of them are researchers (PhD students). One of them is a software engineer who has experience in developing static analyzers. All participants are very familiar with taint analysis. We denote them with User 1–5 in the following.

5.2 Study Design

Due to the low number of participants, we designed a within-subjects study for each tool, i.e., each participant tests all the conditions. We compare the condition with tool support to without tool support. Table 4 shows the tasks we designed for the study. While tasks **VSC** (control condition) and **VSC+TB-Viewer** (experimental condition) are used to test TB-Viewer, the tasks **Jadx** (control condition) and **Jadx+TB-Extractor** (experimental condition) are for testing TB-Extractor. The independent variable in our study is which tool a participant uses to perform a task we designed. The dependent variables we measured were the time a participant used to complete a task and the perceived usability with the System Usability Scale (SUS) (Brooke, 1996). We measured the time, since we wanted to know whether our extensions TB-Viewer and TB-Extractor help participants to work more efficiently. The SUS scores reflect how usable the participants think our tools are.

Tasks for testing TB-Viewer: For TB-Viewer, the two tasks are about the inspection of taint flows. These tasks simulate the manual inspection one has to do when evaluating a tool's precision. The participants were asked to judge whether taint flows reported by a taint analysis tool are false positives or not. We prepared six taint flows reported by FlowDroid when applying it to

Table 4 Descriptions of Tasks

Task	Description
VSC (control)	The participant was given plain VS Code, decompiled source code of an app X and 3 taint flows in X. The participant was asked to judge whether these taint flows are true positives or false positives in VS Code.
VSC+TB-Viewer (experimental)	The participant was given VS Code with TB-Viewer installed, decompiled source code of the app X and 3 taint flows in X (different 3 than in task VSC). The participant was asked to judge whether these taint flows are true positives or false positives in VS Code.
Jadx (control)	The participant was given the Jadx decompiler, an apk Y from our suite, and two expected taint flows specified for the apk. The participant was asked to document these two flows in TAF-format.
Jadx+TB-Extractor (experimental)	The participant was given the Jadx decompiler extended with TB-Extractor, the apk Y from our suite, and two expected taint flows (different 2 than in task Jadx) specified for the apk. The participant was asked to document these two flows in TAF-format.

an app from our benchmark suite. To avoid unfair distribution, we intentionally chose six true-positive taint flows that we think to be similarly complex. However, the participants are not aware of this and they have to triage the taint flows by searching through and looking at relevant code.

In task **VSC**, the participants were given Visual Studio Code and decompiled source code of the app. We asked them to inspect three taint flows that are documented in an XML file (in AQL-Answer format[4]). For each flow we provide information only about the source and the sink but not about the data-flow paths, as this is also the case when dealing with popular Android taint analysis tools.[5] For each participant, these three taint flows are *randomly* chosen from the six taint flows. In task **VSC+TB-Viewer**, the participants are asked to inspect the remaining three taint flows. In addition, they used Visual Studio with the extension TB-Viewer installed. TB-Viewer can read the taint flows from this XML file and display them directly in Visual Studio Code as described in Section 4.3.

To minimize the ordering/learning effects, we randomize the order of these two tasks for the participants. We make sure that the participants do not always start with task **VSC** nor **VSC+TB-Viewer**.

[4] `https://github.com/FoelliX/AQL-System/wiki/Answers`

[5] Note that FlowDroid does provide an option to compute and output data-flow paths in its current version, not, however, in the version used for this study. To construct the ground truth, we preferred not to use the current version but instead the version from the ReproDroid paper due to the many false negatives that the current FlowDroid version creates (see Section 6).

Tasks for testing TB-Extractor: For TB-Extractor, the participants are asked to document taint flows that are determined in an Android app. Manual inspection and discovering taint flows is a skillful and time-consuming task. To simplify the study, we give the participants taint flows found by us. In other words, they do not need to search taint flows by themselves, but only documenting them. We chose four taint flows from our baseline of an benchmark app. As described in Section 4.3, in Visual Studio Code (including TB-Viewer) each taint flow is displayed with detailed information about source, sink, its attributes and its intermediate flows as well as a general description. Note that we ensure the tasks **VSC** and **VSC+TB-Viewer** to be conducted before the tasks for TB-Extractor. Thus, the participants already know TB-Viewer when conducting the tasks for TB-Extractor.

In task **Jadx**, the participants are given jadx together with the chosen app. They are asked to document two randomly chosen taint flows from the four prepared taint flows with a code editor. They are given a template JSON file using TAF-format in which they only need to fill in the information (code, line number etc.) about taint flows copied from jadx. The TAF-format is explained to the participants before they start the task. In task **Jadx+TB-Extractor**, the participants are given jadx *with* TB-Extractor. We play a short tutorial video (six minutes) to them. This video explains how to document taint flows with the extended jadx. The participants are required to document the other two taint flows.

Similar to the tasks for testing TB-Viewer, we also randomize the order of these two tasks for the participants.

5.3 Data Collection

We conducted the experiment with participants remotely via a video conference tool. Participants were asked to share their screen with us all the time. After a brief introduction to the study and guide for installation of our tools, we gave our tasks to the participants in written form and asked them to solve the tasks independently without our help. In each session, the participants were given maximally 30 minutes to solve each task. We measured the actual time each participant spent for each task. We asked the participants to give us a clear signal when they started and finished each task. After each task, the participants filled out an exit-survey. In this survey, they were asked to evaluate the ten statements from the System Usability Scale (SUS) and tell about their feeling when using the system to do the task. SUS is a questionnaire that is designed to measure the usability of the a system (Brooke, 1996). The survey and the descriptions of all tasks used can be found on our website.[6]

[6] `https://taintbench.github.io/userstudy`

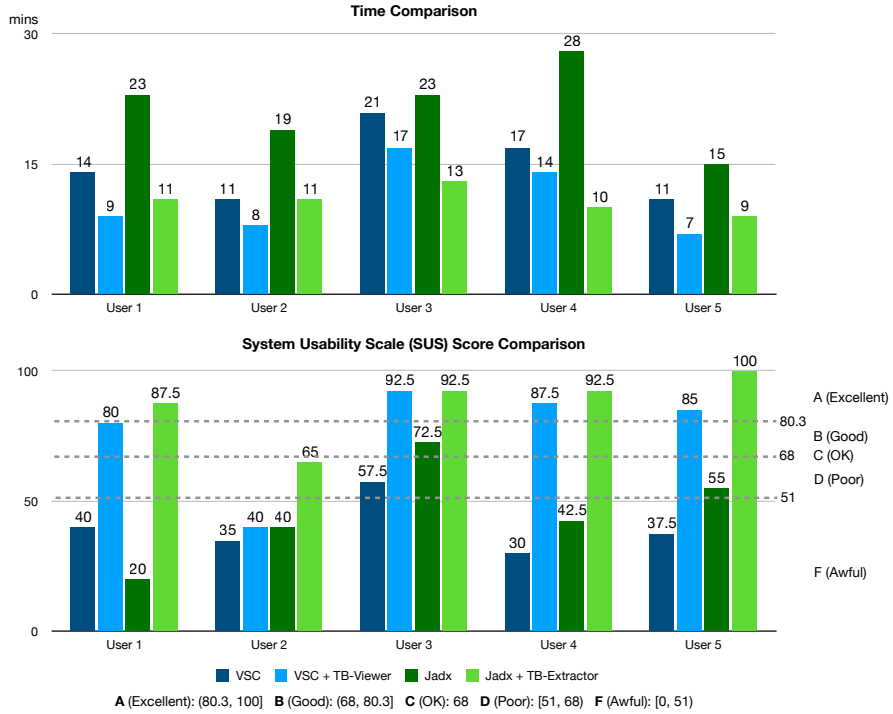


Fig. 6 Experimental Results

5.4 Results

Figure 6 shows the results of our experiment. The measured time is used to answer RQ1 and the SUS score for answering RQ2.

RQ1 (Time) All participants solved the tasks more efficiently with the support of TB-Viewer and TB-Extractor than the ones without. Averagely, the time used for task **VSC+TB-Viewer** is reduced from 14.8 to 11 minutes in comparison to task **VSC**. With the support of TB-Extractor, the average time for solving the task is even halved (from 21.6 to 10.8).

RQ2 (Usability) Overall, the participants responded positively to both tools. Except **User 2**, all users gave both TB-Viewer and TB-Extractor high SUS scores ranging from 80 to 100, which means the usability of both tools is *excellent* or at least *good* from their point of view. **User 2** explained to us why he rated **VSC+TB-Viewer** with low scores. He felt it was very cumbersome to do the task without the data-flow path of a taint flow displayed in the editor. However, this information is not given in the results of FlowDroid, thus, TB-Viewer cannot provide this feature. Actually, if the information

about the data-flow path is given in the analysis results, TB-Viewer can actually display this information as done for the taint flows in our baseline (see Figure 5). While **User 2** complained more about the analysis results missing data-flow path, other users felt well supported by the tool while solving the task. For example, **User 1** told us about his positive feeling about TB-Viewer:

> *"Not having to switch back and forth constantly between VS Code and the XML file took away a lot of possible problem vectors. Like scrolling too far, misreading a line, misunderstanding what a particular line in the XML means. Even though the tool didn't provide a lot more functionality (the 'jump to' feature is much appreciated) than the XML-based solution, I still felt more secure with my results in the end."*

Also **User 4** had similar feeling when solving task **VSC+TB-Viewer** and wrote:

> *"Finding the sources and sinks is much easier than without the system. Still it is not always easy to find the path between source and sink. Overall, the task is much easier to solve than without the system and gives higher confidence in giving the correct evaluation."*

In the documentation tasks, without TB-Extractor, all participants felt doing the task was very tedious and error-prone. They all made some mistakes (e.g., wrong line number, wrong method signature) in the documentation. In contrast, the taint flows documented with TB-Extractor were all correct. The participants felt TB-Extractor was self explanatory and easy to use. Especially **User 5** who had to document taint flows for other work before, gave a full SUS score (100) to TB-Extractor and spent the least time (9 minutes) for the task. **User 3**'s comments also show TB-Extractor eases the task:

> Task **Jadx**: *"Absolutely cumbersome to use. A lot of busy work. No support by the tool at all."*
>
> Task **Jadx+TB-Extractor**: *"Easy to use! However, the description of the taint flow is in a separate window. But the window is well designed."*

User 3's comment on task **Jadx** reflects probably one of the main reasons for why in previous evaluations of taint analysis tools the ground truth was rarely documented. In summary, we see that TB-Extractor allows participants to document taint flows more efficiently and correctly.

6 Evaluation of and with the TaintBench Suite

Our TaintBench suite contains 39 benchmark apps with 249 documented benchmark cases in total as shown in Table 5. 203 of them are expected taint flows. 149 expected taint flows were discovered by us manually as described in Section 3.1. During the evaluation with the benchmark apps, we also inspected

taint flows which were reported by both FlowDroid and Amandroid manually. Thereby additinal 54 expected and 46 unexpected taint flows were added to the suite. We will introduce more details about this in Section 6.2. Each benchmark app comes with the following assets in its own GitHub repository:

- the APK file,
- the decompiled source code project,
- the baseline definition (TAF-file),
- a profile file about the benchmark app containing statically extracted information including target platform version, permissions, sensitive APIs, behavior description, etc.

Furthermore, we classified the taint flows based on their behaviors according to their source and sink categories as shown in the Table 6[7]. We reused the categories defined in the SuSi paper (Rasthofer et al., 2014) and MUDFLOW paper (Avdiienko et al., 2015). We also added new categories such as `INTERNET SOURCE` and `CRITICAL FUNCTION`, since except data leaks our suite includes other types of malicious taint flows such as Path Traversal (CWE-22), Execution with Unnecessary Privileges (CWE-250) and Use of Potentially Dangerous Function (CWE-676).[8] The categorization of the sources and sinks was first done by the lead author. To enhance the reliability, the third author checked and discussed the assigned categories with the lead author whenever there were disagreements. Consensus was achieved for the final categorization.

We present our evaluation of the TaintBench suite (RQ3) and with it (RQ4 and RQ5) by answering the following three research questions:

[7] More detailed information of each flow can be found on `https://taintbench.github.io/img/data/Sources_Sinks_Category_Stats.pdf`

[8] CWEs can be found on `https://cwe.mitre.org`

Table 5 Summary of the TaintBench Benchmark Suite

No.	Name	E.	U.	No.	Name	E.	U.
1	backflash	13	11	21	proxy_samp	17	3
2	beita_com_beita_contact	3	0	22	remote_control_smack	17	0
3	cajino_baidu	12	3	23	repane	1	0
4	chat_hook	12	1	24	roidsec	6	0
5	chulia	4	0	25	samsapo	4	1
6	death_ring_materialflow	1	0	26	save_me	25	6
7	dsencrypt_samp	1	0	27	scipiex	3	0
8	exprespam	2	0	28	slocker[1]	5	0
9	fakeappstore	3	0	29	sms_google	4	0
10	fakebank[1]	5	0	30	sms_send_locker_qqmagic	6	2
11	fakedaum	2	0	31	smssend_packageInstaller	5	0
12	fakemart	2	0	32	smssilience_fake_vertu	2	2
13	fakeplay	2	0	33	smsstealer_kysn_assassincreed[1]	5	0
14	faketaobao	4	0	34	stels_flashplayer_android_update	3	0
15	godwon_samp	6	0	35	tetus	2	0
16	hummingbad[1]	2	0	36	the_interview_movieshow	1	0
17	jollyserv	1	0	37	threatjapan_uracto	2	0
18	overlay[1]	4	2	38	vibleaker[1]	4	0
19	overlaylocker2[1]	7	12	39	xbot[1]	3	0
20	phospy	2	3	Σ		203	46

E.: Number of expected taint flows, **U.**: Number of unexpected taint flows, [1]: Suffix "_android_samp"

Table 6 Overview of Expected and Unexpected Taint Flows according to Source and Sink Categories. New categories we added are marked with * when appearing for the first time in the table.

Source Category	**Sink Category**	**E.**	**U.**
ACCOUNT INFORMATION	NETWORK	11	
ACCOUNT INFORMATION	INTENT	1	
ACCOUNT INFORMATION	FILE	2	
ACCOUNT INFORMATION	LOG	1	
ACCOUNT INFORMATION	DATABASE	2	
CONTACT INFORMATION	NETWORK	11	11
CONTACT INFORMATION	INTENT	2	
CONTACT INFORMATION	SMS MMS	1	
CRITICAL FUNCTION *	CRITICAL FUNCTION *	2	
DATABASE	SMS MMS	3	
DATABASE	FILE	24	
DATABASE	NETWORK	19	
DATABASE	DATABASE	2	3
DATABASE	LOG	3	
DATABASE	CRITICAL FUNCTION	1	
FILE	NETWORK	13	2
FILE	FILE	1	1
FILE	CRITICAL FUNCTION	5	
FILE	INTENT	3	
FILE	LOG		1
INTERNET SOURCE *	SMS MMS	2	
INTERNET SOURCE	OTHER STORAGE *	1	
LOCATION INFORMATION	FILE	4	
LOCATION INFORMATION	NETWORK	4	2
NETWORK INFORMATION	EMAIL	1	
NETWORK INFORMATION	LOG	1	1
NETWORK INFORMATION	FILE	1	
NETWORK INFORMATION	NETWORK	9	
NETWORK INFORMATION	SMS MMS		1
OTHER DATA *	NETWORK	1	
OTHER DATA	LOG	7	1
OTHER DATA	CRITICAL FUNCTION		6
OTHER DATA	INTENT		2
SMS MMS	SMS MMS	5	
SMS MMS	NETWORK	11	
SMS MMS	INTENT	6	2
SMS MMS	LOG	1	
SMS MMS	FILE	1	
SMS MMS	CRITICAL FUNCTION	1	
SYSTEM SETTINGS	NETWORK	3	
SYSTEM SETTINGS	CRITICAL FUNCTION	1	1
UNIQUE IDENTIFIER	FILE	1	
UNIQUE IDENTIFIER	NETWORK	25	6
UNIQUE IDENTIFIER	LOG	3	
UNIQUE IDENTIFIER	EMAIL	1	
UNIQUE IDENTIFIER	CRITICAL FUNCTION	3	6
UNIQUE IDENTIFIER	SMS MMS	2	

E.: Number of expected taint flows, **U.**: Number of unexpected taint flows

RQ3 How does TaintBench compare to DroidBench and Contagio?
RQ4 How effective are taint analysis tools on TaintBench compared to DroidBench?
RQ5 What insights can we gain by evaluating analysis tools on TaintBench?

6.1 RQ3: How does TaintBench compare to DroidBench and Contagio?

With this question we wanted to evaluate the TaintBench benchmark suite under two aspects regarding representativeness: First, we evaluated the *taint flows* in TaintBench and compared them to those in DroidBench in terms of language and framework features involved in the flows. Second, using a set of metrics we compared the benchmark apps in TaintBench to the apps in DroidBench and the whole Contagio dataset.

6.1.1 Comparison of Taint Flows

As introduced in the representativeness criterion in Section 3, one of our goals for TaintBench is to include taint flows which address different language and framework features (attributes). The numbers of expected taint flows involving different language and framework features are listed in Table 7. The attributes of each taint flow are assigned by us and TB-Profiler as mentioned in Section 4.1. Through these attributes the taint flows can be categorized and also mapped to the majority (11/18) of the DroidBench's categories as shown in Table 7. Categories of DroidBench such as "Android Specific" (DroidBench 3-0, 2016) are not uniquely relatable.

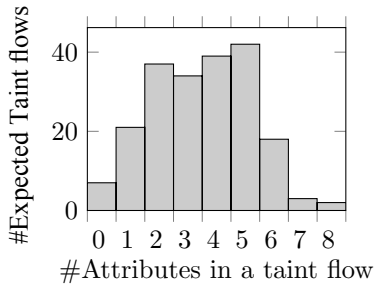


Fig. 7 Distribution of Attributes

Table 7 Attributes Associated to Expected Taint Flows in TaintBench

Attribute	Description	DB Category	# Flows
`nonStaticField`	sensitive values stored in non-static fields	Field and Object Sensitivity	61
`staticField`	sensitive values stored in static fields	Field and Object Sensitivity	31
`reflection`	refection APIs called	Reflection	5
`array`	sensitive values stored in arrays	Arrays and Lists	39
`collections`	sensitive values stored in Java collection objects	Arrays and Lists	67
`threading`	threading mechanisms involved	Threading	80
`appendToString`	sensitive values appended to Strings	General Java	99
`callbacks`	callbacks for UI interactions	Callbacks	23
`lifecycle`	lifecycle methods involved	Lifecycle	104
`payload`	malicious payload is downloaded at runtime	Dynamic Loading	5
`ICC`	inter-component communication involved	ICC	49
`IAC`	inter-app communication involved	IAC	2
`implicitFlows`*	implicit flows	Implicit Flows	6
`pathConstraints`+	path conditions must be satisfied	–	74

*Cannot automatically be assigned by TB-Extractor, +No mapping category in DroidBench (DB).

Moreover, most expected taint
flows in TaintBench address multiple (up to 8) features at the same time as shown in the histogram in Figure 7. This is not modeled in DroidBench. The majority (175/203) of expected taint flows in TaintBench address multiple features at the same time rather than a single one as designed in DroidBench.

6.1.2 Comparison of Benchmark Apps

We compare the benchmark apps in TaintBench to apps in DroidBench as well as to the entire Contagio dataset (from which TaintBench apps are sampled) in three aspects: (i) the usage of sources and sinks, (ii) call-graph and (iii) code complexity. For each aspect, we used a set of metrics for a quantitative evaluation.

Usage of Sources and Sinks The usage of sources and sinks is a predictor of how many data-flow propagations starting from sources/sinks (depending on forward/backward analysis) are required to capture all possible taint flows. Thus, we quantify the usage of sources and sinks with the following metrics:

- **#Sources/Sinks**: number of different source/sink APIs appeared in a benchmark app.
- **#Usage Sources/Sinks**: number of different code locations where source/sink APIs are used in a benchmark app.

To measure these metrics, we first compiled a list of potential source and sink APIs. This list consists of three parts: (i) sources and sinks from SuSi (Rasthofer et al., 2014) GitHub repository; (ii) sources and sinks detected by the machine-learning approach SWAN (Piskachev et al., 2019) when applying it to Android platform jars (API level 3 to 29); (iii) sources and sinks documented in both TaintBench and DroidBench. Based on this list, we computed the values of above metrics for each app. The summarized results are shown in Table 8. We report the minimum, the maximum, and the geometric mean[9] of each suite and the Contagio dataset. Clearly, TaintBench employs more usages of sources and sinks than DroidBench, since all values of TaintBench are at least six times higher than those of DroidBench. In addition, the geometric means of TaintBench are in the same order of magnitude as for the entire Contagio dataset. Especially regarding the number of different sources and sink APIs appeared in an app, the difference between TaintBench and Contagio is less than 6. In conclusion, this indicates that a tool must be scalable to handle more data-flow propagations on TaintBench to achieve equally good results as on DroidBench.

Call-graph Complexity One of the most important tasks for inter-procedural analysis is to construct the call graph. We use AndroGuard (Androguard,

[9] When there are 0s in the dataset, we computed the geomean of all positive numbers.

Table 8 Usage of Sources and Sinks

	#Sources	#Usage Sources	#Sink	#Usage Sinks
		DroidBench		
min	0	0	1	1
max	8	27	13	22
geomean	2.2	2.4	2.8	3.3
		TaintBench		
min	4	9	6	13
max	514	4284	369	3486
geomean	49	149	38.4	133.2
		Contagio		
min	2	2	0	0
max	699	8044	495	7870
geomean	55.1	201.6	43.7	182.0

Table 9 Call-Graph Complexity

	CG Size	Max. CCL	LC Size
	DroidBench		
min	11	1	0
max	144	4	0
geomean	27.28	1.4	0
	TaintBench		
min	112	1	0
max	83981	16	31
geomean	1895.68	4.79	3
	Contagio		
min	43	0	0
max	139298	44	31
geomean	2599.6	6.3	2.5

Table 10 Analysis Time(s) measured for FlowDroid and Amandroid

	FlowDroid		Amandroid	
	TaintBench	Contagio	TaintBench	Contagio
min	2.5	2.3	16.2	0.6
max	361.6	361.6	762	6949.3
geomean	8.5	8.2	71.8	113.8

2011) to generate context-insensitive call graphs. To compare fairly, we excluded call graph edges from Android platform APIs to the actual application and edges between Android platform APIs themselves. Based on the resulting sub call graph, we compute:

- **Call Graph Size (CG Size)**: Number of edges in the call graph.
- **Maximal Call Chain Length (Max. CCL)**: Number of edges in the longest acyclic call chain (Rountev et al., 2004; Eichberg et al., 2015).
- **Longest Cycle Size (LC Size)**: Number of edges in the longest cycle in the call graph.

The call-graph comparison is shown in Table 9. The geometric mean indicates, that call graphs in TaintBench are much larger and more complex than in

DroidBench. In comparison to the entire Contagio dataset, the call graphs in TaintBench are smaller. However, as shown in Table 10, the minimum, maximum and geometric mean of analysis time used by FlowDroid for an app in TaintBench is almost the same as the time required with respect to the entire Contagio set.

While the call graphs in DroidBench have no cycles (recursions), since the values of LC size are all zeros, the call graphs in TaintBench include even large cycles (LC size up to 31). Recursion is considered as an important problem that needs to be solved in a context-sensitive analysis. The absence of it in DroidBench makes it impossible to evaluate the implemented solution for handling recursions. Max. CCL can be seen as an indicator for the choice of context string length (call string length) of a context-sensitive analysis. If the context string length is chosen too small, the analysis can lose precision and soundness. When the length is too big, the analysis may not scale. To build a scalable context-sensitive analysis that produces proper results, the context string length for a best trade-off between precision and scalability may be easy to find for DroidBench with Max. CCL varying from 1 to 4, however, it is much more difficult to find the best fit in TaintBench, since the maximum is up to 16.

Code Complexity We compare TaintBench and DroidBench by computing the following Chidamber and Kemerer (CK) metrics (Chidamber and Kemerer, 1994): Coupling between object classes (CBO), Depth of Inheritance Tree (DIT), Response for a Class (RFC) and Weighted Method per Class (WMC). These were often used to evaluate software complexity (Prokopec et al., 2019; Blackburn et al., 2006). Beside the CK metrics, we also compare number of fields and static fields in the benchmark apps. We used the ck tool (Maurício Aniche, 2015) on the source code project of each benchmark app to calculate the metrics. We also computed other common metrics such as number of methods and number of classes. The in-depth results are listed on our website.[10] In summary, all measurements show that TaintBench benchmark apps are more complex than DroidBench benchmark apps. While this is not surprising, we find it important to compute these numbers for future reference.

6.2 RQ4: How effective are taint analysis tools on TaintBench compared to DroidBench?

Tool and Benchmark Selection: We evaluate two taint analysis tools, namely Amandroid and FlowDroid. These two tools were chosen because they lately scored best in two independent studies (Pauck et al., 2018; Qiu et al., 2018) when evaluated on DroidBench and they are based on distinct analysis frameworks. Two different versions of both tools are employed: (1) the

[10] `https://taintbench.github.io/evaluation`

respective up-to-date version, and (2) the version used by Pauck et al. (Pauck et al., 2018) in order to compare our reproduced values to theirs. In the following, we mark the current tool versions by a *-symbol as shown in Table 11. TaintBench and DroidBench (3.0) are selected as benchmark suites for all the experiments. Note, because both tools cannot analyze inter-app communication scenarios, the related benchmark cases of DroidBench are not considered in our setup.

Evaluation Objectives: In the context of TaintBench, we focus on evaluating analysis accuracy in terms of precision, recall and F-measure but less on analysis time.

Execution Environment: The TaintBench framework was setup on an Debian (9 – Stretch) virtual machine with two cores of an Intel©Xeon®CPU (E5-2695 v3@2.30GHz), 128 GB memory and Java 8 (Oracle 1.8.0_231) installed. 96 GB memory were reserved for the analysis tools.

Experiments: Table 12 lists all conducted experiments. The first column ID refers to the benchmark suite and experiment number (e.g., TB2 refers to Experiment 2 w.r.t. TaintBench). This ID is used throughout the whole section. The second column provides a brief description of each experiment. The last column indicates the comparability of experimental results. Accordingly, the results of all experiments except TB5 and TB6 are comparable with one another.

We first conducted all experiments with the 149 expected taint flows identified by us manually in the benchmark construction phase as described in Section 3.1. Afterwards, we manually checked and rated newly discovered taint flows reported by all tools in TB3 and added them as expected and unexpected taint flows into the baseline and re-run all experiments. In the following, we report the results using the final baseline of the TaintBench suite presented in Table 5. The accuracy metrics are computed with the following equations:

$$Precision = \frac{TP}{TP + FP} \quad , \quad Recall = \frac{TP}{P} \quad , \quad F - measure = \frac{2Precision \cdot Recall}{Precision + Recall}$$

where TP is the number of true positives, FP the number of false positives and P the number of expected cases in the benchmark suite.

Table 11 Tools Evaluated

Tool	Version
Amandroid (Amandroid, 2017)	November 2017 (3.1.2)
Amandroid* (Amandroid*, 2018)	December 2018 (3.2.0)
FlowDroid (FlowDroid, 2017)	April 2017 (Nightly)
FlowDroid* (FlowDroid*, 2019)	January 2019 (2.7.1)

* Up-to-date tool versions.

Table 12 Descriptions of Experiments

ID	Description	
DB1	Default configuration; evaluated on *DB*	✔
TB1	Default configuration; evaluated on *TB*	✔
DB2	Sources & sinks w.r.t. *DB* (**Suite**-Level)	✔
TB2	Sources & sinks w.r.t. *TB* (**Suite**-Level)	✔
TB3	Sources & sinks w.r.t. *TB* (**App**-Level)	✔
TB4	Sources & sinks w.r.t. *TB* (**Case**-Level)	✔
TB5	w.r.t. minified apps per *TB* case	✘
TB6	w.r.t. delta apps per *TB* case	✘

DB: DroidBench, *TB*: TaintBench

6.2.1 Experiment 1 (DB1 & TB1):

The tools are executed in their default configuration. Figure 8 presents precision, recall and F-measure for DroidBench and TaintBench in column DB1 and TB1, respectively.

> The results obtained for FlowDroid and Amandroid in configuration DB1 are identical to those in the ReproDroid study (Pauck et al., 2018), replicating the results obtained there.

The metrics (shown in the bar charts) are calculated based on the table in Figure 8, which also shows the expected cases and unexpected cases defined for each benchmark suite. The evaluation is based on these cases only. Al-

	DroidBench				TaintBench											
	Expected Cases: 163				Expected Cases: 186											
	Unexpected Cases: 41				Unexpected Cases: 35											
	DB1		DB2		TB1		TB2		TB3		TB4		TB5		TB6	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
Amandroid	80	12	95	15	2	2	31	19	31	19	31	18	29	4	8	42
Amandroid*	41	10	46	10	5	1	6	7	6	7	6	6	7	1	1	12
FlowDroid	89	12	96	12	26	0	84	10	102	11	101	9	59	8	51	62
FlowDroid*	87	12	95	12	16	1	41	14	43	14	42	13	28	5	23	34

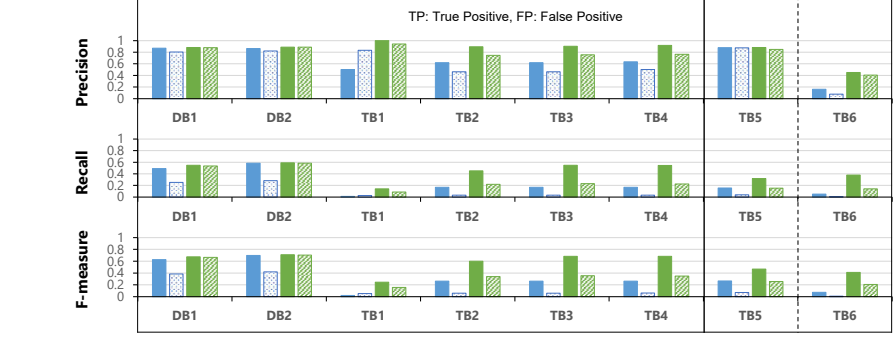


Fig. 8 Experiments 1-6 Result-Overview

Table 13 Intersection (∪) and difference (∖) of source and sink sets used by analysis tools (AMANDROID, FLOWDROID) and involved in benchmark cases of DROIDBENCH and TAINTBENCH.

A = \ B =	AMANDROID		FLOWDROID		DROIDBENCH		TAINTBENCH	
	Sources	Sinks	Sources	Sinks	Sources	Sinks	Sources	Sinks
Intersection ($\|A \cap B\|$)								
AMANDROID	30	42	24	38	4	8	6	4
FLOWDROID	24	38	89	133	7	9	12	8
DROIDBENCH	4	8	7	9	15	23	7	4
TAINTBENCH	**6**	**4**	**12**	**8**	**7**	**4**	**44**	**44**
Difference ($\|A \setminus B\|$)								
AMANDROID	0	0	6	4	26	34	24	38
FLOWDROID	65	95	0	0	82	124	77	125
DROIDBENCH	11	15	8	14	0	0	8	19
TAINTBENCH	**38**	**40**	**32**	**36**	**37**	**40**	**0**	**0**

though the baseline definition of TAINTBENCH contains 203 expected and 46 unexpected taint flows, REPRODROID can only reflect 186 expected cases and 35 unexpected cases, since it does not distinguish different flows, when the sources and sinks look exactly the same in Jimple. Jimple statements are not differentiable (by their textual representation) if they are (1) occuring in the same method of the same class, (2) use variables with the exact same names as well as constants with the same contents, (3) and refer to the same source code line number. As the figure shows, the precision of AMANDROID is dramatically decreased to 50% when evaluated on TAINTBENCH. It only found 4 flows in our baseline and 2 of them are false positives. The precision of AMANDROID* stays almost unchanged, however, this is calculated only from 6 flows. In contrast, the precisions of both FLOWDROID and FLOWDROID* are high (over 90%). However, on TAINTBENCH all tools show a significantly lower recall and F-measure than for DROIDBENCH. In the default configuration most taint flows in TAINTBENCH remain undetected. With 14% (26/186), FLOWDROID's recall is still the highest.

6.2.2 Experiment 2 (DB2 & TB2):

To understand if the low recall values for TAINTBENCH in Experiment 1 are mainly caused by the tools' source and sink configurations, we compared the different source and sink sets involved. The results of this comparison are summarized in Table 13. Tool names refer to source and sink sets defined in a tool's default configuration. Benchmark suite names refer to the sets occurring in their benchmark cases. While the upper part of the table row-wise shows the intersections of these sets in terms of numbers of sources and sinks, the lower part enumerates sources and sinks contained in one set A but not in another set B. The two rows labeled with TAINTBENCH show that (i) the sets of sources and sinks used by tools and DROIDBENCH have only minor intersections with the set of TAINTBENCH; (ii) TAINTBENCH holds at least 32 different sources and 36 sinks (see column FLOWDROID).

In consequence, we generated a list of sources and sinks for each benchmark suite based on the comprised taint flows, using TB-Loader (see Section 4.2). These lists, generated on the suite-level, are configured to be used by the two tools. As shown in Figure 8, when re-configuring sources and sinks this way, the results for DroidBench are affected only slightly (DB1 vs. DB2) but the recall and F-measure values for TaintBench are more than doubled (TB1 vs. TB2). Nonetheless, even under this configuration the tools are still less effective on TaintBench than on DroidBench. Closest is FlowDroid, which achieves a recall of 45% (84/186) for TaintBench while reaching 59% (96/163) for DroidBench. While FlowDroid reports 84 true positives, Amandroid and Amandroid* detect only 31 and 6 true positives in TaintBench, respectively.

Surprisingly, on TaintBench the current tool versions (Amandroid* and FlowDroid*) show a lower recall than their predecessors (Amandroid and FlowDroid). This argues for the use of TaintBench also for regression testing.

For FlowDroid and FlowDroid*, the difference is small (1 or 2 flows) regarding DroidBench. Considering TaintBench, FlowDroid* finds only half (41/84) of the true positives that can be found by its old version even under the same source and sink configuration. In addition, FlowDroid* is less precise, since it reports more false positives than FlowDroid (14 vs. 10).

6.3 RQ5: What insights can we gain by evaluating analysis tools on TaintBench?

6.3.1 Experiment 3 & 4 (TB3 & TB4):

Since the results of Experiment 2 show that source and sink configurations affect the recall values heavily, we conduct two more experiments in which sources and sinks are configured regarding not just each suite but even each benchmark app (Experiment 3) and each benchmark case (Experiment 4). To this end, we are now using smaller but more precise sets. We expected that the results would be the same compared to Experiment 2, and this also holds for Amandroid(*).

Surprisingly, FlowDroid and FlowDroid* find *more* taint flows in Experiment 3 and 4. The number of true positives increases from 84 to 102 for FlowDroid and from 41 to 43 for FlowDroid*. This indicates that the configuration of "superfluous" sources and sinks, which are actually irrelevant for a specific benchmark app or case, has some shadow effect on the taint computation in FlowDroid and FlowDroid*.

Furthermore, we made the following observation: One true-positive flow (A) is detected by FlowDroid* in Experiment 3, but not in Experiment 4. Instead, in Experiment 4 a *different* true-positive flow (B) is detected[11].

Considering flow (A), we found that FlowDroid* sometimes does not find a taint flow ($source \rightarrow Child.sink$) when $Parent.sink$ was *not* declared in the list of sources and sinks, where $Child$ is a subclass of the class $Parent$. By intuition the reason seems to be that the flow is only detected when $Parent.sink$ is configured as in Experiment 3. Thus, when $Parent.sink$ is not configured in the list, the flow to $Child.sink$ remains undetected as in Experiment 4.

Moreover, in case of (B) there are two flows ($source_1 \rightarrow sink_1$) and ($source_2 \rightarrow sink_1$) with the same sink but FlowDroid* reports only one of them in Experiment 3. However, the internal analysis of FlowDroid* is actually capable of finding both flows in Experiment 4, namely, the new true positive (B) is detected. After a closer investigation, we found out that when more sources and sinks than the source and sink of the expected taint flow are configured for FlowDroid* (Experiment 3) one sink overshadows the other. The order of the relevant sources and sinks appear on two parallel paths in the inter-procedural control-flow graph (ICFG). These path can be illustrated as follows:

$path_1$: $source_1 \rightarrow sink_2 \rightarrow sink_1$
$path_2$: $source_2 \rightarrow sink_1$

In Experiment 3, two flows are found: ($source_1 \rightarrow sink_2$), ($source_2 \rightarrow sink_1$). However, the expected one ($source_1 \rightarrow sink_1$) remains undetected which is not the case in Experiment 4. Because $sink_1$ appears later than $sink_2$ in $path_1$, we think that FlowDroid* stops the propagation of taints from $source_1$ when the taints reach $sink_2$. We reported our findings to the tool maintainers.

6.3.2 Experiment 5 (TB5)

With this experiment we seek to test if the call-graph complexity of real-world apps in TaintBench is a cause of some unsatisfactory results. To do so, irrelevant call-graph edges are removed by MinApk-Generator (see Section 4.2.1). This leads to fewer timeouts in all cases in Experiment 5 in comparison to Experiment 2 (cf. Table 14; −5 for Amandroid* and −1 for all other tools). Overall, 27 new true positives are uniquely detected in Experiment 5. 12 of these by Amandroid and 8 by FlowDroid. The newer tools are less affected with 3 and 4 newly found true positives in case of Amandroid* and FlowDroid*. The fact that Experiment 5 mostly simplifies the call graph indicates that the tools likely miss these flows in the original benchmark apps due to incomplete call graphs.

As a side-effect of the changes made by MinApk-Generator, the tools were unable to detect some previously detected true positives. Amandroid also reported one new false positives. For example, when MinApk-Generator

[11] A: Flow with ID=1 in *overlay_android_samp*. B: Flow with ID=7 in *cajino_baidu*.

uses an `ActionBarActivity` class for entry-point creation, this class does not appear in the `dummyMain` method generated by FlowDroid(*). Hence, the results of Experiment 5 look overall worse in Figure 8.

6.3.3 Experiment 6 (TB6)

With this experiment we check if the tools handle data sanitization properly. We use the DeltaApk-Generator to kill each expected taint flow in the baseline definition (see Section 4.2.1). If a previously detected true positive in Experiment 3 is still detected in the respective delta APK (Experiment 6), this flow is a known false positive (TB3 vs. TB6). If the tools were to fully handle the killing of flows, also known as "strong updates", the results of both experiments should be identical. This is not the case: both precision and recall decreased (see TB6 in Figure 8). To this effect, and because of the adapted interpretation, the results should not be compared to the experiments above.

> The tools over-approximate the killing of taint flows, i.e., miss the ability to perform strong updates, which produces a number of false positives on real-world apps.

6.3.4 Further Important Insights

Timeouts and Unsuccessful Exits: The maximal execution time per app is set to 20 minutes during all experiments. On DroidBench (DB1, DB2), neither Amandroid(*) nor FlowDroid(*) reach this timeout. On TaintBench, though, Amandroid* exceeds the timeout on 11 apps. All other tools rarely do so (see Table 14). However, for one DroidBench and seven TaintBench apps FlowDroid* claimed to find no analysis entry point, even though FlowDroid was able to find those — a clear regression. In case of two other TaintBench apps FlowDroid* failed its analysis, since it was unable to calculate callbacks. Consequently, on TaintBench FlowDroid* finds fewer than half of the true positives that can be found by FlowDroid.

> It is alerting that both newer tool versions fail to analyze a striking number of benchmark apps, particularly where earlier analysis versions succeeded.

Table 14 Timeouts, Unsuccessful Exits and Analysis Time in Experiment 2

	DroidBench (DB2)				TaintBench (TB2)			
	AD	AD*	FD	FD*	AD	AD*	FD	FD*
Timeouts	0	0	0	0	1	11	1	1
Unsuccessful Exits	0	0	0	1	0	0	0	9
Analysis Time (min)	58	61	20	13	98	41	17	5
↪ incl. Timeouts	58	61	20	13	118	261	37	27

AD: Amandroid, FD: FlowDroid

Analysis Time: In all scenarios, both versions of FlowDroid are faster than any version of Amandroid. With respect to DroidBench, FlowDroid* is 35%[12] faster than its predecessor (see Table 14). Considering TaintBench, FlowDroid*'s speed-up is even larger (71%[13]). Amandroid* is not faster than Amandroid on DroidBench but 58%[14] faster in case of TaintBench. However, the amount of timeouts thrown by Amandroid* and unsuccessful exits of FlowDroid* impede a fair comparison. While new tool versions appear to be faster, the number of timeouts or unsuccessful exits has risen.

7 Continuous Benchmarking

From our experiments, we found out that newer tool versions perform worse than their predecessors when evaluating on TaintBench. To help the tool authors avoid such regressions in the future, we aim to provide a way in which Android taint analysis tools can be evaluated on TaintBench on a continuous basis. We set up GitHub Actions (GitHub, 2020) for both versions of Amandroid and FlowDroid[15]. Using the TaintBench framework, we were able to configure the evaluation of each tool as an automated workflow of Github Actions. The source and sink configuration of each tool is at app-level as in Experiment 3 (see Section 6.3.1). The outcome of each workflow includes a benchmark file computed by ReproDroid containing performance metrics (precision, recall, F-measure, analysis time) and raw analysis results of the tool. Each workflow will be triggered on pushes or pull requests to the TaintBench GitHub repository. This way we can easily obtain performance improvements and regressions of newer tool releases evaluated on the newest version of the TaintBench suite in the future. We also provide detailed instructions allowing other Android taint analysis tools to be easily adapted for such continuous benchmarking on TaintBench.

8 Threats to Validity

The *external* validity of the TaintBench suite itself is threatened by the fact that we were forced to exclude obfuscated applications. Also, while all taint specifications have been checked multiple times by at least three authors, there is the potential threat that the baseline definition nonetheless misses some actual taint flows (expected cases). Regarding the usability test, one external threat is embodied by the generalization of the presented results. Given that the participants were invited by us and voluntary, it is possible that they are not representative of the general population of experts in taint analysis.

[12] 35%=(20-13)/20
[13] 71%=(17-5)/17
[14] 58%=(98-41)/98
[15] More information can be found on `https://taintbench.github.io/ci`

The *internal* validity of the usability test is threatened by the fact that participants were aware of the measurement of time used for each task. This may have influenced their behavior during the study. The internal validity of the TaintBench suite is impeded by the time passed since the malware apps were created. These malware apps do not target the latest Android API level. Nevertheless, we were able to install and execute almost all of the benchmark apps[16] on a Nexus 4 Android emulator with API level 25. Many apps composed in the TaintBench suite are not functional anymore, since they have been communicating with public servers which are not accessible anymore. Most-likely the servers were actively taken down when the apps were identified as malware by researchers and field experts. Because of this, only parts of TaintBench can be used to benchmark dynamic taint analysis tools. This in turn embodies another reason, why dynamic tools could not be used to verify our baseline definition.

Note, this paper focuses on the construction and evaluation of our novel benchmark suite; an in-depth investigation of reasons for our findings in the tools' implementations exceeds the scope of this paper. During our experiments we noticed that FlowDroid's results are non-deterministic (two runs of FlowDroid(*) with the same inputs produce different results). This is a known issue first mentioned by Benz et al. (Benz et al., 2020), which has not yet been fixed. This issue is hardly notable in DroidBench experiments but well recognizable when analyzing larger apps as in TaintBench.

Regarding our experiments using TaintBench, we are aware of the internal threat caused by the measurement used in ReproDroid. If there are two source or sink statements at different positions in the source code (e.g., `s=source()` at line 5 and `s=source()` at line 10 in a method `foo()`), which look identical in Jimple, ReproDroid cannot distinguish them due to missing exact code locations in the results produced by the taint analysis tools. Thus, if one of them is detected, ReproDroid regards both flows as detected. In consequence, the actual recall on TaintBench might be lower than observed. This issue could be mitigated if the analysis tools were to include unique statement identifiers e.g., line numbers in their results. We added this functionality to FlowDroid for future releases. The related pull request is already merged.[17]

9 Future Work and Conclusion

In future, we will update and maintain the TaintBench suite. We also want to include more modern apps targeting updated API levels into the TaintBench suite to increase its complexity and hope that the community will join us. In addition, we plan to reconstruct the malicious taint flows in new apps targeting higher API levels. These new artificial apps can be used for benchmarking dynamic approaches. To simplify the use of TaintBench in continu-

[16] Except the app cajino_baidu

[17] `https://github.com/secure-software-engineering/FlowDroid/pull/222`

ous benchmarking and in regressions tests, we will improve the TaintBench framework such that any benchmark constructed with it can be exported as a set of JUnit test cases.

Regarding future work on improving Android taint analysis tools, we case studied some false negatives that were not detected by FlowDroid during our experiments. Our first result reveals that many call sites of sources and sinks in those false negatives were not in the call graphs used by FlowDroid. In future, we plan to work on designing new call graph construction algorithms that can improve the recall. We are also considering organizing competitions for Android taint analysis tools with our benchmark suite to encourage further improvement and development in this area.

To conclude, our novel real-world malware benchmark suite TaintBench, constructed with respect to the criteria determined for taint analysis benchmark suites, reveals insights that could not be gained with the micro benchmark DroidBench. The associated experiments revealed surprising facts about taint analysis tools: (i) Android taint analysis tools have difficulties in detecting real-world taint flows in malware apps, yielding very low recall. (ii) For Amandroid the situation is particularly bad: the latest version detects almost no taint flow. (iii) While FlowDroid shows better recall, a configuration using superfluous sources and sinks that are actually irrelevant for specific taint flows has a shadow effect on its taint computation, causing it to miss some actual flows. (iv) For both Amandroid and FlowDroid, new tool releases are less accurate than their predecessors: as we have shown in Experiments 2, 3 and 4, new releases have lower precision, recall and F-measure. Hence, we recommend to involve TaintBench in future evaluations of Android taint analyses and their regression tests.

References

Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K (2014) DREBIN: effective and explainable detection of android malware in your pocket. In: Proceedings of the 21st NDSS, The Internet Society

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon YL, Octeau D, McDaniel PD (2014) Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of PLDI, ACM

Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, Bodden E (2015) Mining apps for abnormal usage of sensitive data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol 1, pp 426–436, DOI 10.1109/ICSE.2015.61

Benz M, Krogh Kristensen E, Luo L, P Borges Jr N, Bodden E, Zeller A (2020) Heaps'n leaks: How heap snapshots improve android taint analysis. In: Proceedings of the 42nd International Conference on Software Engineering, URL `https:`

//2020.icse-conferences.org/details/icse-2020-papers/114/Heaps-n-Leaks-How-Heap-Snapshots-Improve-Android-Taint-Analysis

Blackburn SM, Garner R, Hoffmann C, Khan AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking AL, Jump M, Lee HB, Moss JEB, Phansalkar A, Stefanovic D, VanDrunen T, von Dincklage D, Wiedermann B (2006) The dacapo benchmarks: java benchmarking development and analysis. In: Proceedings of the 21th OOPSLA, ACM

Bohluli Z, Shahriari HR (2018) Detecting privacy leaks in android apps using inter-component information flow control analysis. In: Proceedings of the 15th ISCISC, IEEE

Bosu A, Liu F, Yao DD, Wang G (2017) Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: Proceedings of AsiaCCS, ACM

Brooke J (1996) Sus: a "quick and dirty'usability. Usability evaluation in industry p 189

Cam N, Hau P, Nguyen T (2016) Android Security Analysis Based on Inter-application Relationships, pp 689–700. DOI 10.1007/978-981-10-0557-2_68

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. Transactions on Software Engineering IEEE

Do LNQ, Eichberg M, Bodden E (2016) Toward an automated benchmark management system. In: Proceedings of the 5th SOAP@PLDI, ACM

Eichberg M, Hermann B, Mezini M, Glanz L (2015) Hidden truths in dead software paths. In: Proceedings of the 10th ESEC/FSE, ACM

Enck W, Gilbert P, Chun B, Cox LP, Jung J, McDaniel PD, Sheth A (2014) Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. Communications of the ACM

Gordon MI, Kim D, Perkins JH, Gilham L, Nguyen N, Rinard MC (2015) Information flow analysis of android applications in droidsafe. In: Proceedings of the 22nd NDSS, The Internet Society

Grech N, Smaragdakis Y (2017) P/taint: Unified points-to and taint analysis. Proc ACM Program Lang 1(OOPSLA), DOI 10.1145/3133926, URL https://doi.org/10.1145/3133926

Huang J, Zhang X, Tan L, Wang P, Liang B (2014) Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In: Proceedings of the 36th ICSE, ACM

Huang W, Dong Y, Milanova A, Dolby J (2015) Scalable and precise taint analysis for android. In: Proceedings of ISSTA, ACM

Lam P, Bodden E, Lhoták O, Hendren L (2011) The Soot framework for Java program analysis: a retrospective. In: Proceedings of CETUS, URL http://www.bodden.de/pubs/lblh11soot.pdf

Li L, Bartel A, Bissyandé TF, Klein J, Traon YL, Arzt S, Rasthofer S, Bodden E, Octeau D, McDaniel PD (2015) Iccta: Detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th ICSE, IEEE Computer Society

Livshits VB, Lam MS (2005) Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th USENIX Security Symposium, USENIX Association

Luo L, Bodden E, Späth J (2019a) A qualitative analysis of android taint-analysis results. In: Proceedings of the 34th ASE, IEEE

Luo L, Dolby J, Bodden E (2019b) Magpiebridge: A general approach to integrating static analyses into ides and editors (tool insights paper). In: Proceedings of the 33rd ECOOP, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, LIPIcs, vol 134

Contagio Mobile Malware (2018) URL `http://contagiomobile.deependresearch.org/index.html`, Accessed 02/18/2021

Mitra J, Ranganath V (2017) Ghera: A repository of android app vulnerability benchmarks. In: Proceedings of the 13th PROMISE, ACM

Pauck F, Wehrheim H (2019) Together strong: cooperative android app analysis. In: Proceedings of ESEC/FSE, ACM

Pauck F, Zhang S (2019) Android app merging for benchmark speed-up and analysis lift-up. In: Proceedings of the 2nd A-Mobile@ASE, IEEE

Pauck F, Bodden E, Wehrheim H (2018) Do android taint analysis tools keep their promises? In: Proceedings of ESEC/FSE, ACM

Piskachev G, Do LNQ, Bodden E (2019) Codebase-adaptive detection of security-relevant methods. In: Proceedings of the 28th ISSTA, ACM

Prokopec A, Rosà A, Leopoldseder D, Duboscq G, Tuma P, Studener M, Bulej L, Zheng Y, Villazón A, Simon D, Würthinger T, Binder W (2019) Renaissance: benchmarking suite for parallel applications on the JVM. In: Proceedings of the 40th PLDI, ACM

Qiu L, Wang Y, Rubin J (2018) Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In: Proceedings of the 27th ISSTA, ACM

Ranganath V, Mitra J (2020) Are free android app security analysis tools effective in detecting known vulnerabilities? Empirical Software Engineering 25(1):178–219, DOI 10.1007/s10664-019-09749-y, URL `https://doi.org/10.1007/s10664-019-09749-y`

Rasthofer S, Arzt S, Bodden E (2014) A machine-learning approach for classifying and categorizing android sources and sinks. In: Proceedings of the 21st NDSS, The Internet Society

Rastogi V, Chen Y, Jiang X (2013) Droidchameleon: evaluating android anti-malware against transformation attacks. In: 8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013, ACM

Reif M, Eichberg M, Hermann B, Mezini M (2017) Hermes: assessment and creation of effective test corpora. In: Proceedings of the 6th SOAP@PLDI, ACM

Rountev A, Kagan S, Gibas M (2004) Static and dynamic analysis of call chains in java. In: Proceedings of ISSTA, ACM

Vallée-Rai R, Co P, Gagnon E, Hendren LJ, Lam P, Sundaresan V (1999) Soot - a java bytecode optimization framework. In: Proceedings of CASCON, IBM

Wei F, Roy S, Ou X, Robby (2014) Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of CCS, ACM

Wei F, Li Y, Roy S, Ou X, Zhou W (2017) Deep ground truth analysis of current android malware. In: Proceedings of the 14th DIMVA, Springer, Lecture Notes in Computer Science, vol 10327

Wong MY, Lie D (2016) Intellidroid: A targeted input generator for the dynamic analysis of android malware. In: Proceedings of the 23rd NDSS, The Internet Society

Yang T, Qian K, Li L, Lo DC, Tao L (2016) Static mining and dynamic taint for mobile security threats analysis. In: Proceedings of SmartCloud, IEEE Computer Society

Youssef A, Shosha AF (2017) Quantitave dynamic taint analysis of privacy leakage in android arabic apps. In: Proceedings of the 12th International Conference on Availability, Reliability and Security, Association for Computing Machinery, New York, NY, USA, ARES '17, DOI 10.1145/3098954.3105827, URL https://doi.org/10.1145/3098954.3105827

Zhang J, Tian C, Duan Z (2019) Fastdroid: efficient taint analysis for android applications. In: Proceedings of the 41st ICSE, IEEE / ACM

Zheng C, Zhu S, Dai S, Gu G, Gong X, Han X, Zou W (2012) Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: Proceedings of SPSM@CCS, ACM

Amandroid (2017) URL https://bintray.com/arguslab/maven/argus-saf/3.1.2, Accessed 02/16/2020

Amandroid* (2018) URL https://bintray.com/arguslab/maven/argus-saf/3.2.0, Accessed 02/16/2020

Androguard (2011) URL https://github.com/androguard/androguard, Accessed 02/16/2020

AQL (2020) Android app analysis query language (aql). URL https://foellix.github.io/AQL-System, Accessed 02/16/2020

Contagio Mobile (2012) URL http://contagiominidump.blogspot.com, Accessed 02/16/2020

DroidBench 3-0 (2016) URL https://github.com/secure-software-engineering/DroidBench/tree/develop, Accessed 02/16/2020

F-Droid (2020) F-droid. URL https://F-Droid.org, Accessed 02/16/2020

FlowDroid (2017) URL https://github.com/secure-software-engineering/soot-infoflow-android/wiki, Accessed 02/16/2020

FlowDroid* (2019) URL https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.7.1, Accessed 02/16/2020

GitHub (2020) Github actions. URL https://docs.github.com/en/actions, Accessed 07/05/2020

Gitpod (2019) Online ide for github. URL https://www.gitpod.io, Accessed 02/16/2020

JADX (2020) Dex to Java decompiler. URL https://github.com/skylot/jadx, Accessed 02/16/2020

Maurício Aniche (2015) Java code metrics calculator (CK). URL https://github.com/mauricioaniche/ck, Accessed 02/16/2020

Micro T (2020) New tekya ad fraud found on google play. URL https://blog.trendmicro.com/trendlabs-security-intelligence/new-tekya-ad-fraud-found-on-google-play/, Accessed 29.06.2020

Microsoft (2020) VSC - Visual Studio Code. URL https://code.visualstudio.com, Accessed 02/16/2020

OASIS (2019) Static analysis results interchange format (sarif) version 2.0. URL https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.html, Accessed 02/16/2020

OWASP (2021) Owasp benchmark. URL https://owasp.org/www-project-benchmark/, Accessed 07.05.2021

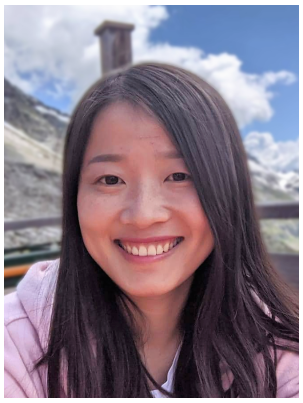
Rasthofer S (2013) The android logging service – a dangerous feature for user privacy? URL https://blogs.uni-paderborn.de/sse/2013/05/17/privacy-threatened-by-logging, Accessed 02/16/2020

Soni J (2020) This dangerous malware got around google play store security. URL https://www.techradar.com/news/phantomlance-malware-breaches-google-play-store-security, Accessed 29.06.2020

statcounter (2019) Operating system market share worldwide jan - dec 2019. URL https://gs.statcounter.com/os-market-share#monthly-201901-201912-bar, Accessed 02/16/2020

VirusShare (2014) URL https://virusshare.com, Accessed 02/16/2020

Short Biography



Linghui Luo is a PhD candidate and a research associate in the Secure Software Engineering Group at the Heinz Nixdorf Institute of Paderborn University, Germany. She received the silver medal in the ACM Student Research Competition at the ESEC/FSE conference 2021. Her research interests include program analysis (mainly taint analysis for Java and Android applications), empirical software engineering and usability of program analysis tools. She is the creator of the MagpieBridge framework for IDE integration and a contributor to the static analysis framework Soot.

Felix Pauck received his M.Sc. degree in Computer Science from Paderborn University (2017), since then he works as a doctoral candidate at the Paderborn University. He started his work on the cooperative analysis of Android apps by finishing his masters' thesis. Currently his main research interests are software analysis and benchmarks.

Goran Piskachev is a research associate at Fraunhofer IEM in Paderborn and a PhD student at Paderborn University. He received his master degree in computer science from Paderborn University. Previously, he completed an engineering degree at the Ss. Cyril and Methodius University in Skopje. His research interests include static code analysis, security testing, domain specific languages, and machine learning for code analysis.

Manuel Benz earned his master's degrees in Computer Science and IT Security in 2016 from Technische Universität Darmstadt. During that time he also worked with Prof. Bodden on detecting misusages of cryptographic APIs at Fraunhofer SIT. After his master's, Manuel joined Prof. Bodden's working group Secure Software Engineering at Paderborn University as a Ph.D. student. His research was focused on combining static and dynamic program analysis techniques to mitigate the shortcomings of either approach. Based on their research, he and his colleagues Andreas Dann and Johannes Späth founded the cloud-native-security-focused start-up CodeShield in 2019. As a co-founder and CTO of CodeShield, Manuel carries further his research to help develops build secure cloud-native applications.

Ivan Pashchenko (PhD 2019) is a Research Assistant Professor at the University of Trento. He holds the silver medal for the ACM/Microsoft student Research competition in the graduate category in 2017. His research interests include open-source software security, software verification, and machine learning for security. Contact him at ivan.pashchenko@unitn.it.

Martin Mory (M.Sc. 2016) works in the research group Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. His main focus of research is static data-flow analysis for C/C++ software, particularly pointer analysis.

Eric Bodden is a full professor for Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. He is further the director for Software Engineering and IT Security at the Fraunhofer Institute for Engineering Mechatronic Systems Design. Prof. Bodden has been recognized several times for his research on program analysis and software security, most notably with the German IT-Security Price and the Heinz Maier-Leibnitz Price of the German Research Foundation, as well as with several distinguished paper and distinguished reviewer awards. He is an ACM Distinguished Member.

Ben Hermann is an assistant professor at the Technical University of Dortmund. He works on evolutionary software security and has been the author of several works in the field of static program analysis. Prof. Hermann worked on several static analysis frameworks including PhASAR, Soot, and OPAL and has significant experience in engineering these frameworks and the analyses build on top of them. He received his doctorate degree from the University of Darmstadt for his work on Java security.

Fabio Massacci (PhD 1997) is a professor at the University of Trento, Italy, and Vrije Universiteit Amsterdam, The Netherlands. He received the Ten Years Most Influential Paper award by the IEEE Requirements Engineering Conference in 2015. He coordinates the education activities of the European Competence Center Pilot CyberSec4Europe and is the European coordinator of the AssureMOSS project. Contact him at fabio.massacci@ieee.org.