Systematic Mutation-Based Evaluation of the Soundness of Security-Focused Android Static Analysis Techniques

AMIT SEAL AMI and KAUSHAL KAFLE, Department of Computer Science, William & Mary
KEVIN MORAN, Department of Computer Science, George Mason University
ADWAIT NADKARNI and DENYS POSHYVANYK, Department of Computer Science, William & Mary

Mobile application security has been a major area of focus for security research over the course of the last decade. Numerous application analysis tools have been proposed in response to malicious, curious, or vulnerable apps. However, existing tools, and specifically, static analysis tools, trade soundness of the analysis for precision and performance and are hence sound*y*. Unfortunately, the specific unsound choices or flaws in the design of these tools is often not known or well documented, leading to misplaced confidence among researchers, developers, and users. This article describes the *Mutation-Based Soundness Evaluation* (μSE) framework, which systematically evaluates Android static analysis tools to discover, document, and fix flaws, by leveraging the well-founded practice of mutation analysis. We implemented μSE and applied it to a set of prominent Android static analysis tools that detect private data leaks in apps. In a study conducted previously, we used μSE to discover 13 previously undocumented flaws in FlowDroid, one of the most prominent data leak detectors for Android apps. Moreover, we discovered that flaws also propagated to other tools that build upon the design or implementation of FlowDroid or its components. This article substantially extends our μSE framework and offers a new in-depth analysis of two more major tools in our 2020 study; we find 12 new, undocumented flaws and demonstrate that all 25 flaws are found in more than one tool, regardless of any inheritance-relation among the tools. Our results motivate the need for systematic discovery and documentation of unsound choices in soundy tools and demonstrate the opportunities in leveraging mutation testing in achieving this goal.

CCS Concepts: • **Security and privacy → Software security engineering**;

Additional Key Words and Phrases: Security and privacy, software security engineering

ACM Reference format:
Amit Seal Ami, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2021. Systematic Mutation-Based Evaluation of the Soundness of Security-Focused Android Static Analysis Techniques. *ACM Trans. Priv. Secur.* 24, 3, Article 15 (February 2021), 37 pages.
https://doi.org/10.1145/3439802

This work is supported in part by NSF-1815336 and NSF-1815186.
Authors' addresses: A. S. Ami, K. Kafle, A. Nadkarni, and D. Poshyvanyk, Department of Computer Science, College of William and Mary, P.O. Box 8795, Williamsburg, VA 23187-8795, USA; emails: aami@email.wm.edu, {kkafle, nadkarni, denys}@cs.wm.edu; K. Moran, 4400 University Dr, Computer Science Department, George Mason University, Fairfax, VA 22030, USA; email: kpmoran@gw.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
2471-2566/2021/02-ART15 $15.00
https://doi.org/10.1145/3439802

1 INTRODUCTION

Mobile devices such as smartphones and tablets have become the fabric of our consumer computing ecosystems; by the year 2020, more than 80% of the world's adult population is projected to own a smartphone [29]. This popularity of mobile devices is driven by the millions of diverse, feature-rich, third-party applications or "apps" they support. However, in fulfilling their functionality, apps often require access to security and privacy-sensitive resources on the device (e.g., GPS location, security settings). Applications can neither be trusted to be well-written or benign, and to prevent misuse of such access through malicious or vulnerable apps [30, 34, 43, 64, 90, 97, 110], it is imperative to understand the challenges in securing mobile apps.

Security analysis of third-party apps has been one of the dominant areas of smartphone security research in the last decade, resulting in tools and frameworks with diverse security goals. For instance, prior work has designed tools to identify malicious behavior in apps [5, 33, 111], discover private data leaks [6, 8, 32, 42], detect vulnerable application interfaces [16, 37, 57, 67], identify flaws in the use of cryptographic primitives [30, 34, 97], and define sandbox policies for third-party apps [46, 51]. To protect users from malicious or vulnerable apps, it is imperative to assess the challenges and pitfalls of existing tools and techniques. However, *it is unclear whether existing security tools are sufficiently robust to expose particularly well-hidden unwanted behaviors.*

Our work is motivated by the pressing need to discover the limitations of application analysis techniques for Android. Existing application analysis techniques, specifically those that employ static analysis, must in practice trade soundness for precision, as there is an inherent conflict between the two properties. A sound analysis requires the technique to over-approximate (i.e., consider instances of unwanted behavior that may not execute in reality), which in turn deteriorates precision. This tradeoff has practical implications on the security provided by static analysis tools. That is, *in theory*, static analysis is expected to be sound, yet, in practice, these tools must purposefully make unsound choices to achieve a feasible analysis that has sufficient precision and performance to scale. For instance, techniques that analyze Java generally do not over-approximate analysis of certain programming language features, such as reflection, for practical reasons (e.g., Soot [101], FlowDroid [6]). Although this particular case is well-known and documented, many such unsound design choices are neither well-documented, nor known to researchers outside a small community of experts.

Security experts have described such tools as sound*y*, i.e., having a core set of sound design choices, in addition to certain practical assumptions that sacrifice soundness for precision [66]. Although soundness is an elusive ideal, sound*y* tools certainly seem to be a practical choice: *but only if the unsound choices are known, necessary, and properly documented.* However, the present state of sound*y* static analysis techniques is dire, as unsound choices (1) may not be documented, and unknown to non-experts, (2) may not even be known to tool designers (i.e., implicit assumptions), and (3) may propagate into future research. The sound*i*ness manifesto describes the misplaced confidence generated by the insufficient study and documentation of sound*y* tools, in the specific context of language features [66]. Motivated by the manifesto, we leverage sound*i*ness at the general, conceptual level of design choices, and attempt to resolve the status quo of sound*y* tools by making them more secure as well as transparent.

We describe the *Mutation-Based Soundness Evaluation* (μSE, read as "muse") framework that enables systematic security evaluation of Android static analysis tools to discover unsound design assumptions, leading to their documentation, as well as improvements in the tools themselves. In particular, this article describes our extension of our original μSE paper published in USENIX '18 in August [11]. μSE leverages the practice of mutation analysis from the software engineering (SE) domain [4, 19, 21, 25, 44, 68, 81, 82, 86, 109], and specifically, more recent advancements in mutating Android apps [63]. In doing so, μSE adapts a well-founded practice from SE to security, by making useful changes to contextualize it to evaluate security tools.

μSE creates *security operators*, which reflect the security goals of the tools being analyzed (e.g., data leak or SSL vulnerability detection). These security operators are seeded, i.e., inserted into one or more Android apps, as guided by a *mutation scheme*. This seeding results in the creation of multiple mutants (i.e., code that represents the target unwanted behavior) within the app. Finally, the mutated application is analyzed using the security tool being evaluated, and the undetected mutants are then subjected to a deeper analysis. We propose a semi-automated methodology to analyze the uncaught mutants, resolve them to flaws in the tool, and confirm the flaws experimentally.

We demonstrate the effectiveness of our approach by evaluating static analysis research tools that detect data leaks in Android apps. Based on our analysis of the discovered flaws, we provide immediate patches that address one flaw, and identify classes of design-level flaws that may be hard to address without significant research effort. Further, we perform a flaw propagation study that checks for the presence of these flaws among *seven* data leak detectors. The general contributions of this article can be summarized as follows:

— *We introduce the novel paradigm of μSE*, which provides a systematic methodology for discovering flaws in static analysis tools for Android, leveraging the well-understood practice of mutation analysis. We adapt mutation analysis for security evaluation and design the abstractions of *security operators* and *mutation schemes*.
— *We design and implement the μSE framework* for evaluating Android static analysis tools. μSE adapts to the security goals of a tool being evaluated and allows the detection of unknown or undocumented flaws.
— *We demonstrate the effectiveness of μSE by evaluating several widely used Android security tools* that detect private data leaks in Android apps. μSE detects undocumented flaws, and demonstrates their propagation. Our analysis leads to the documentation of unsound assumptions at the design level, and immediate security patches for an easily fixable but evasive flaw.

We published an earlier version of this work in USENIX '18 [11] where we analyzed FlowDroid [6] using μSE and discovered 13 previously undocumented flaws. The current version of this study substantially extends upon the previous work, in the following manner:

— **Design and implementation of μSE:** We designed a new scope-based mutation placement approach, in addition to the mutation schemes originally explored in the USENIX '18 paper [11]. Further, we refined the implementation of the reachability-based scheme by integrating class declaration-level placement. Moreover, this extension includes a discussion on leveraging the abstractions invented in μSE for evaluating tools with security goals other than data leak detection, such as the detection of cryptographic-API misuse vulnerabilities. Such a discussion on the general applicability of μSE's abstractions was not present in our USENIX '18 paper. Finally, we enhance μSE's execution engine as a part of this extension, thereby improving its accuracy in terms of associating execution traces with mutants.
— **Multiplicative improvement in mutants generated:** In the USENIX '18 paper [11], seven Android apps were used as base applications for mutation, to produce 7,584 compilable mutants. In this extension, we added eight new apps, resulting in a total of 15 real-world, open source Android apps used for mutation. We were able to seed 30,117 compilable mutants into these eight new apps, of which 4,385 were executable, and were used for evaluating additional data leak detectors (see next). More importantly, to gauge the impact of

our design and implementation improvements on μSE's ability to generate compilable mutants, we also mutated the seven original apps, which resulted in 24,819 mutants, i.e., 17,235 more mutants (a 2.27× *increase in compilable mutants*) over the USENIX '18 study.

— **Evaluation of the effectiveness of mutation schemes:** In this extension, we perform a data-driven evaluation of our implemented mutation schemes, across two primary directions: (1) their ability to create executable mutants, and (2) their ability to create mutants that may lead to the discovery of flaws. Our analysis, and the resultant insights regarding schemes is novel, as no such evaluation was performed in our USENIX '18 paper [11].

— **Significant additional extrinsic evaluation with an in-depth analysis of Argus and HornDroid:** We studied FlowDroid in depth in our USENIX '18 paper, which formed *the core of the extrinsic evaluation*. In this study, we effectively *tripled our extrinsic evaluation* by performing an in-depth soundness evaluation of two additional state-of-the-art data leak detecting static analysis approaches for Android, namely, Argus (previously known as AmanDroid) [38] and HornDroid [14]. We used the newly generated set of 4,385 executable mutants generated from the eight new base apps to evaluate Argus and HornDroid.

— **New findings:** Our extended analysis led to significant new findings over our USENIX'18 study [11], which can be summarized into the following four points. (1) **New flaws.** We found 12 novel flaws in Argus and HornDroid, in addition to the 13 flaws discovered in FlowDroid in our USENIX '18 study [11], which brings the total number of discovered flaws to 25. (2) **New flaw class.** We discovered a *new flaw class*, aside from the four classes discussed in the USENIX '18 version, due to a distinctive and novel pattern exhibited by certain flaws, i.e., a fundamentally flawed analysis of critical Android lifecycle methods that are present in all applications (e.g., onCreate). (3) **New insight on propagation.** Furthermore, we discovered a *new insight in terms of how flaws propagate across tools*, relative to the USENIX '18 paper. That is, in the USENIX paper we found that flaws generally propagate when a direct inheritance relationship is present among two tools (i.e., directly relying on the code base), but also observed that the flaws did not propagate to tools that did not have a direct relationship, but were simply built for similar design goals. However, in this work, on studying the propagation of flaws with four additional tools (i.e., across a total of seven tools), we discovered that every single flaw was present in at least one other tool, which means that flaws propagate across tools purely because of the shared design goal, even without any direct inheritance relationship. Such propagation based purely on the design goal was speculated in the USENIX '18 paper, but was not evident, as it is in this work. (4) **New insight on re-emergence of flaws.** Finally, we found that flaws that are fixed after reporting, *can re-emerge in future updates* of a tool. Our findings demonstrate that soundness issues can be introduced at any point in tools' lifecycles, which further necessitates continuous evaluation with μSE.

We have released the μSE framework, the security operators and mutation schemes constructed for evaluating data leak detectors, as well as all of the experimental data, to facilitate the reproducibility of our results, as well as to enable security researchers, tool designers, and analysts to uncover undocumented flaws and unsound choices in sound*y* security tools [100].

2 MOTIVATION AND BACKGROUND

This work is motivated by the pressing need to help researchers and practitioners identify instances of unsound assumptions or design decisions in their static analysis tools, thereby *extending the sound core* of their sound*y* techniques. That is, security tools may already have a core set of sound design decisions (i.e., the sound core) and may claim soundness based on those decisions. The sound*i*ness manifesto [66] defines the *sound core* in terms of specific language features, whereas we use the term in a more abstract manner to refer to the design goals of the tool. Systematically identifying unsound decisions may allow researchers to resolve flaws and help extend the sound core of their tools.

Moreover, research papers and tool documentations indeed do not articulate many of the unsound assumptions and design choices that lie outside their sound core, aside from some well-known cases (e.g., choosing not to handle reflection, race conditions), as confirmed by our results (Section 6). In addition, there is a chance that developers of these techniques may be unaware of some implicit assumptions/flaws due to a host of reasons: e.g., because the assumption was inherited from prior research or a certain aspect of Android was not modeled correctly. Therefore, our objective is to discover instances of such hidden assumptions and design flaws that affect the security claims made by tools, document them explicitly, and possibly, help developers mend existing artifacts.

2.1 Motivating Example

Consider the following motivating example of a prominent static analysis tool, FlowDroid [6]:

FlowDroid is a highly popular static analysis framework for detecting private data leaks in Android apps by performing a dataflow analysis. Some of the limitations of FlowDroid are well-known and stated in the paper [6]; e.g., FlowDroid does not support reflection, like most static analyses for Java. However, through a manual but systematic, preliminary analysis of FlowDroid, we discovered a security limitation that is not well-known or accounted for in the paper, and hence affects guarantees provided by the tool's analysis. We discovered that FlowDroid (i.e., v1.5, which was latest at the time of our preliminary analysis in October 2017) does not support "Android fragments" [3], which are app modules that are widely used in most Android apps (i.e., in more than 90% of the top 240 Android apps per category on Google Play, as demonstrated in our original USENIX '18 paper [11]). This flaw renders any security analysis of general Android apps using FlowDroid unsound, due to the high likelihood of fragment use, even when the app developers may be cooperative and non-malicious. Further, FlowDroid v2.0, which was released on 10/10/2017 [98], claims to address fragments, *but failed to detect our exploit.*

On investigating further, we found that FlowDroid v1.5 was extended or used by at least 13 research tools [1, 8, 55, 58, 59, 61, 65, 76, 79, 89, 92, 95, 108], none of which acknowledge or address this limitation in modeling fragments. This leads us to conclude that this significant flaw not only persists in FlowDroid, but may have additionally propagated to the tools that inherit it directly (i.e., by inheriting its code), or indirectly (i.e., by adhering to similar design principles/goals). Our flaw propagation study in Section 9 confirms this conjecture for inheritors of FlowDroid, as well as the two other data leak detectors that we evaluate in depth, i.e., Argus [38] and Horndroid [14].

We reported the flaws to the authors of FlowDroid, and created two patches to fix it. Our patches were confirmed to work on FlowDroid v2.0 built from source, and were accepted into FlowDroid's repository [99] in December 2017. Thus, we were able to discover and fix an undocumented flaw that significantly affected FlowDroid's soundness claims, thereby expanding its sound core. However, we have confirmed that FlowDroid v2.5 [98] still fails to detect leaks in fragments, and are working with developers to resolve this issue.

Through this example, we demonstrate that unsound assumptions in security-focused static analysis tools for Android are not only detrimental to the validity of their own analysis, but may inadvertently propagate to future research. Thus, identifying these unsound assumptions is not only beneficial for making the user of the analysis aware of its true limits, but in addition for the research community in general. As of today, aside from a handful of manually curated testing tool-kits (e.g., DroidBench [6]) with hard-coded (but useful) checks, to the best of our knowledge,

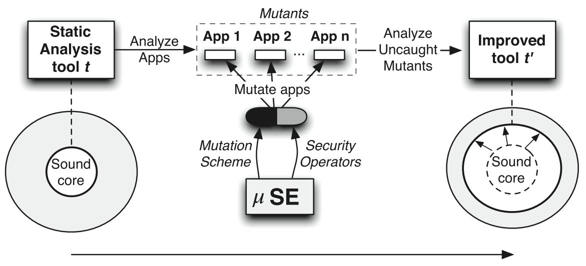

Fig. 1. μSE tests a static analysis tool on a set of mutated Android apps and analyzes uncaught mutants to discover and/or fix flaws.

there has been no prior effort at methodologically discovering problems related to sound*i*ness in Android static analysis tools and frameworks. *This article is motivated by the need to systematically identify and resolve the unsound assumptions in security-related static analysis tools.*

2.2 Background on Mutation Analysis

Mutation analysis has a strong foundation in the field of SE, and is typically used as a test adequacy criterion, measuring the effectiveness of a set of test cases [81]. Faulty programs are created by applying transformation rules, called *mutation operators*, to a given program. The larger the number of faulty programs or *mutants* detected by a test suite, the higher the effectiveness of that particular suite. Since its inception [19, 44], mutation testing has seen striking advancements related to the design and development of advanced operators. Research related to the development of mutation operators has traditionally attempted to adapt operators for a particular target domain, such as the web [86], data-heavy applications [4, 25, 109], or GUI-centric applications [82]. Recently, mutation analysis has been applied to measure the effectiveness of test suites for both functional and non-functional requirements of Android apps [20, 50, 63].

This article builds upon SE concepts of mutation analysis and adapts them to a security context. Our methodology does not simply use the traditional mutation analysis, but rather *re-defines* it to effectively improve security-focused static analysis tools, as described in Sections 4 and 5.

3 μSE

We describe μSE, a semi-automated framework for systematically evaluating Android static analysis tools that adapts the process of mutation analysis commonly used to evaluate software test suites [81]. That is, we aim to help discover concrete instances of flawed security design decisions made by static analysis tools, by exposing them to methodologically mutated applications. We envision two primary benefits from μSE: *short-term* benefits related to straightforwardly fixable flaws that may be patched immediately, and *long-term* benefits related to the continuous documentation of assumptions and flaws, even those that may be hard to resolve. This section provides an overview of μSE (Figure 1) and its design goals.

As shown in Figure 1, we take an Android static analysis tool to be evaluated (e.g., FlowDroid [6] or MalloDroid [34]) as input. μSE executes the tool on *mutants*, i.e., apps to which *security operators* (i.e., security-related mutation operators) are applied, as per a *mutation scheme*, which governs the placement of code transformations described by operators in the app (i.e., thus generating mutants). The security operators represent anomalies that the static analysis tools are expected to detect, and hence, are closely tied to the security goal of the tool. The uncaught mutants indicate flaws in the tool, and analyzing them leads to the broader discovery and awareness of the unsound assumptions of the tools, eventually facilitating security improvements.

Design goals: Measuring the security provided by a system is a difficult problem; however, we may be able to better predict failures if the assumptions made by the system are known in advance. Similarly, although soundness may be a distant ideal for security tools, we assert that it should be feasible to articulate the boundaries of a tool's sound core. Knowing these boundaries would be immensely useful for analysts who use security tools, for developers looking for ways to improve tools, as well as for end users who benefit from the security analyses provided by such tools. To this end, we design μSE to provide an effective foundation for evaluating Android security tools. Our design of μSE is guided by the following goals:

- **G1** ***Contextualized security operators.*** Android security tools have diverse purposes and may claim various security guarantees. Security operators must be instantiated in a way that is sensitive to the context or purpose (e.g., data leak identification) of the tool being evaluated.
- **G2** ***Android-focused mutation schemes.*** Android's security challenges are notably unique, and hence require a diverse array of novel security analyses. Thus, the strategies for defining mutation schemes, i.e., the *placement* of the target, unwanted behavior in the app, must consider Android's abstractions and application model for effectiveness.
- **G3** ***Minimize manual effort during analysis.*** Although μSE is certainly more feasible than manual analysis, we intend to significantly reduce the manual effort spent on evaluating undetected mutants. Thus, our goal is to dynamically filter inconsequential mutants, and to develop a systematic methodology for resolving undetected mutants to flaws.
- **G4** ***Minimize overhead.*** We expect μSE to be used by security researchers as well as tool users and developers. Hence, we must ensure that μSE is efficient so as to promote a wide-scale deployment and community-based use of μSE.

Threat model: The design goals delineated above are ultimately meant to address a specific threat that arises from static analyses that are *thought* to be sound—but are not actually sound—leading to undiscovered security vulnerabilities in Android apps. μSE is designed to help security researchers evaluate tools that detect vulnerabilities (e.g., SSL misuse), *and more importantly*, tools that detect malicious or suspicious behavior (e.g., data leaks). Thus, the security operators and mutation schemes defined in this article are of an adversarial nature. That is, behavior like "data leaks" is intentionally malicious/curious, and generally not attributed to accidental vulnerabilities. Therefore, to evaluate the soundness of existing tools that detect such behavior, μSE has to develop mutants that mimic such adversarial behavior as well, by defining mutation schemes of an adversarial nature. This is the key difference between μSE and prior work on fault/vulnerability injection (e.g., LAVA [27]) that assumes the mutated program to be benign.

4 DESIGN

Figure 2 provides a conceptual description of the process followed by μSE, which consists of three main steps. In Step 1, we *specify* the security operators and mutation schemes that are relevant to the security goals of the tool being evaluated (e.g., data leak detection), as well as certain unique abstractions of Android that separately motivate this analysis. In Step 2, we *mutate* one or more Android apps using the security operators and defined mutation schemes using a Mutation Engine (ME). After this step, each app is said to contain one or more mutants. To maximize effectiveness, mutation schemes in μSE stipulate that mutants should be systematically injected into all potential locations in code where operators can be instantiated. In order to limit the effort required for manual analysis due to potentially large numbers of mutants, we first filter out the non-executing mutants in the Android app(s) using a dynamic Execution Engine (EE) (Section 5). In Step 3, we apply the security tool under investigation to *analyze* the mutated app, leading it to detect some or all of the mutants as anomalies. We perform a methodological manual analysis of the undetected

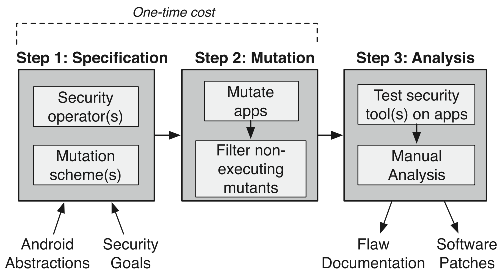

Fig. 2. The components and process of the μSE.

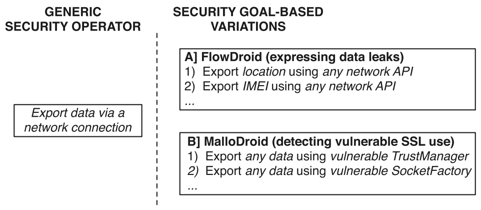
GENERIC SECURITY OPERATOR
SECURITY GOAL-BASED VARIATIONS
Export data via a network connection
A] FlowDroid (expressing data leaks)
1) Export location using any network API
2) Export IMEI using any network API
...
B] MalloDroid (detecting vulnerable SSL use)
1) Export any data using vulnerable TrustManager
2) Export any data using vulnerable SocketFactory
...

Fig. 3. A generic "network export" security operator, and its more fine-grained instantiations in the context of FlowDroid [6] and MalloDroid [34].

mutants, which may lead to documentation of flaws, and software patches. Note that tools sharing a security goal (e.g., FlowDroid [6], Argus [38], HornDroid [14], and BlueSeal [94] all detect data leaks) can be analyzed using the same security operators and mutation schemes, and hence the mutated apps, significantly reducing the overall cost of operating μSE (Goal **G4**).

This section describes the design of μSE, including the additional contributions made in this extension of our USENIX '18 study [11]. Moreover, in Section 4.4, we describe a general approach for leveraging the abstractions introduced in μSE to evaluate security tools built for goals other than data leak detection (e.g., cryptographic API misuse detection).

4.1 Security Operators

A security operator is a description of the unwanted behavior that the security tool being analyzed aims to detect. When designing security operators, we are faced with an important question: *what do we want to express?* Specifically, the operator might be too coarse or fine-grained; finding the correct granularity is the key. For instance, defining operators specific to the implementations of individual tools may not be scalable. On the contrary, defining a generic security operator for all the tools may be too simplistic to be effective. Consider the following example:

Figure 3 describes the limitation of using a generic security operator that describes code which "exports data to the network." Depending on the tool being evaluated, we may need a unique, fine-grained, specification of this operator. For example, for evaluating FlowDroid [6], we may need to express the specific types of private data that can be exported via any of the network APIs, i.e., the data portion of the operator is more important than what network API is used. However, for evaluating a tool that detects vulnerable SSL connections (e.g., CryptoLint [30]), we may want to express the use of vulnerable SSL APIs (i.e., of SSL classes that can be overridden, such as a custom TrustManager that trusts all certificates) without much concern for what data is exported. That is, the requirements are practically orthogonal for these two use cases, rendering a generic operator useless, whereas precisely designing tool-specific operators may not scale.

In μSE, we take a balanced approach to solve this problem: instead of tying a security operator to a specific tool, we define it in terms of the *security goal* of the concerned tool (Goal **G1**). Because the security goal influences the properties exhibited by a security analysis, security operators designed with a particular goal in consideration would apply to all the tools that claim to have that security goal, hence making them feasible and scalable to design. For instance, a security operator that reads information from a private source (e.g., IMEI, location) and exports it to a public sink (e.g., the device log, storage) would be appropriate to use for all the tools that claim to detect private data leaks (e.g., Argus [38], HornDroid [14], BlueSeal [94]) (e.g., see Listing 1 for one such implemented

```
Inject:
  String dataLeak## = java.util.Calendar.getInstance().getTimeZone().getDisplayName();
  android.util.Log.d("leak-##", dataLeak##);
```

Listing 1. Security operator that injects a data leak from the Calendar API access to the device log.

operator). Moreover, security operators generalize to other security goals as well; a simple operator for evaluating tools that detect vulnerable SSL use (e.g., MalloDroid) could add a `TrustManager` with a vulnerable `isServerTrusted` method that returns true.

To derive security operators at the granularity of the security goal, we must examine the claims made by existing tools; i.e., security tools must certainly detect the unwanted behavior that they claim to detect, unless affected by some unsound design choice that hinders detection. We inspect the following sources to precisely identify the security flaws considered by tools:
(1) Research papers: The tool's research paper is often the primary source of information about what unwanted behavior a tool seeks to detect. We inspect the properties and variations of the unwanted behavior as described in the paper, as well as the examples provided, to formulate security operator specifications for injecting the unwanted behavior in an app. However, we do not create operators using the limitations and assumptions already documented in the paper or well known in general (e.g., reflection or dynamic code loading), as μSE seeks to find unknown assumptions.
(2) Open source tool documentation: Due to space limitations or tool evolution over time, research papers may not have the most complete or up-to-date information considering what security flaws a tool can actually address. We used tool documentation available in online appendices and open source repositories to fill this knowledge gap.
(3) Testing toolkits: Manually curated testing toolkits (e.g., DroidBench [6]) may be available and may provide examples of baseline operators.

4.2 Mutation Schemes

To enable the security evaluation of static analysis tools, μSE must seed mutations within Android apps. For this purpose, we define *mutation schemes*, i.e., the methods for choosing *where* to apply security operators to inject mutations within Android apps.

Our design of mutation schemes leverages a number of factors: (1) Android's unique abstractions, (2) the intent to over-approximate reachability for coverage, and (3) the security goal of the tool being analyzed. We design mutation scheme *strategies* based on these factors (Sections 4.2.1–Section 4.2.3), and describe them in the context of our running example first described in Section 2 (but elaborated as follows).

Recall that FlowDroid [6], the target of our analysis in Section 2, detects data leaks in Android apps. Hence, FlowDroid loosely defines a data leak as a flow from a sensitive *source* of information to some *sink* that exports it. FlowDroid lists all of the sources and sinks within a configurable "SourcesAndSinks.txt" file in its tool documentation. A simple data leak mutation may be implemented by using the data leak operator described previously, with a source (e.g., java.util.Calendar.getTimeZone()) and sink (e.g., android.util.Log.d()) from this file. The strategies described in the rest of this section guide our implementation of *four* specific mutation schemes, using which we seed this data leak across target Android applications (see Section 5 for implementation). In particular, we significantly enhance the goal-based mutation scheme strategy by introducing scope as a factor, and implement a scope-based mutation scheme (Section 5), which further increases the expressiveness of the mutation introduced by μSE.

4.2.1 *Mutation Scheme Strategy 1: Leveraging Android Abstractions.*

The Android platform and app model support numerous abstractions that pose challenges to static analysis. One commonly stated example is the absence of a Main method as an entry-point into the app, which compels static analysis tools to scan for the various entry points, and treat them all similarly to a traditional Main method [6, 47]. Based on our domain knowledge of Android and its security, we choose the following features as a starting point in a mutation scheme strategy that models unique aspects of Android, and more importantly, tests the ability of analysis tools to detect unwanted behavior placed within these features (Goal **G2**):

(1) Activity and fragment lifecycle: Android apps are organized into a number of *activity* components, which form the user interface (UI) of the app. The activity lifecycle is controlled via a set of callbacks, which are executed whenever an app is launched, paused, closed, started, or stopped [24]. In addition, Fragments are UI elements that possess similar callbacks, although they are often used in a manner secondary to activities. We design our mutation scheme to place mutants within methods of fragments and activities where applicable, so as to validate a tool's ability to model the activity and fragment lifecycles.

(2) Callbacks: Because much of Android relies on callbacks triggered by events, these callbacks pose a significant challenge to traditional static analyses, as their code can be executed asynchronously in several different potential orders. We place mutants within these asynchronous callbacks to validate the tools' ability to soundly model the asynchronous nature of Android. For instance, consider the example in Listing 2, where the onClick() callback can execute at any point of time.

```java
final Button button = findViewById(R.id.button_id);
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Code here executes on main thread after user presses button
        }
    });
```

Listing 2. Dynamically created onClick callback.

(3) Intent messages: Android apps communicate with one another and listen for system-level events using Intents, Intent Filters, and Broadcast Receivers [22, 23]. Specifically, Intent Filters and Broadcast Receivers form another major set of callbacks into the app. Moreover, Broadcast Receivers can be dynamically registered. Our mutation scheme not only places mutants in the statically registered callbacks such as those triggered by Intent Filters in the app's Android Manifest, but also callbacks dynamically registered within the program, and even within other callbacks, i.e., recursively. For instance, we generate a dynamically registered broadcast receiver inside another dynamically registered broadcast receiver, and instantiate the security operator within the inner broadcast receiver (see Listing 5 in the Appendix for the code).

(4) XML resource files: Although Android apps are primarily written in Java, they additionally include resource files that establish callbacks. Such resource files also allow the developer to register for callbacks from an action on a UI object (e.g., the onClick event, for callbacks on a button being touched). As described previously, static analysis tools often list these callbacks on par with the Main function, i.e., as one of the many entry points into the app. We incorporate these resource files into our mutation scheme, i.e., mutate them to call our specific callback methods.

4.2.2 *Mutation Scheme Strategy 2: Evaluating Reachability.* The objective behind this simple, but important, mutation scheme is to exercise the reachability analysis of the tool being evaluated. We inject mutants (e.g., data leaks from our example) at the start of every method in the app. Furthermore, as part of this extended study, we add leaks at the class declaration level as well. While the previous schemes add methods to the app (e.g., new callbacks), this scheme simply verifies if the app successfully models the bare minimum.

4.2.3 Mutation Scheme Strategy 3: Leveraging the Security Goal. Like security operators, mutation schemes may also be designed in a way that accounts for the security goal of the tool being evaluated (Goal **G1**). Such schemes may be applied to any tool with a similar objective. In keeping with our motivating example (Section 2) and our evaluation (Section 6), we develop an example mutation scheme strategy that can be specifically applied to evaluate data leak detectors. This strategy can be instantiated in terms of the following three methods of seeding mutants:

(1) Taint-based operator placement: This placement methodology tests the tools' ability to recognize an asynchronous ordering of callbacks, by placing *the source in one callback and the sink in another.* The execution of the source and sink may be triggered due to the user, and the app developer (i.e., especially a malicious adversary) may craft the mutation scheme specifically so that the sources and sinks lie on callbacks that generally execute in sequence. However, this sequence may not be observable through just static analysis. A simple example is collecting the source data in the onStart() callback, and leaking it in the onResume() callback. As per the activity lifecycle, the onResume() callback *always* executes right after the onStart() callback.

(2) Complex-path operator placement: Our preliminary analysis demonstrated that static analysis tools may sometimes stop after an arbitrary number of hops when analyzing a call graph, for performance reasons. This finding motivated the complex-path operator placement. In this scheme, we make the path between source and sink as complex as possible (i.e., which is ordinarily one line of code, as seen in Listing 1). That is, the design of this scheme allows the injection of code along the path from source to sink based on a set of pre-defined rules. For our evaluation, we instantiate this scheme with a rule that recreates the String variable saved by the source, by passing each character of the string into a StringBuilder, then sending the resulting string to the sink as shown in Listing 3. μSE allows the analyst to dynamically implement such rules, as long as the input and output are both strings, and the rule complicates the path between them by sending the input through an arbitrary set of transformations.

```java
String dataLeak0 = java.util.Calendar.getInstance().getTimeZone().getDisplayName();
String[] leakArRay0 = new String[] {"n/a", dataLeak0};
String dataLeakPath0 = leakArRay0[leakArRay0.length - 1];
android.util.Log.d("leak-0", dataLeakPath0);
```

Listing 3. Complex path operator placement.

```java
public class ParentClass {
    String dataLeak = "";
    int methodA(){
        android.util.Log.d("leak-0-1", dataLeak);
        return 1; }
    class ChildClass{
        int childMethodA(){
            dataLeak = java.util.Calendar.getInstance().getTimeZone().getDisplayName();
            android.util.Log.d("leak-0-0", dataLeak);
            return 1; }}}
```

Listing 4. Scope-based operator placement at different levels of inheritance.

(3) Scope-based operator placement: This scope-based placement methodology was not defined in our USENIX '18 paper [11], and is a new addition to μSE as part of this extension. As the name suggests, μSE can inject code by analyzing scopes based on *visibility.* For example, as shown in Listing 4, childMethodA is visible from both ChildClass and ParentClass. Therefore, we declare a variable dataLeak at the ParentClass, while assigning the leak source at the childMethodA. Consequently, we insert corresponding sinks in childMethodA and methodA. Note that this scheme is not restricted by callbacks and can be useful for different variations of method declarations. As application developers may arbitrarily organize class scopes, seeding mutants into applications

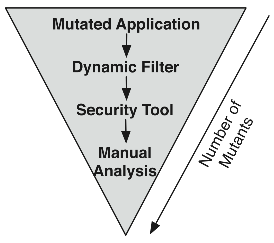

Fig. 4. The number of mutants (e.g., data leaks) to analyze drastically reduces at every stage in the process.

using this scheme generally results in interesting and often complicated mutant placement, which further assists in stress-testing security techniques that assume an adversarial threat model (e.g., data leak detectors).

In a traditional mutation analysis setting, the mutation placement strategy would seek to minimize the number of non-compilable mutants. However, as our goal is to evaluate the soundness of Android security tools, we design our mutation scheme to over-approximate. Once the mutated apps are created, for a feasible analysis, we pass them through a dynamic filter that removes the mutants that cannot be executed, ensuring that the mutants that each security tool is evaluated against are all executable i.e., the data leaks can indeed happen in runtime.

Note that although mutation schemes using the first two strategies (Section 4.2.1 and Section 4.2.2) may be generally applied to any type of static analysis tool (e.g., SSL vulnerability and malware detectors), the third strategy, as the description suggests, will need to be adapted per security goal (e.g., data leak detection). We elaborate on this point by presenting a general approach for leveraging our mutation abstractions for other security goals, in Section 4.4.

4.3 Analysis Feasibility and Methodology

μSE reduces manual effort by filtering out mutants whose security flaws are not verified by dynamic analysis (Goal **G3**). As described in Figure 2, for any given mutated app, we use a dynamic filter (i.e., the Execution Engine (EE), described in Section 5) to purge non-executable leaks. If a mutant (e.g., a data leak) exists in the mutated app, but is not confirmed as executable by the filter, (i.e., data does not leak from source to sink), we discard it. For example, data leaks injected in dead code are filtered out. Thus, when the Android security tools are applied to the mutated apps, only mutants that were executed by EE are considered.

Furthermore, after the security tools were applied to mutant apps, only *undetected* mutants are considered during analyst analysis. The reduction in the number of mutants subject to analysis at each step of the μSE process is illustrated in Figure 4.

The following methodology is used by an analyst for each undetected mutant after testing a given security tool to isolate and confirm flaws:

(1) Identifying the source and sink: During mutant generation, μSE's ME injects a unique mutant identifier, as well as the source and sink using util.Log.d statements. Thus, for each undetected mutant, an analyst simply looks up the unique IDs in the source to derive the source and sink.

(2) Performing leak call-chain analysis: Since the data leaks under analysis went undetected by a given static analysis tool, this implies that there exists one (or multiple) method call sequences (i.e., call-chains) invoking the source and sink that could not be modeled by the tool. Thus, a security analyst inspects the code of a mutated app and identifies the observable call sequences from various entry points. This is aided by dynamic information from the EE so that an analyst can examine the order of execution of detected data leaks to infer the propagation of leaks through different call chains.

(3) Synthesizing minimal examples: For each of the identified call sequences invoking a given undetected data leak's source and sink, an analyst then attempts to synthesize a minimal example by re-creating the call sequence using only the required Android APIs or method calls from the mutated app. This info is then inserted into a pre-defined skeleton app project so that it can be again analyzed by the security tools to confirm a flaw.
(4) Validating the minimal example: Once the minimal example has been synthesized by the analyst, it must be validated against the security tool that failed to detect it earlier. If the tool fails to detect the minimal example, then the process ends with the confirmation of a flaw in the tool. If the tool is able to detect the examples, the analyst can either iteratively refine the examples, or discard the mutant, and move on to the next example.

4.4 Leveraging Security Operators and Mutation Schemes for Other Security Goals

Our design describes the μSE framework in terms of the security goal of data leak detection, primarily due to the popularity of the goal, as well as the proliferation of data leak detectors in academic research. However, μSE may be applied as a general framework, to evaluate detection techniques that target other security goals. We briefly describe the generality of μSE, and specifically, its abstractions of security operators and mutation schemes for evaluating tools that detect *cryptographic-API misuse*, which is the *second most* prolific cause of vulnerabilities, after data leaks [103]. In this context, μSE will evaluate tools such as CryptoGuard [88] and CrySL [56], which try to detect misuse of crypto APIs.

μSE's abstractions may be naturally leveraged to evaluate crypto-API misuse detectors. To elaborate, the *security operators* for evaluating such tools would represent well-known cryptographic vulnerabilities, such as passing a weak algorithm name as a parameter to an encryption-related cryptography API. For example, one operator could be represented by passing insecure parameters in an API such as `Cipher.getInstance()`, such as *only passing* `AES` as a parameter, which would initialize an insecure default cipher that uses the `ECB` mode.

Similarly, μSE's mutation schemes may also be directly leveraged to place such mutations in a hard-to-detect manner. First, the scheme that evaluates reachability (Section 4.2.2) can be directly applied, by placing such vulnerable API invocations (i.e., mutations) at as many reachable locations within the target app as possible. Similarly, the Android-specific scheme strategies (Section 4.2.1) can be leveraged to place the vulnerable invocation in commonly used Android abstractions such as fragments and broadcast receivers. Finally, the security goal-based scheme strategies (Section 4.2.3) can be leveraged to evaluate crypto-API misuse detectors as well. For example, we can consider the parameter passed to the `Cipher.getInstance()` API, i.e., the `AES` string, as a source value, and the API call as the sink, and use the taint-based mutation scheme to distribute these source and sink values across different Android lifecycle methods, such that they would execute in the right sequence, but be hard to detect. Similarly, our newly defined scope-based mutation scheme can be used to inject the source parameter and the API in different program scopes, in a way that would execute flawlessly at runtime, but would be hard to detect. Other than syntactical changes required to implement the security operator (i.e., defining new *source-sink pairs*, but with Cipher APIs) and adjusting it for mutation using the schemes (i.e., scoping them within required *try-catch* block), no other changes will be required to the framework, for making it applicable for evaluating crypto-API misuse detectors. We elaborate on the estimated effort required to make these changes in Section 12.

5 IMPLEMENTATION

This section provides the implementation details of μSE 's components: (1) the ME for mutating apps, and (2) the EE for exercising and filtering out non-executable mutants. We have made

μSE available to the research community [100], along with all the data and code generated or used.

(1) ME: The ME allows μSE to automatically mutate apps according to a fixed set of security operators and mutation schemes. The ME is implemented in Java and extends the MDroid+ mutation framework for Android [63]. More specifically, μSE's ME implements the seeding of mutants according to our defined *mutation schemes*. This required extensions to MDroid+'s implemented static analyses to identify a more diverse array of source code locations (e.g., analyzing the visibility scope for the *scope-based operator placement* scheme). MDroid+'s previous implementation only supported identifying strings and specific API patterns. These additions help to make μSE's ME generic, as it could be applied to support additional security operators or mutation schemes in the future. To achieve these extensions, we designed ME to do the following.

Firstly, the ME derives a mutant injection profile (MIP) of all possible injection points for a given mutation scheme, security operator, and target app source code. The MIP is derived through one of two types of analyses: (i) text-based parsing and matching of `xml` files in the case of app resources; or (ii) using Abstract Syntax Tree (AST) –based analysis for identifying potential injection points in code. μSE takes a systematic approach toward applying mutants to a target app, and for each mutant location stipulated by the MIP for a given app, a mutant is seeded. The injection process additionally uses either text or AST-based code transformation rules to modify the code or resource files. In the context of our evaluation, μSE further marks injected mutants in the source code with log-based indicators that include a unique identifier for each mutant, as well as the source and sink for the injected leak. This information can be customized for future security operators and exported as a ledger that tracks mutant data. μSE can be extended to additional security operators and mutation schemes by adding methods to derive the MIP and perform target code transformations.

Given the time cost in running the studied security-focused static analysis tools on a set of `apks`, μSE breaks from the process used by traditional mutation analysis frameworks that seed each mutant into a separate program version, and seeds all mutants into a single version of a target app. Finally, the target app is automatically compiled using its build system (e.g., Gradle [49], ant [39]) so that it can be dynamically analyzed by the EE.

(2) Mutation schemes: We implemented four mutation schemes using the strategies outlined in Section 4.2. First, we implemented the Reachability scheme, using the reachability-based strategy (Section 4.2.2). Then, we implemented three other schemes, Scope, Taint, and Complex Reachability, using the security goal-based strategy (Section 4.2.3). Finally, we integrated the Android-specific mutation scheme strategy (Section 4.2.1) in these four schemes by prioritizing placement into several abstractions unique to Android (e.g., fragments, receivers, asynchronous task classes). Among these, the Scope-based scheme was not implemented in our prior USENIX '18 paper [11].

(3) EE: To facilitate a feasible manual analysis of the mutants that are undetected by a security analysis tool, μSE uses the EE to dynamically analyze target apps, verifying whether or not injected mutants can be executed in practice. To further elaborate with respect to our implementation, an *executable mutant* represents a data leak from source to sink as observed through our execution engine. This EE builds upon prior work in automated input generation for Android apps by adapting the systematic exploration strategies from the CrashScope tool [70, 71] to explore a target app's GUI. We made several improvements to CrashScope in order to tailor the automated app execution to the goal of discovering as many *executable* mutants as possible. For instance, CrashScope's execution strategies were originally intended to uncover crashes in Android apps and thus included several mechanisms that executed common crash inducing actions on apps including rotating the screen, and injecting text with special characters. In order to be used in μSE, we discarded these crash inducing operations, and developed strategies that focused upon uncovering as many execution states of the app as possible. Additionally, as part of extending our

USENIX '18 paper [11], we made a practical improvement to the manner in which CrashScope analyzes applications as part of our current study. That is, during initial testing, we discovered that even after uninstalling an application from an Android virtual device used in CrashScope, certain background services may persist, which can contaminate the runtime mutant execution logs across applications. We addressed this problem by modifying CrashScope to instantiate a new, clean Android virtual device for each application analyzed, thereby fully isolating application execution logs. We discuss the limitations of the EE in Section 12 (see the Appendix for additional details).

6 EVALUATION OVERVIEW

The primary objective of this evaluation is to measure the effectiveness of μSE at uncovering flaws in security-focused static analysis tools for Android apps, and to demonstrate the extent of such flaws. As a case study, we focus our evaluation on the security goal of data leak detection, which has received significant attention from the security research community in the last decade [31, 32, 42, 54, 58–60, 89]. Our evaluation is guided by six key research questions that provide an objective measure of the effectiveness and practicality of μSE, while also shedding light on the general nature of unsound decisions/flaws in static analysis security tools:

- **RQ_1** ***Effectiveness of μSE.*** *Can μSE find security problems in static analysis tools for Android, and help resolve them to flaws/unsound choices?*
- **RQ_2** ***Relative effectiveness of mutation schemes.***[1] *How effective are the individual mutation schemes for (a) generating executable mutants and (b) discovering security flaws?*
- **RQ_3** ***Propagation of flaws.*** *Are flaws inherited when a tool is reused, or built based on similar principles?*
- **RQ_4** ***Addressing flaws.*** *Are all flaws unearthed by μSE hard to address, or can some be patched?*
- **RQ_5** ***Scalability of μSE.*** *Does the semi-automated methodology of μSE for analyzing mutants allow for a feasible analysis (i.e., in terms of the manual effort)?*
- **RQ_6** ***Performance of μSE.*** *What is the runtime performance of μSE?*

To address **RQ_1–RQ_6**, we performed several experiments for a period of over 2 years (i.e., October 2017–January 2020). We started by creating a data-leak security operator (i.e., as described in Section 4.1), and used μSE's expressive mutation schemes (Section 4.2) to seed the corresponding mutated code in a set of 15 open source Android applications obtained from F-droid [62] (see Table 9 in the Appendix for the list), creating 54,936 mutants representing injected data leaks. Section 7 describes this process, along with the refinement made possibly by μSE's EE, and addresses questions pertaining to μSE's intrinsic evaluation (**RQ_2**, **RQ_5**, and **RQ_6**). We selected three prominent data leak detectors, namely, FlowDroid [6], Argus [38] (previously known as AmanDroid), and HornDroid [14], as the target of our in-depth evaluation using the end-to-end approach as described in Section 4. The in-depth study of FlowDroid was reported in our USENIX '18 paper [11] and is not re-performed in this study. Due to the longitudinal nature of this study, we strived to use the latest release of the data leak detectors whenever available. Section 8 describes this evaluation, the 13 flaws that we previously discovered (**RQ_1**) in FlowDroid, and the 12 new flaws found in HornDroid and Argus. Further, we also briefly describe the one flaw that we could fix (**RQ_4**), and the relative effectiveness of μSE's mutation schemes in unearthing flaws (**RQ_2**). We developed minimal examples of the discovered flaws, and performed a *flaw propagation study* (Section 9) to discover the extent to which flaws discovered in one tool manifest in others developed for the same security goal (**RQ_3**). Particularly, we used four additional data leak detection tools, i.e., in addition to the three tools analyzed in depth in Section 8, bringing the total to seven tools,

[1] **RQ_2** was not investigated in our USENIX '18 paper [11].

for which propagation was studied. Note that compared to our USENIX '18 paper [11], which only studied the propagation of flaws discovered in FlowDroid, we describe the propagation of flaws discovered in three tools, namely, FlowDroid, Argus, and HornDroid. Our results demonstrate that flaws generally propagate to other tools, and more so if the tools rely on common design principles.

7 EXECUTING μSE TO CREATE MUTANTS REPRESENTING DATA LEAKS

We applied μSE to 15 target Android apps obtained from F-Droid [62], and created 54,936 mutants (i.e., data leaks).[2] These leaks were generated by the μSE's ME using the data-leak security operator, and the four mutation schemes described in Section 5, namely, (1) *reachability*, (2) *scope*, (3) *taint*, and (4) *complex reachability*.

Filtering non-executable leaks: We then used our EE to filter out non-executable leaks, as described in Section 5, and confirmed 8,250 out of 54,936 leaks as executable. The remaining 46,686 non-executable leaks were then removed. Note that this number is independent of the tools involved, i.e., the filtering only happens once, and the mutated APKs can then be passed to any number of tools for analysis. By filtering out a large number of potentially non-executable leaks (i.e., *46,686 /54,936 or about 85 %*), our dynamic filtering is tremendously effective at reducing the number of mutants used to evaluate security tools, and in turn, the manual effort required to analyze the uncaught mutants, which demonstrates the feasibility of μSE (**RQ$_5$**).

Runtime performance: μSE took 19 hours in total to create the mutants and filter out those that were non-executable, which is a one-time cost for each security goal, i.e., which does not have to be repeated for any of the tools we analyze in particular (**RQ$_6$**). To elaborate, it took us 74 minutes on average to mutate each of the 15 apps, with minimum and maximum times of 16 and 170 minutes, and a standard deviation of about 53 minutes. The increase in runtime compared to our original work (92 minutes in worst case) [11] is due to two reasons: (1) heterogeneity of the applications selected for mutation, and (2) a bulk of the time spent can be attributed to our *improvements* to CrashScope (see Section 5) that require the Android virtual device it uses to be recreated more frequently, leading to an increase in runtime. However, the improvements also increase the reliability of CrashScope's mutant detection by preventing cross-contamination of mutation logs across apps, and hence, are desirable.

Correlation between executable mutants and μSE's schemes: To improve our understanding of what factors contribute to more executable leaks, we further examined the number of executable leaks generated as a factor of the mutation schemes used to generate them (i.e., since there was a single security operator used) in this extended study.

Table 1 shows the number of executable and non-executable leaks seeded using each of the mutation schemes described in Section 4.2 (**RQ$_2$**). As seen in the table, the Reachability and Complex Reachability result in the insertion of a somewhat similar number of leaks (i.e., 7,867 and 6,868, respectively), and the fraction of leaks deemed executable by our EE is similar as well, i.e., 1,708 (about 22%) and 1,378 (i.e., 20%) for the Reachability and Complex Reachability scheme, respectively. Our intuition is that this equivalence results from the inherent similarity in the nature of the two schemes. Moreover, these two schemes produced the highest fraction of executable mutants (i.e., over 20%). Further, the Scope -based scheme inserted a total of 7,431 leaks, out of which, 1,251 (i.e., or 16%) were confirmed as executable by the EE. Note that the number of leaks inserted using the Scope-based scheme for individual apps is highly variable, primarily due to wide variations in the developers' usage of scope. Finally, the Taint -based scheme was the most numerous, both in terms of the number of leaks inserted (i.e., 32,770) as well as the number of executable leaks

[2] We use "mutants" and "leaks" interchangeably to refer to data-leak-related mutants used in the evaluation, i.e., Sections 6–9.

Table 1. The Number of Leaks Inserted by μSE, and the Final Number Marked as Executable by μSE's EE

App ID	Inserted Leaks per Scheme				Executable Leaks per Scheme			
	Reachability	Scope	Taint	Complex Reachability	Reachability	Scope	Taint	Complex Reachability
App 01	24	48	66	22	13 (≈54%)	12 (≈12%)	8 (≈12%)	11 (≈50%)
App 02	106	-	231	83	63 (≈59%)	-	80 (≈34%)	44 (≈14%)
App 03	248	-	668	191	51 (≈20%)	-	162 (≈24%)	36 (≈26%)
App 04	45	192	346	41	20 (≈44%)	156 (≈81%)	51 (≈14%)	23 (≈2%)
App 05	3,181	-	12,104	2,777	598 (≈18%)	-	1362 (≈11%)	508 (≈42%)
App 06	434	-	2,699	384	57 (≈13%)	-	110 (≈4%)	51 (≈5%)
App 07	204	-	551	174	111 (≈54%)	-	243 (≈44%)	95 (≈34%)
App 08	975	1,193	5,266	839	215 (≈22%)	56 (≈4%)	567 (≈10%)	168 (≈32%)
App 09	23	8	20	21	15 (≈65%)	6 (≈75%)	13 (≈65%)	12 (≈7%)
App 10	476	-	5,283	449	59 (≈12%)	-	681 (≈12%)	54 (≈11%)
App 11	316	1,111	827	277	50 (≈15%)	287 (≈25%)	83 (≈10%)	41 (≈5%)
App 12	250	354	428	213	77 (≈30%)	111 (≈31%)	59 (≈13%)	38 (≈7%)
App 13	156	203	828	147	89 (≈57%)	112 (≈55%)	295 (≈35%)	84 (≈71%)
App 14	125	844	663	107	79 (≈63%)	55 (≈6%)	27 (≈4%)	59 (≈9%)
App 15	1,304	3,478	2,790	1,143	211 (≈16%)	456 (≈13%)	172 (≈6%)	154 (≈41%)
Total	7,867	7,431	32,770	6,868	1,708 (≈22%)	1,251 (≈16%)	3,913 (≈11%)	1,378 (≈20%)

Note that the "-" indicates that the scheme is not applicable to a particular app, due to the app's particular characteristics (e.g., the absence of fragments).

(i.e., 3,913) it caused. However, the Taint -based scheme did not lead to a high rate of executable leaks, which we suspect to be due to the sheer number of leaks seeded with it. The high number of leaks (and more importantly, non-executable leaks) seeded by the Taint-based scheme is primarily due to its design, i.e., it places one source per method in a class, thus creating a total of n sources distributed in n methods. Then, it places n sinks per source in each method as scope allows. As a result, around n^2 number of sinks are created in total for n sources in each class.

The superior performance of the Reachability and Complex Reachability schemes over the other two is expected, i.e., as both Scope-based and Taint-based schemes insert leaks with sources and sinks distributed across methods, thereby creating unreachable sinks at a higher rate. In contrast, the Reachability and Complex Reachability schemes place leaks with the sources and sinks placed together, resulting in a lower number of unreachable sinks, and hence, a higher rate of creating executable mutants.

8 IN-DEPTH EVALUATION OF DATA LEAK DETECTION TOOLS WITH μSE

To demonstrate the utility of μSE, we evaluated three prominent data leak detectors with it: FlowDroid [6], HornDroid [14], and Argus [38]. Among these, FlowDroid was analyzed in depth as part of the previous study [11], whereas we analyze Argus and HornDroid as part of our current, extended study. We selected FlowDroid and Argus for the in-depth analysis as they are regularly maintained, with multiple publicly available versions, and form a representative sample of the current state-of-the-art. Additionally, we selected HornDroid as it is the first Android static analysis tool with a formal proof of soundness, making it an interesting case for a sound*i*ness evaluation. Compatibility with our mutant apps and general analysis feasibility were also major factors that influenced tool selection for the in-depth analysis. Specifically, we avoided tools that had not been updated recently (i.e., were built for outdated Android versions), and hence were incompatible with many of our mutant apps.

Methodology: Our methodology for evaluating a security tool with μSE is as follows: First, we analyze executable mutants with the tool being evaluated. Then, we systematically examine the surviving (i.e., undetected) mutants using the methodology described in Section 4.3, and resolve the undetected mutants to design/implementation flaws. Using this approach, our analysis of FlowDroid, HornDroid, and Argus led to the discovery of 25 unique flaws, among which 13 were reported in our USENIX '18 paper [11]. *We confirmed that these flaws were undocumented, i.e., mentioned neither in the tools' corresponding papers nor in the documentation.*

8.1 Evaluating FlowDroid

FlowDroid was introduced by Artz et al. [6] in 2014 as a data leak detection tool for Android. It models the Android lifecycle to handle callbacks invoked by the Android Framework to perform information flow analysis and data leak detection. Furthermore, it applies context, flow, field, and object sensitivity to reduce the number of false-positive sensitive data leaks the tool detects. FlowDroid has been cited over 1,400 times, and the tool has been continuously maintained since 2014, which motivated its analysis.

For our in-depth analysis, we evaluated FlowDroid v2.0, which was the version available during our USENIX '18 study [11], using the 7,584 mutants originally created from our first seven base apps (i.e., apps 01–07), leveraging the Reachability, Complex Reachability, and Taint schemes. In this extension, we used the substantially improved version of μSE to create 30,117 mutants (4,385 executable) from the remaining eight apps to evaluate HornDroid and Argus, using all four mutation schemes (i.e., including the additional Scope scheme developed in this extension). This separation is mainly due to the order in which the tools were evaluated, and the longitudinal nature of our study. Moreover, this two-phase evaluation with disjoint sets of mutants led to the discovery of flaws in all three tools, which indicates that μSE may be effective at revealing flaws in security tools, irrespective of the apps used as input.

Results: Out of the 2,026 mutants that we analyzed using FlowDroid, 987 were undetected. On analyzing the undetected mutants, we discovered 13 unique flaws in FlowDroid, demonstrating that μSE can be effectively used to find problems that can be resolved to flaws (**RQ$_1$**). Using the approach from Section 4.3, we needed less than 1 hour to isolate a flaw from the set of undetected mutants, in the worst case. In the best case, flaws were found in a matter of minutes, demonstrating that the amount of manual effort required to quickly find flaws using μSE is minimal (**RQ$_5$**). We provide descriptions of the flaws discovered in FlowDroid in Table 2.

We have reported these flaws and are working with the FlowDroid developers to resolve them. In fact, we developed two patches [99] to correctly implement Fragment support (i.e., F5 in Table 2), which were accepted by developers. To gain insight about the practical challenges faced by static analysis tools, and their design flaws, we further categorize the flaws into the following classes:

FC1: Missing callbacks: The security tool (e.g., FlowDroid) did not recognize some callback method(s) and will not find leaks placed within them. Tools that use lists of APIs or callbacks are susceptible to this problem, as prior work has demonstrated that the generated list of callbacks (1) may not be complete, and (2) or may not be updated as the Android platform evolves. We observed both such cases in our analysis of FlowDroid. That is, DialogFragments was added in API 11 *before FlowDroid was released*, and NavigationView was added after. These limitations are well known in the community of researchers at the intersection of program analysis and Android security, and have been documented by prior work [15]. However, μSE helps evaluate the robustness of existing security tools against these flaws and helps in uncovering these undocumented flaws for the wider security audience. Additionally, *some of these flaws may not be resolved even after*

Table 2. Descriptions of Flaws Uncovered* in FlowDroid v2.0

ID: Flaw Name	Description
	FC1: Missing Callbacks
F1: DialogFragmentShow	FlowDroid misses the DialogFragment.onCreateDialog() callback registered by DialogFragment.show().
F2: PhoneStateListener	FlowDroid does not recognize the onDataConnectionStateChanged() callback for classes extending the PhoneStateListener abstract class from the telephony package.
F3: NavigationView	FlowDroid does not recognize the onNavigationItemSelected() callback of classes implementing the interface NavigationView.OnNavigationItemSelectedListener.
F4: SQLiteOpenHelper	FlowDroid misses the onCreate() callback of classes extending android.database.sqlite.SQLiteOpenHelper.
F5: Fragments	FlowDroid 2.0 does not model Android Fragments correctly. We added a patch, which was promptly accepted. However, FlowDroid 2.5 and 2.5.1 remain affected. We investigate this further in the next section.
	FC2: Missing Implicit Calls
F6: RunOnUIThread	FlowDroid misses the path to Runnable.run() for Runnables passed into Activity.runOnUIThread().
F7: ExecutorService	FlowDroid misses the path to Runnable.run() for Runnables passed into ExecutorService.submit().
	FC3: Incorrect Modeling of Anonymous Classes
F8: ButtonOnClickToDialogOnClick	FlowDroid does not recognize the onClick() callback of DialogInterface.OnClickListener when instantiated within a Button's onClick="method_name" callback defined in XML. FlowDroid will recognize this callback if the class is instantiated elsewhere, such as within an Activity's onCreate() method.
F9: BroadcastReceiver	FlowDroid misses the onReceive() callback of a BroadcastReceiver implemented programmatically and registered within another programmatically defined and registered BroadcastReceiver's onReceive() callback.
	FC4: Incorrect Modeling of Asynchronous Methods
F10: LocationListenerTaint	FlowDroid misses the flow from a source in the onStatusChanged() callback to a sink in the onLocationChanged() callback of the LocationListener interface, despite recognizing leaks wholly contained in either.
F11: NSDManager	FlowDroid misses the flow from sources in any callback of a NsdManager.DiscoveryListener to a sink in any callback of a NsdManager.ResolveListener, when the latter is created within one of the former's callbacks.
F12: ListViewCallbackSequential	FlowDroid misses the flow from a source to a sink within different methods of a class obtained via AdapterView.getItemAtPosition() within the onItemClick() callback of an AdapterView.OnItemClickListener.
F13: ThreadTaint	FlowDroid misses the flow to a sink within a Runnable.run() method started by a Thread, only when that Thread is saved to a variable before Thread.start() is called.

* Reported in our USENIX '18 paper [11].

adding the callback to the list; e.g., PhoneStateListener and SQLiteOpenHelper, both added in API 1, are not interfaces, but abstract classes. Therefore, adding them to FlowDroid's list of callbacks (i.e., AndroidCallbacks.txt) does not resolve the issue.

FC2: Missing implicit call: The security tool did not identify leaks within some method that is implicitly called by another method. For instance, FlowDroid does not recognize the path to

Runnable.run() when a Runnable is passed into the ExecutorService.submit(Runnable). The response from the developers indicated that this class of flaws was due to an unresolved design challenge in Soot's [101] SPARK algorithm, upon which FlowDroid depends. This limitation is also generally well known within the program analysis community [15]. However, the documentation of this gap in analysis, thanks to μSE, would certainly benefit researchers in the wider security community.

FC3: Incorrect modeling of anonymous classes: The security tool did not detect data leaks expressed within an anonymous class. For example, FlowDroid did not recognize leaks in the onReceive() callback of a dynamically registered BroadcastReceiver, which is implemented within another dynamically registered BroadcastReceiver's onReceive() callback. It is important to note that finding such complex flaws is only possible due to μSE's semi-automated mechanism and may be rather prohibitive for an entirely manual analysis.

FC4: Incorrect modeling of asynchronous methods: The security tool did not recognize a data leak whose source and sink are called within different methods that are asynchronously executed. For instance, FlowDroid did not recognize the flow between data leaks in two callbacks (i.e., onLocationChanged and onStatusChanged) of the LocationListener class, which the adversary may cause to execute sequentially (i.e., as our EE confirmed).

Apart from **FC1**, which may be patched with limited effort, the other three categories of flaws may require a significant amount of research effort to resolve. However, documenting them is critical to increase awareness of real challenges faced by Android static analysis tools.

8.2 Evaluating Argus

As part of this extended study, we evaluated Argus v3.1.2 (i.e., the latest version at the time of our analysis) with 4,385 executable mutants generated from the eight new base apps, i.e., apps 08–15. Further, we analyzed the 7,708 uncaught mutants (i.e., the leaks not detected by Argus) using the methodology previously described in Section 4.3. Through the addition of eight new base apps, we aim to add diversity to the created mutants, as these new apps are likely to encompass a new set of development practices that might impact our findings. Therefore, this new set of apps might also help us find new flaws through μSE due to the additional, diverse mutations created. While we do not consider the original apps from our USENIX '18 study for Argus and HornDroid, we consider this to be a balanced tradeoff as we later study the propagation of flaws from both old and new sets of apps across tools.

Results: Through analyzing the uncaught mutants, we discovered nine unique flaws in Argus, as shown and classified in Table 3. Note that these flaws are *separate from* the ones found in FlowDroid (Section 8.1). This demonstrates that the flaws found from our USENIX '18 study [11] were not inherently coupled to the base apps chosen, and that μSE can be effective with different sets of apps representing a reasonable level of diversity. A description of each flaw is provided in Table 3. Our project repository provides minimal APKs representing the flaws in Argus [100].

Of the nine flaws, three are based on Fragment usage and are of the FC4 flaw class, while the remaining six fall in FC1. Note that while we found fragment-based flaws in FlowDroid, the four fragment-related flaws discovered for Argus are independent, although they could be interpreted as variants of those found in FlowDroid. The flaws discovered in Argus are described as follows.

Argus flaws in FC4: Similar to FlowDroid, Argus struggles with identifying leaks in fragments. We observe that most of these problems are at the design level, and occur because Argus does not track flows between the GUI components defined in fragments. To elaborate, Argus misses asynchronous flows between GUI components defined in the relevant XML files for fragments and the corresponding click-event listeners. All of our fragment-based flaws (F14–F16) exploit this design gap in Argus to avoid detection.

Table 3. Descriptions of Flaws Uncovered in Argus v3.1.2

ID: Flaw Name	Description
	FC4: Incorrect Modeling of Asynchronous Methods
F14: FragmentEventToExternalMethod	Fragment declared within an Activity class requires its click event listening methods to be defined in the Activity class. When we placed source in such click event listening method and the sink in an Activity method callable by, Argus missed the leak.
F15: FragmentCrossClick-EventListeners	In similar construction of F14, when source and sink are distributed across click event listener methods connected to Fragment GUI components, Argus missed the leak.
F16: OnCreateFragmentClick-EventListener	Android Activity lifecycle method onCreate can be used to create a source for leak. If it is then leaked through a method called by a fragment component click event listening method, it is undetected by Argus.
	FC1: Missing Callbacks
F17: FragmentClickEventListener	When sink and source are placed in an event listener method defined in an Activity class are coupled with components in a Fragment through relevant XML resource file, Argus misses the leak. In similar construction to F14, if both source and sink are placed in event listener methods for fragment components, Argus does not report it.
F18: RecyclerViewHolder	A RecyclerView.ViewHolder abstract class is used to describe each item within a RecyclerView. This is required to be extended when used inside the extending class of RecyclerView.Adapter. When leak source and sink are placed within the scope of the class implementing the RecyclerView.ViewHolder, Argus is unable to detect it.
F19: RecyclerViewConstructor	This flaw is similar to F18, but where the leak source and sink are placed within the constructor of the class extending RecyclerView.ViewHolder. Argus is unable to detect the placed leak in this scenario.
F20: RecyclerOnCreate-ViewHolder	The class extending RecyclerView.Adapter implements the abstract method OnCreateViewHolder, when ViewHolder needs to represent an item. If a leak is placed within OnCreateViewHolder, Argus's analysis cannot detect it.
F21: RecyclerViewOnBind-ViewHolder	Similar to RecyclerCreateViewHolder, the class extending RecyclerView.Adapter implements the abstract method onBindViewHolder, when ViewHolder needs to display the data at the specified position. If a leak is placed within onBindViewHolder, Argus's analysis cannot detect it.
F22: RecyclerViewGetItemCount	getItemCount is an abstract method required to be overridden to return the total number of items bound in RecyclerView. This method is placed within a class extending RecyclerView.Adapter. Argus is unable to find a leak if the source and sink are placed within getItemCount.

Argus flaws in FC1: Of the remaining six flaws we discovered in Argus, five are based on RecyclerView widgets. The RecyclerView is used to display a collection of data within a limited, scrolling window. Each RecyclerView widget implementation includes classes extending RecyclerView.ViewHolder and RecyclerView.Adapter. Furthermore, these classes contain several abstract methods which must be implemented, namely, onCreateViewHolder, onBindViewHolder, and getItemCount. Our analysis reveals that Argus fails to detect leaks placed in any of these components and methods. In addition, Argus does not detect flaws when leaks are placed in click-event listeners statically connected to fragment components via XML resource files. This demonstrates that further work is required to analyze relations in between methods not only through source code, but resource files as well. A summary of these flaws is tabulated in Table 3, and the example minimal APKs, each exhibiting a single flaw, are available in our online appendix [100].

Table 4. Descriptions of Flaws Uncovered in HornDroid

ID: Flaw Name	Description
	FC5: Android Lifecycle Callbacks
F23: OnCreateFragmentConstructor	HornDroid does not detect a leak if the source is placed in the onCreate method of an activity, and the sink is placed in the constructor of the fragment within the activity.
F24: ActivityOncreate	When a leak is placed in the onCreate method of Activity, i.e., both source and sink are placed in the method, HornDroid is unable to detect the leak.
F25: FragmentOnCreateView	When the source of the leak is placed at the onCreate method of activity class, and the sink at the onCreateView method of the fragment class, HornDroid does not detect the leak.

8.3 Evaluating HornDroid

HornDroid was proposed by Calzavara et al. [14] as the first static analysis tool for Android with a formal proof of soundness. This unique attribute motivated our choice of HornDroid for in-depth evaluation in this extended study. HornDroid abstracts Android applications as a set of Horn clauses to formulate security properties, which can then be processed by Satisfiability Modulo Theories (**SMT**) solvers.

HornDroid's formal proof and over-approximation require far more resources than the other tools we evaluate, i.e., as stated in the HornDroid paper, the authors tested HornDroid on a server with 64 multi-thread cores and 758 Gb of memory, although they reported that the most memory utilization was around 10 GB. To match this maximum 10 GB memory utilization, we conducted our study on a server with 32 GB of RAM and 8 cores. When analyzing a mutated app with HornDroid, we set a timeout of 36 hours, after which we would abort the analysis and report whatever mutants were caught until that time.

Results: We were not able to analyze many of our mutated apps using HornDroid. Specifically, out of the 31 mutated APKs (i.e., created after mutating the base APKs 08–15), HornDroid could only analyze four without crashing or timing out. This was in spite of us taking care to compile the APKs with the API level that HornDroid was built to analyze, i.e., API 19. This outcome is unfortunately not surprising; indeed, prior work on analyzing the feasibility of Android static analysis tools has reported similar results for research prototypes [91]. As a result, HornDroid successfully analyzed 46 executable data leaks, out of which, it caught 14, leaving 32 undetected mutants for further analysis. We discovered three flaws from analyzing these undetected mutants, as shown in Table 4.

All of the three flaws (F23–F25) we discovered are related to the lack of appropriate support for fragments; however, unlike prior fragment-related flaws, these flaws are generally centered around the Android lifecycle methods for fragments and activities. These flaws could broadly fit in FC1; however, they demonstrate a more fundamental, design-level lack of support for fragments, as HornDroid fails to detect leaks in even basic lifecycle methods (e.g., Fragment onCreateView). Hence, we create a new flaw class to represent such flaws, *FC5: Android Lifecycle Callbacks*, as a special variant of FC1, described as follows.

FC5: Android lifecycle callbacks: This class incorporates flaws in detecting leaks in fundamental lifecycle methods, such as onCreate. To elaborate, there are only six lifecycle callback methods in total for any Android activity [24], where onCreate is the only one considered mandatory. It is also the first method to be called when the Activity is initialized, i.e., it is the starting point of the Activity. Hence, analyzing such callbacks is imperative for a practical analysis of dataflows within the app. However, HornDroid fails to detect leaks in three specific instances of lifecycle

Table 5. Impact of Operator Placement Approaches in Finding Flaws*

Flaw ID	Reachability	Complex-Reachability	Taint	Scope
Flaws found from FlowDroid v2.0				
F1	✓	✓	✓	✓
F2	✓	✓	✓	✓
F3	✓	✓	✓	✓
F4	✓	✓	✓	✓
F5	✓	✓	✓	✓
F6	✓	✓	✓	✓
F7	✓	✓	✓	✓
F8	✓	✓	✓	✓
F9	✓	✓	✓	✓
F10	-	-	✓	✓
F11	-	-	✓	✓
F12	-	-	✓	✓
F13	-	-	✓	✓
Flaws found from Argus v3.1.2				
F14	-	-	✓	✓
F15	-	-	✓	✓
F16	-	-	✓	✓
F17	✓	✓	✓	✓
F18	✓	-	-	-
F19	✓	✓	✓	✓
F20	✓	✓	✓	✓
F21	✓	✓	✓	✓
F22	✓	✓	✓	✓
Flaws found from HornDroid				
F23	✓	✓	✓	✓
F24	✓	✓	✓	✓
F25	-	-	-	✓

*A "✓" indicates flaws may be resolved through a relevant operator placement approach, whereas "-" indicates it may not.

callbacks for activities and fragments (F23–F25), which is concerning, as the tool strongly claims soundness, i.e., quoted as follows: *"In order to support a sound analysis of fragments, HornDroid over-approximates their lifecycle by executing all the fragments along with the containing activity in a flow-insensitive way."*

8.4 Effectiveness of Individual Schemes in Finding Flaws

As described in Section 4.2, μSE uses four different mutation schemes to place operators in an app. These schemes may sometimes create overlapping mutants wherein operators are placed in the same position according to two or more schemes. As part of this extended study, we trace the flaws we found back to the operator placement strategies described in Sections 4.2.2 and 4.2.3 to determine the usefulness of these approaches in finding flaws (**RQ$_2$**).

As shown in Table 5, of the 25 flaws, 24 could be discovered using the scope-based scheme, and 22 using the taint-based scheme. In addition, we found one flaw (i.e., F18) that could only be reached using the reachability-based scheme. Another interesting observation is that both the Reachability and Complex Reachability schemes are similar in terms of usefulness in finding flaws, which may be a positive indicator of a general reachability-based approach over more complex strategies.

Based on the data in Table 5 alone, it may seem that only using the scope-based scheme is a viable option. However, recall from Section 7 that mutation schemes may also display widely different performance in terms of creating *executable* mutants. As a result, using a single scheme may improve flaw detection, but may also generate a tremendous overhead in terms of non-executing mutants. Furthermore, even if the same mutation/leak placement can be achieved using multiple schemes, the common placement may help the researcher or tool developer resolve the flaw behind an uncaught mutant faster, i.e., by helping them see the common or different factors (i.e., among the schemes). For example, the Complex Reachability scheme makes the path between source and sink indirect while residing within the same scope, whereas the reachability scheme places the source and sink at the same location without establishing any indirect path. Failure to detect the leak placed through Complex Reachability would indicate that the flaw is due to the indirect path, rather than the location of the leak.

9 FLAW PROPAGATION STUDY

The objective of this study is to determine whether flaws from one tool propagate to other tools, which are either implemented directly on top of the original tool, or inherit certain design attributes (**RQ_3**). To carry out this study, we utilized the minimal APKs created for each of our 25 flaws, and analyzed them with other tools built with the same security goal (i.e., other data leak detectors excluding the tools from which the specific flaws were discovered). Each minimal APK contains only one type of flaw, i.e., we built 25 APKs, one for each flaw mentioned in Tables 2, 3, and 4.

Moreover, in order to prevent tools from crashing due to backwards compatibility issues, we built several versions of the minimal APKs from the same code base, varying the SDK versions, as well as the build tools (i.e., Android Studio and Gradle vs. building manually using Android SDK tools). This was done because APK building procedure has changed over the years, and as a result, many of the studied tools, which are well over 4–5 years old, would break for apps built using the latest build configuration and procedure. For example, Android Studio's Gradle-based building overrides the target SDK version defined in the AndroidManifest.xml file, and also uses newer versions of build tools and target platforms. We discovered that decompilers such as dare [78] used in some of the tools we analyzed do not function correctly when analyzing such builds (i.e., either crash, or run into infinite loops). Considering these factors, we try our best to customize minimal APKs for individual tools, in order to minimize crashes or timeouts, and only report results from variants of minimal APKs that worked., i.e., resulted in a successfully completed analysis.

9.1 Propagation of FlowDroid's Flaws (F1–F13)

To determine if the flaws present in FlowDroid are also present in other data leak detectors, as well as tools that inherit it, we checked if the newer release versions of FlowDroid (i.e., v2.5, v2.5.1, and v2.7.1), as well as six other tools (i.e., Argus, DroidSafe, IccTA, BlueSeal, HornDroid, and DidFail), are susceptible to any of the flaws discussed in Table 2. Among these, flaw propagation across versions of FlowDroid was done for v2.5 and v2.5.1 in our original μSE study. As part of this extended study, due to availability of FlowDroid v2.7.1, we include it.
Results: Table 6 provides an overview of the propagation of F1–F13. In Table 6, all the versions of FlowDroid are susceptible to the flaws discovered from our analysis of FlowDroid v2.0. Note that while we fixed the Fragment flaw and our patch was accepted to FlowDroid's codebase, the latest releases of FlowDroid (i.e., v2.5, v2.5.1, and v2.7.1) still seem to have this flaw. We have reported this issue to the developers.

Table 6. Analysis of the Propagation of Flaws in FlowDroid 2.0 to Other Data Leak Detectors*

Flaw ID	FD v2.0	FD v2.5	FD v2.5.1	FD v2.7.1	Argus	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
F1	✓	✓	✓	✓	x	x	✓	x	✓	✓
F2	✓	✓	✓	✓	x	x	✓	x	✓	✓
F3	✓	✓	✓	✓	✓	-	✓	-	-	✓
F4	✓	✓	✓	✓	✓	x	✓	x	✓	✓
F5	✓	✓	✓	✓	✓	✓	✓	-	✓	✓
F6	✓	✓	✓	✓	✓	x	✓	x	✓	✓
F7	✓	✓	✓	✓	✓	x	✓	x	✓	✓
F8	✓	✓	✓	✓	x	x	✓	✓	x	✓
F9	✓	✓	✓	✓	x	x	✓	x	x	✓
F10	✓	✓	✓	✓	x	x	✓	x	x	✓
F11	✓	✓	✓	✓	✓	x	✓	x	x	✓
F12	✓	✓	✓	✓	x	x	✓	x	x	✓
F13	✓	✓	✓	✓	x	x	✓	x	x	✓

* Note that a "-" indicates tool crash with the minimal APK, a "✓" indicates presence of the flaw, and an "x" indicates absence; FD* = FlowDroid.

A significant observation from Table 6 is that the tools that directly inherit FlowDroid (i.e., IccTA, DidFail) are similarly flawed as FlowDroid. This is especially true when the tools do not augment FlowDroid in any manner, and use it as a black box (**RQ$_3$**). On the contrary, Argus, which is motivated by FlowDroid's design, but augments it on its own, does not inherit as many of FlowDroid's flaws.

BlueSeal, HornDroid, and DroidSafe use a significantly different methodology, and are also not susceptible to many of μSE's uncovered flaws for FlowDroid. Interestingly, BlueSeal and DroidSafe are similar to FlowDroid in that they use Soot to construct a control flow graph and rely on it to identify paths between sources and sinks. However, BlueSeal and DroidSafe both augment the graph in novel ways, and thus do not exhibit most of the flaws found in FlowDroid.

Finally, it is important to note that our analysis does not imply that FlowDroid is weaker than the tools, which have fewer flaws in Table 6. However, it does indicate that the flaws discovered may be typical of the design choices made in FlowDroid and inherited by the tools such as IccTA and DidFail.

9.2 Propagation of Argus's Flaws (F14–F22)

Unique to this extended study (i.e., previously not reported in our original μSE paper [11]), we analyzed the minimal APKs developed for F14–F22 with FlowDroid v2.5.1, v2.6, v2.6.1, v2.7, and v2.7.1; BlueSeal, DroidSafe, DidFail, HornDroid, and IccTA to examine their prevalence. Note that FlowDroid 2.0 had become unavailable at this point in the study (i.e., deprecated in favor of later versions), which is why we focus on FlowDroid versions from 2.5 to 2.7.1.

Results: As shown in Table 7, flaws found in Argus largely affect FlowDroid, HornDroid, and IccTA. Specifically, recall that while some of the fragment and RecyclerView-based flaws were missing callbacks that could potentially be fixed with patches, some were design-level gaps in the dataflow tracking performed by these tools. For example, in F14, an event listener reads sensitive data (i.e., the source), and calls another method, which leaks the data (i.e., the sink). While such event listeners may not be directly called from any of the lifecycle methods in the app, they are likely to be invoked when user interacts with the GUI components of the app. The fact that none of the RecyclerView-based (i.e., F18–F22) or fragment-based (i.e., F14–F17) leaks were detected by

Table 7. Analysis of the Propagation of Flaws in Argus 3.1.2 to Other Data Leak Detectors*

Flaw ID	Argus	FD* v2.5.1	FD* v2.6	FD* v2.6.1	FD* v2.7	FD* v2.7.1	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
F14	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F15	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F16	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F17	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F18	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F19	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F20	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F21	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F22	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓

* Note that a "-" indicates tool crash with the minimal APK, a "✓" indicates presence of the flaw, and an "x" indicates absence; FD* = FlowDroid.

Table 8. Analysis of the Propagation of Flaws in HornDroid to Other Data Leak Detectors*

Flaw ID	Argus	FD* v2.5.1	FD* v2.6	FD* v2.6.1	FD* v2.7	FD* v2.7.1	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
-	Argus	FD2.5.1	2.6	2.6.1	2.7	2.7.1	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
F23	x	x	-	x	-	✓	x	-	✓	✓	x
F24	x	x	-	x	-	✓	x	-	✓	✓	x
F25	x	✓	-	✓	-	✓	x	-	✓	✓	✓

* Note that a "-" indicates tool crash with the minimal APK, a "✓" indicates presence of the flaw, and an "x" indicates absence; FD* = FlowDroid.

any of the tools we evaluated demonstrates the fragility of these tools in the face of commonly used Android GUI abstractions.

DroidSafe and BlueSeal crashed when analyzing the minimal APKs for F18–F22, while DidFail crashed for all flaws, i.e., F14–F22. These crashes are expected, and primarily occur due to lack of support for newer API. To elaborate, DroidSafe was specifically built to focus on Android 4.4.1 (API 19), and hence crashes on encountering RecyclerView, i.e., the set of APIs at the root of F14–F22, which was introduced in Android 5.0 (API 21). Similarly, BlueSeal customizes an older version of Soot for analysis, which fails when analyzing RecyclerView -based APKs. Finally, DidFail was built using customized but outdated versions of several other tools, namely, FlowDroid, Epicc [80], and dare [78], and hence crashes on all the newer flaws. Note that while tools such as FlowDroid and dare may be separately updated and maintained, the current implementation of DidFail customizes these dependencies to a sufficient degree, which prevents us from simply replacing the dependency with a newer version. However, we can still infer that since DidFail uses FlowDroid as a component, it is likely to inherit all of FlowDroid's flaws.

9.3 Propagation of HornDroid's Flaws (F23–F25)

To understand whether the flaws we found in HornDroid propagate to other tools, we prepared minimal APKs based on the found flaws of HornDroid, and analyzed them using Argus, FlowDroid versions v2.5.1, v2.6, v2.6.1, v2.7, and v2.7.1; BlueSeal, DroidSafe, DidFail, and IccTA. None of these flaws, nor their propagation, were reported in our original μSE paper [11].

Results: As shown in Table 8, we find that Argus did not exhibit any of the fragment-based flaws identified in HornDroid. This is surprising, considering that HornDroid was susceptible to every single flaw discovered in Argus, as seen previously in Table 7. This finding may indicate that Argus may be relatively more sound than HornDroid in practice, in spite of the latter providing a formal proof of soundness.

Furthermore, although FlowDroid's earlier versions (v2.5.1, v2.6.1) were able to detect leaks F23 and F24, FlowDroid version v2.7.1 was not. This is not an isolated case: recall that our patch for another fragment flaw F5 [99] fixed it in FlowDroid v2.0, but future versions of FlowDroid (i.e., v2.5 onwards) still exhibit the flaw, as seen in Table 6. Interestingly, we note that F23 and F24 are absent in IccTA, an approach that relies on FlowDroid. This is because IccTA uses an older version of FlowDroid as a component, which is resistant to these flaws, and hence, remains unaffected as well. Moreover, F25, a fragment-based flaw, is exhibited by all versions of FlowDroid. This finding is an indicator of the lack of systematic fragment support in FlowDroid, as well as other major tools, when in fact fragments are a widely used GUI element that may contain data leaks.

Finally, the propagation study also allows us to derive certain general conclusions regarding the quality of the tools studied. First, we conclude that all of the tools we analyzed are incapable of finding leaks that are exhibited in RecyclerView s. Second, we find evidence to suggest that Argus is relatively better for detecting fragment-based leaks, relative to HornDroid and FlowDroid. This may seem counterintuitive considering there was only one fragment-related flaw (F5) in FlowDroid, two in HornDroid (F23, F25), and four in Argus (F14–F17). However, when we consider the propagation of fragment-related flaws as well, Argus is affected by five (F5, F14–F17), while HornDroid and FlowDroid (considering its latest release, v2.7.1) are both affected by seven (i.e., both are affected by flaws F5, F14–F17, and F23, F25). Thus, we argue that Argus provides more holistic support for fragments, relative to the other tools studied.

10 DISCUSSION

μSE has demonstrated efficiency and effectiveness at revealing real undocumented flaws in prominent Android security analysis tools. While experts in Android static analysis may be familiar with some of the flaws we discovered (e.g., some flaws in FC1 and FC2), we aim to document these flaws for the entire scientific community. Further, μSE indeed found some design gaps that were surprising to expert developers; e.g., FlowDroid's design does not consider callbacks in anonymous inner classes (flaws 8–9, Table 6), and in our interaction with the developers of FlowDroid, they acknowledged handling such classes as a non-trivial problem. During our evaluation of μSE, we were able to glean the following pertinent insights.

Insight 1: *Most mutation schemes are generally effective.* While certain mutation schemes may be Android-specific, our results demonstrate limited dependence on these configurations. Out of the 25 flaws discovered using μSE (i.e., both in our USENIX '18 paper [11] as well as this extension), we discover that each mutation scheme is necessary for detecting certain flaws (i.e., which may not be detected with other schemes), as shown in Section 8.4.

Insight 2: *Security-focused static analysis tools exhibit undocumented flaws that require further evaluation and analysis.* Our results clearly demonstrate that previously unknown security flaws or undocumented design assumptions, which can be detected by μSE, pervade existing Android security static analysis tools. Our findings not only motivate the dire need for systematic discovery, fixing, and documentation of unsound choices in these tools, but also clearly illustrate the power of mutation-based analysis adapted in security context.

Insight 3: *Current tools inherit flaws from legacy tools.* A key insight from our work is that while inheriting code of the foundational tools (e.g., FlowDroid) is a common practice, some of the researchers may not necessarily be aware of the unsound choices they are inheriting as well. As our study results demonstrate, when a tool inherits another tool directly (e.g., IccTA inherits FlowDroid), all the flaws propagate.

Insight 4: *Tools which follow similar design principles but do not have a direct relationship (e.g., inheriting a codebase), have similar flaws.* Through our experiments and evaluation performed in this extended study, we effectively demonstrate that FlowDroid, HornDroid, and Argus, three different static analysis tools built independently from each other, but which share similar design principles, can and do exhibit similar flaws. This indicates that certain unsound decisions or flaws may be tied to the common security goal or could be occurring due to fundamental gaps in the high-level design decisions that are common across the board for such tools.

Insight 5 : *Flaws which were not present in previous versions of a static analysis tool can appear in later versions, as the tool evolves.* As we found in this extended study, certain flaws found in HornDroid were not present in earlier versions of FlowDroid but appeared in the latest version (Table 8). This shows that unsound choices can be made at any stage and at any iteration of software lifecycle, and further establishes the necessity of automatically and systematically evaluating these tools. μSE lays the groundwork for the development of such a holistic, dynamic, testing framework.

Insight 6: *As tools, libraries, and the Android platform evolve, security problems become harder to track down.* Due to the nature of software evolution, all the analysis tools, underlying libraries, and the Android platform itself evolve asynchronously. A few changes in the Android API may introduce undocumented flaws in analysis tools. μSE handles this fundamental obstacle of continuous change by ensuring that each version of an analysis tool is systematically tested, as we realize while tracking the Fragment flaw in multiple versions of FlowDroid.

Insight 7: *Benchmarks need to evolve with time.* While manually curated benchmarks (e.g., DroidBench [6]) are highly useful as a "first line of defense" in checking if a tool is able to detect well-known flaws, the downside of relying too heavily on benchmarks is that they only provide a known, finite number of tests, leading to a false sense of security. Due to constant changes (insight #6) benchmarks are likely to become less relevant unless they are constantly augmented, which requires tremendous effort and coordination. μSE significantly reduces this burden on benchmark creators via its suite of extensible and expressive security operators and mutation schemes, which can continuously evaluate new versions of tools. The key insight we derive from our experience building μSE is that *while benchmarks may check for documented flaws, μSE 's true strength is in discovering new flaws.*

11 RELATED WORK

μSE builds upon the theoretical underpinnings of mutation analysis from SE, and to our knowledge, is the first work to adapt mutation analysis to evaluate the soundness claimed by security tools. Moreover, μSE adapts mutation analysis to security, and makes fundamental and novel modifications (described previously in Section 4). We now describe prior work in three other related areas.

Formally verifying soundness: While an ideal approach, formal verification is one of the most difficult problems in computer security. For instance, prior work on formally verifying apps often requires the monitor to be rewritten in a new language or use verification-specific programming constructs (e.g., verifying reference monitors [41, 102], information flows in apps [72, 73, 107]), which poses practical concerns for tools based on numerous legacy codebases (e.g., FlowDroid [6], CHEX [67]). Further, verification techniques generally require correctness to be specified, i.e., the policies or invariants that the program is checked against. Concretely defining what is "correct" is hard even for high-level program behavior (e.g., making a "correct" SSL connection) and may be infeasible for complex static analysis tools (e.g., detecting "all incorrect SSL connections"). μSE does not aim to substitute formal verification of static analysis tools; instead, it aims to uncover existing limitations of such tools.

Evaluating static analyses: Recently, there has been significant work in the area of experimentally evaluating the features and effectiveness of static analysis techniques. For instance, Qiu et al. [87] performed a comparative evaluation of precision and runtime performance, among FlowDroid+IccTA, AmanDroid, and DroidSafe, by using a common configuration setup, using a benchmark that extends DroidBench [28] and ICC-Bench [48]. Pauck et al. [84] propose the ReproDroid framework that automatically evaluates the effectiveness of Android taint analysis tools using user-labeled ground truth in Android apps. However, there are fundamental differences in our work, and these related approaches, in terms of the primary goal, scope, and the actual techniques leveraged.

To elaborate, μSE focuses on exhaustively generating security test cases for evaluating Android security tools, and systematically performing in-depth evaluations of such tools to discover gaps in the soundness that directly affect their ability to detect security vulnerabilities such as data leaks. μSE's security focus is evident in our additional efforts toward designing security-focused mutation, and attributing undetected mutants to actual flaws and design choices in tools that affect security. On the contrary, both prior approaches focus on evaluating either the presence of promised static analysis *features*, or the general precision, i.e., with a lack of specific focus on security. Furthermore, μSE is a holistic, automated, mutation framework that generates thousands of expressive, *security-goal-focused test cases* for evaluating security tools, while related work generally relies on handcrafted benchmarks [87] or user-specified ground truth [84], which may be sufficient for evaluating features, but not for a thorough security evaluation of tools. That is, μSE *empowers* security researchers to discover flaws without delving into the intricate details of program analysis techniques. However, we do note that certain aspects of prior work (e.g., the automated bootstrapping of tools in ReproDroid [84]) are complementary to μSE, and may be incorporated into its pipeline in the future. Finally, while μSE evaluates soundness, our mutation-based approach may be used to evaluate precision as well, which is a separate research direction that we aim to explore in the future.

Android application security tools: The popularity and open-source nature of Android has spurred an immense amount of research related to examining and improving the security of the underlying OS, SDK, and apps. Recently, Acar et al. have systematized Android security research [2], and we discuss work that introduces static analysis-based countermeasures for Android security issues according to Acar et al.'s categorization.

Perhaps the most prevalent area of research in Android security has concerned the permissions system that mediates access to privileged hardware and software resources. Several approaches have motivated changes to Android's permission model, or have proposed enhancements to it, with goals ranging from detecting or fixing unauthorized information disclosure or leaks in third-party applications [6, 32, 42, 53, 74, 75, 106] to detecting over privilege in applications [7, 36, 104]. Similarly, prior work has also focused on benign but vulnerable Android applications, and proposed techniques to detect or fix vulnerabilities such as cryptographic API misuse API [30, 34, 35, 97] or unprotected application interfaces [16, 37, 57]. Moreover, these techniques have often been deployed as modifications to Android's permission enforcement [12, 13, 17, 26, 32, 33, 37, 40, 45, 77, 83, 85, 93, 96, 112], SDK tools [7, 36, 104], or inline reference monitors [9, 10, 18, 52, 105]. While this article demonstrates the evaluation of only a small subset of these tools with μSE, our experiments demonstrate that μSE has the potential to impact nearly all of them. For instance, we can apply μSE to vet SSL analysis tools by purposely introducing complex SSL errors in applications, or privilege or permission misuse analysis tools, by developing security operators that attempt to misuse permissions.

12 LIMITATIONS

(1) Soundness of μSE: As acknowledged in Section 11, μSE does not aim to supplant formal verification (which would be sound) and does not claim soundness guarantees. Rather, μSE provides a systematic approach to semi-automatically uncover flaws in existing security tools, which is a significant advancement over manually curated tests.

(2) Manual effort: Presently, the workflow of μSE requires an analyst to manually analyze the result of μSE (i.e., uncaught mutants). However, as described in Section 7, μSE possesses enhancements that mitigate the manual effort by dynamically eliminating non-executable mutants, that would otherwise impose a burden on the analyst examining undetected mutants. In our experience, this analysis was completed in a reasonable time using the methodology outlined in Section 4.3.
(3) Limitations of execution engine: Like any dynamic analysis tool, the EE will not explore all possible program states, thus, there may be a set of mutants marked as non-executable by the EE, that may actually be executable under certain scenarios. However, the CrashScope tool, which μSE's EE is based upon, has been shown to perform comparably to other tools in terms of coverage [71]. Future versions of μSE's EE could rely on emerging input generation tools for Android apps [69].
(4) Dependency on android framework APIs: We designed μSE to be as generic as possible, as discussed in Section 4.4 and Section 5. For example, the mutation seeding methodology relies on the AST of the target source code that *selects* the target location for mutation. As a result, as long as the Android framework changes through extension and target code are parseable as AST, the seeding methodology will not have to be changed. Indeed, the base apps we used for μSE (Table 9 in the Appendix) rely on different versions of Android SDK with both a Gradle [49] and a pre-Gradle build system, which demonstrates the *versatility* of μSE. On the other hand, calls to functions/methods from the Android framework APIs are introduced through mutation. Therefore, these *will* have to be changed as Android framework changes over time, as it is *not possible* to generalize such calls.
(5) Adaptation to different goals: μSE requires defining security operator through manual examination of the claims made by existing tools, i.e., the security goal of the concerned tool (Section 4.1). Further changes might be necessary for satisfying syntactical requirements related to the defined, new, security operator. For example, for `Cipher.getInstance`, the security operator will have to be enclosed within a try-catch scope because of its `throws-exception` signature. Thus, the implementation of security operators to mutate cryptographic APIs may require manual intervention by domain experts initially, but we expect it to be a one-time effort, similar to how we defined security operators for data leak APIs once for detecting flaws in this article. Finally, we note that μSE's modular implementation separates the operators from its core components, i.e., the ME and the EE simply seed and execute security operators as per the operator specification, and hence, are decoupled from the security goal per se. Hence, the implementation of μSE framework would not have to change for different security goals, limiting the amount of code change to only the addition of goal-specific security operators.

13 CONCLUSION

We proposed the μSE framework for performing systematic security evaluation of Android static analysis tools to discover (undocumented) unsound assumptions, adopting the practice of mutation testing from SE to security. μSE not only detected major flaws in popular, open-source Android security tools, but also demonstrated how these flaws propagated to other tools that inherited the security tool or followed similar principles. With μSE, we demonstrated how mutation analysis can be feasibly used for gleaning unsound assumptions in existing tools, benefiting developers, researchers, and end users, by making such tools more secure and transparent.

APPENDIX

CrashScope (Execution Engine): The EE functions builds upon CrashScope [70, 71], which statically analyzes the code of a target app to identify activities implementing potential contextual features (e.g., rotation, sensor usage) via API call-chain propagation. This execution is guided

```
    BroadcastReceiver receiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
        BroadcastReceiver receiver = new BroadcastReceiver(){
            @Override
                public void onReceive(Context context, Intent intent) {
                    String dataLeak = Calendar.getInstance().getTimeZone().getDisplayName();
                    Log.d("leak-1", dataLeak);}};
        registerReceiver(receiver, new
             IntentFilter().addAction("android.intent.action.SEND"));
    }};
    registerReceiver(receiver, new IntentFilter().addAction("android.intent.action.SEND"));
```

Listing 5. A dynamically created BroadcastReceiver, created inside another, with data leak. Whenever the onReceive() callback of the receiver object is invoked, it will create another receiver object of similar type, with a leak inside its own onReceive() callback. This can be further evolved to use anonymous object declaration that can leak information in a similar nature.

Table 9. List of App Names, URLs, and IDs Assigned by Us for the Purpose of the μSE Study

App ID	Andorid App name	URL
app 01	2048	https://f-droid.org/en/packages/com.uberspot.a2048/
app 02	Protect Baby Monitor	https://f-droid.org/en/packages/protect.babymonitor/
app 03	QR Scanner	https://f-droid.org/en/packages/com.secuso.privacyFriendlyCodeScanner/
app 04	Location Share	https://f-droid.org/en/packages/ca.cmetcalfe.locationshare/
app 05	Camera Roll	https://f-droid.org/en/packages/us.koller.cameraroll/
app 06	AndroidPN Client	https://f-droid.org/en/packages/org.androidpn.client/
app 07	Activity Launcher	https://f-droid.org/en/packages/de.szalkowski.activitylauncher/
app 08	Man Man	https://f-droid.org/en/packages/com.adonai.manman/
app 09	BMI Calculator	https://f-droid.org/en/packages/com.zola.bmi/
app 10	A Time Tracker	https://f-droid.org/en/packages/com.markuspage.android.atimetracker/
app 11	AFH Downloader	https://f-droid.org/en/packages/org.afhdownloader/
app 12	Android Explorer	https://f-droid.org/en/packages/com.iamtrk.androidexplorer/
app 13	Kaltura Device Info	https://f-droid.org/en/packages/com.oF2pks.kalturadeviceinfos/
app 14	Apod Classic	https://f-droid.org/en/packages/com.jvillalba.apod.classic/
app 15	Calendar Trigger	https://f-droid.org/en/packages/uk.co.yahoo.p1rpp.calendartrigger/

by one of several exploration strategies, organized along three dimensions: (i) GUI exploration, (ii) text entry, and (iii) contextual features.

Note that because the goal of the EE is to explore as many screens of a target app as possible, the EE forgoes certain combinations of exploration strategies from CrashScope [70, 71] (e.g., entering unexpected text or disabling contextual features) prone to eliciting crashes from apps. The approach uses adb and Android's `uiautomator` framework to interact with and extract GUI-related information from a target device or emulator. Further implementation details of exploration strategies can be found in [70, 71].

Apps used in the study: For our study, we collected a set of 15 open-source apps from F-Droid [62], as shown in Table 9. The apps come from different heterogeneous build configuration settings, with compile SDK API level 23 (Marshmallow) to 27 (Oreo), minimum SDK API level 9 (Gingerbread) to 16 (Jelly Bean), and target SDK API level from 17 (Jelly Bean) to 26 (Oreo), with sizes ranging from several hundred KB to a maximum of 3.5 MB.

ACKNOWLEDGMENTS

We thank the developers of the evaluated tools for making their tools available to the community, and for being open to suggestions. We thank Richie Bonnett for his contributions to the conference paper version of this work.

REFERENCES

[1] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. Hare hunting in the wild Android: A study on the threat of hanging attribute references. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, New York, NY, 1248–1259. DOI: https://doi.org/10.1145/2810103.2813648

[2] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. 2016. SoK: Lessons learned from Android security research for appified software platforms. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP'16)*.

[3] Android Developers. [n.d.]. Fragments. Retrieved July 7, 2019 from https://developer.android.com/guide/components/fragments.html.

[4] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated testing for SQL injection vulnerabilities: An input mutation approach. In *International Symposium on Software Testing and Analysis, (ISSTA'14)*. 259–269.

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS'14)*.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*.

[7] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 217–228.

[8] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering—Volume 1*. 426–436.

[9] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged app sandboxing for stock Android. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*.

[10] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard: Enforcing user requirements on Android apps. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*.

[11] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, 1263–1280. https://www.usenix.org/conference/usenixsecurity18/presentation/bonett.

[12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2012. Toward taming privilege-escalation attacks on Android. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'12)*.

[13] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2011. Practical and lightweight domain isolation on Android. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM'11)*.

[14] S. Calzavara, I. Grishchenko, and M. Maffei. 2016. HornDroid: Practical and sound static analysis of Android applications by SMT solving. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS P'16)*. 47–62.

[15] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS'15)*.

[16] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*.

[17] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. 2010. CRePE: Context-related policy enforcement for Android. In *Proceedings of the 13th Information Security Conference (ISC'10)*.
[18] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. 2012. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications.
[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (April 1978), 34–41.
[20] Lin Deng, N. Mirzaei, P. Ammann, and J. Offutt. 2015. Towards mutation analysis of Android apps. In *IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15)*. 1–10.
[21] Anna Derezińska and Konrad Hałas. 2014. *Analysis of Mutation Operators for the Python Language*. Springer International Publishing, Cham, 155–164.
[22] Android Developers. 2019. Android Developer Documentation—Broadcasts. Retrieved July 7, 2019 from https://developer.android.com/guide/components/broadcasts.html.
[23] Android Developers. 2019. Android Developer Documentation—Intents and Intent Filters. Retrieved July 7, 2019 from https://developer.android.com/guide/components/intents-filters.html.
[24] Android Developers. 2019. Android Developer Documentation—The Activity Lifecycle. Retrieved July 7, 2019 from https://developer.android.com/guide/components/activities/activity-lifecycle.html.
[25] Daniel Di Nardo, Fabrizio Pastore, and Lionel C. Briand. 2015. Generating complex and faulty test data through model-based mutation analysis. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation, (ICST'15)*. 1–10.
[26] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the USENIX Security Symposium*.
[27] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*.
[28] DroidBench [n.d.]. DroidBench 2.0. Retrieved June 27, 2020 from https://github.com/secure-software-engineering/DroidBench.
[29] The Economist. 2015. Planet of the Phones. Retrieved July 7, 2019 from http://www.economist.com/news/leaders/21645180-smartphone-ubiquitous-addictive-and-transformative-planet-phones.
[30] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS'13*). ACM Press, 73–84. DOI: https://doi.org/10.1145/2508859.2516693
[31] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'11)*.
[32] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.
[33] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*.
[34] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 50–61. DOI: https://doi.org/10.1145/2382196.2382205
[35] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL development in an Appified world. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS'13)*. ACM, New York, NY, 49–60. DOI: https://doi.org/10.1145/2508859.2516655
[36] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'11)*.
[37] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. Permission re-delegation: Attacks and defenses. In *Proceedings of the USENIX Security Symposium*.
[38] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'14)*.
[39] The Apache Software Foundation. 2019. Apache Ant Build System. Retrieved July 7, 2019 from http://ant.apache.org.
[40] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. 2012. Modeling and enhancing Android's permission system. In *Computer Security —ESORICS 2012*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.). Springer, Berlin, 1–18.

[41] Jason Franklin, Sagar Chaki, Anupam Datta, and Arvind Seshadri. 2010. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *IEEE Symposium on Security and Privacy (SP'10)*. 365–379.

[42] Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST'12)*.

[43] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: Scalable and accurate zero-day Android malware detection. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys'12)*.

[44] R. G. Hamlet. 1977. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.* 3, 4 (July 1977), 279–290.

[45] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. ASM: A programmable interface for extending Android security. In *Proceedings of the USENIX Security Symposium.*

[46] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. 2014. PREC: Practical root exploit containment for Android devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY'14)*.

[47] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek. 2013. Flow permissions for Android. In *Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 652–657. DOI: https://doi.org/10.1109/ASE.2013.6693128

[48] iccbench [n.d.]. ICC-Bench. Retrieved June 27, 2020 from https://github.com/fgwei/ICC-Bench.

[49] Gradle Inc. 2019. Gradle Build System. Retrieved July 7, 2019 from https://gradle.org.

[50] Reyhaneh Jabbarvand and Sam Malek. 2017. *mu*Droid: An energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 208–219. DOI: https://doi.org/10.1145/3106237.3106244

[51] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. 2016. Mining sandboxes. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. 37–48.

[52] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: Fine-grained permissions in Android applications. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM'12)*.

[53] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-time enforcement of information-flow properties on Android (extended abstract). In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'13)*.

[54] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy Oracle: A system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security.* ACM New York, NY, 279–288.

[55] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis.* 1–6.

[56] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE Press, Piscataway, NJ, 931–936.

[57] Youn Kyu Lee, Jae young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A SEALANT for inter-app security holes in Android. In *Proceedings of the 39th International Conference on Software Engineering.* 312–323.

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

[59] L. Li, A. Bartel, J. Klein, and Y. L. Traon. 2014. Automatically exploiting potential component leaks in Android applications. In *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications.* 388–397. DOI: https://doi.org/10.1109/TrustCom.2014.50

[60] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2014. I know what leaked in your pocket: Uncovering privacy leaks on Android apps with static taint analysis. In *CoRR.*

[61] M. Lillack, C. Kastner, and E. Bodden. 2017. Tracking load-time configuration options. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. DOI: https://doi.org/10.1109/TSE.2017.2756048

[62] F-Droid Limited. 2019. F-Droid—Free and Open Source Android App Repository. Retrieved August 10, 2019 from https://f-droid.org/en/.

[63] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 233–244. DOI: https://doi.org/10.1145/3106237.3106275

[64] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis—-1,000,000 apps later: A view on current Android malware behaviors. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS'14)*.

[65] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*. ACM, New York, NY, 89–103. DOI: https://doi.org/10.1145/2742647.2742668

[66] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Communications of the ACM* 58, 2 (Jan. 2015).

[67] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'12)*. 229–240.

[68] Yu-Seung Ma, Yong Rae Kwon, and Jeff Offutt. 2002. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*. 352–366.

[69] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, New York, NY, 94–105.

[70] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. 2017. CrashScope: A practical tool for automated testing of Android applications. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C'17)*. 15–18. DOI: https://doi.org/10.1109/ICSE-C.2017.16

[71] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically discovering, reporting and reproducing Android application crashes. In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation, (ICST'16)*. 33–44.

[72] Andrew C. Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL'99)*.

[73] Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology* 9, 4 (October 2000), 410–442.

[74] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC enforcement on Android. In *Proceedings of the 25th USENIX Security Symposium*.

[75] Adwait Nadkarni and William Enck. 2013. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'13)*.

[76] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. UIPicker: User-input privacy identification in mobile applications. In *USENIX Security Symposium*. 993–1008.

[77] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 2010. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*.

[78] D. Octeau, S. Jha, and P. McDaniel. 2012. Retargeting Android applications to Java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*.

[79] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to Android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering—Volume 1 (ICSE'15)*. IEEE Press, Piscataway, NJ, 77–88. http://dl.acm.org/citation.cfm?id=2818754.2818767.

[80] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security'13)*. USENIX, 543–558. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau.

[81] A. Jefferson Offutt and Roland H. Untch. 2001. *Mutation 2000: Uniting the Orthogonal*. Springer US, Boston, MA, 34–44. DOI: https://doi.org/10.1007/978-1-4757-5939-6_7

[82] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon. 2015. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *Proceedings of the International Conference on Software Testing, Verification, and Validation—Workshops, (ICSTW'15)*. 1–10.

[83] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. 2009. Semantically rich application-centric security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC'09).* 340–349.

[84] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18).* ACM, New York, NY, 331–341. DOI: https://doi.org/10.1145/3236024.3236029

[85] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'12).*

[86] Upsorn Praphamontripong, Jeff Offutt, Lin Deng, and Jingjing Gu. 2016. An experimental evaluation of web mutation operators. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICSTW'16).* 102–111.

[87] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM Press, Amsterdam, Netherlands, 176–186. DOI: https://doi.org/10.1145/3213846.3213873

[88] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19).* ACM Press, London, United Kingdom, 2455–2472. DOI: https://doi.org/10.1145/3319535.3345659

[89] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. 2014. DroidForce: Enforcing complex, data-centric, system-wide policies in Android. In *Proceedings of the 2014 9th International Conference on Availability, Reliability and Security.* 40–49. DOI: https://doi.org/10.1109/ARES.2014.13

[90] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic large-scale dynamic analysis of Android applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY'13).*

[91] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. 2016. * droid: Assessment and evaluation of Android application analysis tools. *ACM Computing Surveys (CSUR)* 49, 3 (2016), 55.

[92] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA'14) and Software and System Performance Testing, Debugging, and Analytics (PERTEA) (WODA+PERTEA'14).* ACM, New York, NY, 1–5. DOI: https://doi.org/10.1145/2632168.2632169

[93] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the USENIX Security Symposium.*

[94] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Steven Y. Ko, and Lukasz Ziarek. 2014. Information flows as a permission mechanism. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'14).*

[95] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in Android application code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16).* ACM, New York, NY, 25–36. DOI: https://doi.org/10.1145/2884781.2884855

[96] Stephen Smalley and Robert Craig. 2013. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS'13).*

[97] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-Hunter: Large scale, automated detection of Ssl/Tls man-in-the-middle vulnerabilities in Android apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14).*

[98] Steven Artz. [n.d.]. FlowDroid 2.0. Retrieved July 7, 2019 from https://github.com/secure-software-engineering/soot-infoflow/releases.

[99] stream101. [n.d.]. Possible to Integrate Fragment Lifecycle? Retrieved July 7, 2019 from https://github.com/secure-software-engineering/soot-infoflow-android/issues/52.

[100] μSE Developers. 2019. μSE Sources and Data.Retrieved July 7, 2019 from https://muse-security-evaluation.github.io.

[101] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot—A Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research.* IBM Press, 13.

[102] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, implementation and verification of an extensible and modular hypervisor framework. In *IEEE Symposium on Security and Privacy (SP'13)*. 430–444.

[103] Veracode. 2020. Veracode's 10th State of Software Security Report Finds Organizations Reduce Rising 'Security Debt' via Devsecops, Special Sprints. Retrieved July 7, 2019 from https://www.veracode.com/veracodes-10th-state-software-security-report-finds-organizations-reduce-rising-security-debt.

[104] Timothy Vidas, Nicolas Cristin, and Lorrie Faith Cranor. 2011. Curbing Android permission creep. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP'11)*.

[105] Rubin Xu, Hassen Saidi, and Ross Anderson. 2012. Aurasium: Practical policy enforcement for Android applications. In *Proceedings of the USENIX Security Symposium*.

[106] Yuanzhong Xu and Emmett Witchel. 2015. Maxoid: Transparently confining mobile applications with custom views of state. In *Proceedings of the 10th European Conference on Computer Systems*. 26.

[107] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[108] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. 2015. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 303–313. DOI: https://doi.org/10.1109/ICSE.2015.50

[109] Chixiang Zhou and Phyllis G. Frankl. 2009. Mutation testing for Java database applications. In *Proceedings of the 2nd International Conference on Software Testing Verification and Validation, (ICST'09)*. 396–405.

[110] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.

[111] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'12)*.

[112] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. 2011. Taming information-stealing smartphone applications (on Android). In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST'11)*.

Received February 2020; revised September 2020; accepted November 2020