Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation

Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk, *William & Mary*

https://www.usenix.org/conference/usenixsecurity18/presentation/bonett

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation

Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, Denys Poshyvanyk
{*rfbonett, kkafle, kpmoran, nadkarni, denys*}*@cs.wm.edu*
William & Mary

Abstract

Mobile application security has been one of the major areas of security research in 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 are often not known or well-documented, leading to a misplaced confidence among researchers, developers, and users. This paper proposes 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 implement μSE as a semi-automated framework, and apply it to a set of prominent Android static analysis tools that detect private data leaks in apps. As the result of an in-depth analysis of one of the major tools, we discover 13 undocumented flaws. More importantly, we discover that all 13 flaws propagate to tools that inherit the flawed tool. We successfully fix one of the flaws in cooperation with the tool developers. Our results motivate the urgent need for systematic discovery and documentation of unsound choices in sound*y* tools, and demonstrate the opportunities in leveraging mutation testing in achieving this goal.

1 Introduction

Mobile devices such as smartphones and tablets have become the fabric of our consumer computing ecosystem; by the year 2020, more than 80% of the world's adult population is projected to own a smartphone [31]. 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 [59, 44, 98, 80, 35, 87, 32], 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 [34, 99, 12], discover private data leaks [33, 13, 42, 15], detect vulnerable application interfaces [38, 22, 62, 54], identify flaws in the use of cryptographic primitives [35, 32, 87], and define sandbox policies for third-party apps [47, 50]. 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 if existing security tools are robust enough 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 trade-off 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 in-

stance, techniques that analyze Java generally do not over-approximate analysis of certain programming language features, such as reflection, for practical reasons (*e.g.,* Soot [90], FlowDroid [13]). While 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 [61]. While 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 clearly 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 to 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 [61]. While our work is 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.

This paper proposes 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. μSE leverages the practice of mutation analysis from the software engineering (SE) domain [74, 45, 25, 63, 27, 78, 11, 97, 75, 28], and specifically, more recent advancements in mutating Android apps [58]. 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 μSE by evaluating static analysis research tools that detect data leaks in Android apps (*e.g.,* FlowDroid [13], IccTA [55]). We evaluate a set of *seven* tools across three experiments, and reveal 13 flaws that were undocumented. We also discover that when a tool inherits another (*i.e.,* inherits the codebase), *all the flaws propagate*. Even in cases wherein a tool only conceptually inherits another (*i.e.,* leveraging decisions from prior work), *just less than half of the flaws propagate*. We provide immediate patches that fix one flaw, and in other cases, we identify flaw classes that may need significant research effort. Thus, μSE not only helps researchers, tool designers, and analysts uncover undocumented flaws and unsound choices in sound*y* security tools, but may also provide immediate benefits by discovering easily fixable, but evasive, flaws.

This paper makes the following contributions:

- *We introduce the novel paradigm of Mutation-based Soundness Evaluation*, 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 13 unknown flaws, and validates their propagation. Our analysis leads to the documentation of unsound assumptions, and immediate security fixes in some cases.

Threat Model: 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 paper 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 [30]) that assumes the mutated program to be benign.

The rest of the paper proceeds as follows: Section 2 motivates our approach, and provides a brief background. Section 3 describes the general approach and the design goals. Section 4 and Section 5 describe the design and implementation of μSE, respectively. Section 6 evaluates the effectiveness of μSE, and Section 7 delivers the insights distilled from it. Section 8 describes related work. Section 9 describes limitations. Section 10 concludes.

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. While the soundiness manifesto [61] defines the *sound core* in terms of specific language features, 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). There is also 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 and researchers mend existing artifacts.

2.1 Motivating Example

Consider the following motivating example of a prominent static analysis tool, FlowDroid [13]:

FlowDroid [13] is a highly popular static analysis framework for detecting private data leaks in Android apps by performing a data flow analysis. Some of the limitations of FlowDroid are well-known and stated in the paper [13]; *e.g.,* FlowDroid does not support reflection, like most static analyses for Java. However, through a systematic evaluation 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, latest as of 10/10/17) does not support "Android fragments" [10], 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, see Appendix A). 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 recently released [88], claims to address fragments, *but also failed to detect our exploit*. On investigating further, we found that FlowDroid v1.5 has been extended by at least 13 research tools [55, 53, 96, 15, 73, 82, 60, 85, 8, 79, 56, 57, 71], 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 also propagated to the tools that inherit it. We confirm this conjecture for inheritors of FlowDroid that also detect data leaks, and are available in source or binary form (*i.e.,* 2 out of 13), in Section 6.

Finally, 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 [89]. Thus, we were able to discover and fix an undocumented design flaw that significantly affected FlowDroid's soundness claims, thereby expanding its sound core. However, we have confirmed that FlowDroid v2.5 [88] 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 also 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 also for the research community in general. As of today, aside from a handful of manually curated testing toolkits (*e.g.,* DroidBench [13]) with hard-coded (but useful) checks, to the best of our knowledge, there has been no prior effort at methodologically discovering problems related to sound*i*ness in Android static analysis tools and frameworks. *This paper is motivated by the need*

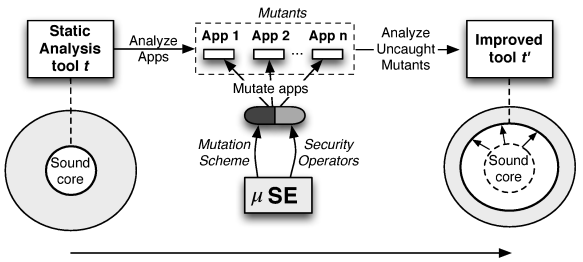

Figure 1: μSE tests a static analysis tool on a set of mutated Android apps and analyzes uncaught mutants to discover and/or fix flaws.

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 [74]. 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 [45, 25], mutation testing has seen striking advancements related to the design and development of advanced operators. Research related to development of mutation operators has traditionally attempted to adapt operators for a particular target domain, such as the web [78], data-heavy applications [11, 97, 28], or GUI-centric applications [75]. Recently, mutation analysis has been applied to measure the effectiveness of test suites for both functional and non-functional requirements of Android apps [26, 49, 58].

This paper 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 *redefines* this methodology to effectively improve security-focused static analysis tools, as we describe in Sections 4 and 8.

3 μSE

We propose 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 [74]. 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 [13] or MalloDroid [35]) 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, while 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 scheme.* Android's security challenges are notably unique, and hence require a diverse array of novel security analyses. Thus, the 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.* While 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, as well as to develop a systematic methodology for resolving undetected mutants to flaws.

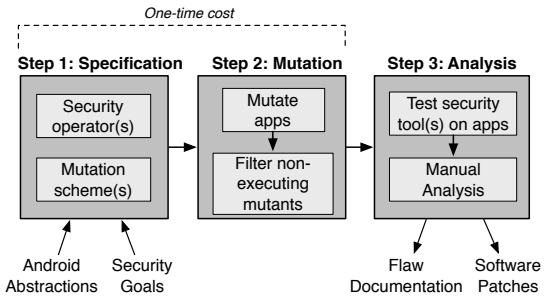

Figure 2: The components and process of the μSE.

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 to promote a wide-scale deployment and community-based use of μSE.

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 mutants, which may lead to documentation of flaws, and software patches.

Note that tools sharing a security goal (*e.g.,* FlowDroid[13], Argus [39], HornDroid [20] and BlueSeal [84] 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**). The rest of this section describes the design contributions of μSE. The precise implementation details can be found in Section 5.

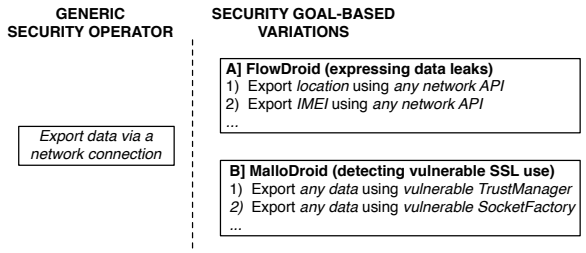

Figure 3: A generic "network export" security operator, and its more fine-grained instantiations in the context of FlowDroid [13] and MalloDroid [35].

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 [13], 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 [32]), 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, while 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**). Since 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

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

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

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 [39], HornDroid [20], BlueSeal [84]). For instance, one of our implemented operators for evaluating tools that detect data leaks is as described in Listing 1. 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, which, when combined with our expressive mutation schemes (Section 4.2), would generate a diverse set of mutants.

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. In order to precisely identify what a tool considers as a security flaw, and claims to detect, we inspected the following sources:

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.,* leaks in reflection and dynamically loaded code), 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 always 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 [13]) 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. We define the specific methods for choosing *where* to apply security operators to inject mutations within Android apps as the mutation scheme.

The mutation scheme depends on 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 (*i.e.,* similar to security operators). Note that while mutation schemes using the first two factors may be generally applied to any type of static analysis tool (*e.g.,* SSL vulnerability and malware detectors), the third factor, as the description suggests, will only apply to a specific security goal, which in the light of this paper, is data leak detection.

We describe each factor independently, as a mutation scheme, in the context of the following running example described previously in Section 2:

Recall that FlowDroid [13], 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, from which it first selects a simple source `java.util.Calendar.getTimeZone()` and a simple sink `android.util.Log.d()`. We then design a data leak operator, as shown in Listing 1. Using this security operator, we implement the following three different mutation schemes.

4.2.1 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 [13, 48].

Based on our domain knowledge of Android and its security, we choose the following features as a starting point in a mutation scheme 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 [3]. Fragments are also UI elements that possess similar callbacks, though they are often used in a manner secondary to activities. We design our mutation scheme to

```
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

place mutants within methods of fragments and activities where applicable, so as to test a tool's ability to model the activity and fragment lifecycles.

2) Callbacks: Since 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 test 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.

3) Intent messages: Android apps communicate with one another and listen for system-level events using Intents, Intent Filters, and Broadcast Receivers [2, 1]. 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 3 in Appendix B for the code).

4) XML resource files: Although Android apps are primarily written in Java, they also 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 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. 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 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 that can be specifically applied to evaluate data leak detectors. This scheme infers two ways of adding 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 predefined rules. In 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. 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.

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 cre-

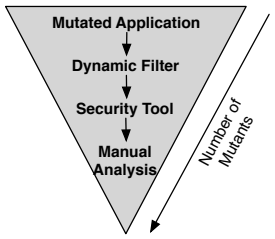

Figure 4: The number of mutants (*e.g.,* data leaks) to analyze drastically reduces at every stage in the process.

ated, 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.

4.3 Analysis Feasibility & 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, 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 recreating 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.

5 Implementation

This section provides the implementation details of μSE: (1) ME for mutating apps, and (2) EE for exercising mutants to filter out non-executing ones. We have made μSE available for use by the wider security research community [89], along with the data generated or used in our experiments (*e.g.,* operators, flaws) and code samples.

1. Mutation Engine (ME): The ME allows μSE to automatically mutate apps according to a fixed set of security operators and mutation schemes. ME is implemented in Java and builds upon the MDROID+ mutation framework for Android [58]. Firstly, 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 analysis: (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 also 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 [6], ant [4]) so that it can be dynamically analyzed by the EE.

2. Execution Engine (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. This EE builds upon prior work in automated input generation for Android apps by adapting the systematic exploration strategies from the CRASHSCOPE tool [66, 65] to explore a target app's GUI. We discuss the limitations of the EE in Section 9. For more details, please see Appendix C.

6 Evaluation

The main *goal* of our 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. For this study, we focus on tools that detect private data leaks on a device. Specifically, we focus on a set of *seven* data leak detectors for Android that use static analysis, primarily due to the availability of their source code, namely FlowDroid [13], Argus [39] (previously known as AmanDroid), DroidSafe [43], IccTA [55], BlueSeal [84], HornDroid [20], and DidFail [53]. For all the tools except FlowDroid, we use the latest release version when available; in FlowDroid's case, we used its v2.0 release for our μSE analysis, and confirmed our findings with its later releases (*i.e.,* v2.5 and v2.5.1). Additionally, we use a set of 7 open-source Android apps from F-droid [5] that we mutate. These 7 apps produced 2026 mutants to inspect, which led to the discovery of 13 flaws. A larger dataset of apps is likely to generate more mutants, and lead to more flaws.

In this section, we describe the highlights of our evaluation (Section 6.1), along with the *three* experiments we conduct, and their results. In the first experiment (Section 6.2), we run μSE on *three* tools, and record the number of leaks that each tool fails to detect (*i.e.,* the number of uncaught mutants). In the second experiment (Section 6.3), we perform an in-depth analysis of FlowDroid by applying our systematic manual analysis methodology (Section 4.3) on the output of μSE for FlowDroid. Finally, our third experiment (Section 6.4) measures the propagation and prevalence of the flaws found in FlowDroid, in tools from our dataset apart from FlowDroid, and two newer versions of FlowDroid.

These experiments are motivated by the following research questions:

RQ1 *Can μSE find security problems in static analysis tools for Android, and help resolve them to flaws/ unsound choices?*

RQ2 *Are flaws inherited when a tool is reused (or built upon) by another tool?*

RQ3 *Does the semi-automated methodology of μSE allow a feasible analysis (in terms of manual effort)?*

RQ4 *Are all flaws unearthed by μSE difficult to resolve, or can some be isolated and patched?*

RQ5 *How robust is μSE's performance?*

6.1 Evaluation overview and Highlights

We insert a total of 7,584 data leaks (*i.e.,* mutants) in a set of 7 applications using μSE. 2,026 mutants are verified as executable by the EE, and 83-1,480 are not detected depending on the studied tool. During our analysis, μSE exhibits a maximum one-cost runtime of 92 minutes (**RQ5**), apart from the time taken by the analyzed tool (*e.g.,* FlowDroid) itself. Further, our in-depth analysis of the output of μSE for FlowDroid discovers 13 unique flaws that are not documented in either the paper or the source code repository (**RQ1**). Moreover, it takes our analyst, a graduate student with background in Android security, *one* hour per flaw (in the worst case), due to our systematic analysis methodology, as well as our dynamic filter (Section 4.3), which filters out over 73 % of the seeded non-executable mutants (**RQ3**). Further, we demonstrate that two newer versions of FlowDroid, as well as the *six* other tools set apart from FlowDroid (including those that inherit it), are also vulnerable to at least *one* flaw detected in FlowDroid (**RQ2**). This is confirmed, with negligible effort, using minimal examples generated during our analysis of FlowDroid (**RQ3**). Finally, we are able to generate patches for a specific flaw discovered in FlowDroid, and our pull request has been accepted by the tool authors (**RQ4**).

6.2 Executing μSE

The objective of this experiment is to demonstrate the effectiveness of μSE in filtering out non-executable injected leaks (*i.e.,* mutants), while illustrating that this process results in a reasonable number of leaks for an analyst to manually examine.

Methodology: We create 21 mutated APKs from 7 target applications, with 7,584 leaks among them,

Table 1: The number and percentage of leaks not detected by 3 popular data leak detection tools.

Tool	Undetected Leaks	Undetected Leaks (%)
FlowDroid v2.0	987/2,026	48.7%
Argus	1,480/2,026	73.1%
DroidSafe	83/2,026	4.1%

by combining the security operators described in Section 4.1, with mutation schemes from Section 4.2. First, we measure the total number of leaks injected across all apps, and then the total number of leaks marked by the EE as non-executable. 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. The non-executable leaks are then removed. Next, we configure FlowDroid, Argus, and DroidSafe and evaluate each tool with μSE individually, by running them on the mutated apps (with non-executable leaks excluded) and recording the number of leaks not detected by each tool (*i.e.,* the *surviving* mutants).

Results: μSE injects 7,584 leaks into the Android apps, of which, 5,558 potentially non-executable leaks are filtered out using our EE, leaving only 2,026 leaks confirmed as executable in the mutated apps. By filtering out a large number of potentially non-executable leaks (*i.e.,* over 73%), our dynamic filtering significantly reduces manual effort (**RQ3**).

Table 1 shows the statistics acquired from μSE's output over FlowDroid, Argus, and DroidSafe. We observe that FlowDroid cannot detect over 48% of the leaks, while Argus cannot detect over 73%. Further, DroidSafe does not detect a non-negligible percentage of leaks (*i.e.,* over 4%), and as these leaks have been confirmed to execute by our EE, it is safe to say that DroidSafe has flaws as well. Note that this experimental result validates our conceptual argument, that security operators designed for a specific goal may apply to tools with that goal. However, given its popularity, we limit our in-depth evaluation to FlowDroid.

Finally, we measure the runtime of the μSE-specific part of the analysis, *i.e.,* up to executing the tool to be evaluated, to be a constant 92 minutes in the worst case, a majority of which (*i.e.,* 99%) is taken up by the EE. Note that the time taken by μSE is a one-time cost, and does not have to be repeated for tools with a similar security goal (**RQ5**).

6.3 FlowDroid Analysis

This experiment demonstrates an in-depth, manual analysis of FlowDroid, which we choose for two reasons: (1) impact (FlowDroid is cited by 700 papers and numerous other tools depend on it), and (2) potential for change (since FlowDroid is being maintained at the moment, any contributions we can make will have immediate benefits).

Methodology: We performed an in-depth analysis using the list of surviving mutants (*i.e.,* undetected leaks) generated by μSE for FlowDroid v2.0 in the previous experiment. We leveraged the methodology for systematic manual evaluation, described in Section 4.3, and discovered 13 unique flaws. *We confirmed that none of the discovered flaws have been documented before*; *i.e.,* in the FlowDroid paper or in their official documentation.

Results: We discovered 13 unique flaws, from FlowDroid alone, demonstrating that μSE can be effectively used to find problems that can be resolved to flaws (**RQ1**). Using the approach from Section 4.3, the analyst needed less than an 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 (**RQ3**). We give descriptions of the flaws discovered as a result of μSE's analysis in Table 2.

We have reported these flaws, and are working with the developers to resolve the issues. In fact, we developed patches to correctly implement Fragment support (*i.e.,* flaw 13 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 discovered flaws into the following *flaw classes*:

FC1: Missing Callbacks: The security tool (*e.g.,* FlowDroid) does 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 as the generated list of callbacks (1) may not be complete, and (2) or may not be updated as the Android platform evolves. We found both these cases in our analysis of FlowDroid. That is, DialogFragments was added in API 11, *i.e.,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 [21]. 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 adding the callback to the list*; *e.g.,*PhoneStateListener and SQLiteOpen-

Table 2: Descriptions of flaws uncovered in FlowDroid v2.0

Flaw	Description
FC1: Missing Callbacks	
1. DialogFragmentShow	FlowDroid misses the DialogFragment.onCreateDialog() callback registered by DialogFragment.show().
2. PhoneStateListener	FlowDroid does not recognize the onDataConnectionStateChanged() callback for classes extending the PhoneStateListener abstract class from the telephony package.
3. NavigationView	FlowDroid does not recognize the onNavigationItemSelected() callback of classes implementing the interface NavigationView.OnNavigationItemSelectedListener.
4. SQLiteOpenHelper	FlowDroid misses the onCreate() callback of classes extending android.database.sqlite.SQLiteOpenHelper.
5. 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	
6. RunOnUIThread	FlowDroid misses the path to Runnable.run() for Runnables passed into Activity.runOnUIThread().
7. ExecutorService	FlowDroid misses the path to Runnable.run() for Runnables passed into ExecutorService.submit().
FC3: Incorrect Modeling of Anonymous Classes	
8. 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.
9. 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	
10. 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.
11. 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.
12. 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.
13. 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.

Helper, 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 does 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 [90] SPARK algorithm, upon which FlowDroid depends. This limitation is also known within the program analysis community [21]. However, the documentation of this gap, thanks to μSE, would certainly help developers and researchers in the wider security community.

FC3: Incorrect Modeling of Anonymous Classes: The security tool misses data leaks expressed within an anonymous class. For example, FlowDroid does 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 does not recognize a data leak whose source and sink are called within different methods that are asynchronously executed. For instance, FlowDroid does 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 efforts, 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.

6.4 Flaw Propagation Study

The objective of this experiment is to determine if the flaws discovered in FlowDroid have propagated to the tools that inherit it, and to determine whether other static analysis tools that do not inherit FlowDroid are similarly flawed.

Methodology: We check if the two newer release versions of FlowDroid (*i.e.,* v2.5, and v2.5.1), as well as 6 other tools (*i.e.,* Argus, DroidSafe, IccTA, BlueSeal, HornDroid, and DidFail), are susceptible to any of the flaws discussed previously in FlowDroid v2.0, by using the tools to analyze the minimal example APKs generated during the in-depth analysis of FlowDroid.

Results: As seen in the Table 3, all the versions

Table 3: Flaws present in data leak detectors. Note that a "−" indicates tool crash with the minimal APK, a "✓" indicates presence of the flaw, and a "x" indicates absence, and *FD = FlowDroid.

Flaw	FD* v2.5.1	FD* v2.5	FD* v2.0	Blueseal	IccTA	HornDroid	Argus	DroidSafe	DidFail
DialogFragmentShow	✓	✓	✓	x	✓	✓	x	x	✓
PhoneStateListener	✓	✓	✓	x	✓	✓	x	x	✓
NavigationView	✓	✓	✓	-	✓	-	✓	-	✓
SQLiteOpenHelper	✓	✓	✓	x	✓	✓	✓	x	✓
Fragments	✓	✓	✓	✓	✓	✓	✓	-	✓
RunOnUIThread	✓	✓	✓	x	✓	✓	✓	x	✓
ExecutorService	✓	✓	✓	x	✓	✓	✓	x	✓
ButtonOnClickToDialogOnClick	✓	✓	✓	x	✓	x	x	✓	✓
BroadcastReceiver	✓	✓	✓	x	✓	x	x	x	✓
LocationListenerTaint	✓	✓	✓	x	✓	x	x	x	✓
NSDManager	✓	✓	✓	x	✓	x	✓	x	✓
ListViewCallbackSequential	✓	✓	✓	x	✓	x	x	x	✓
ThreadTaint	✓	✓	✓	x	✓	x	x	x	✓

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 and v2.5.1) still seem to have this flaw. We are working with the developers on a solution.

A significant observation from the Table 3 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 (**RQ2**). On the contrary, Argus, which is motivated by FlowDroid's design, but augments it on its own, does not exhibit as many flaws.

Also, BlueSeal, HornDroid, and DroidSafe use a significantly different methodology, and are also not susceptible to these flaws. 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 don't exhibit the flaws found in FlowDroid.

Finally, our analysis does not imply that FlowDroid is weaker than the tools which have fewer flaws in Table 3. 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. A similar deep exploration into the results of μSE for the other tools may be explored in the future (*e.g.,* of the 83 uncaught leaks in DroidSafe from Section 6.2).

7 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 3), 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: *Simple and security goal-specific mutation schemes are effective.* While certain mutation schemes may be Android-specific, our results demonstrate limited dependence on these configurations. Out of the 13 flaws discovered by μSE, the Android-influenced mutation scheme (Section 4.2.1) revealed one (*i.e., BroadCastReceiver* in Table 3), while the rest were evenly distributed among the other two mutation schemes; *i.e.,* the schemes that evaluate reachability (Section 4.2.2) or leverage the security goal (Section 4.2.3).

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. More importantly, even in those cases where the tool does not directly inherit the code-

base, unsound choices may still propagate at the conceptual level and result in real flaws.

Insight 4: *As tools, libraries, and the Android platform evolve, security problems become harder to track down*. Due 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 5: *Benchmarks need to evolve with time*. While manually-curated benchmarks (*e.g.,* DroidBench [13]) 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 #3) 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.*

8 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). In this section, we survey related 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, 91], information flows in apps [67, 68, 95]), which poses practical concerns for tools based on numerous legacy codebases (*e.g.,* FlowDroid [13], CHEX [62]). 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.

Mutation Analysis for Android: Deng *et al.* [26] introduced mutation analysis for Android and derived operators by analyzing the syntax of Android-specific Java constructs. Subsequently, a mutation analysis framework for Android (μDroid) has been introduced to evaluate a test suite's ability to uncover energy bugs [49]. μSE incorporates concepts from the general mutation analysis proposed by prior work (especially on Android [49, 26, 58]), but adapts them in the context of security. We design mSE to focus on undetected mutants, providing a semi-automated methodology to resolve such mutants to design/implementation flaws (Section 4.3). The derivation of security operators (Section 4.1) represents a notable departure from traditional mutation testing that seeds simple syntactic code changes. Our mutation schemes (Section 4.2) evaluate coverage of OS-specific abstractions, reachability of the analysis, or the ability to detect semantically-complex mutants, providing the expressibility necessary for security testing, while building upon traditional approaches. Further, μSE builds upon the software infrastructure developed for MDROID+ [58] that allows a scalable analysis of mutants seeded according to security operators. In particular, μSE adapts the process of deriving a potential fault profile for mutant injection and relies on the EE to validate the mutants seeded according to our derived security operators.

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 [9], 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 appli-

cations [33, 13, 42, 70, 69, 94, 52] to detecting overprivilege in applications [37, 14, 92]. 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 [35, 32, 87, 36] or unprotected application interfaces [38, 22, 54]. Moreover, these techniques have often been deployed as modifications to Android's permission enforcement [34, 33, 72, 40, 29, 38, 18, 76, 23, 100, 86, 19, 46, 77, 83, 81], SDK tools [37, 14, 92], or inline reference monitors [93, 51, 24, 17, 16]. While this paper 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, μSE could be applied to further vet SSL analysis tools by purposely introducing complex SSL errors in real applications, or tools that analyze overprivilege or permission misuse, by developing security operators that attempt to misuse permissions to circumvent such monitors. Future work may use μSE to perform an in-depth analysis of these problems.

9 Limitations

1) Soundness of μSE: As acknowledged in Section 8, mSE does not aim to supplant formal verification (which would be sound), and does not claim soundness guarantees. Rather, mSE 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 6.2, 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's EE is based upon, has been shown to perform comparably to other tools in terms of coverage [66]. Future versions of μSE's EE could rely on emerging input generation tools for Android apps [64].

10 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 a popular, open-source Android security tool, but also demonstrated how these flaws propagated to other tools that inherited the security tool in question. 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.

11 Acknowledgements

We thank Rozda Askari for his help with setting up experiments. We thank the FlowDroid developers, as well as the developers of the other tools we evaluate in this paper, for making their tools available to the community, providing us with the necessary information for our analysis, and being open to suggestions and improvements. The authors have been supported in part by the NSF-1815336, NSF-714581 and NSF-714161 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Android developer documentation - broadcasts `https://developer.android.com/guide/components/broadcasts.html`.

[2] Android developer documentation - intents and intent filters `https://developer.android.com/guide/components/intents-filters.html`.

[3] Android developer documentation - the activity lifecycle `https://developer.android.com/guide/components/activities/activity-lifecycle.html`.

[4] Apache ant build system - `http://ant.apache.org`.

[5] F-droid.`https://f-droid.org/`.

[6] Gradle build system - `https://gradle.org`.

[7] Soot java instrumentation framework `http://sable.github.io/soot/`.

[8] AAFER, Y., ZHANG, N., ZHANG, Z., ZHANG, X., CHEN, K., WANG, X., ZHOU, X., DU, W., AND GRACE, M. 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* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1248–1259.

[9] ACAR, Y., BACKES, M., BUGIEL, S., FAHL, S., MCDANIEL, P., AND SMITH, M. Sok: Lessons learned from android security research for appified software platforms. In *37th IEEE Symposium on Security and Privacy (S&P '16)* (2016), IEEE.

[10] ANDROID DEVELOPERS. Fragments. https://developer.android.com/guide/components/fragments.html.

[11] APPELT, D., NGUYEN, C. D., BRIAND, L. C., AND ALSHAHWAN, N. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014* (2014), pp. 259–269.

[12] ARP, D., SPREITZENBARTH, M., HÜBNER, M., GASCON, H., AND RIECK, K. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2014).

[13] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. 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)* (2014).

[14] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 217–228.

[15] AVDIIENKO, V., KUZNETSOV, K., GORLA, A., ZELLER, A., ARZT, S., RASTHOFER, S., AND BODDEN, E. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (May 2015), pp. 426–436.

[16] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., AND VON STYP-REKOWSKY, P. Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium (USENIX Security 15)* (Aug. 2015).

[17] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., AND VON STYP-REKOWSKY, P. 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)* (2013).

[18] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Toward Taming Privilege-Escalation Attacks on Android. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)* (2012).

[19] BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHASTRY, B. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)* (2011).

[20] CALZAVARA, S., GRISHCHENKO, I., AND MAFFEI, M. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)* (March 2016), pp. 47–62.

[21] CAO, Y., FRATANTONIO, Y., BIANCHI, A., EGELE, M., KRUEGEL, C., VIGNA, G., AND CHEN, Y. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2015).

[22] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2011).

[23] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings of the 13th Information Security Conference (ISC)* (Oct. 2010).

[24] DAVIS, B., SANDERS, B., KHODAVERDIAN, A., AND CHEN, H. I-arm-droid: A rewriting framework for in-app reference monitors for android applications.

[25] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (April 1978), 34–41.

[26] DENG, L., MIRZAEI, N., AMMANN, P., AND OFFUTT, J. Towards mutation analysis of android apps. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on* (April 2015), pp. 1–10.

[27] DEREZIŃSKA, A., AND HAŁAS, K. *Analysis of Mutation Operators for the Python Language*. Springer International Publishing, Cham, 2014, pp. 155–164.

[28] DI NARDO, D., PASTORE, F., AND BRIAND, L. C. Generating complex and faulty test data through model-based mutation analysis. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015* (2015), pp. 1–10.

[29] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium* (Aug. 2011).

[30] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)* (may 2016).

[31] ECONOMIST, T. Planet of the phones. http://www.economist.com/news/leaders/21645180-smartphone-ubiquitous-addictive-and-transformative-planet-phones, Feb. 2015.

[32] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013).

[33] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. 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)* (Oct. 2010).

[34] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2009).

[35] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. 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* (2012).

[36] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL Development in an Appified World. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 49–60.

[37] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[38] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (Aug. 2011).

[39] FENGGUO WEI, SANKARDAS ROY, X. O., AND ROBBY. 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)* (Nov. 2014).

[40] FRAGKAKI, E., BAUER, L., JIA, L., AND SWASEY, D. Modeling and enhancing android's permission system. In *Computer Security – ESORICS 2012* (Berlin, Heidelberg, 2012), S. Foresti, M. Yung, and F. Martinelli, Eds., Springer Berlin Heidelberg, pp. 1–18.

[41] FRANKLIN, J., CHAKI, S., DATTA, A., AND SESHADRI, A. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), pp. 365–379.

[42] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. 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)* (June 2012).

[43] GORDON, M. I., KIM, D., PERKINS, J., GILHAM, L., NGUYEN, N., AND RINARD, M. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2015).

[44] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys)* (2012).

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

[46] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the USENIX Security Symposium* (Aug. 2014).

[47] HO, T.-H., DEAN, D., GU, X., AND ENCK, W. PREC: Practical Root Exploit Containment for Android Devices. In *Proceedings of the Fourth ACM Conference on Data and Application Security and Privacy (CODASPY)* (Mar. 2014).

[48] HOLAVANALLI, S., MANUEL, D., NANJUNDASWAMY, V., ROSENBERG, B., SHEN, F., KO, S. Y., AND ZIAREK, L. Flow permissions for android. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2013), pp. 652–657.

[49] JABBARVAND, R., AND MALEK, S. μdroid: An energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 208–219.

[50] JAMROZIK, K., VON STYP-REKOWSKY, P., AND ZELLER, A. Mining sandboxes. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on* (May 2016), pp. 37–48.

[51] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGEL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. 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)* (2012).

[52] JIA, L., ALJURAIDAN, J., FRAGKAKI, E., BAUER, L., STROUCKEN, M., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract). In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (Sept. 2013).

[53] KLIEBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. 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* (2014), pp. 1–6.

[54] LEE, Y. K., BANG, J. Y., SAFI, G., SHAHBAZIAN, A., ZHAO, Y., AND MEDVIDOVIC, N. A sealant for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering* (May 2017), pp. 312–323.

[55] LI, L., BARTEL, A., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), pp. 280–291.

[56] LI, L., BARTEL, A., KLEIN, J., AND TRAON, Y. L. Automatically exploiting potential component leaks in android applications. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications* (Sept 2014), pp. 388–397.

[57] LILLACK, M., KASTNER, C., AND BODDEN, E. Tracking load-time configuration options. *IEEE Transactions on Software Engineering PP*, 99 (2017), 1–1.

[58] LINARES-VÁSQUEZ, M., BAVOTA, G., TUFANO, M., MORAN, K., DI PENTA, M., VENDOME, C., BERNAL-CÁRDENAS, C., AND POSHYVANYK, D. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 233–244.

[59] LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (2014).

[60] LIU, B., LIU, B., JIN, H., AND GOVINDAN, R. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2015), MobiSys '15, ACM, pp. 89–103.

[61] LIVSHITS, B., SRIDHARAN, M., SMARAGDAKIS, Y., LHOTÁK, O., AMARAL, J. N., CHANG, B.-Y. E., GUYER, S. Z., KHEDKER, U. P., MØLLER, A., AND VARDOULAKIS, D. In defense of soundiness: A manifesto. *Commun. ACM 58*, 2 (Jan. 2015).

[62] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012), pp. 229–240.

[63] MA, Y., KWON, Y. R., AND OFFUTT, J. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering (ISSRE 2002), 12-15 November 2002, Annapolis, MD, USA* (2002), pp. 352–366.

[64] MAO, K., HARMAN, M., AND JIA, Y. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, ACM, pp. 94–105.

[65] MORAN, K., LINARES-VASQUEZ, M., BERNAL-CARDENAS, C., VENDOME, C., AND POSHYVANYK, D. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (May 2017), pp. 15–18.

[66] MORAN, K., VÁSQUEZ, M. L., BERNAL-CÁRDENAS, C., VENDOME, C., AND POSHYVANYK, D. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016* (2016), pp. 33–44.

[67] MYERS, A. C. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL)* (January 1999).

[68] MYERS, A. C., AND LISKOV, B. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (October 2000), 410–442.

[69] NADKARNI, A., ANDOW, B., ENCK, W., AND JHA, S. Practical DIFC Enforcement on Android. In *Proceedings of the 25th USENIX Security Symposium* (Aug. 2016).

[70] NADKARNI, A., AND ENCK, W. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (Nov. 2013).

[71] NAN, Y., YANG, M., YANG, Z., ZHOU, S., GU, G., AND WANG, X. Uipicker: User-input privacy identification in mobile applications. In *USENIX Security Symposium* (2015), pp. 993–1008.

[72] NAUMAN, M., KHAN, S., AND ZHANG, X. 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)* (2010).

[73] OCTEAU, D., LUCHAUP, D., DERING, M., JHA, S., AND MCDANIEL, P. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 77–88.

[74] OFFUTT, A. J., AND UNTCH, R. H. *Mutation 2000: Uniting the Orthogonal*. Springer US, Boston, MA, 2001, pp. 34–44.

[75] OLIVEIRA, R. A. P., ALÉGROTH, E., GAO, Z., AND MEMON, A. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *International Conference on Software Testing, Verification, and Validation - Workshops, ICSTW'15* (2015), pp. 1–10.

[76] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)* (Dec. 2009), pp. 340–349.

[77] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2012).

[78] PRAPHAMONTRIPONG, U., OFFUTT, J., DENG, L., AND GU, J. An experimental evaluation of web mutation operators. In *International Conference on Software Testing, Verification, and Validation, ICSTW'16* (2016), pp. 102–111.

[79] RASTHOFER, S., ARZT, S., LOVAT, E., AND BODDEN, E. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *2014 Ninth International Conference on Availability, Reliability and Security* (Sept 2014), pp. 40–49.

[80] RASTOGI, V., CHEN, Y., AND ENCK, W. AppsPlayground: Automatic Large-scale Dynamic Analysis of Android Applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)* (Feb. 2013).

[81] ROESNER, F., AND KOHNO, T. Securing embedded user interfaces: Android and beyond. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (2013).

[82] SASNAUSKAS, R., AND REGEHR, J. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)* (New York, NY, USA, 2014), WODA+PERTEA 2014, ACM, pp. 1–5.

[83] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the USENIX Security Symposium* (2012).

[84] SHEN, F., VISHNUBHOTLA, N., TODARKA, C., ARORA, M., DHANDAPANI, B., KO, S. Y., AND ZIAREK, L. Information Flows as a Permission Mechanism. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2014).

[85] SLAVIN, R., WANG, X., HOSSEINI, M. B., HESTER, J., KRISHNAN, R., BHATIA, J., BREAUX, T. D., AND NIU, J. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 25–36.

[86] SMALLEY, S., AND CRAIG, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (2013).

[87] SOUNTHIRARAJ, D., SAHS, J., LIN, Z., KHAN, L., AND GREENWOOD, G. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2014).

[88] STEVEN ARTZ. FlowDroid 2.0. `https://github.com/secure-software-engineering/soot-infoflow/releases`.

[89] µSE DEVELOPERS. µSE sources and data. `https://muse-security-evaluation.github.io`.

[90] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), IBM Press, p. 13.

[91] VASUDEVAN, A., CHAKI, S., JIA, L., MCCUNE, J., NEWSOME, J., AND DATTA, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), pp. 430–444.

[92] VIDAS, T., CRISTIN, N., AND CRANOR, L. F. Curbing Android Permission Creep. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)* (2011).

[93] XU, R., SAIDI, H., AND ANDERSON, R. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the USENIX Security Symposium* (2012).

[94] XU, Y., AND WITCHEL, E. Maxoid: transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems* (Apr. 2015), p. 26.

[95] YANG, J., YESSENOV, K., AND SOLAR-LEZAMA, A. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2012).

[96] YANG, W., XIAO, X., ANDOW, B., LI, S., XIE, T., AND ENCK, W. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (May 2015), vol. 1, pp. 303–313.

[97] ZHOU, C., AND FRANKL, P. G. Mutation testing for java database applications. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009* (2009), pp. 396–405.

[98] ZHOU, Y., AND JIANG, X. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2012).

[99] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. 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)* (Feb. 2012).

[100] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. W. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)* (June 2011).

A Fragment Use Study

We performed a small-scale app study using the Soot [7] static analysis library to deduce how commonly fragments were used in real apps. That is, we analyzed 240 top apps from every category on Google Play (*i.e.*, a total of 8,664 apps collected as of June 2017 after removing duplicates), and observed that at least 4,273 apps (49.3%) used fragments in their main application code, while an additional 3,587 (41.4%) used fragments in packaged libraries. Note that while we did not execute the apps to determine if the fragment code was really executed, the fact that 7,860 out of 8,664 top apps, or 91% of popular apps contain fragment code indicates the possibility that fragments are widely used, and that accidental or malicious data leaks in a large number of apps could evade FlowDroid due to this flaw.

```
1  BroadcastReceiver receiver = new BroadcastReceiver()
   {
2      @Override
3      public void onReceive(Context context, Intent
        intent) {
4      BroadcastReceiver receiver = new
        BroadcastReceiver(){
5          @Override
6              public void onReceive(Context context,
                Intent intent) {
7                  String dataLeak = Calendar.
                    getInstance().getTimeZone().
                    getDisplayName();
8                  Log.d("leak-1", dataLeak);
9              }
10     };
11     IntentFilter filter = new IntentFilter();
12     filter.addAction("android.intent.action.SEND");
13     registerReceiver(receiver, filter);
14 }};
15 IntentFilter filter = new IntentFilter();
16 filter.addAction("android.intent.action.SEND");
17 registerReceiver(receiver, filter);
```

Listing 3: Dynamically created Broadcast Receiver, created inside another, with data leak.

B Code Snippets

In Listing 3, we dynamically register a broadcast receiver inside another dynamically registered broadcast receiver, and add the mutant (*i.e.*, a data leak in this case) inside the onReceive() callback of the inner broadcast receiver.

C CrashScope (Execution Engine)

The EE functions by statically analyzing the code of a target app to identify activities implementing potential contextual features (*e.g.*, rotation, sensor usage) via API call-chain propagation. It then executes an app according to one of several exploration strategies while constructing a dynamic event-flow model of an app in an online fashion. These strategies are organized along three dimensions: (i) GUI-exploration, (ii) text-entry, and (iii) contextual features. The Execution Engine uses a Depth-First Search (DFS) heuristic to systematically explore the GUI, either starting from the top of the screen down, or from the bottom of the screen up. It is also able to dynamically infer the allowable text characters from the Android software keyboard and enter expected text or no text. Finally, the EE can exercise contextual features (*e.g.*, rotation, simulating GPS coordinates). Since 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 (*e.g.*, entering unexpected text or disabling contextual features) prone to eliciting crashes from apps. The approach utilizes `adb` and Android's `uiautomator` framework to interact with and extract GUI-related information from a target device or emulator.