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

Friday, Aug 17th, 2018

Richie Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni & Denys Poshyvanyk
Motivation
MOTIVATION
• Security tools have diverse security goals

• Security analysis of apps is highly beneficial to end users

• Keeps the ecosystem clear of malicious or vulnerable apps
Security Analysis of Applications

• Security tools have diverse security goals

Q: Do we really know how well these tools work?

• Keeps the ecosystem clear of malicious or vulnerable apps

Security tools have diverse security goals, beneficial to end users, and can keep the ecosystem clear of malicious or vulnerable apps. However, we need to know how well these tools work.
• 2015: *Soundiness* manifesto\(^1\)

• Static analysis tools are implicitly expected to be sound (i.e., they over-approximate)

• In practice, all tools are *soundy*: A sound core, but with some unsound assumptions to be practical; e.g. JNI, Reflection

• Soundy tools are practical

• However, developers might not document unsound choices for various reasons

---

• 2015: *Soundiness* manifesto¹

• Static analysis tools are implicitly expected to be sound (i.e., they over-approximate)

• In practice, all tools are **soundy**: A sound core, but with some unsound assumptions to be practical; e.g. JNI, Reflection

• Soundy tools are practical

• *However*, developers might not document unsound choices for various reasons

• **We want to discover the extent of the unsound decisions**

---

• The scope of the soundness manifesto is language features

• We target security analysis of mobile apps (e.g., data leak detection, SSL vuln, etc.)

• This paper: A general discussion on the design/implementation choices in the context of the target platform, i.e., Android, and its unique abstractions:
  • Application model
  • Inter-component communication
  • Asynchronous invocation and component lifecycles
A framework that enables systematic evaluation of existing security tools to identify and document unsound decisions, eventually expanding the sound core

• Benefits:

• Researchers: *discover* undocumented flaws in tools

• Developers: *build* more effective tools by discovering easily fixed but evasive bugs

• Users: *benefit* from better detection, and hence a better application ecosystem

---

**OUR VISION**
µSE: Mutation-based Soundness Evaluation

• µSE leverages *mutation analysis* for systematic evaluation of security tools

• Contextualizes mutation analysis to security

• Develops the abstractions of
  
  1. Security operators and
  
  2. Mutation schemes
**Mutation analysis Background**

Mutation operators → Mutation Engine → Mutated Apks → Test Suite to evaluate → Killed Mutants, Unkilled Mutants
μSE Overview

• μSE leverages *mutation analysis* for systematic evaluation of security tools
μSE OVERVIEW

• μSE leverages *mutation analysis* for systematic evaluation of security tools

![Diagram](image)
µSE Overview

• µSE leverages mutation analysis for systematic evaluation of security tools
• \( \mu\)SE leverages \textit{mutation analysis} for systematic evaluation of security tools

\[ \mu\SE \overset{\text{Overview}}{=} \]

- Analyze Apps
  - App 1
  - App 2
  - \ldots
  - App \( n \)

Mutate apps

\( \mu\SE \)

- Sound core

Mutants

- Analyze Uncaught Mutants

Improved tool \( t' \)

- Sound core

\textbf{Mutation Scheme}

\textbf{Security Operators}
\begin{itemize}
  \item Basic Components and their definitions:
    \begin{itemize}
      \item Security operator: What anomaly/mutation to insert in the app
      \item Mutation scheme: Where to place/seed it
    \end{itemize}
\end{itemize}
**μSE Design: Security Operators**

- Challenges:
  - Too fine-grained —> Not scalable
  - Too generic —> Not effective as different tools have different security focus

- μSE defines security operator in terms of *security goals* of the tools
  - Scalable to tools with similar security goals (e.g., data leak detection)
μSE Design: Security Operators

1. Operator for data leak detectors

\[
\text{dataLeak} = \text{Location.read()}
\]

\[
\text{log.d(dataLeak)}
\]

2. Operator for SSL vulnerability detectors

\[
\text{boolean isServerTrusted()} \{
\text{return true }
\}
\]
• Multiple strategies with different objectives

  1. Reachability analysis
  2. Android Abstractions
  3. Security goals
1. Reachability analysis

• Placing operator at the start of every method
  • Helps in the evaluation of the coverage of flaws
  • Simplest mutation scheme for operator placement
2. Android abstractions

• Model unique aspects of Android platform

• Mutants are built specifically for Android by choosing its unique abstractions as the starting point

  • Activity & Fragment Lifecycles

  • Callbacks

  • Intent messages

  • Android Resource files
2. Android abstractions

- Model unique aspects of Android platform
- Mutants are built *specifically for Android* by choosing its unique abstractions as the starting point
  - Activity & Fragment Lifecycles
  - Callbacks
  - Intent messages
  - Android Resource files
2. Android abstractions

- Model unique aspects of Android platform
- Mutants are built *specifically for Android* by choosing its unique abstractions as the starting point
  - Activity & Fragment Lifecycles
  - Callbacks
  - Intent messages
  - Android Resource files
3. Security goal

• Accounting for the specific objective of the tool under scrutiny

• Can be applied to other tools with similar goals

• E.g., a taint-based scheme for data leak detection tools
3. Security goal

- Accounting for the specific objective of the tool under scrutiny
- Can be applied to other tools with similar goals
- E.g., a taint-based scheme for data leak detection tools
IMPLEMENTATION

Step 1: Specification

- Security operator(s)
- Mutation scheme(s)

Android Abstractions

Security Goals
IMPLEMENTATION

Step 1: Specification
- Android Abstractions
- Security Goals

Security operator(s)
Mutation scheme(s)

Step 2: Mutation
- Mutation Engine
IMPLEMENTATION

Step 1: Specification
- Security operator(s)
- Mutation scheme(s)
- Android Abstractions
- Security Goals

Step 2: Mutation
- Mutation Engine

Step 3: Analysis
- Test tool(s) on mutants
- Manual analysis
- Vulnerability Documentation
- Software Patches
IMPLEMENTATION

Step 1: Specification
- Security operator(s)
- Mutation scheme(s)
  - Android Abstractions
  - Security Goals

Step 2: Mutation
- Mutation Engine
- Execution Engine
  - Non-executing mutants

Step 3: Analysis
- Test tool(s) on mutants
- Manual analysis
- Vulnerability Documentation
- Software Patches
IMPLEMENTATION

Step 1: Specification
- Security operator(s)
- Security Goals
- Android Abstractions

Step 2: Mutation
- Mutation Engine
- Execution Engine
- Non-executing mutants

Step 3: Analysis
- Test tool(s) on mutants
- Manual analysis
- Vulnerability Documentation
- Software Patches

Generated Mutants
- Security Tool
- Manual Analysis

Before: 7584
After: 2026
• We evaluate the effectiveness of μSE using a case study

• Security goal we chose for our case study: Data leak detection
EVALUATION: TESTING DATA LEAK DETECTORS

- 7,584 mutants in total, 2,026 verified as executable
- 3 data leak detection tools evaluated using 2,026 mutants

<table>
<thead>
<tr>
<th>TOOLS</th>
<th>UNDETECTED LEAKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowdroid 2.0</td>
<td>987/2026 (48.7%)</td>
</tr>
<tr>
<td>Argus-SAF</td>
<td>1480/2026 (73.1%)</td>
</tr>
<tr>
<td>DroidSafe</td>
<td>83/2026 (4.1%)</td>
</tr>
</tbody>
</table>
**EVALUATION: TESTING DATA LEAK DETECTORS**

- 7,584 mutants in total, 2,026 verified as executable
- 3 data leak detection tools evaluated using 2,026 mutants

<table>
<thead>
<tr>
<th>TOOLS</th>
<th>UNDETECTED LEAKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowdroid 2.0</td>
<td>987/2026 (48.7%)</td>
</tr>
<tr>
<td>Argus-SAF</td>
<td>1480/2026 (73.1%)</td>
</tr>
<tr>
<td>DroidSafe</td>
<td>83/2026 (4.1%)</td>
</tr>
</tbody>
</table>

- **Impact:** Cited over 900 times
- **Immediate response and benefits:** Flowdroid is actively being maintained
# Evaluation: Flaws Discovered

<table>
<thead>
<tr>
<th>Vulnerability Class (VC)</th>
<th>Example Flaw in VC</th>
<th>Description of the Flaw</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Missing Callbacks (5 flaws)</td>
<td>Fragments</td>
<td>Doesn't model Android Fragments correctly.</td>
</tr>
<tr>
<td><strong>2</strong> Missing Implicit Calls (2 flaws)</td>
<td>RunOnUiThread</td>
<td>Misses a path to Runnable.run() for runnables passed into Activity.runOnUiThread()</td>
</tr>
<tr>
<td><strong>3</strong> Incorrect Modeling of Anonymous Classes (2 flaws)</td>
<td>BroadcastReceiver</td>
<td>Misses the onReceive() callback of a BroadcastReceiver implemented programmatically and registered within another programmatically defined BroadcastReceiver's onReceive() callback.</td>
</tr>
<tr>
<td><strong>4</strong> Incorrect Modeling of Asynchronous Methods (4 flaws)</td>
<td>LocationListenerTaint</td>
<td>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.</td>
</tr>
</tbody>
</table>
# Evaluation: Flaw Propagation

<table>
<thead>
<tr>
<th>Flaw</th>
<th>FD 2.5.1</th>
<th>FD 2.5.0</th>
<th>FD 2.0</th>
<th>BlueSeal</th>
<th>IccTA</th>
<th>HornDroid</th>
<th>Argus</th>
<th>DroidSafe</th>
<th>DidFail</th>
</tr>
</thead>
<tbody>
<tr>
<td>DialogFragmentShow</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>PhoneStateListener</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>NavigationView</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>-</td>
<td>✘</td>
<td>-</td>
<td>✘</td>
<td>-</td>
<td>✘</td>
</tr>
<tr>
<td>SQLiteOpenHelper</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Fragments</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>RunOnUiThread</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>ExecutorService</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>ButtonOnClickToDialogOnClick</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>BroadcastReceiver</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>LocationListenerTaint</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>NSDManager</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>ListViewCallbackSequential</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>ThreadTaint</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
</tr>
</tbody>
</table>

- **Inheriting flowdroid as a black box** - IccTA (13/13), DidFail (13/13)
- **Motivated by flowdroid’s design** (but augmented to their need) - Argus-SAF (6/13)
- **Implementing their own methodologies** - BlueSeal (1/13), HornDroid (6/13), DroidSafe (1/13)

✘ - Fails to detect
We could fix one of the problems (fragment, FlowDroid 2.0)

However, fixing flaws is significantly challenging

Some flaws are design-choices that are hard to immediately fix (e.g. Runnable)

Some are unsolved research challenges (e.g., BroadcastReceiver)

μSE effectively serves the function of discovering/documenting these for future research
Caveats

• μSE doesn’t claim soundness

• Aims to increase the confidence in the results of sound tools by discovering and documenting unsound choices

• μSE doesn’t replace formal verification

• Rather a framework for systematically uncovering flaws in security tools

• Significant advancement over manually curated toolkits
CONCLUDING REMARKS

• μSE demonstrates the effectiveness of mutation analysis at discovering undocumented flaws in security tools

• Flaws not only affect individual tools, but propagate to future research

• Android evolves, and μSE is a significant improvement over manually curated benchmarks that need keep up with Android’s fast-paced evolution

• μSE allows patching of easily fixable but evasive flaws; however, this is a hard problem in general
Thank you!

Code and data at: https://muse-security-evaluation.github.io/

Kaushal Kafle
William & Mary
kkafle@cs.wm.edu