Replay – Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps

David Schmidt
University of Vienna
Faculty of Computer Science
Christian Doppler Laboratory AsTra
Doctoral School Computer Science
Vienna, Austria
d.schmidt@unvie.ac.at

Sebastian Schrittwieser
University of Vienna
Faculty of Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
sebastian.schrittwieser@univie.ac.at

Abstract

In this work, we replay the paper *Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps* [6]. The paper presents a static analysis that uncovers hidden behavior in Android apps through input validation logic. While the original analysis, published in 2020, examined 150,000 apps, we re-execute their pipeline on 10,331 apps collected in 2023 and 2024, to study how hidden behavior in Android apps has evolved over the past five years. Overall, we observe a decline in the prevalence of hidden behavior. Nevertheless, we also identify backdoors similar to those originally reported, indicating that such techniques remain present.

ACM Reference Format:
David Schmidt and Sebastian Schrittwieser. 2025. Replay – Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *Proceedings of the 2025 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks (CheckMATE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3733817.3765609

1 Introduction

We conduct a replay study of InputScope [6], a static analysis designed to automatically uncover hidden behavior in Android apps by analyzing input validation logic. The original analysis, was performed before 2020 as the paper was published in May 2020 and examined a dataset of 150,000 Android apps. We re-executed their analysis pipeline on a dataset of 10,331 Android apps collected in 2023 and 2024. This replay allows us to answer the question: *How has hidden behavior in Android apps evolved over the past five years?*

InputScope. InputScope operates in two static analysis phases. In the first phase, it applies taint analysis to identify locations where Android apps perform string equality checks involving user input. In this context, user inputs are the taint sources, and string comparison APIs are the sinks.

In the second phase, InputScope executes a Value Set Analysis (VSA). The analysis begins with a backward trace from the string comparison with the user-input. This step resolves operations such as string concatenation, which may require recursively tracing multiple string variables. Once all dependencies are resolved, a forward simulation reconstructs the string's value along the execution trace.

Finally, the analysis classifies the reasons behind the discovered comparisons into one of four categories: (1) *Secret access key*, a comparison with a hardcoded value that controls conditional functionality, such as triggering a password reset; (2) *Master password*, a comparison involving a hardcoded string that unlocks functionality universally, while other, regular paths to access remain; (3) *Blacklist secret*, an input compared against a list of strings to suppress certain behaviors, e.g., filtering offensive vocabulary; (4) *Secret command*, a set of hardcoded comparisons used to trigger internal features, such as debug menus or hidden configuration options.

2 Methodology

To replay the original analysis, we used the publicly available implementation of InputScope [6]. The code includes the core components required to detect string comparisons involving user input and reconstructing the corresponding string values.

Dataset. We based our evaluation on Android apps collected during a previous study [3, 4]. The dataset comprises 10,331 apps downloaded from the Google Play Store in 2023. We re-downloaded the same set in October 2024 and successfully retrieved 8,702 apps. We refer to these two datasets as *2023* and *2024*, respectively.

While the original study analyzed 150,000 apps from Google Play and alternative marketplaces, we focused exclusively on the Google Play Store, the most widely used distribution channel for Android apps, and analyzed a subset of both popular and randomly selected apps to provide a holistic view of the apps available.

Environment. We executed our analysis on a Debian 12 virtual machine equipped with 64 GB RAM and a 32-core Intel(R) Xeon(R) 6230 CPU. To improve throughput, we parallelized the execution using `GNU parallel` [2], running four concurrent analyses.

3 Evaluation

In this section, we discuss challenges, present our findings, and compare them to the results reported in the original work.

3.1 Reproducibility Issues

Although the GitHub repository provides the core components of InputScope, including taint analysis and VSA, it omits the implementation of heuristics used to classify hidden behaviors. We re-implemented the classification logic in Python based on

CheckMATE '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1906-6/2025/10
https://doi.org/10.1145/3733817.3765609

Table 1: Results of our replay study. The row *Equality checks* refers to the total number of apps in each dataset. All other values represent the proportion of findings relative to the number of detected equality checks.

	InputScope	2023	2024
# Apps	150,000	10,331	8,702
# Equ. Checks	114,797 (76.53%)	2,702 (33.80%)	2,844 (32.68%)
# Backdoors	12,706 (11.07%)	315 (11.66%)	217 (7.63%)
# Access Keys	7,584 (6.61%)	237 (8.77%)	166 (5.84%)
# Master Pwds	501 (0.44%)	66 (2.44%)	32 (1.13%)
# Priv. Cmds	6,013 (5.24%)	43 (1.59%)	32 (1.13%)
# Blacklists	4,601 (4.01%)	116 (4.29%)	65 (2.29%)

descriptions provided in the original paper. However, we cannot guarantee that we implemented it identically. We integrated Gradle build scripts into the Java analysis to improve maintainability and simplify the build process. This modification holds advantages due to annual updates to the Android ecosystem, which also result in updates of FlowDroid [1] and Soot [5]. Two component on which InputScope depends.

We provide an updated implementation using FlowDroid version 2.14.1 (released October 2024). The updated code, along with our re-implementation of the classification heuristics, is available at: https://github.com/CDL-AsTra/replay-inputscope.

Based on the publication timeline of the original study (May 2020), we infer that their dataset was collected before 2020.

4 Results

We present an overview of the original results and our replay results on the *2023* and *2024* datasets in Table 1. Compared to the original study, which detected equality checks in 76.53% of analyzed apps, our detection rates were notably lower, 33.8% in 2023 and 32.68% in 2024. As discussed in Section 3.1, changes in recent Android versions and the evolving app ecosystem may contribute to this reduced detection rate.

Despite this gap, the relative proportions of detected behaviors remain comparable. For access keys, we observed similar results: 6.61% in the original study, 8.77% in 2023, and 5.84% in 2024. For master passwords, our analysis flagged more apps than the original study, 2.44% in 2023 and 1.13% in 2024, compared to 0.44%. In contrast, we identified fewer privileged command behaviors: 1.59% in 2023 and 1.13% in 2024, versus 5.24% originally.

A comparison between the *2023* and *2024* datasets reveals further insights. Focusing on apps for which the analysis completed successfully in both years, we identified 14 master passwords, 20 access keys, and 3 secret commands that appeared only in the 2023 versions. This suggests developers may have removed or refactored hidden logic over time. In contrast, only a few new hidden behaviors emerged in 2024. Specifically, 3 master passwords and 2 access keys, indicating that such practices are not being introduced at scale.

4.1 Case-studies

Similar to the original paper, we manually analyze the findings of the most popular apps for further insights.

Blacklists. Among the 20 most popular apps containing blacklist behavior, we identified 13 cases filtering special characters such as < or >, one instance filtering domain names, one filtering offensive English words, and one applying a phone number prefix filter. In three cases, the analysis failed to reconstruct the comparison values, as they were dynamically loaded from shared preferences. Additionally, one instance was misclassified as a blacklist.

Backdoors. Of the 20 most popular apps flagged for access keys, master passwords, or secret commands, we confirmed only ten as true positives. Most false positives originated from benign input validation logic. However, consistent with the original paper, we identified a total number of 31 cases where backdoors enable developer or debugging features, disable advertisements, unlock paid services, or act as master passwords. Since most backdoors in the original paper were censored, direct comparison of the discovered commands is challenging. However, the authors disclosed four cases without censorship, mentioning that those apps had received patches. Notably, we rediscovered the same secret commands in `com.th.ringtone.maker`. Further, four out of five access keys used to bypass service payments were censored with the pattern `q***d`. We identified a similar payment bypass with a matching pattern in a dictionary app with over one million installations. This suggests that the same access key pattern is still in use.

5 Conclusion

We discovered fewer hidden functionalities compared to the original study. One reason is the reduced number of detected equality checks, which may result from changes in Android that would require updates to the static analysis. However, even within our own datasets, we observed a slightly positive trend: fewer hidden behaviors were discovered in the 2024 version compared to 2023. Nevertheless, we identified backdoors in several popular apps that closely resemble those reported in the original paper, indicating that such patterns are still in use.

Acknowledgments

The financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[2] Free Software Foundation, Inc. [n. d.]. *Coreutils - GNU Core Utilities*. https://www.gnu.org/software/coreutils/ Archived at https://archive.ph/9cQ2P.

[3] David Schmidt, Alexander Ponticello, Magdalena Steinböck, Katharina Krombholz, and Martina Lindorfer. 2025. Analyzing the iOS Local Network Permission from a Technical and User Perspective. In *46th IEEE Security & Privacy (S&P)*.

[4] David Schmidt, Sebastian Schrittwieser, and Edgar Weippl. 2025. Leaky Apps: Large-scale Analysis of Secrets Distributed in Android and iOS Apps. In *32nd ACM Conference on Computer and Communications Security (CCS)*.

[5] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *9th Compiler Construction*.

[6] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *41st IEEE Security & Privacy (S&P)*.