# Automated End-to-End Dynamic Taint Analysis for WhatsApp

Sopot Cela*
Meta
London, United Kingdom
scela@meta.com

Andrea Ciancone
Meta
London, United Kingdom
aciancone@meta.com

Per Gustafsson†
Meta
London, United Kingdom
pergu@meta.com

Ákos Hajdu
Meta
London, United Kingdom
akoshajdu@meta.com

Yue Jia
Meta
London, United Kingdom
yuej@meta.com

Timotej Kapus
Meta
London, United Kingdom
kapust@meta.com

Maksym Koshtenko
Meta
London, United Kingdom
mkosh@meta.com

Will Lewis
Meta
London, United Kingdom
willjwlewis@meta.com

Ke Mao
Meta
London, United Kingdom
kemao@meta.com

Dragos Martac
Meta
London, United Kingdom
dragosmartac@meta.com

## ABSTRACT

Taint analysis aims to track data flows in systems, with potential use cases for security, privacy and performance. This paper describes an end-to-end dynamic taint analysis solution for WhatsApp. We use exploratory UI testing to generate realistic interactions and inputs, serving as data sources on the clients and then we track data propagation towards sinks on both client and server sides. Finally, a reporting pipeline localizes tainted flows in the source code, applies deduplication, filters false positives based on production call sites, and files tasks to code owners. Applied to WhatsApp, our approach found 89 flows that were fixed by engineers, and caught 50% of all privacy-related flows that required escalation, including instances that would have been difficult to uncover by conventional testing.

## CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**.

## KEYWORDS

Taint analysis, simulation, testing

**ACM Reference Format:**
Sopot Cela, Andrea Ciancone, Per Gustafsson, Ákos Hajdu, Yue Jia, Timotej Kapus, Maksym Koshtenko, Will Lewis, Ke Mao, and Dragos Martac. 2024. Automated End-to-End Dynamic Taint Analysis for WhatsApp. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3663529.3663824

*Authors are in alphabetical order, which is not intended to denote any information about the relative contribution.
†All of Per Gustafsson's contribution to this work was conducted at Meta.



*FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil*

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663824

## 1 INTRODUCTION

The purpose of taint analysis is to track the propagation of data in systems or programs in order to show the presence (or prove the absence) of certain flows. Taint analysis has various applications in security [4, 7, 8, 15, 18], privacy [14, 17, 19, 22] and performance [5] analysis. At Meta, we develop various code analysis tools to help our programmers write robust code. The goal of this particular work is to provide WhatsApp developers with an *automated* and *end-to-end* taint analysis tool. Automation means that developers only have to specify the source and the sink of the flow, and being end-to-end refers to the fact that propagation can be tracked both on the clients (Android and iOS) and the server.

Making an automated and end-to-end taint analysis tool practical in WhatsApp's environment comes with a few challenges. First, we cannot record and replay real user data to remain private. Furthermore, we cannot use release builds as we need symbolication for precise localization and further instrumentation (e.g., redirecting client traffic to sandbox server). As a consequence of using non-release builds and artificial data, we might report flows that do not exist in production (false positives). Therefore, we need heuristics to mitigate this in a post-processing step. Finally, it is technically challenging to support the wide range of platforms (server and clients) and programming languages (Java, Kotlin, Objective-C, Swift, Erlang, Hack) in WhatsApp's ecosystem.

To tackle the aforementioned challenges, we propose a dynamic analysis built on top of Sapienz [12]. Sapienz uses exploratory user interface (UI) testing to generate realistic (but artificial, and thus privacy-safe) interactions and data for a pool of test users. We can configure which data acts as a source on the clients. Sapienz then aims to generate interactions that maximize the coverage of the analysis. We use custom instrumented builds that can track what data reached our pre-defined sinks (e.g., application logs) including symbols (e.g., module/function names) both on client and server sides. Furthermore, custom sinks can also be added via (platform-specific) instrumentation. Our analyzer then uses both pre-defined and configurable rules (e.g., regular expressions) to match the data in the sinks with the sources to find tainted flows. Finally, a reporting pipeline to localizes the flows in the source

code, applies deduplication, filters false positives and attributes tasks to the appropriate code owner. Localization and ownership attribution is based on querying the source control system and it helps to make the tasks actionable. False positives are filtered by heuristics matching the flows (found in non-release builds) against production call sites.

We developed and rolled out our analyzers iteratively, collecting feedback and improving precision and coverage over time. Our initial focus was on privacy use cases and we filed 174 and 33 tasks with a true positive rate of 39% and 64% for Android and iOS, respectively. Moreover, our automated analyzers have detected 40% (Android) and 60% (iOS) of all privacy-related data flows that required further investigation/escalation during the observed period. Some of the detected flows occur under very special circumstances that would have been hard to uncover by conventional testing. This leads us to the conclusion that our analyzers are effective in helping developers write and maintain robust and high quality code. To summarize, the contributions of the paper are twofold: we propose a novel way to leverage Sapienz for dynamic taint analysis and we report on a large-scale industrial case study at WhatsApp.

## 2 BACKGROUND

*Taint analysis.* Given a set of *sources*, a set of *sinks* and a set of *sanitizers*, the goal of taint analysis is to determine whether there is a data flow from a source to a sink without passing through a sanitizer. Such flows are called *tainted flows* and the associated data is said to be *tainted*. As an example, consider the following pseudocode.

```
d = getData()
try: process(d)
catch err: log("Error " + err + " for data " + d)
```

In this example, we can define `getData` as a source and `log` as a sink. It is easy to see that if an error happens during the execution of `process`, then there is a tainted flow from `getData` to `log` via the variable `d`. However, if we introduce a sanitizer function `hash`, and pass `hash(d)` into `log`, there would be no tainted flows.

*Sapienz.* Sapienz [12] is an automated tool for fault-discovery in mobile or web applications based on user-interface (UI) interactions. Automation here means that application developers are not required to define and hard-code a specific user journey for their tests, but can instead rely on the built-in capabilities of Sapienz to explore the application under test (AUT) and discover potential issues. Sapienz tries to simulate user behavior and maximize UI coverage by spawning a pre-defined number of parallel runs for the specified application version. In each run, the UI hierarchy of the AUT is read at every step, a list of eligible actions (e.g., tap on views, insert text, swipe, etc.) is built based on a set of human-crafted heuristics and then ranked based on a recursive reward algorithm that tries to optimize for novelty discovery. More specifically, the algorithm aims to rank those actions higher that are more likely to unlock unexplored features of the AUT (e.g., an action that takes to a page that has not yet been explored).

Another mechanism that Sapienz uses to maximize coverage is injecting rich test user states before any action is performed (e.g., spawn multiple test users, form connections, etc.) [2]. Due to the social nature of the applications developed at WhatsApp (and Meta), augmenting the application with content before the run proved to unlock a wider variety of testing scenarios. On top of this, Sapienz allows customizing its behavior at various stages of the testing through a plug-in mechanism called "external listeners". This enables custom logic (code) to be executed before starting the run, in-between the UI actions and before run completion.

As described above, Sapienz has been built with the purpose of maximizing fault-discovery, which requires completeness- and novelty-seeking UI interaction generation. This behavior makes Sapienz suitable for our current work as well, with the goal of producing and analyzing a diverse set of data flows related to privacy.

*WhatsApp. WhatsApp* consists of different *clients* communicating with the *server*. Our current work targets the server (written in Erlang/Hack), the Android client (written in Java/Kotlin) and the iOS client[1] (written in Objective-C/Swift). Each of these components consist of millions of lines of code. As described later in the paper, doing such multi-platform analysis is challenging and we had to come up with an architecture that supports platform specific extensions (e.g. instrumentation or post processing depending on the platform/language).

In this particular work at WhatsApp, we focus on privacy and define certain UI elements and device APIs (related to user data) on the clients as sources; logging APIs and sensitive endpoints on both client and server side as sinks; and redacting/obfuscating data transformations as sanitizers.

## 3 DYNAMIC TAINT ANALYSIS WITH SAPIENZ

An overview of the architecture can be seen in Figure 1. The main inputs of the workflow are a platform (Android/iOS) and a build handle with the corresponding commit hash (defaulting to the latest version). For Android, we use a customized debug build, which includes symbols in logs and has instrumentation for some additional (WhatsApp specific) sinks besides the standard application logs. We can also redirect traffic to a sandbox server for end-to-end taint analysis. The iOS integration is more recent, and it uses beta builds (having symbols) with no extra instrumentation or sandbox redirection (yet). The sources, sinks and sanitizers are defined in code or configuration. In the following, we describe the three main phases in detail: (1) data generation and simulation with Sapienz, (2) log parsing and tainted data matching, and (3) reporting.

### 3.1 Data Generation and Simulation

The initial step for a Sapienz run is to acquire a device in the form of an emulator (Android) or simulator (iOS). Once the connection to the corresponding device is established, the AUT is installed and a test phone number is registered as a WhatsApp user on the device. The test phone number is leased from a pool dedicated to our use case, subject to a number of isolation mechanisms such that interactions with numbers outside the pool are blocked. As Sapienz relies on multiple parallel runs to achieve higher coverage, it generates a list of contact test numbers for each test user that are likely to be active in other runs. We use this list of contacts to generate content for the current client, such as creating groups or communities. The benefit of this step is twofold. On one hand, by

[1]While Sapienz provides capabilities for testing web applications as well, the focus of the current work is narrowed to the two main mobile clients.

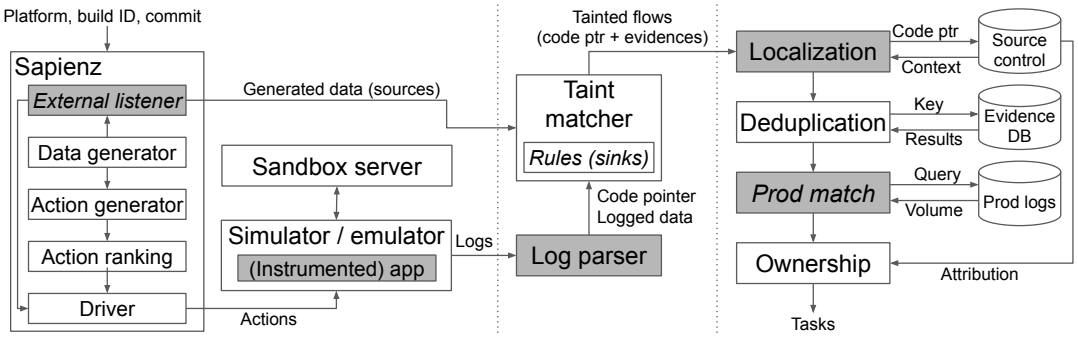


**Figure 1: Overview of the architecture. Components with a grey background are platform dependent.**

simply adding the extra content in the initial state of the application, we extend the list of eligible actions, thereby giving Sapienz more possible paths to choose from. On the other hand, by choosing contacts that are likely to be active, we leverage the network effect to increase the chance of multi-client interactions such as VoIP calls or opening disappearing messages.

We use the external listener mechanism of Sapienz to specify and gather the subset of generated data to be used as sources in the rest of the pipeline. Examples include phone numbers, contacts, UI textbox contents, IP addresses, locations, etc. Finding the relevant data was challenging and required domain- and platform-specific knowledge from WhatsApp developers. Note also that Sapienz treats the AUT mostly as a black-box, therefore, we have to make sure that the generated data is unique enough to be identified at the sinks without tracing its complete path throughout the application.

Our simulation termination criteria include multiple conditions, most commonly a limit on the number of UI actions. Sapienz outputs the generated data at sources and the logged sinks. The output also includes each UI action performed (e.g., tap, type, scroll), UI hierarchies at each step, and a screen recording.

Some of the sinks require very specific actions in order to be potentially triggered (e.g., account linking). We use external listeners to directly inject such actions to make Sapienz's search easier. This can be seen as slightly breaking our promise of a fully automated analysis. However, in practice we only needed to define such actions in a few specific cases and the majority of the analysis still remains fully automated.

## 3.2 Log Parsing and Tainted Data Matching

We get standard application logs from the clients and the server, as well as custom (WhatsApp specific) logs provided by our builds with instrumented sinks. First, we parse the logs with a platform dependent parser. Each log file is split up into individual logging statements (which might be multi-line depending on the platform), and we extract pairs of logged data and code pointers. Depending on the platform, the code pointer can include filename, module, function and line number, as well as stack traces.

The external listeners in Sapienz provide the generated data (sources) that we match against the logged data (sinks). Matching is based on *rules*, which can be defined by implementing a particular interface, but there are some general rules that can be reused and configured. For example, the most basic rule is to simply find the occurrence of a generated data in the log using regular expressions. However, we also have rules that filter based on logging level, or require a combination of sources to reach the sink (e.g., pairs of phone numbers). Each occurrence of a rule matching a logging statement is called an *evidence*. We group evidences together by code pointers (to avoid duplicated reports) resulting in the preliminary set of *tainted flows*.

## 3.3 Reporting

*Localization.* The first step of the reporting pipeline is *localization*. Depending on the platform and language, code pointers might only include partial or limited information. Localization uses the source control manager (SCM) to fill in the missing bits so that code pointers precisely identify the full file path and line number (plus we know the commit hash from the input). For example, in Swift we only have the file name, but not the path, so we use an SCM query to find the path (the file name is almost always unique). We also get the line's content and the surrounding context to be used in subsequent steps.

*Deduplication.* The purpose of deduplication is to avoid reporting the same flow multiple times. Potential duplication can occur for various reasons. First, a previous run might have already found the flow and there is a task open, but the developers are still working on it. Moreover, lines might shift around in the code due to other (unrelated) modifications in the file. To avoid this so-called *false splitting* we use the file name and the line's content as a key to save and query already reported flows in our evidence database. If line content is not available, we fall back on using line number. This step can also reopen a task, which was closed but not addressed properly and the flow still exist (new evidence is found).

*Production Matching.* As described earlier, we use non-production builds for technical reasons. This can, however, cause false positives because we can miss certain sanitization that is happening only in production builds. Furthermore, non-release builds often include initialization code with additional logging for testing purposes. To mitigate this, we use various heuristics (depending on the platform/language) to match the found flows against call sites found in production logs.

Production logs do not always have symbols (especially for clients) and line numbers might shift around. Therefore we use the logging context (obtained from localization) to search. For example, if we have a logging statement

```
log("Error " + err + " for data " + d)
```

then we search for the strings `"Error "` and `" for data "`. Interestingly, this can actually result in more false postivies for certain words that commonly appeared elsewhere in the logs so we maintain a blocklist of common words.

Sanitizers are also implemented here as a post processing step: we add further filters to the query to eliminate flows that do not happen in production call sites. For example, checking for digits replaced by ****** in phone numbers is one of the sanitizers we use. We also add filters for the date range and build version number (to avoid reporting flows that happened in the past but not recently), and the build type (e.g., avoid local debug or alpha builds).

Besides filtering false positives, we also rely on production matching for signal boosting: depending on the volume of flows present in production, we can assign different priorities to the flows.

*Ownership Attribution.* The final step of reporting is to file a task to the appropriate owner. For this we use the same Meta subsystem that is used to determine potential reviewers for a code change. The owner usually depends on information obtained from the source code, such as blame information or oncall annotations.

*Tasks.* The resulting tasks include various information for the developers to be actionable and easy to debug, including source and sink description, code pointer and stack trace (depending on platform), Sapienz recorded video of the simulation, (artificial) samples from the analyzer run, production volume and the query. Developers can follow-up on tasks by attaching a code change (acknowledging that the flow is a true positive) or closing the task and attaching some tag (e.g., false-positive). Such follow-up actions are recorded in our evidence database and can be used in future runs (e.g., do not report a false positive again).

## 4 RESULTS

*Deployment.* We deployed our analyzers in an iterative process. Initially, we started with *dry runs* that would just log the flows without filing tasks. Then we moved on to *shadow runs*, where tasks were filed but they were attributed to our team for further investigation. We checked these tasks, triaged the (likely) true positives to developers, and refined our analysis based on false positives and feedback from developers. Discussion with developers also helped us to improve analysis coverage (e.g., by adding new sinks and sources) and reduce false negatives. Our analyses now run thousands of times a day in the continuous integration (CI) pipeline for the latest WhatsApp Android and iOS builds, filing and triaging tasks to code owners directly. In the meantime we still collect feedback from developers and refine our analyses as needed.

*Overview.* Table 1 presents an overview of our results so far. The table breaks down tasks *fixed* (closed with a code change attached, which we consider as a confirmation for being true positive) and *suppressed* (closed with no fix attached, tagged as false positives). Note that the two percentages do not add up to 100% because there can be tasks pending and open. The high difference between the number of tasks filed for the two platforms can be attributed to the fact that the iOS analysis is a more recent development and has been running for a significantly shorter time than the Android analysis (few months vs. more than a year).

**Table 1: Tainted flows (including high-priority SEVs) detected and closed with or without a fix.**

| | Tasks | | | SEVs | | |
|---|---|---|---|---|---|---|
| Platform | Filed | Fixed | Suppressed | All | Detected | Missed |
| Android | 174 | 68 (39%) | 102 (59%) | 10 | 4 (40%) | 6 (60%) |
| iOS | 33 | 21 (64%) | 9 (27%) | 10 | 6 (60%) | 4 (40%) |
| Total | 207 | 89 (43%) | 111 (54%) | 20 | 10 (50%) | 10 (50%) |

*False Positives.* False positives can occur due to various reasons, mostly attributed to the fact that we cannot use production builds and data, and instead rely on post-processing heuristics as a mitigation. First of all, Sapienz performs certain steps to set up the initial state. We needed to explicitly exclude flows originating from here as they cannot happen in production. An other prominent example is the inconsistent usage of APIs: instead of using the appropriate API function to log at a given level, developers sometimes use alternative workarounds, slipping through our production matching heuristics (e.g., using `if isDebug(): Log.production(...)` instead of `Log.debug(...)`). Furthermore, in some cases the sanitizers are vaguely defined (e.g., no explicit limit for production volume), which also causes some inevitable false positives. Our experience so far suggests that the current false positive rate is acceptable for production, but as mentioned earlier, we are continuously monitoring developer feedback. We report less false positives (and as a consequence also less total tasks) on iOS due to using beta builds (as opposed to debug builds on Android). Note also that false positives can usually be identified and suppressed within a few minutes, thanks to the comprehensive output from Sapienz, including precise code pointers and video recordings of the simulation.

*Escalation.* Some of the tasks have actually been escalated to so-called *SEV*s, which is an internal mechanism and rigorous process for fixing and tracking high-priority tasks. These numbers are reported in the SEVs Detected column (and are included in the figures of the Tasks Fixed column as well). We also monitored all the other SEVs related to privacy, which were not detected by our analyzers, but filed through other means, such as manual reporting (SEVs Missed column). This gives us a sense of the percentage of privacy-related data flows that slipped through our analyzers (false negatives). There were examples of false negatives simply attributed to the fact that our analyzers were developed incrementally: sometimes we learned about the existence of an interesting, but not yet covered source or sink via a manually filed SEV. In these cases we added the relevant configurations to make sure these flows are detected by our tools should they happen again. However, some of the false negatives are very specific from a technical point of view and require a case by case remediation. For instance, one slip through was related to a locale-aware rule that needed custom (country-specific) test users to be triggered. Another class of false negatives comes from new features that are originally gated in

debug builds (e.g., through test user configurations or UI toggles), but are enabled by default in production. This indicates that the full potential of the tool can be exploited with effective developer collaboration. To facilitate this, we created a dashboard for monitoring coverage (see paragraph below) and we are maintaining communication channels for feedback and requests. Note also that our approach – as usual for testing – is not intended prove the absence of tainted flows; instead, it can only confirm their presence.

An other interesting aspect of the results is that the number of SEVs for both platforms is similar while the number of tasks differs significantly. While we do not have a definite explanation for this, our hypothesis is the "true negative effect" as described by a study from Amazon [10]: recommendations coming from automated tools are internalized by developers over time, who will avoid making the same mistakes again. Furthermore, when an analyzer is deployed the first time, it is likely to find tainted flows that have been present for a longer time and have a higher chance of being escalated. However, once an analyzer runs frequently (e.g., daily), tainted flows are often detected *before* needing an escalation, which is in fact one of the main points of having such automated tooling.

*Performance.* Table 2 highlights the p50 and p90 execution times[2] of the full analysis (Sapienz + taint matching + reporting) and exploration (Sapienz) only. The key takeaway from the numbers is that the analysis is fast enough to be triggered multiple times per day. Sapienz runs are slightly slower on iOS because exploration often leaves WhatsApp (e.g., clicking on a link that opens a browser) and actions that take Sapienz back to WhatsApp are more expensive.

**Table 2: Execution time, UI actions volume and coverage.**

| | Execution time (mins) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Sapienz | | Total | | Jobs | Actions | | |
| Platform | p50 | p90 | p50 | p90 | /day | /job | /day | Coverage |
| Android | 10.2 | 13.8 | 14.7 | 18.2 | 1920 | 100 | 1.92M | 34% |
| iOS | 14.5 | 16.7 | 18.2 | 21.1 | 1920 | 100 | 1.92M | 46% |

Table 2 also presents the number of jobs and actions executed (excluding setup actions). Sapienz deployments at Meta typically work with 50–100 actions per job, which has proven to give a good trade-off between performance and depth of exploration. Having thousands of jobs with millions of actions per day increases our chance to catch rare flows, e.g., flows with rare exceptions or flows depending on race conditions.

*Coverage.* We measured `Activity`[3] and `UIViewController`[4] coverage for Android and iOS respectively, which roughly correspond to a single page or screen in the apps. These numbers are relatively high for an industrial use case [20]. Monitoring coverage also helps us to detect regressions and identify parts of the apps that need more coverage (e.g., injecting actions with external listeners).

*Server Side.* The numbers presented in Table 1 correspond to sinks on the clients. End-to-end analysis on the server side is more preliminary, most notably it does not have production matching yet. Therefore, it has not yet been rolled out to automatically triage tasks to developers. However, we did conduct a few experiments which resulted in 3 true positive tasks (manually triaged), including 2 SEVs. Furthermore, the server has already seen good coverage using traffic generation and tests directly (without clients) [13].

*Example.* One particularly interesting example was when data from a source ended up being used as the file name of a file locally cached on the iOS client. In certain race conditions, this file was about to be deleted by two concurrent processes. The first one succeeded, but the second crashed (because the file was already gone) and an error message containing the file name (and consequently the tainted data) ended up being logged. This tainted flow happened non-deterministically, meaning that it would have been hard to uncover or debug by conventional testing. However, our analysis runs automatically and continuously in large scale, having a much higher chance to find such flows.

## 5 RELATED WORK

FAUSTA [13] is an automated testing tool for WhatsApp server that generates traffic based on client-server traffic specifications. FAUSTA can be used to detect crashes, performance regressions, and it also supports taint analysis. InfERL [6] is a static analysis tool for Erlang, which supports taint properties described by automata. Both FAUSTA and InfERL are deployed to scan WhatsApp server, complementing the Sapienz-based analysis described in this paper and all of them are modules in the PrivacyCAT system [11].

Zoncolan [9] is a static taint analysis tool deployed for Hack code at Meta, primarily targeting security properties and running on code modifications as part of the review process.

PTPDroid [17] and GUILeak [21] are two automated tools that leverage a static taint analyzer tool named FlowDroid [3] aiming to identify privacy policy violations in popular Android applications. The former focuses on identifying violations coming from API calls to 3rd parties, while the latter is aiming to discover leaks coming from user inputs (extending the classical approach to look at device information such as location or device ID). Similarly, in the current work, we construct the sources as a combination of device APIs and generated inputs.

ViaLin [1] is a recent dynamic analysis tool that focuses on precise tracking of tainted flows (i.e., not only the endpoints) on Android applications via instrumentation.

Privee [23] and Hermes [16] convert requirements into structured formats based on natural language analysis. Such tools could reduce our manual effort when defining sinks and sources from plain English requirements.

## 6 CONCLUSIONS

In this paper we presented an automated and end-to-end dynamic taint analysis system for WhatsApp. Our solution relies on Sapienz to generate realistic test user interactions and data on the client side. We track data propagation into pre-defined and customizable sinks on both client and server sides. A reporting pipeline with various heuristics then localizes the flows, applies deduplication, filters false positives and files tasks to programmers. Our analyzers have been deployed at WhatsApp with promising results on fix rates and detecting privacy-related flows that needed escalation.

[2] p50/p90 means that 50%/90% of jobs finished within the indicated time.
[3] https://developer.android.com/reference/android/app/Activity
[4] https://developer.apple.com/documentation/uikit/uiviewcontroller

# REFERENCES


[1] Khaled Ahmed, Yingying Wang, Mieszko Lis, and Julia Rubin. 2023. ViaLin: Path-Aware Dynamic Taint Analysis for Android. In *Proc. of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. ACM, 1598–1610. https://doi.org/10.1145/3611643.3616330

[2] Nadia Alshahwan, Arianna Blasi, Kinga Bojarczuk, Andrea Ciancone, Natalija Gucevska, Mark Harman, Simon Schellaert, Inna Harper, Yue Jia, Michal Krolikowski, Will Lewis, Dragos Martac, Rubmary Rojas, and Kate Ustiuzhanina. 2024. Enhancing Testing at Meta with Rich-State Simulated Populations. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice*. (Accepted, in press).

[3] 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. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269. https://doi.org/10.1145/2666356.2594299

[4] Subarno Banerjee, Siwei Cui, Michael Emmi, Antonio Filieri, Liana Hadarean, Peixuan Li, Linghui Luo, Goran Piskachev, Nicolás Rosner, Aritra Sengupta, Omer Tripp, and Jingbo Wang. 2023. Compositional Taint Analysis for Enforcing Security Policies at Scale. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1985–1996. https://doi.org/10.1145/3611643.3613889

[5] Marcin Copik, Alexandru Calotoiu, Tobias Grosser, Nicolas Wicki, Felix Wolf, and Torsten Hoefler. 2021. Extracting Clean Performance Models from Tainted Programs. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 403–417. https://doi.org/10.1145/3437801.3441613

[6] Ákos Hajdu, Matteo Marescotti, Thibault Suzanne, Ke Mao, Radu Grigore, Per Gustafsson, and Dino Distefano. 2022. InfERL: Scalable and Extensible Erlang Static Analysis. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*. ACM, 33–39. https://doi.org/10.1145/3546186.3549929

[7] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 175–185. https://doi.org/10.1145/1181775.1181797

[8] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. 2006. Improving Software Security via Runtime Instruction-Level Taint Checking. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. ACM, 18–24. https://doi.org/10.1145/1181309.1181313

[9] Francesco Logozzo, Manuel Fahndrich, Ibrahim Mosaad, and Pieter Hooimeijer. 2019. Zoncolan: How Facebook uses static analysis to detect and prevent security issues. https://engineering.fb.com/2019/08/15/security/zoncolan/.

[10] Linghui Luo, Rajdeep Mukherjee, Omer Tripp, Martin Schäf, Qiang Zhou, and Daniel Sanchez. 2023. Long-term static analysis rule quality monitoring using true negatives. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 315–326. https://doi.org/10.1109/ICSE-SEIP58684.2023.00034

[11] Ke Mao, Cons T Åhs, Sopot Cela, Dino Distefano, Nick Gardner, Radu Grigore, Per Gustafsson, Ákos Hajdu, Timotej Kapus, Matteo Marescotti, Gabriela Cunha Sampaio, and Thibault Suzanne. 2024. PrivacyCAT: Privacy-Aware Code Analysis at Scale. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice*. (Accepted, in press).

[12] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105. https://doi.org/10.1145/2931037.2931054

[13] Ke Mao, Timotej Kapus, Lambros Petrou, Ákos Hajdu, Matteo Marescotti, Andreas Löscher, Mark Harman, and Dino Distefano. 2022. FAUSTA: Scaling Dynamic Analysis with Traffic Generation at WhatsApp. In *Proceedings of 15th IEEE Conference on Software Testing, Verification and Validation*. IEEE, 267–278. https://doi.org/10.1109/ICST53961.2022.00036

[14] Sydur Rahaman, Iulian Neamtiu, and Xin Yin. 2021. Algebraic-Datatype Taint Tracking, with Applications to Understanding Android Identifier Leaks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 70–82. https://doi.org/10.1145/3468264.3468550

[15] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26

[16] John W. Stamey and Ryan A. Rossi. 2009. Automatically Identifying Relations in Privacy Policies. In *Proceedings of the 27th ACM International Conference on Design of Communication*. ACM, 233–238. https://doi.org/10.1145/1621995.1622041

[17] Zeya Tan and Wei Song. 2023. PTPDroid: Detecting Violated User Privacy Disclosures to Third-Parties of Android Apps. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering*. 473–485. https://doi.org/10.1109/ICSE48619.2023.00050

[18] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 87–97. https://doi.org/10.1145/1542476.1542486

[19] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. 2020. Scaling Static Taint Analysis to Industrial SOA Applications: A Case Study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1477–1486. https://doi.org/10.1145/3368089.3417059

[20] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 738–748. https://doi.org/10.1145/3238147.3240465

[21] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D. Breaux, and Jianwei Niu. 2018. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 37–47. https://doi.org/10.1145/3180155.3180196

[22] Chengxu Yang, Yuanchun Li, Mengwei Xu, Zhenpeng Chen, Yunxin Liu, Gang Huang, and Xuanzhe Liu. 2021. TaintStream: Fine-Grained Taint Tracking for Big Data Platforms through Dynamic Code Translation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 806–817. https://doi.org/10.1145/3468264.3468532

[23] Sebastian Zimmeck and Steven M. Bellovin. 2014. Privee: An Architecture for Automatically Analyzing Web Privacy Policies. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association, 1–16.


Received 2024-02-08; accepted 2024-04-18