PrivacyCAT: Privacy-Aware Code Analysis at Scale

Ke Mao
Meta
London, UK
kemao@meta.com

Cons T Åhs
Meta
London, UK
cons@meta.com

Sopot Cela
Meta
London, UK
scela@meta.com

Dino Distefano
Meta UK and Queen Mary University of London
ddino@meta.com

Nick Gardner
Meta
London, UK
nikgardner@meta.com

Radu Grigore
Meta
London, UK
rgrigore@meta.com

Per Gustafsson
Meta
London, UK
pergu@meta.com

Ákos Hajdu
Meta
London, UK
akoshajdu@meta.com

Timotej Kapus
Meta
London, UK
kapust@meta.com

Matteo Marescotti
Meta
London, UK
mmatteo@meta.com

Gabriela Cunha Sampaio
Meta
London, UK
gabrielasampaio@meta.com

Thibault Suzanne
Meta
London, UK
tsuzanne@meta.com

Abstract

Static and dynamic code analyses have been widely adopted in industry to enhance software reliability, security, and performance by automatically detecting bugs in the code. In this paper, we introduce PrivacyCAT[1], a code analysis system developed and deployed at WhatsApp to protect user privacy. PrivacyCAT automatically detects privacy defects in code at early stages (before reaching production and affecting users), and therefore, it prevents such vulnerabilities from evolving into privacy incidents. PrivacyCAT comprises of a collection of static and dynamic taint analysers.

We report on the technical development of PrivacyCAT and the results of two years of its large-scale industrial deployment at WhatsApp. We present our experience in designing its system architecture, and continuous integration process. We discuss the unique challenges encountered in developing and deploying such kind of analyses within an industrial context.

Since its deployment in 2021, PrivacyCAT has safeguarded data privacy in 74% of privacy site events (SEVs). It has prevented 493 potential privacy SEVs from being introduced into the codebases, enabling developers to maintain a high privacy standard for the code that supports over two billion WhatsApp users.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**.

Keywords: program analysis, dynamic analysis, static analysis, privacy

ACM Reference Format:
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 *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3639477.3639742

1 Introduction

WhatsApp is a messaging app developed at Meta. With over two billion users in more than 180 countries, it is one of the most popular apps in the world. User privacy is a fundamental value embedded in WhatsApp's DNA. Notably WhatsApp provides end-to-end encrypted messages, audio, and video calls.

In this paper, we introduce PrivacyCAT, an automatic tool developed at WhatsApp that is used internally to detect possible privacy vulnerabilities in the code as a layer of privacy protection, in addition to privacy by design (e.g., end-to-end encryption). This helps programmers ship more robust code from the privacy point of view. The name PrivacyCAT stands for *Privacy-Aware Code Analysis Tools.* PrivacyCAT in this work refers to a single privacy risk detection system composed of multiple dynamic and static analyzers. It aims to find privacy vulnerabilities in WhatsApp code early in the development cycle, and report them to developers through a unified channel for fixes before the code reaches

[1]Authors after the first author 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.

ACM ISBN 979-8-4007-0501-4/24/04…$15.00
https://doi.org/10.1145/3639477.3639742

production, and consequently affects users. PrivacyCAT performs dynamic taint analysis based on synthesized, realistic user inputs and traffic generation (via tests, Sapienz [1] and FAUSTA [26]), and static taint analysis based on Infer [8]. It traces the propagation of synthesized sensitive data, and its processing at data sinks and exchanging APIs for leakage detection. PrivacyCAT is designed to run on both WhatsApp client and server code changes in order to prevent the introduction of privacy violations into the code. Moreover, it performs hourly scans of the codebases to search for possible privacy defects hidden in the code. By detecting potential privacy leaks before the code is deployed in production, it helps protect WhatsApp users' privacy.

We evaluated PrivacyCAT's performance based on two-year long deployment at three large codebases at WhatsApp. The datasets cover both client and server sides' daily changes from developers. Our findings show that PrivacyCAT is effective in revealing privacy risks and preventing privacy incidents at an early stage.

This paper makes the following main contributions:

- **It introduces the use of privacy-oriented code analysis technology at scale in the software development process.** We present the system design of PrivacyCAT, including both dynamic and static analyses. We showcase the integration, deployment, and maintenance of PrivacyCAT to support developers' daily work at a large scale for the purpose of automating user privacy protection.
- **We report on the privacy use-case for code analysis.** Previous related studies were conducted on apps from app stores without deployments under industrial contexts [14]. We report on our deployments where real privacy risks were detected, confirmed and fixed by developers. This provides empirical data from an industry perspective under the context of large-scale code-bases.
- **We discuss the challenges we faced during two years of PrivacyCAT application at WhatsApp.** Similar challenges may provide directions for future research and push the boundaries of the code analysis techniques and their applications at industry.

The rest of this paper is organized as follows: Section 2 presents background information on taint analysis. Section 3 and 4 describe the dynamic and static analysis approaches of PrivacyCAT. Section 5 demonstrates how PrivacyCAT was integrated and deployed into continuous integration. Section 6 evaluates the performance of PrivacyCAT with industrial empirical data. Section 7 summarizes previous code analysis studies for privacy. Finally Section 8 concludes.

2 Background

In this section, we present background information on taint analyses that empower PrivacyCAT and give an overview of the WhatsApp systems that are the focus of our analysis.

Taint Analysis. Taint analysis is a program analysis technique used to analyze properties of code related to a variety of use cases such as: security analysis [30, 31, 36], software testing and debugging [6, 29]. In our setting, taint analysis performs a dataflow analysis trying to establish that

> *no sensitive data flows exist from any source to any sink, for a given set of sources and sinks.*

We apply taint analysis in PrivacyCAT to detect privacy leaks. We track (artificially generated) Personally Identifiable Information (PII) and check that they do not reach (i.e., flow into) sinks. In our case, a sink could be a database where the storage of PII is prohibited or endpoints where data transfer should be restricted.

System Under Analysis. The goal of PrivacyCAT is to find privacy issues in WhatsApp as early as possible (i.e., shift-left privacy protection). From PrivacyCAT's perspective, WhatsApp code can be classified into three categories:

- WhatsApp Clients are various apps and other surfaces through which people experience WhatsApp. These are written in languages specific for the platform, for example Java/Kotlin for Android, Objective-C/Swift for iOS or JavaScript for Web. In this paper we only consider Android and iOS clients.
- WhatsApp Server is mainly written in Erlang and responsible for routing messages between clients. For some advanced features such as integrity, WhatsApp Server talks to other services via a RPC protocol.
- WhatsApp Services that are written as standalone services supporting advanced features of WhatsApp such as integrity checks and business features.

Due to the variety of systems we aim to cover, PrivacyCAT is not a single analysis, but a collection of analyses (§3-4) sharing the same goal of finding privacy issues with a shared reporting pipeline (§5).

Dataflow Properties. PrivacyCAT checks privacy policies that we were able to translate into dataflow problems. An example property we check is a `message-pair` logging. We've translated this into a dataflow requirement of: "Two distinct values tagged as phone numbers do not end up in a logging sink together". Other policies are of the form "certain sources do not end up in specific sinks". For sources we keep a set of fields we consider sensitive, these include but aren't limited to: phone number, input text, GPS location, payment information, locale and others. For sinks we consider various logging frameworks and database systems used by WhatsApp.

3 Dynamic Analysis

Overview. PrivacyCAT dynamic taint analysis consists of two components: 1) a way of exercising WhatsApp code; 2) a way of detecting a flow of data. For 1) we use FAUSTA traffic generation [26] for server code and Sapienz [1, 25] for client

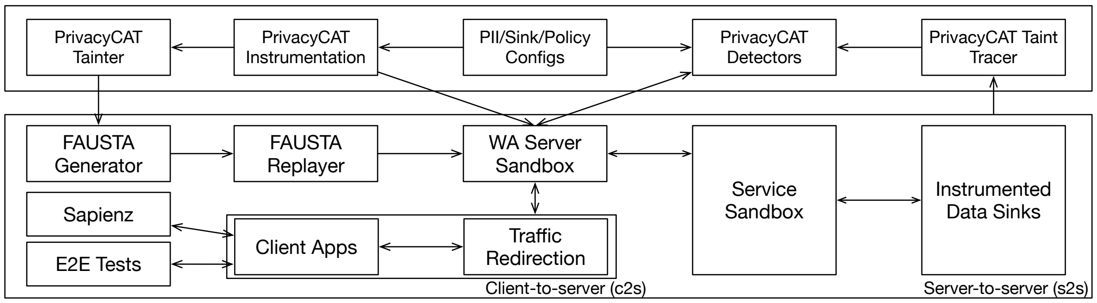

Figure 1. An Overview of PrivacyCAT Dynamic Analysis

code, we also use pre-existing tests authored by developers, for both client and server. For detecting dataflows we use two methods: a light-weight coarse taint analysis, that does not track how the data propagates and a finer taint analysis that does. The different components can be combined, but we have not deployed all combinations.

Coarse Taint Analysis. Coarse taint analysis is a simple substring check of tainted values in a sink. In other words, we "grep" for sources (that we generated) in sinks. This means that coarse taint analysis only knows if the tainted data ended up in the sink, but not what paths it took to get there. In addition, it cannot detect tainted data that has been transformed. However, due to the simplicity and ease of deployment we have found this analysis to be very effective.

First, we explain how coarse taint analysis implemented with tests: We have manually changed commonly used testing APIs for getting test users to write what would be PII values – such as phone numbers – into a file, called `sources`. We have instrumented the code to write all data going to sinks into another file called `sinks`. After the tests are done running, PrivacyCAT checks the `sinks` file for any values that are present in the `sources` file.

Second, FAUSTA [26] is a traffic generation system which has been deployed at WhatsApp for analysis of reliability issues such as *crashes*. To summarise, FAUSTA's [26] *traffic data generators* materialize server inputs (called stanzas) based on specs, and send them to the server. PrivacyCAT extends this materialization process to taint some of the data, based on the annotations for the specs. In addition it instruments the server using the same instrumentation as tests. Such instrumentation enables profiling on coverage, stack trace and tracing of tainted values. The code under analysis runs in a non-production, controlled environment to prevent the instrumentation from introducing any side effects to production.

The server is instrumented by an Erlang parse transform, that takes a list of functions as input, which are considered sinks to be instrumented. It then injects code at the start of these functions to write all arguments as well as the current stack trace into a file. This `sinks` file is later checked by PrivacyCAT.

For example, given a function `foo` in `foo.erl`, which concatenates its two arguments and calls `bar`:

```
foo(A, B) ->
  bar(A ++ B).
```

It would get rewritten into a function shown below. The instrumentation adds a call to write the two arguments into a file. In addition, current callstack and source location are also written to the file. This enables us to localise issues found back to code, making it easier to report. Note that the code below is shortened for brevity.

```
foo(A, B) ->
  file:write_file(
    "/tmp/sinks",
    "A:~p␣B:~p,CallStack:~p␣Src:~p~n",
    [
      A,
      B,
      erlang:process_info(self()),
      "foo.erl:2"
    ]
  ),
  bar(A ++ B).
```

Lastly, Sapienz [1, 25] is a client side search-based testing tool that automatically explores the client UI. PrivacyCAT uses Sapienz to exercise Android and iOS client code in a testing environment. Sapienz uses a framework to login into the app with test users. We record sensitive attributes of these test users as sources, such as their phone numbers and profile names. In addition, we record any texts Sapienz types into the UI as an "input text" source. We use this input text source as an over approximation for "message content" source. For sinks we use Logcat in Android and app logs in iOS, as well as manually patched-in instrumentation for WhatsApp specific sinks.

Finer Taint Analysis. Finer Taint Analysis is our dynamic taint analysis for Erlang. It aims to detect the flow of data from sources to sinks through data transformations and report steps on how the data got there. It is our implementation of Karim et.al [20] taint analysis for Erlang. Detailed exposition of this analysis is beyond the scope of this paper, below we give a brief outline:

Their idea is to shadow the real execution by an execution of an abstract-machine operating on taint values. The analysis is performed in three steps:

1. Instrument the program under analysis with instrumentation that emits instructions (via a side channel) for the abstract machine.
2. Run the program under analysis. This results in a trace of the execution in the form of a sequence of instructions for the abstract machine.
3. Execute the emitted instructions on the abstract machine to get the taint analysis result.

To implement the analysis proposed by Karim et.al [20] in Erlang, we have built a parse transform that traverses the Erlang Abstract Syntax Tree (AST) and inserts instruction emitters for the taint abstract machine between expressions in the bodies of Erlang abstract forms. These instruction emitters write instructions for the abstract machine to a file, during execution of the code under analysis. The emitters are inserted in such a way to represent the effect of the Erlang expressions would have on the taint value. Additionally, we have implemented the abstract machine itself in Erlang, which executes the emitted instructions.

Karim et al.'s JavaScript implementation [20] is single-threaded. We have extended their approach to support parallelism by leveraging the inherent parallelism of Erlang. In Erlang, each process runs independently and only communicates via message passing. Therefore, we can run one taint abstract machine per process and only need to handle message passing. Since in our implementation of the abstract machine is also in Erlang, we made it pass "taint" messages between each other.

Cross-repo Analysis. PrivacyCAT dynamic analysis also performs end-to-end analysis to detect privacy leakages that may occur across multiple systems. Specifically, the *interop analysis* detects privacy leakages in the presence of data exchanges, such as Client-to-Server (*C2S*) and Server-to-Server (*S2S*), as demonstrated by Fig. 1. The end-to-end analysis connects the full pipeline and targets use cases that depend on other internal services.

The *C2S* interop analysis is based on dynamic tracing with input generation for both client and server, as described above. For interop analysis, PrivacyCAT redirects Sapienz client traffic to WhatsApp Server sandboxes. This can be seen as the merge of Sapienz and FAUSTA driven analysis described before.

For *S2S*, PrivacyCAT uses a combination of client and server traffic to trigger interop code paths, such as fetching a business user profile stored by another heterogeneous service. *S2S* is an extension of *C2S* interop analysis where in addition to connecting client to WhatsApp server sandbox, it also connects the WhatsApp server sandbox to another heterogeneous server sandbox. PrivacyCAT also instruments the target sinks of cross-repo services.

Finally, note that these solutions would be infeasible to perform in a production environment mainly due to two reasons:

- Using production data may lead to privacy concerns;
- Runtime checking in production would introduce a non-trivial performance overhead to the system.

Instead, PrivacyCAT generates artificial sensitive and insensitive inputs, and inject them into an isolated, controlled sandbox environment where no user data is involved.

4 Static Analysis

Static taint analysis in PrivacyCAT refers to two distinct extensions of Infer, specifically when used for detecting dataflows. Infer [8, 12] is a general-purpose static-analysis platform supporting multiple languages. The two extensions of Infer that we developed are *Topl* and *Lineage*, which we describe next. These extensions are currently only deployed for WhatsApp server's Erlang codebase, but we plan to analyze client code (that the dynamic analysis already covers) in the future.

Topl. Topl [18] refers to both a language for specifying user-defined properties, and the engine in Infer that checks whether the property holds for a given program. Topl properties are expressed in terms of state machines and therefore are well suited to monitor executions. Here is an example of a typical taint property:

```
property Taint
 start -> start: *
 start -> track: "source/0"(Ret) => data:=Ret
 track -> error: "sink/1"(Arg, Ret) when Arg~~>
     data
```

The state machine has three states: `start`, `track`, and `error`. There is a non-deterministic loop transition in the `start` state to be able to track multiple instances of tainted data. Then, if we see a call to a function named `source`, we store it's return value in the `data` register of the state machine and go to the `track` state. Finally, if we see a call to function `sink` where the argument is a term containing the term we stored in `data`, we report an error. Static PrivacyCAT is currently deployed to check three similar, but more complex properties where sources and sinks are defined based on WhatsApp's privacy policies and are a subset of properties checked by dynamic analysis.

We check the same properties with both dynamic and static analysis for several reasons. First, dynamic and static analyses have different strengths: dynamic analysis only explores real executions; static analysis can see executions missed by dynamic analysis; dynamic analysis is easier to extend to cover multi-language codebases, for example including both a client and a server. Second, dynamic and static analyses offer complementary diagnostic information: dynamic analysis offers a way to reproduce the problematic execution and a stack trace at the violation time; static analysis offers a trace akin to stepping through the code, thus covering several time moments, ending at violation time.

Under the hood Topl is built on top of Infer's Pulse analysis, which performs symbolic execution over the program. Topl augments Pulse by defining a monitor based on the property (state machine) and performing abstract interpretation over both the program and the monitor.

Lineage. Topl is an analysis designed to find violations of temporal properties, which we use mostly for finding data-flows such as violation of taint properties. Lineage is an analysis that we designed from the start with the goal of detecting data-flows but no more. Thus, the analyses are similar in what they achieve (detect dataflows) but differ in some design decisions, which we discuss next.

Topl is based on symbolic execution, and thus underapproximates program semantics. In contrast, Lineage is based on abstract interpretation and overapproximates program semantics, which is the classic approach in static analysis. In practice, this means that Topl has few false positives while Lineage has few false negatives.[2]

Whereas Topl works by taking a property and a program and analysing them both together, Lineage works in two distinct phases, first taking only the code into account, and then taking the source/sink query into account. The first phase produces a graph view of the data-flow in the code. The second phase performs a CFL-reachability (context free language reachability) check, from the source to the sink, on that graph.

Topl relies on Pulse's sophisticated reasoning about values. Lineage does not reason about values, and therefore cannot see that certain data-flows cannot happen.

In short, Lineage is simple, fast, and has few false negatives. For use-cases where it is important to have few false negatives we use Lineage; for use-cases where it is important to have few false-positives we use Topl.

Erlang support in Infer. Infer – including Topl and Lineage – is language agnostic, meaning that it can analyze programs in different languages, modulo a language frontend and some specialization needed to improve analysis precision. As part of PrivacyCAT, we added an Erlang frontend to Infer (called InfERL [19]), and modeled Erlang's built-in data structures and a subset of library functions.

5 System Deployment

PrivacyCAT has been deployed at WhatsApp as a tool for detecting privacy regressions as early as possible in the development cycle and, most importantly, before the code reaches

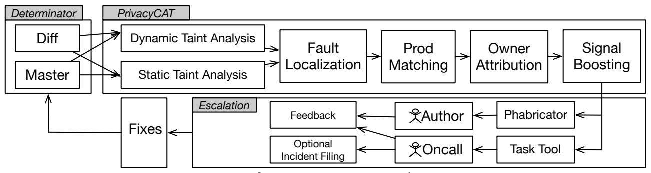

Figure 2. Overview of PrivacyCAT's Integration into CI

production. It is deployed as part of the continuous integration (CI) system and runs together with several other checks of different nature and purposes (e.g. tests, linters, etc).

The deployment covers both server and client side codebases in various programming languages. At the time of writing, the system deployment covers three large repositories: Erlang server, Android and iOS clients. Each of these repositories contains several millions lines of code.

We started PrivacyCAT deployment in the second quarter of 2021 firstly with WhatsApp server repository to run hourly against master branch. Later in third quarter of 2021, we deployed it to run on every WhatsApp server code change (called *diff*). Starting from 2022, we extended the system from server to client repositories, covering both Android and iOS platforms.

Shift-left deployments. Fig. 2 presents an overview of PrivacyCAT's CI integration for providing developers with early signal (shift-left) about potential privacy issues in the code. It is deployed to run against both master branches and also every code change (a.k.a. *diff*) before these are committed in the codebase. The reason of this double deployment is twofold:

i. Running PrivacyCAT on diffs prevents new privacy issues to land in the codebase.
ii. Running every hour on master cleans-up the code base of lurking privacy issues, as well as catch leaks left undetected by the diff analysis (e.g. two diffs are safe individually but merging them causes an issue).

When we look at these deployments from the SEV point of view, diff analysis prevents incidents from happening as SEV-worthy issues are caught before being landed in the codebase. Such diff-time jobs are expected to have minimal time-to-signal to help developers increase dev efficiency. Fig. 3 shows that most PrivacyCAT dynamic and static analysis jobs could finish within 10 minutes, according to over 20k latest prod job samples. Note that PrivacyCAT's time-to-signal may take longer than the analysis jobs considering other end-to-end overhead such as environment setup and harness preparation. Continuous runs on master detect the existence of privacy issues in the code base, open tasks that are triaged to the owner of the code, and if the issue has been confirmed with production volume (see below), a privacy site event (also known as a SEV [13]) is filed.

Fault localization. Fault localization determines the code location of the root cause of the vulnerability (some analyzers

[2]In theory Lineage being an over-approximation would not have false negatives, and Topl being an under-approximation would not have false positives. In reality when applied to real code any program analysis would not be neither fully over-approximating nor fully under-approximating and therefore it would have both false positives and false negatives.

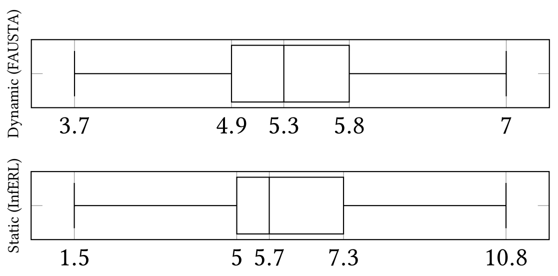

Figure 3. PrivacyCAT diff-time job cost in minutes

also include a full execution or stack trace). This information helps developers quickly locate the source of the issue, and it also helps PrivacyCAT to automatically deduplicate issues and get the necessary context for matching against production data.

Prod matching. One of the main problems with program analysis is high false positive (FP) rate, that is warnings reported by the analysis which cannot actually happen in production. To attenuate the problem, or to decide if it's necessary to open a SEV, we have developed a system which uses heuristics to match warnings reported by the analysis with production data of the last 30 days. These heuristics are based on analyzing the call site and its context in the source code: we extract the constant parts of logging patterns (e.g., constant strings) and search for those in production data. For example, from the context of `log("Error " + err + " for data " + d)` we would extract `"Error "` and `" for data "`. This allows us to estimate the volume of occurrences without relying on the actual data.

Depending on the volume of occurrences found in production, we can add confidence to the privacy risk and file a SEV at the proper level. Having such evidence allows us to escalate the risk to be timely fixed. Note that in principle it would be possible to perform pattern-based detection against production data for finding privacy leaks without PrivacyCAT. But without taint analysis confirming sensitive dataflows, this approach would suffer from high FPs rate. Also, reporting issues already happening in production is useful because PrivacyCAT provides traces for debugging, and errors involving sequences of events can be hard to find in logs.

Owner Attribution. Detected issues are attributed to code owners automatically by PrivacyCAT. For diff-time signals, PrivacyCAT only reports issues related to the diff, thus they can be attributed to the diff author directly. For analysis of master, the attribution relies on the previous fault localization step, which usually points to a root call site. When such a call site is available with evidence from prod matching, a task will be assigned to the author or oncall who owns the call site. For the remaining cases, we assign to PrivacyCAT oncall for a quick sanity check and triage.

Interaction between static and dynamic analysis. We can exploit the fact that we use both static and dynamic analyses to improve results of the single analysis. The Topl static analysis and the PrivacyCAT dynamic taint analysis first produce two independent sets of issues. We then post-process these issues to compute their intersection. Those are the issues detected independently by both analyses and we use this fact as a form of *signal boosting*. Signal boosting consists of:

(a) providing the extra information found by both analyses (e.g. additional traces); and
(b) raising the visibility of those issues because they are more likely to be true positive.

6 Evaluation

In this section, we evaluate PrivacyCAT with data collected during its deployment in a two-year period. We assess PrivacyCAT privacy analysis by answering four questions:

- *Q1: Does* PrivacyCAT *analysis reveal any privacy vulnerabilities? Is the shift-left deployment improving privacy protection?*
- *Q2: What types of privacy vulnerabilities can be detected with* PrivacyCAT *analysis?*
- *Q3: Do developers fix the reported privacy risks?*
- *Q4: What are the challenges and experience learned from the deployment?*

6.1 Data Collection and Methodology

The evaluation considers a data set of PrivacyCAT's results between 2021Q1 and 2023Q2. It covers three major WhatsApp code bases.

PrivacyCAT analysis results were collected continuously, from each of its CI job since deployment. These CI jobs were triggered on every diff submitted by WhatsApp developers to each of the server and Android client repositories[3]. We also collected results of runs on master branches for server and Android/iOS clients, which happens mostly once per hour. The recorded results include details of vulnerability types, description and corresponding code locations for both diff-time signals and also tasks filed to code owners.

To understand the fix status on a diff, we have a continuous job that compares the diff versions before and after PrivacyCAT reports in order to identify possible developer fixes. To understand whether a task has been fixed we look at the diffs attached to the task, plus we use a semi-automated analysis checking developer comments and tag annotations (e.g., we've instructed developers to tag false positive tasks before closing them).

To evaluate the coverage of PrivacyCAT, we manually tracked every privacy incidents in scope with the analysis which happened in the past two years, including those discovered and reported manually by developers at WhatsApp.

[3]The iOS client does not have PrivacyCAT diff-time analysis support yet.

Issues in PrivacyCAT's scope regard leaking sensitive data, while functional requirements (such as failing to display a user notice) are out of scope.

6.2 Effectiveness

During the over two-year's deployment period, PrivacyCAT analysis reported in total 1,715 potential privacy issues for early-stage improvements. Among all cases, 1,426 were raised at diff-time, i.e., before changes were committed into the source control repositories, and 470 were reported when analyzing master branch (i.e., after diff commits) which were automatically triaged to code owners.

Privacy SEVs reported. The most important issues, 26 of the reported cases, were escalated as privacy Site EVents (SEVs). The SEV process ensures the reported incidents are escalated, mitigated, fixed, and retrospectively reviewed. Fig. 4 shows the privacy incidents detected by PrivacyCAT in the server repository over the two years period. The graph compares it with the overall detected incidents including those discovered by human. We can observe that PrivacyCAT has automatically detected 74% of all privacy incidents, while company-wide developers reported the remaining 26%. This indicates that PrivacyCAT performed substantially better than human on the task of detecting data leakage in the code.

Efficacy of Shift-left deployment. Now we assess whether the shift-left deployment of PrivacyCAT enhanced privacy protection. Due to the nature of the system and the property checked by PrivacyCAT, the data-set may not be statistically meaningful, however we can still observe some interesting facts. Also note that, as for any program analysis, PrivacyCAT's warnings are *potential* privacy vulnerabilities about sensitive data leakage, and therefore they do not immediately mean there is a privacy policy violation effectively happening. To confirm that a report is a true positive, PrivacyCAT's reporting pipeline employs the prod-matching mechanism described in Section 5. Based on the matched volume from prod, the code owners may prioritize their fixes such as removing, redacting the sensitive data or limiting the logging volume only for debugging purpose.

The linear trendline of all incidents in Fig. 4 demonstrates that the shift-left practice led to fewer privacy incidents over the two years deployment of PrivacyCAT. Fig. 5 shows the trend of privacy incidents in repository overtime, and compares it with the trend of fixed issues on master. We see that, after the first quarter, PrivacyCAT has filed in increasing number of tasks which were fixed before becoming incidents. See especially the gap between blue and black line in Q2 of 2022. The incident trend has been going down. Without these filed tasks, which were fixed early, the incident number would have been much larger. In 2022, we observed 21% less privacy incidents than 2021. In Q1 of 2023, no privacy incidents were registered on the server code base. Hence those graphs seem to support our intuition that detecting

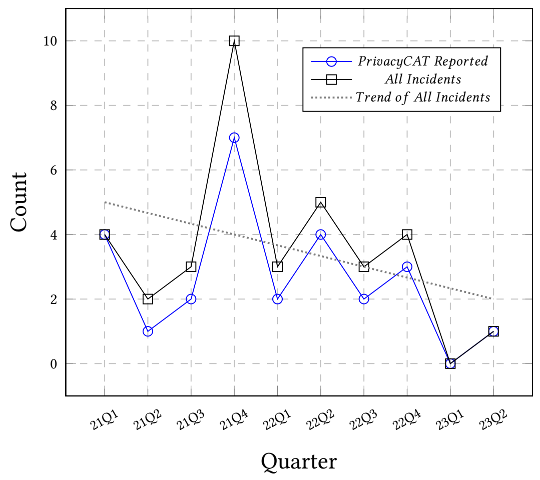

Figure 4. Privacy incidents reported by PrivacyCAT and human (Server Repo)

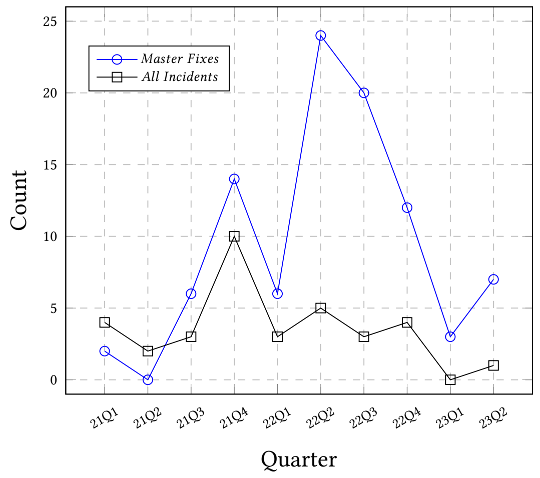

Figure 5. PrivacyCAT master fixes and privacy incidents (Server Repo)

privacy issues as early as possible can decrease the number of incidents. This is important for user privacy, but there is also another advantage. Early detection unlocks developer velocity, because fixing a leak while writing a diff takes minutes, whereas closing an incident usually take days and may involve several departments as stakeholders.

Fig. 6 shows incident-worthy issues detected and fixed at diff time. Those represent potentially prevented incidents. In total, it prevented 493 potential privacy SEVs to land on WhatsApp codebases, Moreover, at some point in the graphs, we observe issues detected at diff time (and also tasks as shown in Fig. 5) decreased. This seems to correspond to what it is called "true negatives" in this paper [24]. Basically, over time, developers become more aware of code analysis rules and learn how to avoid anti-patterns that break the rule.

Effectiveness by analysis. Figure 7 shows the number of errors found and reported broken down by different analysis

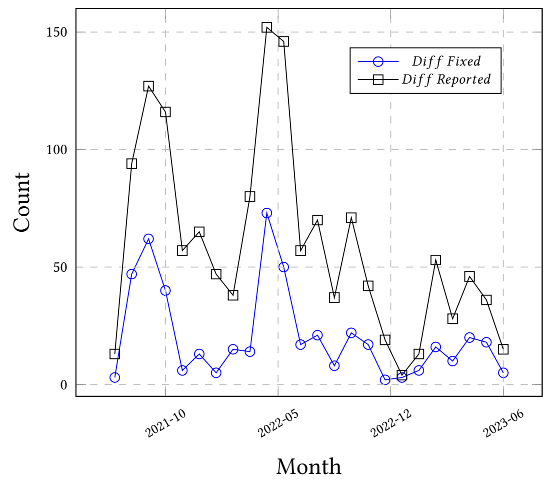

Figure 6. Diff-time monthly reported and fixed vulnerabilities (All Repos)

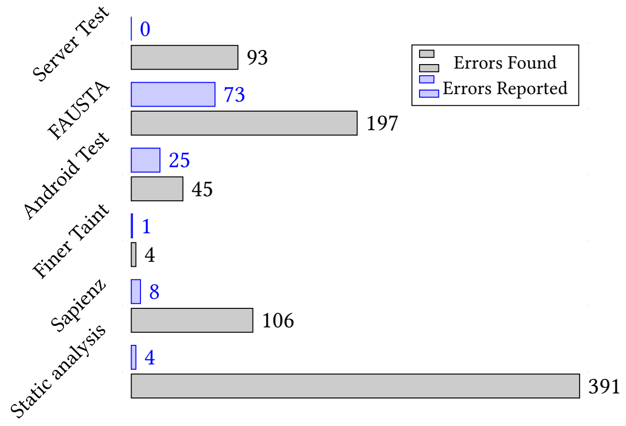

Figure 7. Unique errors found in a 30 day window

in a month. As a reminder, we do not report all issues the tools find, due to various filters we have in place that are described in Section 5 and discussed in Section 6.4. FAUSTA, Server and Android Test, Finer Taint and Sapienz denote the dynamic approaches described in Section 3, while Static analysis is described in Section 4.

While static analysis found most issues, we reported only four of them. This is due to our conservative filtering of false positives. For the dynamic analysis FAUSTA traffic generation and Android tests find the most actionable issues. Finer taint found only four issues, due to its limited deployment.

6.3 Vulnerability Types

At the time of writing, PrivacyCAT is capable of catching vulnerabilities in the following types to help ensure their privacy principles:

Data Collection. Collected user data should follow the data *minimization* and *anonymization* principles. PrivacyCAT reports show that when developers perform data logging, the logging sites are usually generic thus may unintentionally persist sensitive data, such as PII. PrivacyCAT taint analysis identifies such cases of risky logging and help developers avoid over-collection of data.

Data Sharing. Another type of risk is exchanging data with other heterogeneous services. There are policies on how data from WhatsApp can be exchanged with other services. For a WhatsApp service A that communicates with service B, we need to enforce both systems comply with each others privacy policies. From system A's perspective, we break down this problem into two sub problems: Handling incoming traffic and outgoing traffic. Checking outgoing traffic is more interesting for our use case. Our analysis inserts probes into the lower level RPC APIs that performs the data exchange. This instrumentation allows us to perform check of tainted data for all outgoing traffic. When any tainted data is observed at a sink, we also collect the call stack to help trace its propagation. Our deployment shows that PrivacyCAT was able to capture risky call sites that send synthesized sensitive traffic to other internal endpoints. While these results may not directly reveal true positive violations, they have provided us insights to help audit sensitive data consumption and their persistence on the endpoint side.

Data Access Control. Sensitive data usually has restrictions on who can access it, how they can access, and what operations they can perform on the data. Access Control Lists (ACLs) is a common way to define user group permissions including certain operations such as read, write, modify, or delete. PrivacyCAT was able to detect the cases where data sinks write sensitive data, but the target tables are missing the necessary ACLs.

6.4 False Positive, False Negatives, and Fix Rate

Like any program analyses deployed for real code, PrivacyCAT faces challenges in false negatives and false positives. This section discusses the issues and our experience in tackling them.

False negative. The privacy analysis use-case requires high coverage on catching privacy violations, but it's known that the automated approach may miss certain types of regressions. It's challenging to achieve high coverage due to both technical and non-technical reasons. For example, it's infeasible to cover every privacy policy in a large code base. Also not every policy can be formalized or used as a privacy oracle. When tracking tainted traffic, it could miss regressions due to deep states or prod environment that our exploration would not cover.
Mitigation: For policy coverage, we collaborated with privacy experts in order to design policy oracles. To address the limitations on dynamic analysis, we improved FAUSTA's generation strategies and infra efficiency for deeper exploration. Moreover the combination with static analysis makes our analysis more comprehensive. For static analysis, we performed continuous improvements on translating policies

	Reported	Fixed	Fix Rate	FP Rate
Diff	1,426	493	34.6%	-
Master	470	152	32.3%	10.6%

Table 1. Diff and Master Signal Fix Rates (All Repos)

into Topl properties and improving analysis coverage for Erlang syntactic categories.

False positive. False positive is a common issue for any automated detection system. In the privacy use-case it may arise due to:

1. Privacy regressions require human knowledge. For certain cases, we may need policy experts' support to confirm a regression.
2. For early detection, we have to run the analysis in a non-prod environment, which may report issues which will not manifest in production.

Mitigation: We automated privacy policies that can be codified into our analysis (e.g. enforce data with access control). Before we had proof that PrivacyCAT generates low level of false positives (see Fix Rate section below), we had a semi-automated system to review the reported issues ourselves. After tuning and improving the analysis at a point where we could confirm the low false positive rate, we shadowed the signals in CI and rolled out gradually, mitigating therefore the risk in providing a disruptive experience to WhatsApp developers. We also found prod matching is very effective in filtering out false positives caused by the differences between production and testing environments. The prod volume is also useful in deciding the signal severity to help code owners prioritize fixing of the vulnerability.

Fix Rate. One main challenge to apply large-scale code analysis techniques in industry is the potential low fix rates. It's widely reported that code analysis tools suffer from false positives [7, 9].

Table 1 shows that PrivacyCAT's diff-time fix rate is 34.6% and master fix rate is 32.3%. We couldn't find prior industrial studies on detecting privacy vulnerability at large-scale to perform fair benchmarking. However, if we do not restrict to the privacy domain, PrivacyCAT fix rate is higher than a related large-scale study on the usage of SonarQube [27], which reported a fix rate of 13%. When compared with our previous work on FAUSTA, PrivacyCAT scores lower on diffs (FAUSTA has a fix rate of 74%) and it scores higher on master where FAUSTA has a fix rate of 20%.

To measure PrivacyCAT's False Positive (FP) rate, we asked developers to annotate the reported tasks if they are false positive. With this method we found that false positive rate is 10.6%. This meets developer needs (<15%-20%) as suggested by the empirical study [9]. Given the relatively low FP rate, one pertinent question to ask is why developers did not fix the flagged risks. Based on developers' feedback and our investigation, we found the following two main reasons, which indicate the ground truth of false positive rate is likely higher than the measured 10.6% based on explicit developer feedback/annotations.

- **Work in progress**: some reported vulnerability tasks were still in an open state, which may need non-trivial effort to investigate. We found 33 tasks were in such a state at the time of writing.
- **Prior approvals**: Developers sometimes acknowledged PrivacyCAT reported signals, but they do not need fixes due to expected code behaviors. For example, error path logging may be classified as legit for the debugging purpose. There are also exceptional cases previously reviewed and approved by internal audits. Such prior knowledge were usually unavailable in advance and hard to codify in machine-readable properties. Although the signals are worth reporting from PrivacyCAT's perspective, they should not be classified as real policy violations.

6.5 Challenges

In this section we discuss a few challenges and their mitigation, based on the multi-year development, deployment and maintenance of PrivacyCAT.

Privacy in privacy analysis. One aspect we initially underestimated is the importance of enforcing privacy when performing the analysis itself. Some code analysis techniques need to access production traffic or instrument production code, which introduces privacy concerns on user data. PrivacyCAT itself wouldn't have been deployed, if it would require changes to production code, or access user data.

Instead we designed and implement PrivacyCAT analysis to be privacy-safe:

- it does not perform changes to production code and/or environment; and
- it does not rely on user data at all.

PrivacyCAT dynamic analysis simulates user traffic and interactions with artificially generated inputs. It performs most of its analysis in isolated, sandbox environments. This makes PrivacyCAT privacy-safe from the user point of view.

Codifying privacy policies. Previous research on mobile app privacy mainly focused on detection approaches rather than privacy requirements or policies [14]. In our practice, similar to the test oracle problem, discovering privacy requirements and codify them into oracles is the key to PrivacyCAT's success. The problem has been of primarily importance for us to collaborate closely with WhatsApp developers. From them we learned domain knowledge about product features. Moreover we collaborated closely with privacy stakeholders to clarify the nuances of documented policies in natural language. The latter process requires continuous iteration because precise machine-readable properties consumable by both static and dynamic analyses need to be produced from informal policies expressed in natural language. We also built a feedback loop to revise our existing codified policies based on developer feedback, plus tracing any missing privacy SEVs that PrivacyCAT policies missed and should cover in the future.

7 Related Work

Static and dynamic code analysis techniques play a critical role in helping developers build more reliable, performant, secure, and private systems. In the past few decades, their methodologies and applications have been widely studied, for use cases such as identifying reliability errors, code smells, performance, security and privacy risks. [10–12, 21, 22, 26]. In the privacy domain there is a wide body of academic research. Most of this work has seen very limited application or deployment in large-scale industry environments.

On the static analysis side approaches have been developed to analyze code structures, dependencies, and dataflows to identify potential privacy vulnerabilities without executing the code.

Slavin et al. [33] propose a framework to detect privacy policy violations in Android apps. The framework adopted information flow analysis (implemented by the static taint-analysis tool FlowDroid [5]) to detect misalignment between documented policies and actual API behaviors on sensitive data processing. PTPDroid [34] is built on top of FlowDroid, and it checks for privacy policy violations related to 3rd party disclosure. It uses a set of standard APIs as sources and sinks (e.g., network calls).

Gibler et al. [17] present AndroidLeaks to statically find privacy leaks, which they define as "any transfer of PII off of the phone". They consider some permissions as sources (i.e., location) and others as sinks (i.e., internet), then they map permissions to API calls to get a lists of functions to use as sources and sinks. AndroidLeaks performs flow-to style static taint analysis for tracking private data between sources and sinks. Their definition of privacy leak does not attempt to distinguish between intended and unintended leaks. Whereas our work focuses on finding only unintended leaks that should be fixed. They only use static analysis for android apps, while out approach uses multiple analysis across different systems. We also report on integration into CI and how developers reacted to the issues we found.

Egele et al. [15] introduce a system called PiOS for detecting privacy threats that pose to iOS app users. PiOS performs static data flow analysis in binary code compiled from Objective-C source.

Zoncolan [23] is static taint analysis deployed at large scale at Meta. While Zoncolan is only static taint analysis designed for detection of security vulnerability on Hack code, PrivacyCAT is both static and dynamic taint analysis used to detect privacy leakage on Erlang, Android, and iOS.

The dynamic code analysis approach monitors the program during runtime and provides information about how the subject behaves in real-world scenarios, especially those may arise due to unforeseen conditions or unexpected user inputs. PrivacyCAT's dynamic taint analysis is built on top of FAUSTA [26], which is an algorithmic input generation system based on traffic specifications. FAUSTA was initially developed for reliability testing of large-scale services at WhatsApp. Tran et al. [35] introduce a principal-based taint tracing approach. It enables dynamic analysis of JavaScript runtime with limited performance overhead. Enck et al. [16] propose TaintDroid to perform dynamic taint tracking of sensitive data for Android apps. It supports simultaneous variable-level tracking of multiple sources and their information flows within the Android Dalvik VM interpreter.

The combination of dynamic and static code analysis approaches have yielded significant advancements in recent years due to its potential in generating results with less false negatives and false positives. Marescotti et al. [28] show early experiments on boosting static analysis implemented by Infer with dynamic analysis data obtained from FAUSTA and human-authored tests. The results show complementary data flows were detected which would otherwise be missed if perform the two analyses in isolation. Papageorgiou et al. [32] leverage both static and dynamic approaches to evaluate mobile health apps' privacy and security state of practice. The study revealed major concerns that only few apps followed well-established data practices and guidelines. Andow et al. present PoliCheck [3] to support flow-to-policy analysis that can differentiate the entity that receives sensitive data, which is especially useful for detecting unauthorized data sharing to third-parties. The implementation uses PolicyLint [2] for static analysis of contradictory data sharing and collection practices, and AppCensus [4] for dynamic analysis of Android apps.

8 Conclusions

In this paper, we described PrivacyCAT, a scalable code analysis system developed at WhatsApp to automatically detect privacy issues in code. PrivacyCAT uses both static and dynamic analysis to improve the accuracy of results and increase coverage of the code under analysis. PrivacyCAT has been deployed since early 2021 and checks WhatsApp code bases multiple times a day and at every code change. The tool has been successful in automatically identifying the majority of privacy incidents related to sensitive data collection and sharing, enabling WhatsApp developers to promptly resolve these issues. Moreover, it has prevented thousands of privacy issues from being introduced into the WhatsApp codebases.

9 Acknowledgments

We would like to thank WhatsApp and Meta engineering leadership for supporting this work. We want to acknowledge our colleagues for their help while designing, developing, deploying and maintaining PrivacyCAT.

References

[1] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook. In *Search-Based Software Engineering (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 3–45. https://doi.org/10.1007/978-3-319-99241-9_1

[2] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. 2019. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play.. In *USENIX Security Symposium*. 585–602.

[3] Benjami Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. 2020. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*.

[4] AppCensus. 2023. App Search. https://www.appcensus.io/search. Accessed: 2023-05-29.

[5] 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.

[6] Andrea Avancini and Mariano Ceccato. 2010. Towards security testing with taint analysis and genetic algorithms. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*. 65–71.

[7] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[8] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[9] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 332–343. https://doi.org/10.1145/2970276.2970347

[10] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[11] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 269–282.

[12] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

[13] Gareth Eason. 2016. Incident Response @ FB, Facebook's SEV Process. USENIX Association, Dublin.

[14] Fahimeh Ebrahimi, Miroslav Tushev, and Anas Mahmoud. 2021. Mobile app privacy in software engineering research: A systematic mapping study. *Information and Software Technology* 133 (2021), 106466. https://doi.org/10.1016/j.infsof.2020.106466

[15] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting privacy leaks in iOS applications.. In *NDSS*. 177–183.

[16] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[17] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*. Springer, 291–307.

[18] Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. 2013. Runtime Verification Based on Register Automata. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 260–276. https://doi.org/10.1007/978-3-642-36742-7_19

[19] 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*. 33–39.

[20] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

[21] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of Android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[22] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.

[23] 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/.

[24] 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 45th International Conference on Software Engineering: Software Engineering in Practice* (Melbourne, Australia) *(ICSE-SEIP '23)*. IEEE Press, 315–326. https://doi.org/10.1109/ICSE-SEIP58684.2023.00034

[25] 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*. 94–105.

[26] 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

[27] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are static analysis violations really fixed? a closer look at realistic usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 209–219.

[28] Matteo Marescotti, Ákos Hajdu, Dino Distefano, and Ke Mao. 2023. Boosting Static Analysis with Dynamic Runtime Data at WhatsApp Server. In *Proc. of ICSE'2023 (Industry Forum)*.

[29] Wes Masri, Andy Podgurski, and David Leon. 2004. Detecting and debugging insecure information flows. In *15th International Symposium on Software Reliability Engineering*. IEEE, 198–209.

[30] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. Taintpipe: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*. 65–80.

[31] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.. In *NDSS*, Vol. 5. Citeseer, 3–4.

[32] Achilleas Papageorgiou, Michael Strigkos, Efthimios Politou, Eugenia nd Alepis, Agusti Solanas, and Constantinos Patsakis. 2018. Security and privacy analysis of mobile health applications: the alarming state of practice. *Ieee Access* 6 (2018), 9390–9403.

[33] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*. 25–36.

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

[35] Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang. 2012. Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations. In *International Conference on Applied Cryptography and Network Security*. Springer, 418–435.

[36] Zhemin Yang and Min Yang. 2012. Leakminer: Detect information leakage on Android with static taint analysis. In *2012 Third World Congress on Software Engineering*. IEEE, 101–104.