Compositional Static Callgraph Reachability Analysis for WhatsApp Android App Health

Ákos Hajdu
Meta
London, UK
akoshajdu@meta.com

Roman Lee
Meta
Vancouver, Canada
romanlee@meta.com

Gavin Weng
Meta
Menlo Park, USA
gavinweng@meta.com

Nilesh Agrawal
Meta
Menlo Park, USA
niles@meta.com

Jérémy Dubreil*
Unaffiliated
Nantes, France
jeremy.dubreil@lacework.net

Abstract

We report on an industrial use case of static callgraph reachability analysis to improve WhatsApp Android app health. We collaborated with engineers dedicated to app health to annotate/specify the source code. We leveraged the Infer static analyzer to prevent regressions during code changes and to periodically find pre-existing issues on the latest revision. Within three months, the analysis prevented almost a hundred regressions from being introduced and resulted in fixes for a handful of pre-existing issues, including examples with end-user measurable impact.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Keywords: Static analysis, callgraph reachability

ACM Reference Format:
Ákos Hajdu, Roman Lee, Gavin Weng, Nilesh Agrawal, and Jérémy Dubreil. 2025. Compositional Static Callgraph Reachability Analysis for WhatsApp Android App Health. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3735544.3735583

1 Introduction

WhatsApp is a messaging application serving more than 2 billion users in over 180 countries every day. One of WhatsApp's goals is to provide fast and reliable communication, including lower-end consumer devices and poor network conditions. In order to achieve this goal, special emphasis needs to be put on app health and code quality. In this paper we describe a static callgraph reachability analysis on WhatsApp Android app's codebase. The analysis finds potential issues where certain performance critical functions can end up (transitively) calling computationally expensive functions. In order to deploy such an analysis, we needed (1) to specify functions of interest (performance critical and computationally expensive ones), and (2) an automated tool that takes the specification and the source code, and performs the analysis.

Specification is a well-known challenge in program analysis: it is often hard to motivate developers to write specifications, especially at a large scale in a fast-paced company. We tackled this challenge by collaborating with engineers from a team specifically working on app health to provide an initial set of specifications, which was then expanded by other developers as we gradually rolled out the analysis.

The main challenge for the analysis is that it needs to be fast enough to give timely feedback on code changes submitted by developers, while also being sufficiently sound and precise. To tackle this, we leveraged Infer [3, 4], a static analysis platform with multiple language frontends and analyzers (also called checkers). Infer had a so-called *annotation reachability* checker, which could perform static callgraph reachability analysis based on in-code annotations. A distinguishing feature of this checker is that it does not construct and traverse a global callgraph, but uses Infer's compositional and interprocedural abstract interpretation framework. This checker's functionalities were mostly fit for our purposes, but it was mostly unused and unmaintained. We "revived" this checker and added some new features needed for our use case (e.g., supporting regexp-based specification of library functions). In addition, we formalized the algorithm of the checker in this paper for the first time.

We deployed the analysis to run on every code change (*diff*) to prevent regressions, and also to run periodically on the latest revision to find pre-existing issues. Over a period of 3 months, Infer reported 174 issues on diffs with 92 being fixed (53% fix rate). Furthermore, 7 pre-existing issues were also fixed, which is uncommon, as Infer has traditionally

*The author was affiliated with Meta while contributing to this work.

SOAP '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735583

been more successful at diff-time deployments [6]. One notable example issue found by Infer resulted in measurable impact for end-users: app not responding (ANR) issues were reduced by 0.56% in particular regions, and chat loading got 1.25% faster globally.

2 Background

Reachability. Given a program P consisting of a set of functions F and sets of distinguished *source* functions $F_{src} \subseteq F$ and *sink* functions $F_{snk} \subseteq F$, the goal of reachability analysis is to check if any source function can (potentially transitively) call a sink function, that is, whether an execution exists in P with a sequence of calls $f_1 \to f_2 \to \ldots \to f_n$ for some $n > 1$, where $f_1 \in F_{src}$, $f_n \in F_{snk}$, and $f_i \in F$ for $1 \leq i \leq n$. Reachability can be extended with an additional set of *sanitizers* $F_{san} \subseteq F$ imposing an extra condition $f_i \notin F_{san}$ for $1 \leq i \leq n$. In other words, we are not interested in call sequences that pass through any sanitizer function.

Example. Consider the program in Figure 1 with `somethingOnTheUI` being the single source and `readFromDatabase` being the single sink. Then, a sequence of calls `somethingOnTheUI` → `checkSomething` → `readFromDatabase` exists from source to sink. However, if we consider `checkSomething` to be a sanitizer, then there are no source-to-sink call sequences anymore.

```
1 @PerfCrit void somethingOnTheUI() {
2   checkSomething();
3 }
4 void checkSomething() {
5   readFromDatabase();
6 }
7 @Expensive void readFromDatabase() {
8   /* Slow stuff */
9 }
```

Figure 1. Example code with annotated functions.

Static callgraphs. As usual with non-trivial program properties, callgraph reachability is an undecidable problem[1] in theory. In this work, we use *static callgraphs* to solve reachability in an approximate way. The static callgraph of a program P is a directed graph $G = (F, E)$ where the vertices F consist of the functions of the program, and there is a directed edge $(f_1, f_2) \in E$ between functions $f_1, f_2 \in F$ if the body of f_1 contains a call instruction of the form $f_2(\ldots)$. An approximate solution to the reachability problem is to check if a path $f_1 \to \ldots \to f_n$ exists in G where $f_1 \in F_{src}$, $f_n \in F_{snk}$, and $f_i \notin F_{san}$ for $1 \leq i \leq n$.

This approximation can be solved, e.g., by computing a transitive closure of the graph [7]. However, it can report paths that are not feasible in any actual execution (e.g., calls guarded by infeasible conditions) and also miss real call chains (e.g., if dynamic dispatch or lambdas are involved). For us, this is acceptable: we are not aiming for a sound over-approximation. False negatives are acceptable as long as the analysis is fast enough to run on every code change, and can detect enough real issues with a sufficiently low level of false positives to make an impact in practice.

3 Reachability Analysis with Infer

Infer [3, 4] is a static analysis framework that supports multiple language frontends (including Java and Kotlin) and analyzer backends (checkers) through a common intermediate language [2]. One of Infer's checkers is called *annotation reachability*[2] which can perform the aforementioned reachability analysis of programs based on the static callgraph; with a few practical differences and extensions. This checker has been around in Infer for a while, originally developed to solve similar problems on other Android apps, but – to the best of our knowledge – this is the first paper to present it in a formal way. As described later in this section, a key feature of this algorithm is that it does not construct a global callgraph, but rather uses compositional reasoning and abstract interpretation. We have also added some improvements and extensions to the checker to support our particular use case at WhatsApp (Section 3.4).

3.1 Specification

As the name of the checker suggests, the sources, sinks and sanitizers can be defined via sets of Java *annotations*. Given the set of all annotations A, let $A(f) \subseteq A$ denote the annotations of a function[3] f and let $A_{src}, A_{snk}, A_{san} \subseteq A$ denote the source, sink and sanitizer annotations, respectively. Without the loss of generality, we can assume that no sink or source annotation is a sanitizer at the same time ($A_{src} \cap A_{san} = \emptyset$ and $A_{snk} \cap A_{san} = \emptyset$), as otherwise any path containing them would be immediately sanitized. We can derive the set of source, sink and sanitizer functions as follows: $F_{src} := \{f \in F \mid A(f) \cap A_{src} \neq \emptyset\}$, $F_{snk} := \{f \in F \mid A(f) \cap A_{snk} \neq \emptyset\}$ and $F_{san} := \{f \in F \mid A(f) \cap A_{san} \neq \emptyset\}$. Note that we only assume that a source/sink annotation cannot be a sanitizer annotation; a source/sink function can still be a sanitizer at the same time if it has both source/sink and sanitizer annotations.

Additionally, the checker takes as input multiple sets of sources, sinks and sanitizers $(A^1_{src}, A^1_{snk}, A^1_{san}), \ldots, (A^k_{src}, A^k_{snk}, A^k_{san})$, treating each tuple as an independent reachability *property* (but solving all of them in one pass). For example, the configuration in Figure 2 defines a single property with one source, one sink and no sanitizers: ($A^1_{src} = \{$`@PerfCrit`$\}$, $A^1_{snk} = \{$`@Expensive`$\}$, $A^1_{san} = \emptyset$).

[1] We can append a synthetic function call *end*() to the end of a program P and the reachability of *end*() is equivalent to the halting of P.

[2] https://fbinfer.com/docs/checker-annotation-reachability

[3] Infer's Java/Kotlin frontend also looks at annotations from the enclosing class and from overridden functions from base classes / interfaces.

```
"annotation-reachability-custom-pairs": [{
  "sources": ["PerfCrit"],
  "sinks": ["Expensive"]
}]
```

Figure 2. Infer's annotation reachability configuration.

3.2 Analysis

The checker does not explicitly construct a global callgraph to traverse. Instead, it formulates the problem in Infer's compositional and interprocedural abstract interpretation framework. Infer analyzes each function independently by propagating an abstract state through its instructions to obtain a *summary* which can then be applied in every calling context (without having to re-analyze the callee). This also means that the analysis does not need an entry point or a main function: it can start from a set of arbitrary functions, and dependencies are analyzed *on-demand*. As demonstrated in the results (Section 4.3), this allows us to run the checker on every code change at the scale of WhatsApp within a reasonable time. However, recursion needs a special treatment (due to circular dependency in the analysis): either a fixed-point computation, or unrolling to a fixed depth. Infer currently does an unrolling to a depth of one,[4] i.e., performing an under-approximation.

Abstract domain. In annotation reachability, an abstract state $s \subseteq F \times \mathbb{N} \times F \times A$ is a set of tuples, where a tuple $(g, n, h, a) \in s$ represents the information that the current function f being analyzed calls some function g on line n, which then ends up calling sink h, annotated with $a \in A^i_{snk}$ for some i. If $g = h$ in a tuple, it means that the current function f calls a sink *directly*, whereas $g \neq h$ means that the call is *transitive*. Intuitively, this abstract domain implies that we only compute the relevant parts of the callgraph (that can potentially be part of a source-to-sink call chain), and each node only stores a single step towards a given sink. Paths can be reconstructed later, and we have the option to limit the number of paths per source/sink pair, allowing us to scale to large codebases.

Instructions are traversed from the beginning of the function in-order, however, as described below, the join operation and the transfer function is defined in a path- and flow-insensitive way for scalability.[5] The initial state $s_0 := \emptyset$ is the empty set, the join operation is set union $join(s_1, s_2) := s_1 \cup s_2$, and the summary of a function f – denoted by $sum(f)$ – is the abstract state at the exit point[6] of f.

The transfer function $T(s, instr_n)$ takes an abstract state s (the precondition) and an instruction $instr_n$ at line n, and returns a new abstract state that encapsulates the effects (postcondition) of executing $instr_n$ on s. If $instr_n$ is not a call, or the callee cannot be resolved statically (e.g., some unknown external function), then $T(s, instr_n) := s$, that is, the state does not change. If $instr_n$ is a call to some function g then the state gets extended with *direct* and *transitive* entries corresponding to g. Direct entries s_d are added if g itself is a sink, and transitive entries s_t are added for each entry in the summary of g (unless f or g is a sanitizer). Formally, $T(s, instr_n) = s \cup s_d \cup s_t$ where

- $s_d := \{(g, n, g, a) \mid a \in A(g) \wedge \exists i.\, a \in A^i_{snk} \wedge f, g \notin F^i_{san}\}$ and
- $s_t := \{(g, n, h, a) \mid (h', n', h, a) \in sum(g) \wedge \exists i.\, a \in A^i_{snk} \wedge f, g \notin F^i_{san}\}$.

3.3 Reporting

Once we finish analyzing a function f and it happens to be a source ($f \in F^i_{src}$ for some i), we check its summary $sum(f)$ to report if some sink g can be reached. In practice, we can have multiple sets of sources/sinks/sanitizers so we also report which exact annotations are responsible for the call chain. Formally, we report $\{(f, a_f, g, a_g) \mid (h, n, g, a_g) \in sum(f), \exists i, a_f.\, a_f \in A^i_{src} \wedge a_g \in A^i_{snk} \wedge f \notin F^i_{san}\}$. A reported tuple (f, a_f, g, a_g) should be interpreted as "source f annotated with a_f calls sink g annotated with a_g". In Infer's terminology, this is called an *issue*.

Summaries also allow us to reconstruct all distinct paths between a source and a sink. However, in practice, this can lead to an exponential number of paths, so the checker only reports one path per issue. A path for an issue (f, a_f, g, a_g) can be reconstructed recursively:

- $path(f, g, a_g) := f \rightarrow g$ if $\exists n \in \mathbb{N}$ with $(g, n, g, a_g) \in sum(f)$,
- $path(f, g, a_g) := f \rightarrow path(h, g, a_g)$ for some entry $(h, n, g, a_g) \in sum(f)$ otherwise. In practice, the checker picks the entry with the lowest n, i.e., the first callsite that leads towards the sink.

Consider the example code and configuration in Figures 1 and 2 and let us abbreviate each function with its initial letter. The summary $sum(\texttt{r}) = \emptyset$ is empty because `r` does not contain any call. The summary $sum(\texttt{c}) = \{(\texttt{r}, 5, \texttt{r}, \texttt{@Expensive})\}$ contains a single direct call to a sink: `c` calls `r` directly on line 5. The summary $sum(\texttt{s}) = \{(\texttt{c}, 2, \texttt{r}, \texttt{@Expensive})\}$ contains a single transitive call to a sink: `s` calls `r` indirectly via the call to `c` on line 2. The only source is `s`, so Infer would report a single issue: (`s`, `@PerfCrit`, `r`, `@Expensive`).

[4] Infer used to have fixed-point computation, but in practice, using an unrolling to a depth of one still found the majority of the relevant issues in a simpler and more efficient way, that is, the benefits of deeper unrolling were too small compared to the performance loss.

[5] We have also experimented with a path- and flow-sensitive underapproximate checker called Pulse [13], but within our time constraints, it had too many false negatives.

[6] Infer represents functions as control-flow graphs with one common exit node.

The path is reconstructed as $path(\texttt{s}, \texttt{r}, \texttt{@Expensive}) = \texttt{s} \rightarrow path(\texttt{c}, \texttt{r}, \texttt{@Expensive}) = \texttt{s} \rightarrow \texttt{c} \rightarrow \texttt{r}$.

3.4 Checker Improvements

In order to apply Infer to our use case at WhatsApp Android, we had to make some improvements to the annotation reachability checker itself.

Regular expressions. We extended the checker so that functions can be treated (*modeled*) as if they were annotated, based on regular expressions over their (fully qualified) names. This was necessary for library functions (e.g., Android or Java libraries) where the source code is external and annotations could not be added directly in the code. This allows us to detect call paths where the source is in the application code, but the sink is several call steps inside a library. In addition, this also helped to specify a larger set of functions in a concise way (e.g., marking all UI event handler overrides as sources). Figure 3 shows an example, where the configuration specifies that any method (`.*`) of class `MyClass` in package `com.library` should be treated (modeled) as if it was annotated with `@Expensive`. As a side benefit, this opened up the support for languages without annotations.

```
"annotation-reachability-custom-models": {
 "Expensive": ["com\.library\.MyClass\..*"]
}
```

Figure 3. Using regular expressions to model annotations.

Path minimization. Sources or sinks can also appear as intermediate steps in a path. Such paths contain other paths as a sub-path and it might be desirable to only report the shortest, because eliminating that call chain also removes the extended paths. We formalize this by defining minimal source and sink paths. A path $f_1 \rightarrow \ldots \rightarrow f_n$ (where $f_1 \in F_{src}$ and $f_n \in F_{snk}$) is *source-minimal* if $f_i \notin F_{src}$ for $i > 1$ and *sink-minimal* if $f_i \notin F_{snk}$ for $i < n$. In other words, the only source in a source-minimal path is the first function, and the only sink in a sink-minimal path is the last function. In practice though, we are using annotations: given an issue $(f_1, a_{src}, f_n, a_{snk})$ with a path $f_1 \rightarrow \ldots \rightarrow f_n$, we check if f_1 is the only function of the path annotated with a_{src} (source-minimal) and if f_n is the only function annotated with a_{snk} (sink-minimal). Formally, $a_{src} \notin A(f_i)$ for $1 < i \leq n$ must hold for source-minimal paths, and $a_{snk} \notin A(f_i)$ for $1 \leq i < n$ must hold for sink-minimal paths. Note that we only compute one path for each issue, so an issue is source/sink-minimal if and only if its single corresponding path is source/sink-minimal.

Loop highlighting. Consider a source to sink path $f_1 \rightarrow \ldots \rightarrow f_n$ where f_i calls f_{i+1} in a loop (for some index i). We wanted to flag such paths in a special way because executing something computationally expensive within a loop can have a higher impact than just executing it once. Infer's intermediate language is based on control flow graphs (CFG) and there is an implementation of Tarjan's strongly connected components algorithm [16] to compute loops. We added an option to the analyzer that checks for each edge $f_i \rightarrow f_{i+1}$ whether the CFG node in f_i that contains the call to f_{i+1} is part of a loop. If yes, the edge $f_i \rightarrow f_{i+1}$ is highlighted with a special tag in the trace during reporting.

4 Reachability Analysis for WhatsApp Android

The main goal of the use case at WhatsApp Android is to detect potential issues where certain performance critical functions can end up (transitively) calling computationally expensive functions. To achieve this goal, we partnered with engineers from WhatsApp to annotate the code, configure and deploy the checker, gather feedback and make improvements in an iterative process.

4.1 Specification and Configuration

As of writing the paper, there are two reachability properties (sets of sources, sinks and sanitizers) with 8 annotations and 11 regular expressions in total (see breakdown in Table 1). The sources, sinks and sanitizers are defined by Android engineers, including a dedicated team responsible for the app's health. In the first property, source annotations usually refer to something performance critical, such as event handlers on the user interface (overriding functions from Android UI libraries). Sink annotations correspond to (potentially) computationally expensive functions, such as worker threads and file operations (in Java libraries). Sanitizers include functions that are not shipped in production, such as tests, debug utilities (e.g., logging) and Java preconditions. The second property aims to find call chains where the source and sink has incompatible threading annotations (e.g., worker thread as source calling main thread as sink).

Table 1. Number of annotations and regular expressions with their coverage over WhatsApp Android's functions.

Prop.	Annotations			Regexps			Coverage		
	src	snk	san	src	snk	san	src	snk	san
#1	1	2	2	2	3	3	0.356%	2.689%	0.198%
#2	2	1	0	0	3	0	16.267%	0.002%	0.000%

Specifications partially reused existing annotations (e.g., `@WorkerThread`), but some annotations were added in-code just for the purpose of this checker (e.g., known hot spots related to chat loading). One disadvantage of using regular expressions in Infer's configuration file is that it can get out of sync with the codebase. However, we mostly used

regular expressions for well-established Android/Java library functions that we don't expect to change often. Table 1 shows the percentage of WhatsApp Android's functions that are covered as source, sink or sanitizer by the two properties. The second property has a significantly higher coverage due to reusing more of the threading annotations already present in the code. Note that at the scale of WhatsApp Android's codebase, even small percentages amount to a non-trivial number of functions.

4.2 Deployment

Infer has two kinds of deployments in general: continuous and diff-time. *Continuous* scans analyze the latest revision of a repository periodically (typically a few times every day). With a few exceptions where tasks are filed and triaged to code owners [8, 11], these runs are only visible on internal dashboards for Infer developers to get an idea on the baseline of pre-existing issues and to experiment with new checkers. *Diff-time* deployments, on the other hand, analyze every code change (*diff*) and report newly introduced issues inlined as comments during the code review process (along with feedback from other automated tools and human reviewers). Developers can then submit a new version of the diff, triggering a new run of Infer. We measure *fix rate* by checking if issues reported in an intermediate version of the diff disappear in the final version that gets merged to the main branch. Note that fix rate is only a proxy to estimate true positive rate, but in practice it works well because (1) it can be measured automatically, and (2) it indicates the actionability of the checker from a developer's perspective.

Continuous analysis. We first deployed continuous runs of the reachability analysis, allowing us to get an idea on the baseline of pre-existing issues and to iteratively add more annotations, fine tune the checker, and make improvements to Infer. Overall, most of the issues reported by Infer were technically true positives (a path from source to sink indeed exists). However, without path minimization, Infer reported around[7] 12500 issues, which was deemed overwhelming, even if they were all true positives. Applying source and sink minimization individually reduced the number of issues to approximately 10600 and 1600 respectively. Applying both resulted in roughly 1100 issues. We concluded that sink minimization is sufficient as it already reduces the issues to a manageable volume. In addition, sources are usually closer to the business logic of code owners, whereas sinks are typically in common code or lower level libraries. Therefore, not minimizing on the source side allows engineers to fix the issues in (or closer to) the code they own. While this might not fix all related issues at once, developers are usually more confident in changing the code they own. Engineers also added further sanitizers to suppress certain reports that were technically true positives, but still acceptable because of known mitigating factors.

Diff-time analysis. Once we were satisfied with the quality and volume of reported pre-existing issues, we started rolling out the analysis on diffs. We first enabled a so-called *shadow mode* where the analysis was running for all code changes, but issues only showed up on an internal dashboard for us and the app health team to assess. Then we gradually enabled reporting for a few dozen engineers who opted in as early adopters. We set up channels to gather feedback and monitored fix rate. After a few weeks, we rolled out the analysis to all WhatsApp Android engineers.

4.3 Results

Pre-existing issues. Interestingly, as engineers examined the reports from continuous scans, they have found various pre-existing issues that were likely to have a high impact on performance. For such issues, they created tasks, first manually, then later in a semi-automated way, and triaged them to the appropriate code owners to consider fixing. As of writing the paper, 59 tasks have been filed with 7 fixed already (closed with a diff attached). One notable example was directly linked to ANRs (app not responding) from production logs. Infer's call trace provided insight to why the ANRs are happening. The fix reduced ANRs by 0.56% in certain regions where the feature was enabled and also resulted in an end-user measurable 1.25% speedup in chat loading speed globally.[8]

Diff-time reports. Over a period of 3 months, Infer reported 174 issues on code changes in total, with 92 being fixed (53% fix rate). Our analyis is on par with other Infer deployments, typically having a fix rate between 40% and 60%. We looked at some of the unfixed issues and categorized them in the following main groups.

There are diffs that added further source/sink annotations with the intention to be able to see pre-existing issues (without fixing them) and to prevent newly introduced issues. Such diffs caused Infer to find new issues by design.

We have also seen diffs that converted certain modules from Java to Kotlin. Infer has some mechanisms to identify if an issue moves around (e.g., lines shifting, or class renamings), however, moving to a new language was beyond the capabilities of these heuristics.

We encountered false positives due to pre-existing issues being reported as if they were newly introduced. One notable category was related to non-deterministic analysis in case of mutually recursive function calls. As an example, consider a static callgraph of the form $src \rightarrow f \rightleftarrows g \rightarrow snk$. As mentioned before, Infer analyzes recursive calls to a depth of one, which can intuitively be thought of as having to "cut"

[7] Due to the codebase evolving, and internal timeouts, there is a small fluctuation in the number of issues reported on the latest revision.

[8] Measured by a controlled experiment where one group had the fix enabled and the other group did not have it.

edges from the callgraph until it becomes acyclic. It is clear that if $f \leftarrow g$ is cut, Infer still sees the issue (path from source to sink exists), but if $f \rightarrow g$ is cut, then the issue is not found. The edge to be "cut" depends on the order in which functions are scheduled to be analyzed, which can have some non-determinism, especially if the surrounding code changes. On each diff, Infer runs twice: once for the current revision and once for the parent, and computes the difference of issues to only report newly introduced issues. Our observations suggest that a different scheduling order for the current/parent revision can cause an issue to be only seen in one of the runs. As a quick workaround, we mitigated this by switching to a more deterministic scheduler (the issue is either found or missed, but it is consistent across runs). However, if this causes too many missed issues, we can consider bringing back Infer's fixed-point computation (the issue is found regardless of scheduler ordering).

Finally, we have seen false positives related to flow- and path-insensitivity. Two simplified examples are presented in Figure 4: a call to a sink is guarded by a condition that prevents it from happening in production, and a pair of special function calls "enclose" a block that should be treated as a sanitizer. These could be mitigated by making the analysis flow- and path-sensitive, but that would come with its own downsides (e.g., path explosion).

Note that we did not explicitly count false positives, we only sampled unfixed issues and use the fix rate as a measure of checker quality instead.

```
void source() { // Needs path-sensitivity
  if (is_debug()) sink();
}
void source() { // Needs flow-sensitivity
  beginSanitizing();
  sink();
  endSanitizing();
}
```

Figure 4. Example false positives.

Discussion. Currently, we mostly rely on engineers' domain knowledge to identify which issues have a truly significant impact on performance. Post-fix analysis is possible via experiments, as described earlier with the example directly linked to ANRs. We have also not yet measured whether loop highlighting contributes to higher rate of fixes. In the future we plan to improve actionability and prioritization by using runtime tracing data to identify and flag if a reported call chain is on a hot path (executed often).

Analyzer performance. It is not straightforward to independently assess the performance of the analyzer, because production deployments of Infer (1) rely heavily on caching when compiling the source code into the intermediate representation, and (2) have multiple checkers running in parallel (not just annotation reachability). Overall, the p90 compilation and analysis times for continuous runs are 33 and 53 minutes, respectively, including all checkers and the full WhatsApp Android repository, consisting of millions of lines of Java and Kotlin code. Annotation reachability alone (without compilation) takes around 15 minutes. On diffs, the p90 total execution time of Infer is 32 minutes, including compilation and analysis for parent and current revisions, with all checkers, but limited to the changed files[9] (and their dependencies). The key takeaway from these numbers is that the analysis is fast enough (1) to run multiple times a day in continuous mode, (2) to provide timely feedback on developers' diffs, and (3) annotation reachability is not a bottleneck.

5 Related Work

Samhi et al. [14] survey various Android static analysis tools that compute a callgraph. They show that tools fail to cover at least 40% of the calls that can happen in runtime. Infer – including our analyzer – targets bug finding and not soundness, and even with our static approach we are finding interesting bugs. Nevertheless, it would be interesting to compute metrics on coverage and missed bugs.

There are various static taint analysis tools for Android, such as FlowDroid [1] or Difuzer [15]. While these tools employ a callgraph in the background, the primary purpose is to track data flow. ACID [10] relies on callgraphs to detect API invocation and callback incompatibility issues. ARPDROID [5] and NatiDroid [9] focus on security properties (permissions for API invocations). NatiDroid also extends control-flow analysis to native libraries.

Yang et al. [17] proposes a method to enhance the control flow graph with edges corresponding to events and callbacks, which are traditionally missing from static graphs. In our use case, event handlers are usually the entry points of a path (sources) so we don't have to track where they are invoked from. Midtgaard and Jensen [12] approximate interprocedural control-flow using abstract interpretation. Their main focus is on first class functions and tail call optimization, which are not prevalent at the moment in our use case.

6 Conclusions

We reported on our experience on applying static callgraph reachability analysis on WhatsApp Android's codebase to improve app health and performance. Just within its first 3 months, the analysis prevented 92 issues from being introduced, and resulted in fixes for 7 pre-existing issues, including an example with end-user measurable impact.

[9] As described in Section 3.2, Infer's checkers are modular and compositional: they can start from an arbitrary set of functions – for example, functions in the files touched by the code change – and transitively find and analyze their dependencies as needed.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (2014), 259–269. doi:10.1145/2666356.2594299

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 115–137. doi:10.1007/11804192_6

[3] 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. doi:10.1007/978-3-642-20398-5_33

[4] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 3–11. doi:10.1007/978-3-319-17524-9_1

[5] Malinda Dilhara, Haipeng Cai, and John Jenkins. 2018. Automated detection and repair of incompatible uses of runtime permissions in Android apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 67–71. doi:10.1145/3197231.3197255

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

[7] Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345. doi:10.1145/367766.368168

[8] 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. doi:10.1145/3546186.3549929

[9] Chaoran Li, Xiao Chen, Ruoxi Sun, Minhui Xue, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang. 2022. Cross-language Android permission specification. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 772–783. doi:10.1145/3540250.3549142

[10] Tarek Mahmud, Meiru Che, and Guowei Yang. 2022. ACID: An API Compatibility Issue Detector for Android Apps. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, 1–5. doi:10.1145/3510454.3516854

[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 46th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 106–117. doi:10.1145/3639477.3639742

[12] Jan Midtgaard and Thomas P. Jensen. 2009. Control-flow analysis of function calls and returns by abstract interpretation. *SIGPLAN Not.* 44, 9 (2009), 287–298. doi:10.1145/1631687.1596592

[13] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 12225. Springer, 225–252. doi:10.1007/978-3-030-53291-8_14

[14] Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. 2024. Call Graph Soundness in Android Static Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 945–957. doi:10.1145/3650212.3680333

[15] Jordan Samhi, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2022. Difuzer: uncovering suspicious hidden sensitive operations in Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 723–735. doi:10.1145/3510003.3510135

[16] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. doi:10.1137/0201010

[17] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 89–99. doi:10.5555/2818754.2818768

Received 2025-03-04; accepted 2025-04-25