Enhancing Compositional Static Analysis with Dynamic Analysis

Dino Distefano
Meta UK and Queen Mary University of London

Matteo Marescotti
Meta
London, UK

Cons T Åhs
Meta
London, UK

Sopot Cela
Meta
London, UK

Gabriela Cunha Sampaio
Meta
London, UK

Radu Grigore
Meta
London, UK

Ákos Hajdu
Meta
London, UK

Timotej Kapus
Meta
London, UK

Ke Mao
Meta
London, UK

Thibault Suzanne
Meta
London, UK

Abstract

In this paper we introduce a novel method for improving static analysis of real code by using dynamic analysis. We have implemented our technique to enhance the Infer static analyzer [6] for Erlang by supplementing its analysis with data obtained by FAUSTA [24] dynamic analysis. We present the technical details of the algorithm combining static and dynamic analysis and a case study on its evaluation on WhatsApp's Erlang code to detect software defects. Results show an increase in detected bugs in 76% of the runs when data from dynamic analysis is used. In particular, on average, data provided by dynamic analysis for 1 function enables static analysis of 2.1 additional functions. Moreover, dynamic data enabled analysis of a property not verifiable using static analysis alone.

ACM Reference Format:
Dino Distefano, Matteo Marescotti, Cons T Åhs, Sopot Cela, Gabriela Cunha Sampaio, Radu Grigore, Ákos Hajdu, Timotej Kapus, Ke Mao, and Thibault Suzanne. 2024. Enhancing Compositional Static Analysis with Dynamic Analysis. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3691620.3695599

1 Introduction

Source code analyses are techniques to check whether the code satisfies a specification or to detect violations of this specification and therefore detect defects in the program. Traditionally there are two kinds of approaches: *static* and *dynamic*.

Static analysis typically builds a model describing a set of potential behaviours that the code will have at run-time. The static analysis can be over-approximating if the model describes a super-set of all the possible behaviours of the program or under-approximating if the model represents a subset. By definition, an over approximating static analysis has spurious behaviours which may lead to false positives. In contrast, an under-approximating static analysis might miss behaviors, leading to false negatives.

Dynamic analysis works by generating inputs, running the program on those inputs, and checking an oracle on the behaviour of the run. Because only a finite set of runs can be executed, dynamic analysis always under-approximates. However, as dynamic runs are concrete, detected bugs are by definition true positives.

Compositionality – such as [8] – allows static analysis to scale to large code bases (e.g. millions of lines of code). At a high-level, compositionality means that the specification of a function is obtained by composing the specifications of the functions it calls (its *callees*), looking at the code of the caller but not looking at the code of the callees. The analysis usually follows the call graph: it starts from the leaves and walks its way up to the root[1].

During a compositional analysis, sometimes the analysis of a procedure $p(\vec{x})$ fails. This has two negative effects: first, no conclusions can be drawn for $p(\vec{x})$; second, the failure has a knock-on effect hindering the analysis of dependent procedures (i.e. those that transitively call $p(\vec{x})$), reducing therefore the overall analysis coverage in a code base.

In this paper, we introduce a novel technique to combine static and dynamic analysis which helps resolve the above shortcoming of compositional static analysis. The idea is to use dynamic analysis to synthesize specifications for those procedures $p(\vec{x})$ for which static analysis fails. This avoids the knock-on effect on procedures that transitively call $p(\vec{x})$, enabling their static analysis. We call *dynamic specifications* those produced by dynamic analysis in contrast to *static specifications* derived by static analysis. Dynamic specifications are obtained with a process of symbolic abstraction from concrete input/output pairs observed by the dynamic analysis while running the program. Dynamic specifications are then used to enhance subsequent runs of static analysis.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695599

[1]There are other scheduling strategies, and a fixed-point computation might be needed when recursion is used. However, for simplicity, in this paper we will assume the bottom-up strategy following the call graph.

We have experimented with our technology by enhancing the PrivacyCAT [23] analysis on WhatsApp's code. PrivacyCAT is a tool developed at WhatsApp that is used internally to automatically detect possible privacy vulnerabilities helping therefore programmers ship more robust code, from the privacy point of view. Experiments at WhatsApp show that the combination between static and dynamic analysis substantially increases the effectiveness of the original static analysis both in terms of software defects detected and coverage. In particular, 76% of the runs with dynamic specifications detect more bugs, and on average, every dynamic specification allows us to infer 2.1 new static specifications.

In summary, this paper makes the following main contributions:

i. It introduces a novel technique to enhance static analysis by means of dynamic analysis. We present the general algorithm and describe its theoretical foundations.
ii. We implemented the technique in a tool able to run hourly on large scale industrial software.
iii. We report an industrial case study at WhatsApp. This case study provides empirical data on a large Erlang code-base showing the effectiveness of the technique.

The remainder of this paper is organized as follows: Section 2 provides background information on both static and dynamic analysis techniques. Section 3 explains the motivation behind this work and how dynamic analysis is used to enhance static analysis. Section 4 details the symbolic abstraction process. Section 5 presents a case study evaluating our approach on real-world industrial code. In Section 6, we provide an overview of related work, and finally, Section 7 concludes with a summary of our findings.

2 Background

PrivacyCAT [23] is a system composed of different code analyzers to check given privacy rules. PrivacyCAT consists of two separate analyses: *static* PrivacyCAT and *dynamic* PrivacyCAT. There are two reasons why both analyses are utilized. First, dynamic and static analyses have different strengths: dynamic analysis only explores real executions; static analysis can see executions missed by the dynamic one; dynamic analysis is easier to extend to cover multi-language codebases, for example including both a client and a server [9]. 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. Before describing how to combine static and dynamic analysis let's give a brief intro of both.

2.1 Static Analysis

The work in this paper is based on the Infer static analyzer [6, 19]. Infer supports various languages (including Java, Kotlin, C/C++, Objective-C, C#, Erlang) and it is used by several software companies. For example, Meta uses it multiple times everyday to analyze most of their code-bases [7, 12]. Infer is based on Abstract Interpretation [11] and implements a compositional, bottom-up interprocedural analysis using function summaries [8]. Its compositional analysis is performed by synthesising summaries for a piece of code in isolation, usually a function. Summaries are Hoare triples where pre/post-conditions are separation logic formulae [8, 20]. More specifically, for a given piece of code C, Infer synthesises a triple of the form

$$\{P\}\ C\ \{Q\}$$

by inferring suitable P and Q which are formulae in a subset of Separation Logic [3, 13]. P and Q describe the state of the program before and after the execution of C, respectively.

Summaries only talk about the footprint of a function. This fact, combined with the idea of frame inference [2] from separation logic, allows Infer to avoid huge summaries that explicitly tabulate most of the input-output possibilities when performing interprocedural analysis. The theoretical notion allowing Infer to synthesise pre and post-conditions in summaries is *bi-abductive inference* (or bi-abduction) [8]. It consists of automatically inferring parts of the program state that are needed to perform certain computations. Formally, bi-abduction involves solving the following extension of the entailment problem:

$$S_1 * A \vdash S_2 * F$$

where S_1, S_2 are given formulae in separation logic describing states, whereas F (frame) and A (anti-frame) are unknown and need to be automatically inferred by a theorem prover. Summaries of procedures in a program are composed together in a bottom-up fashion to obtain summaries of larger pieces of code. For example, when a function $f(\vec{x})$ calls function $g(\vec{y})$, Infer uses the summary computed for $g(\vec{y})$ when analyzing $f(\vec{x})$.

2.1.1 TOPL. Temporal Object Property Language (TOPL) [18] is a language for specifying user-defined properties. It has an engine built on top of Infer that checks whether the property holds for a given program. Similar to other temporal logics, TOPL properties are expressed in terms of state machines. Here is an example of a typical property modeling *taint analysis*:

```
property Taint
 start -> start: *
 start -> track: "source"(Ret) => data:=Ret
 track -> error: "sink"(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` with arity zero, we store its 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` with arity one where the argument is a term containing the term we stored in `data` (or in other words, if `data` is reachable from the argument following the heap), we go to the special `error` state and report the violation.

Under the hood, TOPL is built on top of Infer's Pulse analysis [25], 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.

2.2 Dynamic Analysis

Dynamic code analysis consists of actually executing the code we want to analyze on a range of inputs (in contrast to static analysis, which does not execute the code). To be effective, the analysis

should use inputs that will achieve a wide coverage of the code. FAUSTA [24] is a traffic generation system used at WhatsApp for analysis of reliability issues such as *crashes*. FAUSTA's *traffic data generators* materialize server inputs (called stanzas) based on client-server communication specifications and send these inputs to the server.

Dynamic PrivacyCAT extends FAUSTA's materialization process to perform taint analysis [23]. It taints some of the input data to check for leakage of personally identifiable information (PII). 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.

3 Combining Static and Dynamic Analysis

In this section we describe how dynamic analysis is used to improve coverage and precision of static analysis.

3.1 Motivation

Compositional static analysis works by composing together analysis results of small parts of the code (usually a function or procedure) to compute the analysis result of a larger part of the code. For example, analysis results of callees are used to compute the analysis result of the caller. In our case, analysis results are procedure summaries expressed in terms of pre/post-conditions:

$$\{P\}\ C\ \{Q\}$$

Such triples are synthesized by Infer. For example consider the following `prepend` function written in C, which prepends an element to a list:

```
void prepend(Node * E, Node * X) {
  E->next = X;
}
```

A summary for this function could be:

$$\{E \mapsto - * list(X)\}\ \texttt{prepend(E, X)}\ \{list(E)\} \tag{1}$$

where $E \mapsto -$ denotes a cell allocated to address E with some value, and $list(X)$ stands for an allocated list starting at address X. In words the spec above says: if `prepend` is called with an allocated cell pointed to by E and a list pointed to by X, then after the execution E points to a list.

However, in reality, synthesizing static summaries for a function f can fail for several reasons. For example, the code could be too complex for the analysis engine, or the computation may take too long and times out. When the derivation of the summary fails, the analysis of f and of all functions depending on f becomes inconclusive. In turn, this cascade effect is problematic as it limits the analysis coverage and has an unfortunate side effect: false negatives (missing possible bugs).

3.2 The Dynamically-Enhanced Static Analysis Algorithm

In this paper we assume a fixed finite set *Vars* of program variables (ranged over by $x, y, \dots$), and an infinite set *LVars* of logical variables (ranged over by $x', y', \dots$). The logical (or primed) variables will not be used within programs, only within logical formulae (where they will be implicitly existentially quantified). Moreover, let *Vals* be a set of values, and *FNames* be the set of functions names.

Dynamic Summaries. The dynamic analysis runs the code starting from inputs either generated by FAUSTA or from tests.

We define a concrete state of a program as a partial map from variables to values:

$$\sigma : Vars \rightharpoonup Vals$$

Let Σ be the set of concrete states σ of a program. Let a *run* of a program be a sequence of concrete states $\sigma_0\sigma_1\sigma_2 \cdots \in \Sigma^*$. We indicate with $[\![C]\!]$ the semantics of a program C consisting of all the runs C can have at run-time. We define a *dynamic summary (or dynamic specs)* to be a triple

$$[\sigma_I]\ f(\vec{x})\ [\sigma_O]$$

where $\sigma_I, \sigma_O \in \Sigma$ are states of the program. The semantics of the triple is: there exists a run $\sigma_0\sigma_1\sigma_2 \cdots \in [\![C]\!]$ such that $f(\vec{x}/\sigma_I) = \sigma_O$. In words: there exists a run of the program such that the function f when called with the concrete state σ_I results in the concrete state σ_O.

For example, for the `prepend` function above, we could observe the following I/O triples:

$[E \mapsto 6 * X \mapsto \{0, 1, 2\}]$ `prepend(E, X)` $[E \mapsto \{6, 0, 1, 2\}]$
$[E \mapsto 7 * X \mapsto \{6, 0, 1, 2\}]$ `prepend(E, X)` $[E \mapsto \{7, 6, 0, 1, 2\}]$
$[E \mapsto 8 * X \mapsto \{7, 6, 0, 1, 2\}]$ `prepend(E, X)` $[E \mapsto \{8, 7, 6, 0, 1, 2\}]$
$[E \mapsto 9 * X \mapsto \{8, 7, 6, 0, 1, 2\}]$`prepend(E, X)`$[E \mapsto \{9, 8, 7, 6, 0, 1, 2\}]$

In the rest of the paper we indicate by $\mathcal{D}$ the set of all dynamic summaries for all functions.

Static Summaries. We have mentioned before that *static summaries* are Hoare's triple of the form

$$\{H_1\}\ C\ \{H_2\}$$

where H_1 and H_2 are formula in a certain logic. In this paper we use H_1, H_2 to be symbolic heaps as defined in [3, 13]. More precisely, a *symbolic heap* H consists of a finite set of equalities and a finite set of heap predicates. The equalities $E = F$ are between expressions E and F, which are program variables x, or primed variables x' or a value $v \in Vals$. The heap predicates describe dynamically allocated data such as lists. A symbolic heap describes a set of concrete program states. We indicate by $\mathcal{H}$ the set of all symbolic heaps and by $\mathcal{S}$ the set of all static specifications.

The Algorithm. One first observation is that static summaries like (1) can be seen as an abstract way to describe a set of concrete dynamic specs. In other words, there should exist an *abstraction function*

$$\alpha : \Sigma \to \mathcal{H}$$

mapping a triple $[\sigma_I]f(\vec{x})[\sigma_O]$ onto a static specification $\{\alpha(\sigma_I)\}\ f(\vec{x})\ \{\alpha(\sigma_O)\}$. Hence, the key idea in this paper is to utilize concrete I/O values observed by running a dynamic analysis to synthesize static summaries which in turn could be used to enhance static analysis.

To implement this idea we adopt the following strategy. We use tracing to collect relevant (σ_I, f, σ_O) triples from dynamic analysis runs for all functions f that static analysis has failed to analyze. Note that since Erlang is a mostly pure language, it is usually

enough to record functions' arguments in σ_I and their return value in σ_O. We transform these I/O triples into static specifications using the abstraction map α defined in Section 4. We then use the synthesized static summaries in later runs of the static analysis to increase its coverage on functions which were not previously statically analysable. The iteration can be repeated to further improve coverage. This process is described in Algorithm 1.

Algorithm 1 Dynamically-Enhanced Static Analysis

1: $(SSpecs, W_s, U_F) \leftarrow SAnalysis(CodeBase, \emptyset)$;
2: $W_{all} \leftarrow W_s$;
3: **while** $U_F \neq \emptyset$ **do**
4: $\quad (D_{U_F}, W_d) \leftarrow DAnalysis(CodeBase, U_F)$;
5: $\quad SDSpecs \leftarrow \alpha_{Spec}(D_{U_F})$;
6: $\quad (SSpecs, W'_s, U'_F) \leftarrow SAnalysis(CodeBase, SSpecs \cup SDSpecs)$;
7: $\quad W_{all} \leftarrow W_{all} \cup W_d \cup W'_s$;
8: $\quad$ **if** $U_F = U'_F$ **then break** ;
9: $\quad U_F \leftarrow U'_F$;
10: **end while**
11: **return** W_{all};

Let $\mathcal{W}$ be the set of all warnings that can be reported by static or dynamic analysis. Moreover, let $\mathcal{D}[f]$ be the set of all the possible dynamic summaries for $f(\vec{x})$. We write $D_f \subseteq \mathcal{D}[f]$ for the set of observed dynamic summaries for $f(\vec{x})$ during a run of the dynamic analysis. Both $\mathcal{D}[f]$ and D_f can be lifted to a set F of functions: $\mathcal{D}[F] = \bigcup_{f \in F} \mathcal{D}[f]$ and $D_F = \bigcup_{f \in F} D_f$.

We assume a function implementing a compositional static analysis:

$$SAnalysis : C \times 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}} \times 2^{\mathcal{W}} \times 2^{FNames}$$

taking the code to analyse ($CodeBase \in C$) and a set of function summaries $2^{\mathcal{S}}$. The latter constitutes a cache of specifications to be used by the compositional analysis. It returns a new set of specifications computed during the analysis, a set of warnings (i.e., possible bugs detected in the code), and a set of non-analyzed function names, for which the analysis has failed to give results. We will refer to these functions as *unspecced* functions. Moreover, we assume the function implementing dynamic analysis:

$$DAnalysis : C \times 2^{FNames} \rightarrow 2^{\mathcal{D}} \times 2^{\mathcal{W}}$$

taking a program to analyse, a set of unspecced functions $U_F \in 2^{FNames}$, and returning a set of dynamic summaries $D_{U_F} \subseteq \mathcal{D}[U_F]$ for U_F, and a set of warnings. Note that the dynamic analysis may not finds specs for some unspecced functions, that is, a function $f \in U_F$ might exist such that $D_f = \emptyset$ (and $\mathcal{D}[f] \cap D_{U_F} = \emptyset$).

Algorithm 1 first runs static analysis from scratch on the code base with an empty cache of function specifications (line 1). The result is: a set of static specifications *SSpecs*, a set of warnings W_s, and a set U_F of unspecced functions where the analysis failed. Then dynamic analysis is run at line 4. It takes as input the set of unspecced functions for which we want to find dynamic specs. It returns a set of warnings W_d and a set of I/O triples D_{U_F}. We abstract these dynamic specifications obtaining a new set of static specifications *SDSpecs* (line 5). We run again static analysis with an augmented cache comprehending both *SSpecs* and *SDSpecs* (line 6). We continue to iterate over dynamic and static analysis on as long we increase the spec coverage, i.e. we reduce the set of unspecced functions. During the execution of the algorithm we collect in W_{all} the set of warnings detected by both static and dynamic analysis and we return it to the user at the end.

Theorem 3.1. *Algorithm 1 terminates.*

Proof. Let U_F^i be the U_F set of the i-th iteration, where $U_F^0 = F$. We can prove by induction that

$$U_F^{i+1} \subseteq U_F^i \tag{2}$$

Let's prove the base case $U_F^1 \subseteq U_F^0$. If $f \in U_F^1$ then f is not in the cache of the static analysis at iteration 1, i.e. $f \notin SSpec_0 \cup SDSpec_0$. By contradiction assume $f \notin U_F^0$ then f has a spec, that is $f \in SSpec_0$ which implies $f \notin U_F^1$ which in turn contradict our hypothesis that $f \in U_F^1$.[2] Therefore we conclude $U_F^1 \subseteq U_F^0$.

The inductive case is similar, which gives us the proof of (2). From $U_F^{i+1} \subseteq U_F^i$ we have two cases

i. If $U_F^{i+1} \subset U_F^i$ then at the i-th iteration we have found more specs and some function which didn't have specs at iteration $i-1$ now got a spec. Hence we have a finite descending chain of set of unspecced functions and therefore the algorithm can be in this case a finite number of iterations.
ii. If $U_F^{i+1} = U_F^i$ then no new specced function have been found and therefore the algorithm terminates (line 8 condition $U_F = U'_F$ holds).

□

Lemma 3.2. *If both dynamic and static analyses are deterministic, the stop criterion of Algorithm 1 gives the smallest U_F set possible, i.e., running more iterations would not increase coverage.*

Proof. Let us assume that $U_F = U'_F$ but we do not stop the algorithm. In this case, U_F will stay the same in the next iteration. Assuming that the codebase also does not change, the dynamic analysis will get the same inputs and hence will return the same result (D_{U_F} and W_d) as in the previous iteration. This results in the same abstracted static specs (*SDSpecs*) and the same input to the static analysis, which will return the same U'_F as in the previous iteration. □

Note that in practice both dynamic and static analyses (including FAUSTA and Infer) are often non-deterministic (e.g. random input generation or internal resource limits) and therefore, it is possible that there is no improvement in coverage for some iterations, but after a while, we do get more specs. For example, if $U_F^3 \subset U_F^2 = U_F^1 \subset U_F^0$ then the algorithm would stop at U_F^2. The termination criterion could be generalized to check for improvement over not only one, but a given k number of iterations, where k could be fine tuned based on experiments. However, in practice (as reported by our experimental evaluations, Section 5), termination criteria based on resources (such as CPU time or memory) also work well.

[2] This proof is based on the assumption that if a function has a spec at one iteration and on the next iteration for some reason a spec is not computed we use the previous spec. Or alternatively, if a function has a spec in one iteration then on later iteration we don't try to re-analyze it.

4 Dynamic Models Abstraction

We have seen that dynamic summaries from observation of functions I/O in real runs can be "too concrete". For each function, the dynamic analysis can observe and record thousands of such summaries. Using them directly as specifications in static analysis is not practical in general. There are two main reasons for that. First of all performance: if a function f has many specifications, then the analysis of f's callers would potentially result in many branches that would slow down static analysis. Second, to avoid this problem, Infer uses an upper bound on the number of specs utilized for a function call. Having more specs above this limit reduces abruptly the precision of the analysis. Consequently, it is better to have a more compact representation of the information conveyed by the dynamic summaries. To achieve that, we abstract the set D_f for each function of interest (see line 5 of Algorithm 1).

Let $\mathcal{D}$ be the set of all dynamic specs and $\mathcal{S}$ the set of all static specifications. Let $\top$ indicate a *non-deterministic* value, and for a domain D, let $D^\top$ be $D \cup \{\top\}$.

In this section we will define the abstraction function

$$\alpha_{Spec} : 2^{\mathcal{D}} \to 2^{\mathcal{S}} \tag{3}$$

abstracting a set of dynamic specifications to a set of static ones. The abstraction function is defined for concrete Erlang states. It is the composition of different kind of abstractions defined in the following.

First we define an abstraction function

$$\alpha : \Sigma \to \mathcal{H}$$

mapping concrete states to logic functions used in static specs. It is defined as

$$\alpha(\sigma) = \bigwedge_{x \in dom(\sigma)} x = \alpha_V(\sigma(x), 0)$$

Informally, the logical formula is a conjunction of equalities: each variable x is equal its abstracted value defined by the abstraction function α_V.

The function $\alpha_V : Vals \times N \to Vals^\top$ abstracts Erlang values. The first argument is the value and the second (integer) argument is a helper for keeping track of nesting (for lists and tuples). The function is defined as:

$$\alpha_V(v, l) = \begin{cases} \alpha_N(v) & \text{if } v \in N \\ \alpha_A(v) & \text{if } v \in A \\ \alpha_{list}(v, l) & \text{if } v \in EL \\ \alpha_{tuple}(v, l) & \text{if } v \in ET \\ \top & \text{otherwise} \end{cases}$$

Where N, A, EL, ET represents Erlang numbers, atoms, lists and tuples, respectively. We now define the per-type abstractions used in α_V. Let $N[p]$ be a set of the first p natural numbers, i.e. $\{0, \ldots, p\}$. For integers we define $\alpha_N : N \to N[p]^\top$

$$\alpha_N(n) = \begin{cases} n & \text{if } 0 \le n \le p \\ \top & \text{otherwise} \end{cases}$$

Informally, this means we only keep explicit the value of the first p integers and treat the rest as unknown. In our experiments, we observed that a good choice for parameter p is to set it to 1 (i.e. only tracking 0 and 1).

For Erlang atoms, we keep some special values explicit, whereas the rest is abstracted away in with the non-deterministic value. Let A be the set of atoms and $L_A \subseteq A$ a fixed set of atoms. Then $\alpha_A : A \to L_A^\top$ is defined as:

$$\alpha_A(a) = \begin{cases} a & \text{if } a \in L_A \\ \top & \text{otherwise} \end{cases}$$

In our implementation for Erlang the set of special atoms is

$$L_A = \{\texttt{false}, \texttt{true}, \texttt{timeout}, \texttt{return}, \texttt{error}, \texttt{exit}, \texttt{undefined}\}.$$

Erlang lists are abstracted in two dimensions: *length* and *level of nesting*. Let EL be the set of Erlang lists and $k, m \in N$, then $\alpha_{list} : EL \times N \to EL^\top$ is defined as

$$\alpha_{list}([e_1, \ldots, e_n], l) = \begin{cases} [\alpha_V(e_1, l+1), \ldots, \alpha_V(e_k, l+1), \top] & \text{if } n > k \text{ and } l \le m \\ [\alpha_V(e_1, l+1), \ldots, \alpha_V(e_n, l+1)] & \text{if } n \le k \text{ and } l \le m \\ \top & \text{otherwise} \end{cases}$$

In this definition m is a fixed constant which stands for the maximum nesting level we want to keep explicit. The constant k is the maximum number of elements per level that we want to keep explicit. Everything beyond that is abstracted to the non-deterministic value $\top$. Also note that in the first line of the definition, the $\top$ as the last element can represent an arbitrary number of elements at the end of the list.

Erlang tuples are abstracted point-wise by abstracting their elements. Also for tuples we fix m as the maximum level of nesting we preserve. Let ET be the set of Erlang tuples then $\alpha_{tuple} : ET \times N \to ET^\top$ is defined as

$$\alpha_{tuple}(\langle e_1, \ldots, e_n \rangle, l) = \begin{cases} \langle \alpha_V(e_1, l+1), \ldots, \alpha_V(e_k, l+1) \rangle & \text{if } l \le m \\ \top & \text{otherwise} \end{cases}$$

To state the sense in which α is sound we can define a concretization function $\gamma : \mathcal{H} \to 2^\Sigma$ mapping logic formula into their semantics:

$$\gamma(H) = \{\sigma \mid \sigma \models H\}$$

With γ we can prove the following result.

Theorem 4.1. *The abstraction function α is sound, that is* $\forall \sigma \in \Sigma : \sigma \in \gamma(\alpha(\sigma))$.

We can lift the abstraction function from states to dynamic summaries in several ways.

$$\alpha\big([\sigma_I]\ f(\vec{x})\ [\sigma_O]\big) = \{\sigma_I\}\ f(\vec{x})\ \{\alpha(\sigma_O)\} \tag{4}$$

$$\alpha\big([\sigma_I]\ f(\vec{x})\ [\sigma_O]\big) = \{\alpha(\sigma_I)\}\ f(\vec{x})\ \{\sigma_O\} \tag{5}$$

$$\alpha\big([\sigma_I]\ f(\vec{x})\ [\sigma_O]\big) = \{\alpha(\sigma_I)\}\ f(\vec{x})\ \{\alpha(\sigma_O)\} \tag{6}$$

For an over-approximating static analysis, definition (4) would preserve over-approximation, and would be sound. Whenever the precondition is abstracted (definition (5) and (6)), there are no guarantees about preserving over- or under-approximation of the analysis. In our implementation, we use definition (6). Despite this choice may introduce false-positives, it is a heuristic consistent with an already existing heuristic in Infer, which over-approximates the behaviour of unknown functions in order to increase coverage (an unknown function is one whose implementation is unavailable at analysis time).

For our use-case, i.e. bug finding, we found that (6) is the most useful for two reasons. First, during the analysis it provides a more

$$
\begin{aligned}
D \quad &= \{ \\
&\quad [E \mapsto 6 * X \mapsto \{0,1,2\}]\texttt{prepend(E, X)}[E \mapsto \{6,0,1,2\}] \\
&\quad [E \mapsto 7 * X \mapsto \{6,0,1,2\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{7,6,0,1,2\}], \\
&\quad [E \mapsto 8 * X \mapsto \{7,6,0,1,2\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{8,7,6,0,1,2\}], \\
&\quad [E \mapsto 9 * X \mapsto \{8,7,6,0,1,2\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{9,8,7,6,0,1,2\}] \\
\}
\end{aligned}
$$

$$
\begin{aligned}
\alpha_{Spec}(D) \quad &= \{ \\
&\quad [E \mapsto \top * X \mapsto \{0,1,\top\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{\top,0,\top\}], \\
&\quad [E \mapsto \top * X \mapsto \{\top,0,\top\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{\top,\top,\top\}], \\
&\quad [E \mapsto \top * X \mapsto \{\top,\top,\top\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{\top,\top,\top\}], \\
&\quad [E \mapsto \top * X \mapsto \{\top,\top,\top\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{\top,\top,\top\}] \\
\} \\
&= \{ \\
&\quad [E \mapsto \top * X \mapsto \{0,1,\top\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{\top,0,\top\}], \\
&\quad [E \mapsto \top * X \mapsto \{\top,0,\top\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{\top,\top,\top\}], \\
&\quad [E \mapsto \top * X \mapsto \{\top,\top,\top\}]\ \texttt{prepend(E, X)}\ [E \mapsto \{\top,\top,\top\}] \\
\}
\end{aligned}
$$

Figure 1: An example of a set of concrete specifications and its abstraction.

compact representation for potentially thousands dynamic models, limiting the amount of branches during analysis of the same function. Second, it increases the bug finding capabilities of the analysis as models express more behaviours.

Join operator. After abstracting dynamic summaries into static ones, we will end up with a large set of static specs. These can still be reduced further by defining a *join operator* able to combine together two summaries into one which summarize both of them. A join operator is a partial function $\uplus : \mathcal{S} \times \mathcal{S} \rightharpoonup \mathcal{S}$. In our case, the join operator we have implemented corresponds to the equality relation between static summaries.

$$H_1 \uplus H_2 = \begin{cases} H_1 & \text{if } \gamma(H_1) = \gamma(H_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

In our case, join can be implemented directly with set inclusion. Therefore we can define the abstraction function (3) mapping sets of dynamic specifications onto sets of static specifications, and used in the algorithm, as a combination of α and the implicit join given by set inclusion:

$$\alpha_{Spec}(D) = \{\ \{\alpha(\sigma_I)\}\ f(\vec{x})\ \{\alpha(\sigma_O)\}\ \mid\ [\sigma_I]\ f(\vec{x})\ [\sigma_O] \in D\ \}$$

In our implementation using α_{Spec} has resulted in dramatic reduction of the number of dynamic specs, often from thousands to few units. We have observed that this abstraction improved speed and precision of the analysis.

Example 4.2. Let's consider again the set D of concrete specifications from Section 3.1 shown on the top of Figure 1. Assuming we abstract away numbers greater than 1 and we keep lists up to 2 elements, the resulting set of abstract specs $\alpha_{Spec}(D)$ is reported in the bottom of Figure 1. Notice that join is obtain by set inclusion: the last two elements of the abstract set are the same and therefore combined into one single element in the final set.

5 Case Study: Analysing WhatsApp Erlang Code

In this section we report results of using the technique described in this paper to enhance the analysis of privacy properties performed by PrivacyCAT [23]. PrivacyCAT performs dynamic taint analysis based on synthesized, realistic user inputs and traffic generation (via tests , FAUSTA [24] and Sapienz [9]), and static taint analysis based on Infer [6, 19]. It traces the propagation of synthesized sensitive data, and its processing at data sinks and exchanging APIs for leakage detection.

In Section 5.1 we present the results of continuous analysis at WhatsApp within the existing PrivacyCAT workflows. Then, in Section 5.2 we provide an in-depth analysis of specific runs dedicated to assessing the isolated impact of dynamic data.

In both sections, the technique presented in this paper is used to analyse the WhatsApp server codebase consisting of millions of lines of Erlang code. Following, we provide details of our implementation of Algorithm 1. The static analysis component is Infer with Erlang support (InfERL [19]) and the dynamic analysis component is FAUSTA traffic generation [24]. The dynamic models are extracted from FAUSTA's runtime execution traces. The traces are transformed into I/O pairs to be used as dynamic models provided to Infer. The termination strategy is *bound by dynamic traces size*: we stop when the size of all dynamic traces exceeds 10Gb.

5.1 Continuous Integration

WhatsApp has a robust Continuous Integration (CI) system in place that includes a static analysis run based on Infer scheduled to occur every hour. Continuous jobs perform deep analysis over the entire repository on the most recent code version. This is designed to complement the quicker diff-time static job, which is a lighter version focused on the changes of each specific diff (i.e. a code change). Using code analysis in CI aims to find privacy vulnerabilities in WhatsApp code early in the development cycle, and to report them

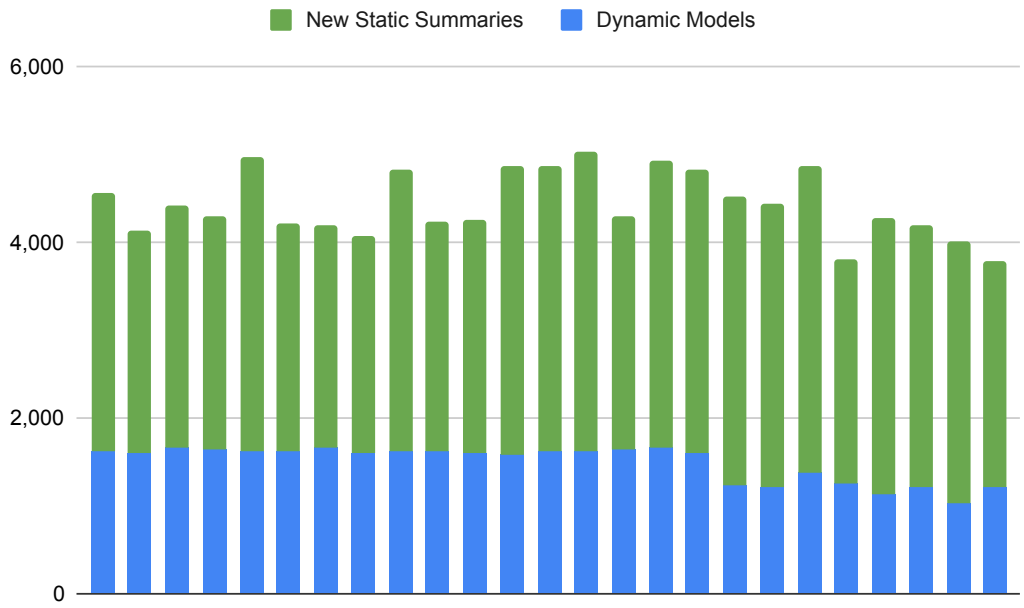

Figure 2: Dynamic models impact on static analysis.

to developers through a unified channel for fixes before the code reaches production, and consequently affects users.

We have implemented the technique presented in this paper and enhanced the hourly static analysis CI continuous workflow of WhatsApp server Erlang codebase. We schedule a *dynamically-enhanced* static analysis job in addition to the existing standard *static-only* Infer job. The dynamic part consists of an ad-hoc FAUSTA traffic generation run with *targeted tracing* enabled. Targeted tracing is the collection of runtime data (input and output values) only for calls to a given set of target Erlang functions. The target functions we provide are those that had no summaries found by the static-only Infer job on the same code version (i.e., the U_F set in Algorithm 1).

Since the first run of this strategy, the continuous CI job has identified 92 issues within the first 6 months. These issues prevented potential privacy site events (SEVs [14]) by early triaging high-priority tasks to respective code owners. The issues were all inspected by the most appropriate human code owner, that if needed decided necessary followups. This demonstrates the effectiveness of our approach in identifying and addressing potential vulnerabilities early in the development cycle.

5.2 Experiments

To evaluate the isolated benefit of dynamic data within static analysis we have conducted a set of experiments and in this section we discuss results. The experiments were performed manually (outside of WhatsApp CI) with verbose logging enabled to allow in-depth analysis of metrics. For each experiment run, all steps involved (Infer and FAUSTA) were fully executed from scratch on the same code version without using any pre-existing data. The code version was set to the most recent at the time of the experiment. The analysis target is set for the entire repository.

5.2.1 Bugs reported. For this experiment, we have sampled 20 pairs of runs on WhatsApp Erlang codebase with and without dynamic summaries and we have made a comparison on their reported bugs. Out of 20 runs, we have seen an increase of detected bugs in 16 cases (80%), a decrease in 3 cases (15%), and no difference in 1 case (5%). Note that these numbers correspond only to the bugs reported by static analysis (W_s in Algorithm 1). That is, using dynamic analysis does not only bring in new bugs itself, but also helps static analysis to discover more.

An interesting result regards a particular TOPL property related to taint analysis. When we initially formulated the property, Infer reported no issues in the codebase but we suspected that there might have been false negatives because most of the functions related to the property could not be analyzed and the cascading effect mentioned earlier in the paper was observed. We filled in the missing summaries by providing them manually and then Infer did start reporting issues. However, specifying summaries manually requires lots of effort (we had to write dozens of these summaries) and simply does not scale (we may need to keep writing manual summaries for every new property). The combination of static and dynamic analysis came to our rescue and was able to automatically provide the needed summaries and find the same violations of the property as when we provided the hand-written summaries.

5.2.2 Analysis of Static Coverage. We executed 25 pairs of runs specifically designed to track *static coverage* improvements solely provided by dynamic models. Static coverage here is defined as the

number of functions that have at least one summary, i.e., that have been analyzed by Infer. Results are reported in Fig. 2. All runs show a static coverage increase when using dynamic models. Additional coverage is already provided by the dynamic models (blue bars). On top of that and more interestingly, we observed additional static summaries (green bars) which could not be computed without the help of dynamic models. We measured that each I/O pair provided by dynamic analysis for 1 function contributed to an additional static coverage of 2.1 functions on average - we call this the *impact*.

Our best run reported impact of 2.9 functions covered for each dynamic model provided. Notably that run is the one with the least amount of dynamic models. In fact, there is a reverse correlation between the number of dynamic models and the impact. This led us to the conclusion that some dynamic models are more important than others, and thus running FAUSTA for longer in order to gain additional dynamic models is not helping linearly in deriving more static summaries. A quicker FAUSTA run already provides substantial impact. This is an important conclusion about the termination strategy of Algorithm 1, therefore we keep this metrics monitored in order to optimize the time spent on the dynamic run. Depending on the context, further time spent on running dynamic analysis might not be worth for the additional impact we can derive from it. Further research on the best practical termination strategy is required to enable better understanding on its impact over performance.

6 Related Work

Combinations of dynamic and static code analysis techniques have been used before.

Concolic testing [15] is mostly a dynamic technique (relying on executing the program) but is aided by symbolic solving (which is more commonly used in static analysis). In the first step, the code under analysis is executed on some random input, and the execution path is observed. Then, a formula is built saying that a different decision is made at one of the branching points along the execution path, and a symbolic solver is asked for some input that satisfies this formula. Finally, the process is iterated, until execution-path coverage is deemed sufficient. In this area, much of the research effort goes into reducing the time needed to achieve a certain path-coverage. (See, for example, [27].) Note that total time is a combination of concrete (instrumented) execution time and symbolic solving time. Unlike our case, concolic testing is mostly a dynamic technique: it uses only some techniques typically used in static analysis, but not a full-blown static analysis.

Dynamic symbolic execution [5] is a technique similar to concolic testing, which tracks multiple executions at once. An idea explored recently [4] was to combine dynamic symbolic execution with (classic, over-approximating) static analysis. The static analyzer helps dynamic symbolic execution to be more efficient: a trace produced by static analysis guides the space exploration of the symbolic execution. In turn, dynamic symbolic execution confirms that reports of static analysis are not false positives, by finding concrete executions exhibiting the bug. In this work, dynamic analysis helps filter out false positives; in our work, dynamic analysis reduces the number of false negatives. Still, this is the closest related work to ours.

Like concolic testing, *predictive monitoring* is also a dynamic analysis that uses some reasoning more commonly found in static analysis, but not a full-blown static analysis. The main idea of predictive monitoring (also known as predictive runtime verification) is to record dynamically one execution of a concurrent program, and reason about other possible executions [10]. (One could argue that reasoning about sets of executions is a typical approach in static analysis.) The typical setting is that of sequential consistent memory models. In this setting, one observes one interleaving, and then reasons about all other interleavings that are allowed by the observed lock operations (acquire/release). The basic property that was studied in this context is the existence of data races. (This was successful, for example, in finding bugs in the Linux kernel [17].) There are, however, extensions to wider classes of properties, such as regular languages [1].

Godefroid et al. [16] proposes an analysis that works at once with over- and under-approximating summaries. The main idea there is alternation: an under-approximating summary can help in computing an over-approximating one and vice versa. This kind of dual reasoning was subsequently used for termination and nontermination [22] (sometimes with input from dynamic methods [21]) and for deciding validity of logical formulas [26]. In our case, both the static analysis and the dynamic analysis are under-approximating, and they are not concerned with termination.

The idea of using dynamic analysis to improve the coverage of a compositional under-approximating static analysis is novel.

7 Conclusions

In this paper we introduced a novel technique to improve compositional static analysis by means of dynamic analysis. Our technique is built on top of the Infer static analyzer and uses FAUSTA, a dynamic analysis system for Erlang developed at Meta.

We have implemented our methodology and used for analysing privacy properties of WhatsApp Erlang server code. The result shows that enhancing Infer static analyzer with dynamic analysis increases the software defects detected as well as it provides a substantial increase on the number of analysed functions.

References

[1] Zhendong Ang and Umang Mathur. Predictive monitoring against pattern regular languages. *Proc. ACM Program. Lang.*, 8(POPL):2191–2225, 2024.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[4] Frank Busse, Pritam M. Gharat, Cristian Cadar, and Alastair F. Donaldson. Combining static analysis error traces with dynamic symbolic execution (experience paper). In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 568–579. ACM, 2022.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[6] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods*, volume 6617 of *Lecture*

Notes in Computer Science, pages 459–465. Springer, 2011.

[7] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–300. ACM, 2009.

[9] Sopot Cela, Andrea Ciancone, Per Gustafsson, Ákos Hajdu, Yue Jia, Timotej Kapus, Maksym Koshtenko, Will Lewis, Ke Mao, and Dragos Martac. Automated end-to-end dynamic taint analysis for WhatsApp. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, page 21–26. ACM, 2024.

[10] Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for Java. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 221–230. ACM, 2008.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[12] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at Facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[13] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[14] Gareth Eason. Incident response @ FB, Facebook's SEV process. Dublin, 2016. USENIX Association.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[16] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56. ACM, 2010.

[17] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 66–83. ACM, 2021.

[18] Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime verification based on register automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2013.

[19] Ákos Hajdu, Matteo Marescotti, Thibault Suzanne, Ke Mao, Radu Grigore, Per Gustafsson, and Dino Distefano. InfERL: Scalable and extensible Erlang static analysis. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*, pages 33–39. ACM, 2022.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[21] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. Dynamite: dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.*, 4(OOPSLA):189:1–189:30, 2020.

[22] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 489–498. ACM, 2015.

[23] 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. PrivacyCAT: Privacy-aware code analysis at scale. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 106–117. ACM, 2024.

[24] Ke Mao, Timotej Kapus, Lambros Petrou, Ákos Hajdu, Matteo Marescotti, Andreas Löscher, Mark Harman, and Dino Distefano. FAUSTA: scaling dynamic analysis with traffic generation at WhatsApp. In *Proceedings of 15th IEEE Conference on Software Testing, Verification and Validation*, pages 267–278. IEEE, 2022.

[25] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification*, volume 12225 of *Lecture Notes in Computer Science*, pages 225–252. Springer, 2020.

[26] Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. Modular primal-dual fixpoint logic solving for temporal verification. *Proc. ACM Program. Lang.*, 7(POPL):2111–2140, 2023.

[27] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. Towards optimal concolic testing. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 291–302. ACM, 2018.