InfERL: Scalable and Extensible Erlang Static Analysis

Ákos Hajdu
Meta
London, UK
akoshajdu@meta.com

Matteo Marescotti
Meta
London, UK
mmatteo@meta.com

Thibault Suzanne
Meta
London, UK
tsuzanne@meta.com

Ke Mao
Meta
London, UK
kemao@meta.com

Radu Grigore
Meta
London, UK
rgrigore@meta.com

Per Gustafsson
Meta
London, UK
pergu@meta.com

Dino Distefano
Meta
London, UK
ddino@meta.com

Abstract

In this paper we introduce InfERL, an open source, scalable, and extensible static analyzer for Erlang, based on Meta's Infer tool. InfERL has been developed at WhatsApp and it is deployed to regularly scan WhatsApp server's Erlang codebase, detecting reliability issues and checking user-defined properties. The paper describes the Erlang specific technical challenges we had to address and our design choices. We also report on our experience in running InfERL on Erlang code at scale, supporting the messaging app used everyday by over 2 billion people.

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

Keywords: Erlang, static analysis, Infer

ACM Reference Format:
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 (Erlang '22), September 11, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3546186.3549929

1 Introduction

WhatsApp is the largest messaging app on the planet. Over 2 billion people rely on it for their personal and business communication, every day.

At Meta we develop a variety of tools to help programmers write robust code. One such tool is Infer [5], an open source static analyzer for C/C++/Objective-C/Java/C#. Infer scales to large codebases comprising millions of lines of code, and is used at Meta to check modifications of code [5, 8].

One part of the code — which has so far seen a limited application of analysis tools — is the WhatsApp server code, written in Erlang. On that code, at each code change, we run a few simple linters as well as Dialyzer [13]. On the one hand, the linters are shallow syntactic checks (e.g., warn if the programmer tries to turn off Dialyzer warnings, or to explicitly invoke garbage collection) and therefore give little signal. On the other hand, although very useful, Dialyzer performs a limited set of checks which does not cover all WhatsApp needs.

In this paper we introduce InfERL, an Erlang extension of the Infer static analyzer. InfERL is *scalable*; that is, the analysis takes time linear in the number of functions in the code, because it uses a compositional approach. Moreover, InfERL is *extensible*; that is, new analyses can be easily added without the need to hardcode them inside the analysis engine. To define new checkers, users write specifications in an automata-like style.

Extending Infer to support Erlang was done in two phases: (1) developing a compiler from Erlang to the assembly-like intermediate language understood by Infer's backend; and (2) extending Infer's analysis engine to Erlang specific language features. Compiling code into an intermediate language for static analysis is a hard task. The compilation needs to strike a balance between semantic precision and meaningful abstraction. For Erlang, it turned out to be particularly challenging due to its extensive use of higher-order functions, dynamic types, concurrency and fault tolerance. These features also required extending Infer's analysis engine in non-trivial ways, even just to provide a basic level of support. Examples include improvements for handling dynamic types in the prover, and built-in models for Erlang's data structures (e.g., tuples, lists, maps). We also implemented support for user-defined models, given as configuration, to ease the analysis of complicated code. Furthermore, we optimized Infer's engine for checking temporal properties, as it has not yet been used at such a scale.

The key contributions of this paper are as follows: (1) We introduce InfERL, a new scalable and extensible static analyzer for Erlang. (2) We describe the technical challenges and decisions of statically analyzing Erlang code in a compositional setting. (3) We give a report on the large-scale industrial application of InfERL to WhatsApp server code.

2 Features and Examples

InfERL takes Erlang code as input and can detect a variety of standard errors as well as check user-specified properties. [1]

2.1 Reliability Issues

InfERL supports the reliability errors listed in the table below (along with simple examples as illustration).

Issue type	Example
Bad key	`M = #{}, M#{2 := 3}.`
Bad map	`L = [1,2,3], L#{1 => 2}.`

Bad record	`R = #rabbit{name="Bun"},`
	`R#person.name.`

No matching branch in try	`tail(X) ->`
	`  try X of [_|T] -> {ok,T}`
	`  catch  _ -> error end.`

No matching case clause	`tail(X) ->`
	`  case X of [_|T] -> T end.`

No matching function clause	`tail([_|Xs]) -> Xs.`
No match of rhs	`[H|T] = [].`

No true branch in if	`sign(X) -> if X > 0 -> pos;`
	`                X < 0 -> neg end.`

2.2 User-Specified Properties

In addition to generic reliability issues, InfERL can be instructed to look for user-specified properties. These properties can be temporal, in the sense that they depend on a sequence of events. A simple example is the following:

Are there any code paths in which `file:write` *is called after* `file:close`?

[1]Further details regarding the examples presented in this section and instructions on running InfERL are available in github.com/facebook/infer/blob/main/infer/src/erlang/README.md

To pose this question to the analyzer, we write it down formally as in Figure 1a. This is an automaton, whose structure is drawn in Figure 1b. When the property is checked on the code in Figure 1c, function good is not flagged but function bad is flagged. The difference is that good calls a function (nop) with no side-effect, while bad calls a function (op) that has the side-effect of closing the file. Notice that the analysis is *interprocedural.*

Taint. Another kind of property that the user may specify is an information flow query:

Does the value returned by a function `source` *ever end up as the argument of a function* `sink`?

Technically, user-specified properties are compiled to nondeterministic automata. The nondeterminism is useful, for example, to track in parallel all tainted values.

Data transformation. It is also possible to pose the following query:

Does the value returned by a function `source` *ever end up as the argument of a function* `sink`*, even if it was transformed by a function* `transform`?

As an example, suppose that `source` returns a credit card number, `sink` stores data in some database, and `transform` takes a credit card number and returns its last four digits. Then, the above query asks the analysis to identify code paths that may store the last four digits (or all digits) of a credit card in a database.

The formal semantics of these properties are defined in terms of register automata [9].

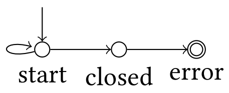
3 Background and Challenges

3.1 Background

Our work is based on the Infer static analyzer [4]. Infer supports various languages (Java, C/C++, Objective-C, C#) via frontends that compile to a common intermediate language based on control flow graph (CFG) called SIL [3]. In the backend, Infer has multiple analyzers. In this work we rely on Pulse [15] – one of Infer's most powerful and actively maintained analyzers – and Topl, which is an extension of Pulse for checking temporal properties described by automata. Infer implements a compositional, bottom-up interprocedural analysis based on function summaries [6]. 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 [6, 10] . Summaries only talk about the footprint of a function. This fact combined with the idea of frame inference [3] from separation logic, allows Infer to avoid huge summaries that explicitly tabulate most of the input-output possibilities. The theoretical notion allowing Infer to synthesise pre and post-conditions in summaries is bi-abductive inference [6] which consists of automatically inferring parts

```
property WriteAfterClose
  start -> start: *
  start -> closed: "file:close/1"(A,Ret) => f:=Ret
  closed -> error: "file:write/2"(F,D,Ret) when F==f
```

(a) User-defined property



(b) Structure of the property

```
good(F) -> nop(F), file:write(F, "hi").
bad(F)  -> op(F), file:write(F, "hi").
nop(_)  -> ok.
op(F)   -> file:close(F).
```

(c) Erlang code

Figure 1. When the property (a) is checked against the code (c), an error is reported in function bad.

of the program state that are needed to perform a certain computation. 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` calls function `g`, Infer uses the summary of `g` when analyzing `f`. Infer generally tries to be under-approximating to avoid false positives. But that's not a strong requirement and occasionally, for pragmatic reasons, it over-approximates (e.g., unknown functions) [15].

3.2 Challenges

To add support for Erlang, we must pay particular attention to language features distinct from those in already supported languages. We list various challenges here, some of which have been addressed, and some left as future work, as described in later sections.

Functional. Erlang programs use recursion, closures and higher-order functions to a far greater extent than programs written in languages already supported by Infer. In the context of a summary-based program analysis, the standard solution to handle recursion is to compute a fixed-point (iteratively update summaries until there is no change). Infer used to do this, until experiments showed that computing a summary at most once per function is faster and leads to results that developers find just as useful. However, Erlang programs might require a fixed-point computation.[2]

Closures are challenging too, as Erlang has no explicit variable declarations and thus, we need to clarify the rules about which variables are captured and which are local.[3] Furthermore, we have to decide how to proceed in the analysis when an unknown closure is invoked. For example, when executing `F()` in the code `F=fun ()->ok end, F()` the closure is known to the analysis; but, in the code `g(F)->F()`, it is unknown (due to the compositional nature of the analysis).

Let it crash. In Erlang there is no hard distinction between user exceptions (that are expected to be thrown and caught in normal execution) and runtime exceptions (that should never be thrown). An analysis that can identify those places where an exception is thrown and never caught has to be a top-down analysis or, at least, aware of what is the entry point of the program. Furthermore, supervisors can detect situations in which a process ends with an exception, and then take appropriate measures. In essence, a supervisor provides an extra-level of fault-tolerance (similar to catching exceptions), which is more dynamic and harder to analyze statically.

Dynamic typing. Erlang uses dynamic types pervasively: every pattern match involves making decisions based on the dynamic type of the value being matched. Infer has some basic support for dynamic types, but as detailed later it turned out to be incomplete as it was rarely used in other languages. A further challenge related to dynamic types and pattern matching is that, strictly speaking, a pattern match is seldom complete. For example, a function might have clauses for empty and non-empty lists, but not for other types. From the analyzer's point of view, it is hard to decide whether non-exhaustiveness is a mistake or intended (let it crash).

Concurrency. Erlang was designed with concurrency in mind, having built-in primitives for sending and receiving messages, and widely used libraries (like `gen_server`) to provide high-level abstractions. The main challenge in handling built-in primitives is to figure out which `receive` may handle a message from a `send`. This is difficult partly because the target of a message is computed dynamically; for example, it could be itself received as content of a previous message. Furthermore, higher-level abstractions (such as `gen_server`) may require special treatment.

Scalability. Scalability is a well-known challenge in program analysis. However, in our case it is worth explicitly calling it out as WhatsApp server is one of the biggest Erlang codebases and our analysis must be suitable to run in a reasonable time to provide early signals to developers. By their approximate nature, Infer's analyzers tend to have a good scalability by sacrificing some coverage. However, certain Erlang constructs might need special approximations. For example, Erlang has a variety of widely used, built-in data structures, including unbounded ones, such as lists or maps. Infer has support for data structures (e.g., we can define a `Cons` structure with a `head` and `tail` field), but unbounded structures in other languages are typically handled with approximations that provide a good trade-off for the particular use cases. For example, Java maps use a recency abstraction of storing only the most recent key, which is sufficient to detect the common error of accessing a key without checking

[2]With a fixed-point computation, a function may be analyzed multiple times, which means that the analysis time may become superlinear in the number of functions.

[3]For example, we found situations in which the compiler and the interpreter disagreed: github.com/erlang/otp/issues/5379

its existence first. We present some of such approximations for Erlang in the following sections.

4 Compilation and Analysis Details

An overview of InfERL's architecture is illustrated in Figure 2. In our work, we added a new frontend to support Erlang. For analysis, we leveraged Pulse, with some Erlang specifics added, and Topl, with some optimizations. The output of Infer is a list of issues (with traces) that is picked up by the reporting phase in the Meta CI to raise signals to developers. The parts within Infer (highlighted by a dashed rectangle) are available as open source at github.com/facebook/infer.

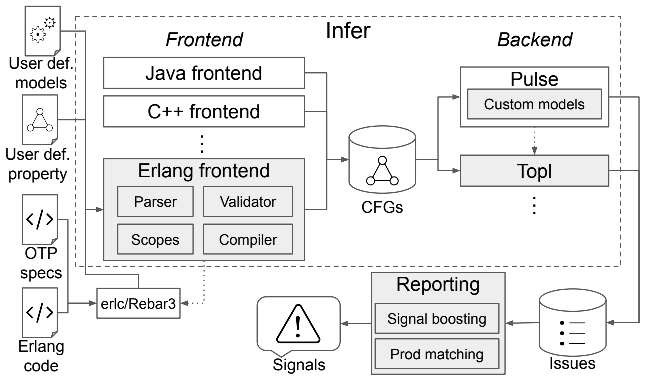


Figure 2. Overview of InfERL's architecture.

4.1 Compilation to SIL

The frontend invokes ERLC or REBAR3 to compile Erlang code into beam files, with the `debug_info` option enabled. Such beam files include the AST of the Erlang code. The next stages of the frontend are modular, that is, each beam file is processed without knowledge of the other beam files. Indeed, we leverage modularity to process beam files in parallel. For each beam file, the next stages are (1) parsing the ASTs, (2) performing basic validation, (3) doing scope analysis (to determine local/captured variables), and (4) compiling each Erlang construct to CFG nodes and instructions in the intermediate language SIL. Optionally, InfERL can be instructed to extract type specs from OTP functions (by discovering beam files of the default OTP version available) to make the analysis more precise (see later). The frontend is extensible to support Elixir as well with minor modifications.

Modules and functions. Modules are processed independently. Each function is translated to a SIL procedure with a corresponding CFG in Infer. On entry, the procedure loads the arguments into fresh variables and matches them against the patterns of the first function clause. On successful matching, the function body is executed, otherwise control flows to matching the pattern of the next function clause, and so on. If there are no more clauses, we report the "no matching function clause" error.

Expressions. Translation covers a significant portion of Erlang expressions, patterns and guards, including literals, variables, blocks, match/case/if expressions (with non-match errors), lists (nil, cons, comprehensions), maps, records, tuples, binary/unary operators, send, and receive. Certain features are currently supported with limitations or approximations: try/catch assumes that no exceptions happen, lambdas cannot be named, and not all forms of dynamic calls are supported. Furthermore, bitstrings, floats, strings and related operations are not supported. This fragment of Erlang was enough for us to start experimenting, but we continuously add/improve support for missing/limited features.

Type specs. Our initial experiments revealed that many false positives come from potential non-exhaustive pattern matching. For example, consider the following code:

```
revapp(Xs, []) -> Xs;
revapp(Xs, [Y|Ys]) -> revapp([Y|Xs], Ys).
```

The pattern matching seems to be complete at a first sight, but one can call `revapp` with, say, a tuple as the second argument. It is not clear whether it is a mistake or intentional (let it crash). Our current solution is to rely on specs, such as

```
-spec revapp(list(), list()) -> list().
```

With the spec, we do not report a non-match error anymore because pattern matching is complete w.r.t. the spec.

A similar problem occurs when we pattern match on the result of a function whose implementation is not seen by the analysis (e.g. a builtin function for which we don't have any model yet). Consider the following code:

```
min(Xs) -> case lists:sort(Xs) of
             [] -> {error};
             [X|_] -> {ok,X} end.
```

In this example, we again rely on the spec of `lists:sort` (extracted from OTP beam files) to deduce that the pattern match is exhaustive. Note that we don't perform type checking, but assume that other tools can ensure that the code is well typed [12, 19].

4.2 Erlang Support in Pulse

Pulse provides good scalability, but comes with some tradeoffs: the size of the summary (which roughly correspond to the number of execution paths explored) for each function is limited, and loops (coming for instance from list comprehensions) are only unrolled to a bounded depth. In order to support Erlang, we added Pulse models for the built-in data structures (lists, tuples, records, maps), library functions, and certain expressions (e.g. list append). Furthermore, the pervasive use of dynamic types in Erlang uncovered problems in Pulse (such as information about dynamic types sometimes being lost during normalization of formulas) that we fixed.This also benefits other languages in Infer.

Data structures. We implemented full support for tuples and records. Lists are also supported, but list comprehensions (modeled as loops) are approximated to a configurable loop unrolling bound. Maps use a recency abstraction where only the latest key/value is stored.

Higher-order functions. Infer's current support for higher-order functions is limited. Proper support was recently added for closures that capture variables.Still, full support for closures requires handling unknown closures, as in our `g(F)->F()` example. To do so, we plan to explore two approaches: defunctionalization and specialization.

Defunctionalization [17] is a global program transformation that removes all uses of higher-order functions but preserves semantics. Being global means that we cannot apply it to the program fragment `g(F)->F()` — we must know the whole program. In particular, we need to know what anonymous function definitions appear in the program text.

Specializationmeans that whenever the analyzer finds a call to `g` with a known closure, it creates and analyzes a special version of `g` in which `F` is the known closure. Conceptually, this is similar to on-demand inlining of the calls to function `g`. The advantage of specialization is that it is language agnostic (for example Objective-C also has closures) and does not require whole program analysis. The downside is that it does not work for mutually recursive higher-order functions. We also observed that most of the higher-order functions in our codebase are from Erlang's standard library, which allows us to inline/specialize such calls, as well as define custom Pulse models for the most common ones.

Concurrency. `send` expressions are currently modeled as a no-op, and `receive` is approximated by returning a non-deterministic value (or timeout). However, we do have plans to add better support for them. Pulse/Infer relies heavily on special treatment of function calls to achieve its compositionality and scalability. Message passing can be viewed as a generalization of function calls. When Pulse/Infer would reach a `receive` expression, it should first figure out all possible corresponding `send` expressions, and then rely on summaries of those `send` expressions. Connecting `send` and `receive` expressions could be done by some pre-analysis or even under-approximated by a dynamic analysis. Conversely, whenever a `send` statement is processed, the analysis would have to produce a summary for it. Finally just as there should be a fixed-point computation for the summaries of functions (when we have mutual recursion), there should also be a fixed-point computation for the summaries of `send` expressions (when we have messages going back-and-forth between processes).

We do not support higher-level abstractions of concurrency, such as `gen_server`. However, we do have plans for them: `gen_server` is essentially used to implement (distributed) objects/actors. One typical use is to (a) send a request, (b) handle the request by updating the state and computing a response, and (c) send back the response. Parts (a) and (c) are similar to calling and returning from a method. Updating the state of the server process (b) is similar to how methods update object fields. Finally, spawning a server process using a certain module for its implementation is similar to instantiating an object from a certain class. Thus, a natural way to handle code using `gen_server` is to treat servers as objects, which Infer supports for other languages.

Operators. Comparison operators are currently limited to integers. List subtraction is not yet supported, and appending is approximated only up to a configurable total length.

Library functions. We provide models for a few, commonly used library functions (e.g. for maps, lists and certain BIFs). However, most OTP functions are modeled by their spec only: we assume a non-deterministic return value, but with the correct type. Furthermore, we allow user-defined models for Erlang functions, provided by the developers in a configuration file.

4.3 Topl

Topl is a pre-existing analyzer in Infer for checking user-defined properties, as seen in Section 2.2. It was, however, not routinely used on large codebases, and we noticed that it does not scale as well on WhatsApp server. The lack of scalability manifested by giving up on exploring some abstract states because an internal limit was hit. To hit the limit on abstract states less often, we implemented several optimizations. First, when the limit is reached, we use a more expensive normalization procedure, to identify equivalent abstract states. Second, we added support for dynamic types in Topl's solver, which led to more abstract states being identified as unsatisfiable. And third, we added a garbage collector: if an abstract state tracks values that became unreachable in the program, we drop it.

Apart from performance, deploying on a large codebase revealed two more problems. One problem is that sometimes code complexity leads to timeouts in Pulse and, since Topl works on top of Pulse, this leads to lack of Topl coverage. We addressed this problem by adding custom Erlang models to Pulse, as described in the previous section. Such models effectively tell the analysis to not analyze certain (complicated) functions in the codebase and, instead, make some simple (and configurable) assumptions about their behavior. Another problem is that the language used to describe user-defined properties is not friendly to Erlang programmers. To address this, we plan to provide an Erlang-specific language (in addition to the current one which works in terms of Infer's intermediate representation, thus being language agnostic). And, further, we plan to allow for easy specification of classes of properties; for example, a taint property could require just a list of sources and sinks, and the corresponding automaton would be hidden from users.

5 Deployment on WhatsApp Server Code

Test functions. We developed an extensive set of tests, consisting of more than 700 small Erlang functions.[4] Each function targets a specific language feature, including corner cases. Some functions are expected to succeed and terminate, while some are expected to crash. We compile and run these functions with Erlang, and separately we analyze them using InfERL. In both cases we compare the results with fixtures to cross check that the results match expectations (i.e. InfERL reports an issue if and only if the function crashes when executed). This checks that the compilation and analysis agree on Erlang's semantics.

WhatsApp server. InfERL is part of the continuous integration process (CI) at Meta. The analysis is fast enough to be ran hourly. InfERL scans the main branch of the WhatsApp server's codebase and files alarms for detected issues to code owners automatically. Code ownership is based on source control commit history or explicit specification by author or team in the module. This approach has been widely practiced at Meta for multiple types of code analyses, such as earlier Infer deployments [8] and dynamic analysis deployments of Sapienz [1] for mobile apps, and FAUSTA [14] for WhatsApp server. One widely reported challenge of deploying static analyzers at scale in industry is the potential high false positive (false alarm) rate of reported issues [7, 11]. When deploying InfERL, we set a high standard on developer experience, which assumes low tolerance on false positives. However, the initial deployment of InfERL has found thousands of false positives, mostly due to missing features, approximations, or limitations in Pulse's solver.

Prod matching. In order to meet our tight FP tolerance, we perform *prod matching*, i.e. report detected issues prioritizing on presence in our production data collected during runtime. The main challenge of this process is to map an InfERL detected issue to the error log with the same cause in production. InfERL implements this by categorizing its detected issues based on traces and locating to a line that is mostly likely the root cause, taking into consideration possible line number shifts due to recent checked-in code. We deployed some user-specified (taint) properties, which resulted in 200 issues being found. With prod matching we surfaced 21 of them to developers. Out of these, developers found 2 to be high-priority and 1 to be very high-priority, and fixed them. Reporting issues already happening in production is still useful because Infer provides traces, and errors involving sequences of events can be hard to find in logs. However, as the precision of our analysis increases, we plan to deploy it to run before changes are shipped to production.

Signal boosting. Issues discovered by InfERL may have already been detected by other testing and verification tools. Raising issues to developers directly in such cases would cause duplication that harms their experience. This is a common problem in industry when working with a large scale codebase that requires various types of testing and analysis. One technique that has been exercised in industry is signal boosting. Instead of duplicating alarms, when the same issue has been detected by multiple tools (e.g., dynamic and static analyses), our CI treats the issue as legit with a higher confidence and notifies developers about the new alarm in the original report. In addition, different tools provide different information about the error, thus helping developers better understand the underlying issue.

[4] github.com/facebook/infer/tree/main/infer/tests/codetoanalyze/erlang

6 Related Work

Infer has been successfully deployed at scale for various repositories at Meta [5, 8] and outside [2, 16, 18].

Dialyzer [12] is likely the most widely deployed static analysis tool for Erlang, bundled with the Erlang/OTP distribution. Our work makes a different trade-off for false positives compared to Dialyzer: InfERL will report an error when it finds an erroneous execution, whereas Dialyzer will only report an error when it happens for all inputs.

RefactorErl [20] is an Erlang source code analyzer and transformer tool, providing a similar translation of Erlang to a CFG format facilitating static analysis. However, its main focus is on source transformations that do not change the semantics of the code rather than extracting signals.

FAUSTA [14] is a dynamic analysis system, also running on WhatsApp server. It tracks taint by creating unique random strings at sources and checking for them at the sinks. The FAUSTA approach has a high chance of detecting a dataflow between a source and sink, given the dataflow occurred in the program executions FAUSTA considers. InfERL on the other hand can consider all paths, but it might miss dataflows due to abstractions needed to run the analysis at scale. Therefore the two approaches are complementary.

7 Conclusions

In this paper we presented InfERL, a scalable and extensible static analyzer for Erlang code. InfERL adds a new frontend to Infer and extends its backend analyzers Pulse and Topl. Our recent deployment of InfERL at WhatsApp shows promising results (especially with user-defined properties), but also highlights challenges for future improvements.

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] Amazon. 2021. Amazon CodeGuru now includes recommendations powered by Infer. http://aws.amazon.com/about-aws/whats-new/2021/10/amazon-codeguru-recommendations-infer/. Online, accessed 29 July 2022.

[3] 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. https://doi.org/10.1007/11804192_6

[4] 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

[5] 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. https://doi.org/10.1007/978-3-319-17524-9_1

[6] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 289–300. https://doi.org/10.1145/1480881.1480917

[7] 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

[8] 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

[9] 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

[10] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[11] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don’t software developers use static analysis tools to find bugs?. In *Proceedings of the 35th International Conference on Software Engineering*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[12] Tobias Lindahl and Konstantinos Sagonas. 2004. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 3302)*, Wei-Ngan Chin (Ed.). Springer, 91–106. https://doi.org/10.1007/978-3-540-30477-7_7

[13] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical type inference based on success typings. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Annalisa Bossi and Michael J. Maher (Eds.). ACM, 167–178. https://doi.org/10.1145/1140335.1140356

[14] 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

[15] 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*, Shuvendu K. Lahiri and Chao Wang (Eds.). Lecture Notes in Computer Science, Vol. 12225. Springer, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14

[16] Franco Raimondi and Bor-Yuh Evan Chang. 2021. How automated reasoning improves the Prime Video experience. http://amazon.science/blog/how-automated-reasoning-improves-the-prime-video-experience. Online, accessed 29 July 2022.

[17] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference, Volume 2*, John J. Donovan and Rosemary Shields (Eds.). ACM, 717–740. https://doi.org/10.1145/800194.805852

[18] Xin S. 2020. Infer#: Interprocedural Memory Safety Analysis For C#. http://devblogs.microsoft.com/dotnet/infer-interprocedural-memory-safety-analysis-for-c/. Online, accessed 29 July 2022.

[19] Josef Svenningsson. 2022. Gradualizer. https://github.com/josefs/Gradualizer. Online, accessed 29 July 2022.

[20] Melinda Tóth and István Bozó. 2011. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School (Lecture Notes in Computer Science, Vol. 7241)*, Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer (Eds.). Springer, 440–498. https://doi.org/10.1007/978-3-642-32096-5_9