Technical Report

Nr. TUD-CS-2015-1167
July 22nd, 2015

Toward a Just-in-Time Static Analysis

Authors
Nguyen Quang Do, Lisa (Fraunhofer SIT)
Karim Ali (Technische Universität Darmstadt)
Eric Bodden (Technische Universität Darmstadt)
Benjamin Livshits (Microsoft Research)

Toward a Just-in-Time Static Analysis

Lisa Nguyen Quang Do
Fraunhofer SIT
lisa.nguyen@sit.fraunhofer.de

Karim Ali, Eric Bodden
Technische Universität Darmstadt
firstname.lastname@cased.de

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

***Abstract*—Despite years if not decades of research and development on static analysis tools, industrial adaption of much of this tooling remains spotty. Some of this is due to familiar shortcomings with the tooling itself: the effect of false positives on developer satisfaction is well known. However, in this paper, we argue that static-analysis results often run against some *cognitive barriers*. In other words, the developer is not able to grasp the results easily, leading to higher *abandonment rates* for analysis tools.**

In this paper, we propose to improve the current situation with the idea of Just-In-Time (JIT) analyses. In a JIT analysis, results are presented to the user in order of difficulty, starting with easy-to-fix warnings. These warnings are designed to gently "train" the developer and prepare them for reasoning about and fixing more complex bugs. The analysis itself is designed to operate in *layers*, so that the next layer of results is being computed while the previous one is being examined. The desired effect is that static-analysis results are available just-in-time, with the developer never needing to wait for them to be computed.

I. Introduction

Static-analysis research has been focusing largely on scalability and precision. While both pose important challenges, in this work we argue that there is a whole other range of challenges that has largely been overlooked so far, which significantly hinders tool adoption in practice. Static analysis is still largely considered to be a batch-style activity. The user runs the static analysis tool at major release points in the product cycle or even as part of a nightly build. The user then proceeds to pour over the warnings, deciding which messages correspond to real errors that require a fix. Despite years of work on eliminating false positives, end-user experience of using highly unsound commercial tools tends to be overwhelming, leading to a high degree of tool abandonment. Attention has been given to prioritizing analysis warnings in an effort to focus developer attention on the issues that are most likely to be true positives. Observing developers' interactions with the static analysis tool, we point out that:

- some error reports are abandoned because they are simply too difficult to deal with;
- reporting an error warning becomes useless if it is not likely to be seriously examined;
- the relative advantages of showing less complex and easy-to-find, therefore, easy-to-fix, bugs first.

```
1  void bar1() {
2    g = f;
3    foo(e, f, g); // args 2 and 3 aliased
4  }
5
6  void bar2() {
7    foo(h, i, j);
8  }
9
10 void foo(a, b, c) {
11   a.f = source(); // assume a is not null
12   b.f = a.f;
13   sink(a.f); // detected in all cases Ⓐ
14   sink(c.f); // detected for bar1 Ⓑ
15 }
```

Fig. 1: Running example.

As a simplified running example, consider the code in Figure 1 and assume an analysis that reports taint flows from sources to sinks. One such taint flow that is easy to identify for both the analysis and the user is the flow from line 11 to line 13. This flow constitutes a strictly intra-procedural flow, without taking into account aliasing information. We propose to report such errors *early*. While the user goes about fixing this problem, the analysis might use that additional time to conduct further analysis. For example, analyzing `foo` in the contexts of `bar1` and `bar2`. In this case, the analysis identifies another possible flow to line 14 given the aliasing introduced by `bar1`.

More generally, we propose the idea of an *interactive just-in-time* (JIT) analysis that starts by giving developers easy-to-fix bugs, while building their understanding of the analysis tool, code, and bug patterns. The approach centers around analysis *layers* and carefully interleaves the process of bug fixing with that of static analysis. The latter runs in the background, while the user fixes the results from a previous layer. Whenever the user is ready for the next layer of bug reports, they are shown to them instantly.

Training the user to train the analysis: Note that due to the process of informal training that occurrs as a result of fixing the previous layers of bugs, the user never encounters issues that they are ill-equipped to resolve. One can combine observing which warnings get user attention with a prioritization scheme for analysis warnings. This approach to static analysis trains the user to face the warnings they are well-equipped to address, while training the analysis to be most helpful to the user. Ultimately, the JIT framework gives the developer the experience they have come to expect from on-the-fly compilation in IDEs like Eclipse, in the context of static analysis.

Last but not least, we propose for later analysis stages to take into account developer feedback in the form of simple questions. Responses to those questions have the potential to weed out many false warnings if answered correctly—another dimension in which static-analysis tools and users can benefit from the synergy of interaction.

Contributions: To summarize, this paper makes the following contributions:

- advocate a renewed focus on the interaction between the static analyzer and the user, with the goals of increasing the number of bugs that actually get fixed and reducing the abandonment rates of the tool.
- propose the idea of JIT analyses designed to interleave the process of computing analysis warnings with that of the user fixing them.
- show how such an analysis can be built for three example applications: explicit-information-flow issues in Java or Android applications, null-dereference detection, and API misuse detection.
- propose an interactive frameaork that makes use of JIT analyses.
- report performance numbers indicating that it is indeed feasible to build the kind of analyses we propose.
- discuss important quality features of the JIT framework that future studies will have to evaluate.

II. Detailed Requirements

We next outline a number of requirements that we have identified and which the JIT framework should ideally fulfill. We will address some of those requirements in later sections of this work. A few of them still raise largely open research questions. Others also conflict, leading to necessary design choices. We highlight those conflicts here.

***Layering*:** JIT analyses must be *layered*, ideally such that computations in later layers can build on top of results computed by earlier ones.

***Quick Response*:** Early analysis layers should present first results within milliseconds, similar to automatic spell checkers.

***Eager Reporting*:** Generally, also in later analysis layers, analysis results should generally be displayed to the user as early as possible.

***Few False Alarms*:** JIT analyses should avoid false alarms, i.e., not report results based on highly imprecise computations. If in doubt about a program property, the analysis should consult the user.

***Helpful Results First*:** Early layers should report only those results that are highly accurate, and likely to be easily fixed by the user. Later stages may then report results that are less accurate and/or less likely to be useful.

***Minimize User Interaction*:** The analysis may ask the user specific questions to weed out false positives, but should only ask questions that are simple to answer and can indeed lead to a large fraction of false positives to be dismissed.

***Up-to-date Results*:** Before later analysis layers show findings, those should be re-confirmed with respect to code changes that the user might have performed after the earlier layers were completed.

***Aid Problem Repair*:** For simple-to-fix problems, the analysis should suggest a "Quick Fix", i.e., an automatic code transformation that would fix the problem and make the error message go away.

***Generalize User Feedback*:** If the user dismisses an analysis result as not helpful, this should suppress the presentation of similar analysis results.

***Change-aware Reporting*:** Problems once dismissed/-suppressed as not interesting should not reappear again after the program code has been modified.

Some of the aforementioned requirements harmonize with each other. For example, while there is a subtle difference between *Quick Response* and *Eager Reporting*, both lead to the similar design decisions, prioritizing efficient analysis and reporting. Other combinations of requirements are contradictory, however, which means that there may be different pareto-optimal solutions, none of which fulfill all requirements perfectly. Design choices determine which one of those "best-effort solutions" an approach will target.

A particularly interesting design choice is the one of whether and to what extent to perform *Eager Reporting*. Such *Eager Reporting* contradicts with the requirement for *Few False Alarms*. In any layer that is based on may-information, i.e., information that may or may not be accurate, the resulting analysis result may or may not cause a false alarm. In such situations, the framework could ask the user for additional input, potentially turning the may-information into a must-information. But given the requirement to *Minimize User Interaction*, such user questions should ideally be pooled, to allow the framework to select questions that may rule out multiple false positives in combination. This in turn will lead to the necessity to report such results lazily, contradicting *Eager Reporting*. But lazy reporting also contradicts, to some extent, the requirement to show *Up-to-date Results*: the longer the framework waits with reporting results, the higher the chance that the user has modified the analyzed code in the meantime, forcing a potentially expensive re-analysis for the program under analysis.

Note that *Eager Reporting* has no contradiction with *Few False Alarms*; quite the contrary, assuming that the first layers only compute must-results, there is no need to consult the user for additional input.

When to consult the user: An interesting design choice to consider is when to ask the user questions, and which ones. As already alluded to above, questions can be useful to turn may-information into must-information. As we found, there is an interesting and quite subtle difference

```
void environmentCausedInaccuracy() {
  String query = UI.read();
  DB.query(query);
}

void internalAnalysisLimitation () {
  String query = Console.readLine();
  if (isPrime(37)) query = sanitize(query);
  DB.query(query);
}

void inputDependentVarability() {
  String query = Console.readLine();
  if (Console.readBool) query = sanitize(query);
  DB.query(query);
}
```

Fig. 2: User input sanitization.

between at least three different types of may-information:

- **Environment-caused inaccuracy:** the information is *inaccurate* (and therefore "may") because the analysis has insufficient knowledge about the application's deployment or execution context.
- **Internal analysis limitation:** the information is inaccurate simply because of shortcomings of the analysis, be it because the analysis problem is undecidable in general or whether the analysis performs program abstractions for other reasons.
- **Input-dependent variability:** accurate information, but the result only applies to executions caused by a fraction of the analyzed program input space.

To illustrate this, consider the three methods in Figure 2 and an analysis that reports strings read from user input and then written to the database without sanitization. In the first case, we assume a method `UI.read()` which concrete semantics are unknown to the analysis. Does that method provide pre-sanitized inputs or not? In this case, it might be useful to ask the developer. In the second case, we see an analysis limitation: 37 is always prime but the analysis fails to determine this fact. Hence, it will falsely assume a potentially unsanitized flow. Whether or not to ask the developer in such situation is a tough design choice. Finally, the last case shows may-information that is of the form "maybe-definitely": there is definitely some user input that will cause an unsanitized flow but there are also others that definitely will not. Such results would be readily reported without additional interaction. An interesting scientific question, which we raise here for future work, is how analyses can efficiently and effectively distinguish those three situations and whether further interesting cases exist.

III. Just-in-Time Framework

In static analysis, some results only take minimal computation to find, whereas others are more complex. This is the case for the two leaks in Figure 1 where one can be found directly and the other requires inter-procedural propagation, aliasing and context-sensitive information.

```
32 F g = new F(), h = new F(), f = null;
33 g = f;
34 if (...)
35   h = f;
36 x = f.a; Ⓒ
37 y = g.a; Ⓓ
38 z = h.a; Ⓔ
```

Fig. 3: Null dereference.

```
39 void foo(Y y, Z z) {
40   Cipher g = new Cipher();
41   z.bar(g);
42   g.doFinal(); Ⓕ
43
44   Cipher h = new Cipher();
45   y.bar(h);
46   h.doFinal();  Ⓖ
47 }
48
49 // X extends Z
50 void bar(h) { h.init(); }
51
52 // Y extends Z
53 void bar(i) { }
```

Fig. 4: API misuse.

For real programs, such flows can get exponentially more complex and take hours of computation to be reported, holding back the delivery time of other simpler results.

The JIT framework proposes a layered system that reports simple flows immediately. Early layers of JIT analyses are run first, yielding results in a matter of seconds. While the user reviews these results, the analyses are enriched by introducing increasingly complex information and adding to the results as the user goes through them.

A. Instantiations of the JIT analysis model

To illustrate our layering system, we use three different analyses and instantiate them following the layer model.

Taint analysis is used to find dangerous flows in code bases. It tracks tainted variables from sources to sinks to detect, for example, the injection of dangerous commands from the user interface to a database. In Figure 1, two data leaks should be reported: Ⓐ and Ⓑ, the latter for the calling context of bar1 due to the alias between f and g at line 2.

Null propagation searches for null dereferences: use of null pointers that would cause a NullPointerException. In Figure 3, `f.a` is dereferenced at line 36 Ⓒ. This is also the case for `g.a` at line 37 Ⓓ due to its alias to `f` at line 33. The dereference at line 38 Ⓔ is also reported, due to the may-alias at line 35.

API misuse detection ensures that programs use APIs correctly by verifying that they follow a certain usage protocol. In Figure 4, we want to verify that a cipher is always initialized before a call to doFinal. Two warnings should be reported: Ⓕ and line 46 Ⓖ. Note that the latter is harder to detect, since the call to bar at line 41 may resolve to either of the two bar methods.

	Taint analysis	**Null detection**	**API misuse**
L1	Ⓐ : direct	Ⓒ : direct	
L2		Ⓓ : must-alias to `f.a`	
L3			Ⓖ : monomorphic call to `Y.bar`
L4	Ⓑ : caller may-alias to `a.f`	Ⓔ : local may-alias to `f.a`	
L5			Ⓕ : polymorphic call to `X.bar` or `Y.bar`

TABLE I: Results reported by the different layers for three just-in-time analyses without user feedback.

B. Layers

We define a set of layers following the requirements of Section II, implemented by the three instantiations presented in Section III-A. Table I presents the results the three analyses would yield for each of the layers, assuming no user interaction.

L1: Intra-procedural, no aliasing: In this layer, the analysis performs data flow propagation in the scope of the current working set (typically, the currently open file, project, etc.), assuming no alias information. Information about assignments is kept in memory for later layers when aliasing is resolved. Only the classes required by the current working set are fully loaded.

L2: Intra-procedural, must-alias: From L1, we compute must-alias information in the scope of the method, and adjust the data flow accordingly.

L3: Inter-procedural, must-alias, monomorphic call sites: From L2, we extend to inter-procedural computation for both the callers and callees of the base methods. Continuing a must-analysis, we traverse only monomorphic calls. For any polymorphic call, we keep the call in memory for a later stage. The necessary classes are loaded in the background, presumably at the package level.

L4: Inter-procedural, may-alias, polymorphic call sites: From L3, we extend the must-alias information to incorporate may-aliases. For callees, we approximate the aliases through a side-effects analysis. If we target incomplete code (e.g. in the case of an IDE such as Eclipse), we need to distinguish between *certain* and *uncertain* may-aliasing, depending on whether or not the alias is caused by the incompleteness of the code. This refers to the Environment-caused inaccuracy explained in Section II.

L5: Inter-procedural, may-alias, propagation in polymorphic call sites: From L4, we resolve the polymorphic call sites from L3 and propagate the data flow information into the callees. This may require more classes to be loaded.

Let us consider how this design addresses the requirements from Section II. *Layering* comes by design. *Quick Response* is ensured, as an intra-procedural analysis needs minimal class loading and computational resources. *Eager Reporting* and *Few False Alarms* are given by the layering policy that only confirmed flows are reported. Pending ones are left for further propagation, unless the user purposely intervenes. *Helpful Results First* is ensured by the observation that a warning found in an earlier layer relies on less assumptions and is most likely true. The other requirements concerning user feedback follow from the design of the JIT framework in Sections III-C and III-D.

C. Developer Interactions

The JIT framework aims at making maximum use of user input, without flooding the user with requests, to facilitate tool understanding. The framework interacts with the user on three levels:

Warning validation: By suppressing uninteresting warnings, the user can facilitate the computation of the next layers. For example, if for Figure 3 the user suppresses warning Ⓒ, the propagation is killed and further layers do not need to be run, saving computational resources. Furthermore, the dependent issues Ⓓ and Ⓔ will not be reported, removing false positives. Instead of blindly suppressing a warning to make developer beliefs explicit, the framework can insert into the code-appropriate assertions such as `assert(f != null)` to document feedback and check its validity in test runs.

Questions to the user: Key questions can be asked to the user to define the need to go to the next layer. If for Figure 1 the user confirms that `bar2` is not reachable, warning Ⓑ can be reported at **L3** instead of **L4**, yielding the warning earlier. Additionally, `bar2` and its callees can be removed from the scan worklist, reducing the code surface to be analyzed.

Fix-oriented output: As the user modifies the source code, warnings can be invalidated. To minimize the consequences of bug-fixing, we propose to report warnings in a fix-oriented manner. In addition of being useful to warning understanding, this system also guides the user as to where and how to fix warnings. It is generally easier to fix simple warnings from early layers. Therefore, the layered system provides an intuitive way to group warnings by fix-similarity, by tracking the dependencies of the warnings from one layer to the other. For example, the propagation of Ⓒ induces warnings Ⓓ and Ⓔ. The JIT framework can suggest to the user to fix warning Ⓒ, which then automatically fixes Ⓓ and Ⓔ. Thus, the changeset

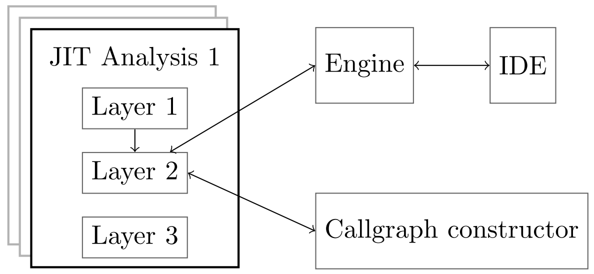

Fig. 5: Architecture of the JIT framework.

for warnings fixing is kept minimal. The reporting system can be further optimized, for example, by computing common flows across different warnings.

D. General Solution Architecture

In Figure 5, we present the JIT framework, which works as follows: The main component is the engine, which runs in an Integrated Development Environment (IDE). While the user writes code and fixes bugs, the engine chooses a suitable layer, based on the past run layers, the working set and user feedback. It launches the JIT analyses on the determined layers. All of these JIT analyses are implemented following the layered model imposed by the engine. Figure 5 illustrates a run at L2 that makes use of the information computed in L1. The different client analyses may run in parallel, while sharing some of the engine-level data such as data flow or call graph information.

The results are reported to the user as soon as they are found, allowing them to either continue developing, or fixing bugs as they are introduced, thus allowing a smooth integration in the workflow. The engine is responsible for:

- Selecting and launching the layers of the JIT analyses;
- Organizing and reporting results to the user;
- Creating quick-fixes;
- Determining and asking key questions to the user;
- Interpreting different user inputs;
- Learning from user input to remove unwanted issues.

The user-centric, adaptive nature of the JIT framework raises the problem of scanning incomplete code. Therefore, we center our JIT analyses around an on-demand model with a varying degree of uncertainty. This motivates the need for fast class loading, in particular for virtual call resolution, when one call can refer to multiple methods depending on the dynamic type of the base object.

IV. Feasibility Study

In static analysis, code loading and call graph creation account for a significant part of the initial analysis cost. Anecdotally, analyzing a “Hello world!” Java program may involve looking at hundreds of standard library classes. To improve responsiveness, a JIT analysis loads classes and generates the call graph *on demand*. This is particularly useful for the layers operating intra-procedurally. Additionally, in the context of an IDE, the user is often only interested in a portion of the codebase. Loading only the

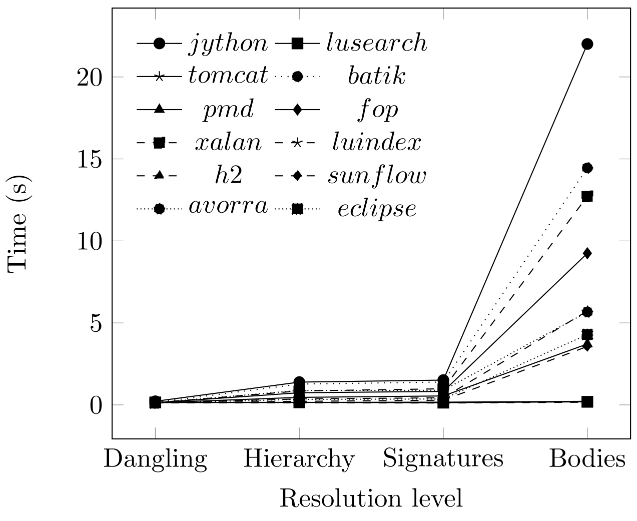

Fig. 6: Resolution times in Soot.

necessary classes shortens and distributes the computation time of both code loading and call-graph creation.

We have performed some initial measurements on the Soot framework for analyzing Java programs. These initial experiments show that it is generally possible to compute a call graph fast based on the class hierarchy. Soot has the ability to resolve Java classes at different levels, listed below in increasing order of complexity:

1) Dangling: creates an empty model of a class `C`.
2) Hierarchy: from Dangling, loads the superclasses, interfaces, and outer classes of `C`.
3) Signatures: from Hierarchy, loads the fields, method signatures and exceptions of `C`. Dependencies are loaded at level Hierarchy.
4) Bodies: from Signatures, loads the bodies of the methods in `C`. Dependencies are loaded at level Signatures.

For our measurements, we loaded the class files of several DaCapo benchmarks at different resolution levels. The results in Figure 6 show the average loading times over 10 runs. Loading at level Bodies takes up to 15x more processing time than at level Signatures (average 8x).

These measurements indicate that one can perform JIT analyses with minimal startup cost. A JIT analysis would first generate a CHA-induced call graph, (based on classes loaded at level Signatures) and refine it by loading classes at level Bodies when required by the analysis.

V. Related Work

In this section, we start by describing several similar efforts and then talk about studies that focus on the usability of static analysis tools and largely motivate our work. Lastly, we mention several relevant commercial tools.

A. Layered or Staged Analyses

Cifuentes et al. [5], [6] present Parfait, a bug checker used internally at Oracle. Parfait computes a list of potential defects and runs different and independent analyses on them in cascading order, from the least to the most expensive. Each analysis confirms the (in)validity of some defects and delegates the unconfirmed cases to the next layer. A JIT analysis has a different notion of *Layering* where each layer reuses previously computed information. This enables full use of the *Quick Response* of the first layers by quickly reporting initial results to the user, which facilitates IDE integration.

Parfait provides interesting usability features, including a fix-oriented defects grouping system, a build integration, and custom categorizations of defects. Our tool will similarly include bug categories within each layer of a given JIT analysis. Higher priority would ne given to bugs that could be easily fixed such that this fix leads to fixing other similar bugs. This eventually reduces the number of issues reported by a JIT analysis, which would encourage developers to continue using the JIT framework.

Ali and Lhotak [1] show that it is possible to precisely reason about parts of a program as long as those parts conform to the Separate Compilation Assumption. Our framework can use this assumption to precisely reason about the pieces of the program it analyzes without having to analyze the rest of the code. This will help scale to large code bases without compromising precision.

B. User Studies

Xiao et al. [12] have conducted a set of interviews to investigate the social factors affecting security tool adoption amongst developers. In general, developers continue using security tools when they feel they can trust these tools. In other words, the more confidence a developer has in a security tool (i.e., *Few False Alarms*), the more likely they will continue using it.

Lewis et al. [11] have conducted a user study at Google to understand whether bug prediction is useful for developers or not. The study finds that developers often prefer warnings to have actionable tasks (e.g., quick-fix suggestions) to *Aid Problem Repair*. In general, developers are more interested in the new parts of code that they have dealt with. Therefore, bug finding tools should have some bias to the new and report *Up-to-date Results*.

Johnson et al. [9] investigate why developers do not use static analysis tools more often. Based on the interactive interviews with developers, the authors compile a list of characteristics that should be in an ideal widely-used static analysis tool. In particular, a static analysis tool should incorporate an interactive mechanism to help developers fix bugs. Developers also expressed the need to, temporarily, suppress warnings they deem irrelevant at the moment (*Generalize User Feedback*). More importantly though, almost all developers who were interviewed agreed that a static analysis tool should not disrupt their workflow (*Minimize User Interaction*).

Ayewah et al. [3] present a study conducted by Google using the FindBugs tool to find and fix bugs. The study shows that not all reported bugs appear in production code. Ideally, our JIT framework could take into consideration whether it is analyzing production code (so it could ignore warnings that it is unsure of) or some code under development and have not undergone rigorous testing yet (so it issues warnings more conservatively).

C. Commercial Tools

Most current commercial tools such as IBM Appscan [2], Coverity [7], HP Fortify [8], and Klockwork [10] only support whole-program scans. However, Checkmarx [4] gives the possibility of running incremental scans where only the change set from the initial scan is considered. This allows a gain of time but it still disrupts the workflow in the same way as a full scan does. Checkmarx also provides a speedy fix feature that proposes optimal locations to fix reported issues. In addition to the common filtering features of commercial tools, Checkmarx also allows the user to customize the analysis itself through the use of queries. However, this requires full knowledge and understanding of the analysis, the scanned code, Checkmarx query language, and the company's scan requirements, which puts a heavy burden on the user.

VI. Conclusion

This paper advocates the idea of the JIT framework, an approach to building static analysis tools that aim to address the computational limitations of a typical analysis engine and the cognitive limitations of a typical software developer. In this framework, JIT analyses proceed in layers, starting with simple errors with lower false positive rates, all the while teaching the developer and preparing them for higher levels of complexity. This paper outlines how several typical analyses can be encoded in this JIT manner and presents some initial measurements, showing that an incremental analysis of this sort is feasible.

References

[1] Karim Ali and Ondrej Lhoták. Application-only call graph construction. In *ECOOP*, pages 688–712, 2012.

[2] IBM Appscan. http://www-03.ibm.com/software/products/en/appscan, May 2015.

[3] Nathaniel Ayewah and William Pugh. The Google FindBugs fixit. In *ISSTA*, pages 241–252, 2010.

[4] Checkmarx. https://www.checkmarx.com/, May 2015.

[5] Cristina Cifuentes. Parfait - A scalable bug checker for C code. In *SCAM*, pages 263–264, 2008.

[6] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, and Manuel Valdiviezo. Transitioning Parfait into a development tool. *IEEE Security & Privacy*, 10(3):16–23, 2012.

[7] Coverity. http://www.coverity.com/, May 2015.

[8] HP Fortify. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/, May 2015.

[9] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, pages 672–681, 2013.

[10] Klockwork. http://www.klocwork.com/, May 2015.

[11] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? Findings from a Google case study. In *ICSE*, pages 372–381, 2013.

[12] Shundan Xiao, Jim Witschey, and Emerson R. Murphy-Hill. Social influences on secure development tool adoption: why security tools spread. In *CSCW*, pages 1095–1106, 2014.