Taming the Static Analysis Beast

John Toman[1] and Dan Grossman[2]

1 **Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA**
jtoman@cs.washington.edu

2 **Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA**
djg@cs.washington.edu

Abstract

While industrial-strength static analysis over large, real-world codebases has become commonplace, so too have difficult-to-analyze language constructs, large libraries, and popular frameworks. These features make constructing and evaluating a novel, sound analysis painful, error-prone, and tedious. We motivate the need for research to address these issues by highlighting some of the many challenges faced by static analysis developers in today's software ecosystem. We then propose our short- and long-term research agenda to make static analysis over modern software less burdensome.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases static analysis, frameworks, api knowledge, library specifications

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.18

1 Introduction

The ubiquitous use of static analysis to ensure the absence of software defects has been a long-held goal of the static analysis research community. As such, we should marvel at and celebrate the mainstream success of scalable code-analysis tools that are now routine for many projects, including at large software companies (such as Microsoft [29, 43], Google [59], and Facebook [17, 16]). Although we can continue to study why static analyses are not more widely deployed [35, 8], industrial-strength static analyses are finally becoming a reality.

Static analysis researchers also now enjoy excellent tool support. Analysis frameworks exist for several popular languages and platforms. These frameworks handle tedious tasks shared across almost all static analyses, such as translation from bytecode or source-code to an intermediate representation, call-graph construction, type information, string analyses, and points-to information [37, 71, 72, 52, 15]. The developers of these frameworks deserve substantial credit: thanks to these platforms, researchers have been able to ignore complex implementation details and focus solely on implementing their analyses.

Unfortunately, writing a sound static analysis that produces useful results for real programs is now harder than ever. Analysis implementations can easily exceed tens of thousands of lines of code [48, 7]. To understand the sources of complexity, one need look no further than today's software environment. Industrial-strength analyses must handle industrial-strength applications in industrial-strength languages. Analyses must handle objects, the pervasive use of callbacks, threads, exceptions, frameworks, reflection, native code, several layers of indirection, metaprogramming, enormous library dependency graphs, etc. In our experience (and those shared by other static analysis authors), getting a realistic static analysis to

run on "real" applications requires a combination of luck,[1] multiple heuristics (which may never see the light of day in published papers), engineering effort, manual annotation, and unsatisfying engineering tradeoffs.

Our community has recognized these difficulties and work continues to be published to tackle these challenges. However, developing a novel, sound static analysis *and* testing it accurately on modern applications often remains excruciatingly painful for fundamental reasons. We begin by describing some of these reasons, using examples drawn from our experience building a static analysis for Java.[2] The difficulties we describe are shared by many other researchers. In particular, we focus on the challenges posed by enormous external libraries, pervasive use of frameworks, and the need for high-level, domain knowledge about API behavior (Section 2). We then describe our research vision for addressing each of these three pain points (Section 3), and our vision for the future of static analysis research (Section 4).

2 Static Analysis Challenges

This section describes the challenges today's static analysis writer faces. Although our descriptions are given in the context of writing an analysis for Java, the challenges we identify are not Java-specific in any fundamental way.

2.1 Libraries

No application is completely self-contained: even a simple "Hello World" application transitively depends on 3,000 classes [41]. The size of an application's transitive dependencies can dwarf the original application code, sometimes by several orders of magnitude, posing significant scaling challenges for static analysis writers [6]. For example, a highly precise, scalable field-sensitive analysis by Lerch et al. [44] exhausted 25 GB of memory when analyzing the Java Class Library (which is comprised of over 18,000 classes). In the same work, an even less precise analysis exhausted the 25 GB memory limit on 6 of 7 non-trivial applications when including external dependencies. Our own experience broadly mirror this trend: when including all external dependencies an analysis that took under a minute exhausts all available memory after running for over 20 minutes.

Some analyses consider all library code along with application code (e.g., [47, 25]). This often limits the sophistication of an analysis: in general the more expressive or complex the analysis, the less scalable it becomes. We do not suggest that useful static analyses that consider library code cannot or do not exist: as mentioned in Section 1, large companies run static analyses regularly on their codebases. Nevertheless, considering an application and all dependent libraries requires tradeoffs in analysis sophistication and enormous engineering effort.

In practice, the challenges of including all library dependencies means many static analysis writers accept incomplete portions of an application's class hierarchy and/or call-graph. However, ignoring these missing pieces is clearly unsound. Analysis writers therefore resort to one of several unappealing options. The analysis writer may provide hand-written summaries for all missing methods. This approach is precise but infeasible for even moderately

[1] We found we needed answers for a type of aliasing query unsupported by all existing off-the-shelf pointer analyses. A few weeks later, an analysis designed to answer these queries was published at a top conference.

[2] Currently under anonymous submission.

sized applications. Another option is to apply a notionally conservative summary of missing library behavior; e.g., "all data flowing into a function are propagated to the return value." This technique is still unsound as it fails to consider "out parameters" and other side effects, and is unacceptably imprecise for pure methods.

In response to this difficulty, several authors have explored how to make analysis tractable in the presence of large libraries. A widely explored technique is caching results across runs of an analysis. Caching forms the core of incremental analyses [55, 65, 5, 16, 51, 50]. However, these approaches can reuse results only from previous executions from the analysis on the same program. If an analysis fails to terminate due to large libraries there is no opportunity for caching. Kulkarni et al. have recently proposed a technique to reuse analysis results on common (i.e., library) code shared between two or more target applications [41]. However, this technique can reuse results only of the same analysis and requires programmer provided predicates describing when cached results may be soundly applied. Even the optimal approach for analyzing libraries in isolation remains an area of active research [57].

An alternative option is to write modular (or bottom-up) analyses [20, 32]. Instead of generating summaries for multiple (or infinite) calling contexts in a top-down setting, bottom-up analyses may generate summaries for methods (including library code) valid over *all* calling contexts. However, as noted by Zhang et al. [76], bottom-up approaches may ultimately need to analyze exponentially many input states limiting their scalability in practice. Thus, although theoretically appealing, "designing and implementing [modular analyses] for realistic languages is challenging" [41].

Finally, instead of relying on the hand-written or unsound rule-of-thumb summaries described above, many authors have explored automatically inferring specifications for missing library methods [12, 21, 45, 53, 56]. For example, in the context of a taint analysis, Bastani et al. [10] infer the specifications for missing methods needed to complete flows from sources to sinks. These specifications are presented to the user as candidate method specifications. Albarghouthi et al. and Zhu et al. [1, 78] have both explored using abduction to infer the minimal method specifications to verify the absence of errors. These techniques are promising, but they are currently limited to relatively simple specifications, require a human oracle, or focus on inferring preconditions for methods. These limitations mean that these techniques are unlikely to infer, e.g., the behavior of Java's thread pool or executor APIs.

The decision to exclude library implementations is motivated by scalability concerns but also affects soundness. What impact do these decisions have on the analysis results reported in the literature? It is hard to say: the answer is certainly "a non-zero number" but to our knowledge there is no empirical study on false negatives due to excluded library code nor is this commonly reported in existing analysis results. It is up to analysis evaluators (who are usually also the analysis designers and implementers) to decide if this unsoundness arises in practice for the applications being analyzed. Unless the community can devise convincing experiments that the effects of excluding library code are negligible, the current approaches used may undermine the credibility of static analysis results.

2.2 Frameworks

Applications in complex domains (e.g., web applications, GUI programs) require a common set of functionality that does not vary significantly from application to application. For example, most web applications must parse incoming HTTP requests and dispatch them to the appropriate handler code. Rather than reimplement this functionality, applications use

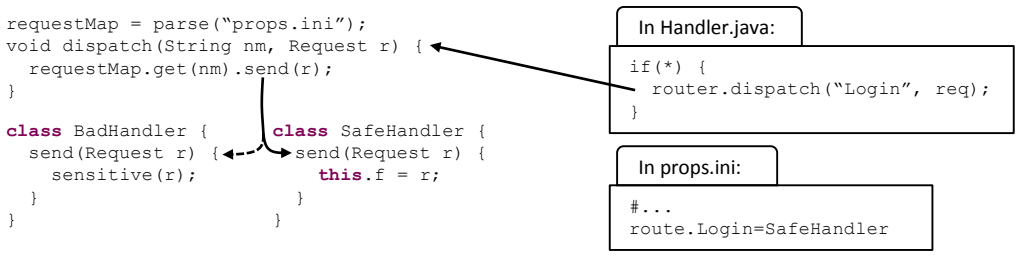

Figure 1 An invented program fragment that demonstrates string indirection commonly found in frameworks. Without the routing information in `props.ini`, the analysis must conservatively assume the program dispatches to `BadHandler` (dashed line). Many framework models incorporate this type of information.

frameworks.[3] Frameworks are skeleton applications with holes for application specific code. Frameworks generally handle "boring" tasks (e.g., parsing HTTP requests or dispatching incoming UI events) and allow the programmer to focus on application specific tasks, e.g., responding to an HTTP request or UI event.

Frameworks are notoriously hard to analyze. In the interest of reusability, framework implementations rely heavily on language features that are difficult or impossible to analyze in general, such as reflection [11, 46]. This design makes basic call-graph construction (a basic requirement of any whole program static analysis) incredibly difficult. In addition to reflection, frameworks often use multiple layers of abstraction that confound most static analyses. For example, in Figure 1, finding the exact callee of `send()` in the `dispatch()` method requires reasoning about the precise key/value pairs present in the `requestMap` variable. Without this information, the static analysis must conservatively assume any handler is invoked, leading to a false report in `BadHandler`.

However, making matters even worse, frameworks are often configured using annotations [27], XML files [66], or other static sources. For example, the mapping information necessary to precisely resolve the `send()` call in Figure 1 is found only in the configuration file `props.ini`! Another example of configurations, simplified from an application we encountered while evaluating our static analysis, is shown in Figure 2. A static analysis must either consider these external artifacts (which requires deep domain knowledge) or make conservative assumptions about the behavior of the framework (leading to a precision loss and corresponding performance hit). Ignoring a framework's code entirely is not a realistic option: applications written using frameworks often lack a distinguished "main" function making even basic call-graph construction impossible.

In practice, static analysis writers either laboriously hand write models[4] of frameworks [70, 7] or avoid evaluating their analysis on framework applications. The latter option is unrealistic considering trends in application engineering but is understandable given the former option: constructing framework models by hand is a time-intensive and frustrating process. Our own experience analyzing Java web applications that use the Servlet framework is representative of this difficulty. The Servlet framework is relatively simple but building a sound model of the framework required reading parts of three specification documents: the Servlet, JSP

[3] The line between a library and framework is fuzzy. In this context, we use framework to refer to code that provides scaffolding upon which an application is built.
[4] A model is a compact, potentially non-executable, description of framework behavior.

```
1 <bean id="filterChain" class="FilterChain">
2   <property name="chain">
3       PATTERN_TYPE_APACHE_ANT
4       /logout=logoutFilter,anonymousProcessingFilter
5       /login=basicProcessingFilter,rememberMeProcessingFilter
6   </property>
7 </bean>
```

Figure 2 A simplified framework configuration fragment. The `filterChain` "bean" is bound to an instance of `FilterChain`. The values of fields of the `chain` bean are configured with property elements (line 2). Lines 3–5 define a tiny url-mapping DSL stored in the `chain` field. Building a complete model of this configuration requires not only a model of bean definitions, but an interpreter for this DSL. In our analysis we opted for a one-off, hand-coded interpretation of the DSL.

(JavaServer Pages) and EL (Expression Language) specifications, which together total 557 pages of prose. The Servlet framework is not an outlier: the reference document for Spring [66], a framework that builds on the Servlet framework, totals *910* pages.

Building a good model requires more than just understanding the framework. In addition to being sound, a model must be precise enough that client analyses can complete in reasonable amounts of time. For example, the largest performance gains in our analysis did not come from optimizations in the core analysis, but from aggressively including more domain specific knowledge into our Servlet model to improve call-graph precision.

Our community recognizes the difficulty of building these models: as recently as 2015 [11], a complete model of the Android framework was a significant research contribution. In addition, there has been work to simplify writing these models using a DSL [67]. However, expecting static analysis writers to build sound and efficient models for every framework is unrealistic. Other techniques [7, 48, 28, 70, 31, 75, 77] also require some form of programmer annotation or development which limits their adoption to new frameworks. However, evaluating new analyses on applications that use older, simple-to-model frameworks is equally undesirable as it ignores trends in modern software development.

2.3 High-Level API Knowledge

Analysis writers often require domain knowledge about the behavior of an API. For example, to soundly construct call graphs, analyses must handle the concurrency and reflection APIs of the Java Class Library. The reflection and concurrency APIs are just one example: many different analyses need high-level knowledge about an API. For example:

- What methods read or write from the database? [68]
- What methods return personal or sensitive information? [7]
- What methods may block execution of the current thread? [40]
- What methods and classes are part of a container abstraction? [23]

The answers to these questions are difficult to extract automatically and require reading the relevant documentation. The unfortunate state-of-the-art is that a static analysis developer interested in the answers to these questions must therefore manually audit an API to find the methods of interest. This is no trivial task: the reflection API alone contains over one hundred methods spread across 17 different interfaces and classes. The methods found during the audit are then usually added to a list of "special" methods; the analysis developer must then incorporate *ad hoc* handling for these methods to the analysis. For example, the call-graph construction facility of Soot [71], a popular analysis framework for Java, contains

a hard-coded list of reflection and thread methods. WALA [72], another framework for Java, maintains its own list in an external XML file.

For many combinations of analysis domains and APIs, it is likely another analysis author has already performed a similar audit. However, no shared infrastructure exists to reuse and share the results of these audits, condemning analysis writers to re-audit APIs. In addition to wasting time, this process is error prone: failure to properly account for high-level API knowledge may make an analysis unsound. We encountered an otherwise sound and precise alias analysis that failed to consider `Class.newInstance()` an allocation site and therefore could not find aliases of reflectively instantiated objects.

3 Future Directions

The problems described in the previous section are not insurmountable. This section sketches future research directions, community initiatives to overcome some of these challenges, and our research agenda.

3.1 Toward Sound Library Handling

As noted in Section 2.1, the size of application code is often dwarfed by library code, leading static analysis authors to exclude the library code out of scalability concerns. We sketch future research directions to address these concerns.

Exhaustive Top-down Summaries. Top-down function summaries are difficult to reuse across analysis runs, as they are highly context-dependent and the probability of reaching a calling context identical to one in cache decreases as the complexity of the domain increases. Exhaustively enumerating input calling contexts as proposed in some work [58] becomes infeasible as the complexity and size of possible input contexts grows.[5] Recent work on StubDroid [6] by Artz and Bodden addresses some of these issues by soundly handling holes in the library call-graph and automatically computing the input contexts of interest. However, their approach assumes a specific representation of dataflow facts within a particular analysis domain. Nevertheless, this technique represents a promising step forward toward library summary precomputation. Our community should explore how to generalize these techniques to work on any combination of dataflow facts and analysis.

Analyzing Analyses. We plan to explore developing automated techniques to compare the power of two or more analyses. In particular, we plan to develop an automated semi-decision procedure that can determine if one analysis always over-approximates another analysis on all code fragments. In other words, the procedure will decide if the results of one analysis imply the results of another analysis on all programs. Recent work on comparing the behaviors allowed by memory models [74] has shown that it is possible to answer these types of queries using automated theorem provers such as Z3 [22].

This technique will have several important applications. This research may enable sound reuse of cached analysis results from different analyses. If the procedure determines analysis A over-approximates analysis B, then cached results from B may be safely reused within A (with some loss of precision). The developed procedure will also allow our community to

[5] Even for seemingly simple domains (e.g., access-paths [69] limited to length 1), this approach is unlikely to scale.

compare the precision of two or more analyses. Finally, this procedure could find soundness bugs in analyses. A developer may choose a concrete interpreter as one "analysis", and query the semi-decision procedure to verify her analysis over-approximates the concrete interpreter.

Analysis Semi-Refinement. Recent work on caching and incremental analysis provide a promising approach to solving scalability concerns on large codebases. However, in the current state-of-the-art, cached results cannot be used across analyses, so every new analysis effectively begins with a blank slate of results to draw upon. No amount of caching helps if the initial run of an analysis never terminates! The research sketched above potentially alleviates this issue, but only if cached results *always* soundly over-approximate the analysis using these results. However, we expect that only highly related analyses in the same problem domain will exhibit this property, which in turn limits opportunity for reuse.

We hypothesize that there are analyses that may not always produce over-approximate results but may sometimes agree under certain conditions. We hope to explore automatically determining when one analysis *conditionally* over-approximates another. For example, two analyses may model the heap incompatibly, but otherwise produce the same results on code with no heap accesses. In this case, cached results for a code fragment may be shared between analyses if the fragment does not access the heap. Given two analyses A and B, we aim to synthesize a predicate such that analysis B over-approximates A on fragments of code for which the predicate is true. If the predicate reduces to a simple syntactic check, results or summaries from unrelated analyses can be easily reused by another analysis to improve efficiency.

Automatic Synthesis of Weakened Analyses. A common technique for static analysis is to add precision "knobs" to an analysis [34, 37, 38]. These knobs allow the analysis user to trade performance for precision. However, constructing these knobs requires careful engineering on the part of the analysis designer and implementer. Similarly, staged analyses (e.g., [33, 26, 36]) exploit a precision/performance tradeoff by iteratively applying more and more precise analyses to suppress false positives or discharge verification of conditions not provable by less precise analyses. Unfortunately, the staged analysis designer must either "luck" into two or more analyses that yield compatible results with different levels of precision, or (more likely) design and implement multiple (related) versions of an analysis.

We plan to research techniques to *synthesize* less precise (but more scalable) versions of existing analyses. These weakened analyses may be used as fast preanalyses, or to handle large library codebases. One possible direction for this work is to develop a technique to take flow-sensitive analysis and create a flow-*insensitive* version (in the style of Andersen points-to analysis [4]). In addition to this flow-sensitive/-insensitive tradeoff (which is well-known within our community), we plan to explore other axes along which analyses may be transformed for performance gains.

3.2 Sound, Automated Handling of Frameworks

The techniques discussed in Section 2.2 for handling frameworks all rely on some manual effort by analysis writers. At times, new frameworks become popular and old ones make changes so fast that keeping up disincentivizes work in the space.[6] We therefore propose two possible research directions that handle frameworks without programmer intervention.

[6] Krishnamurthi reports this experience in work on semantics for JavaScript [39].

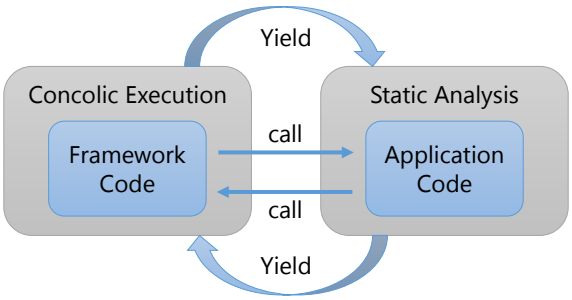

Figure 3 High-level architecture of concolic analysis.

Concolic Analysis. Framework implementations are difficult to analyze statically, but when executed often follow only a handful of paths determined by static values that can be easily accessed at analysis time (e.g., a configuration file or annotations). These characteristics suggest that concolic execution [62, 63, 30] can be an effective approach to analyzing framework implementations. Concolic execution extends traditional symbolic execution by falling back on concrete execution for code that cannot be modeled symbolically (e.g., due to expressions unsupported in the underlying automated theorem prover). For example, a concolic executor can precisely handle framework code that reflectively instantiates objects based on static configuration values by simply executing the relevant code.

However, concolic execution by itself cannot analyze entire framework-based applications. Although the scalability of both concolic and symbolic executors has improved, and there have been amazing advances on semi-decision procedures like CEGAR [19], it remains a challenge to verify programs with many paths of execution and complicated data dependencies. In particular, on programs with infinite paths of execution, these techniques either fail to terminate, artificially finitize the program, or limit tool execution with a fixed time budget. Thus, concolic execution will struggle to verify, e.g., Android applications that process unbounded streams of input events, or web applications that accept infinite sequences of requests.

Given the scalability concerns of concolic execution (and other formal methods techniques) and the difficulty of precisely analyzing extremely flexible framework implementations, we believe that a single, unified analysis approach is insufficient to verify or analyze framework-based applications. We instead plan to explore a *hybrid* analysis technique that combines concolic execution and traditional static analysis. We have termed this technique concolic analysis. Under concolic analysis, framework code is executed concolically, whereas application code is over-approximated with a meet-over-all-paths static analysis. A visualization of this technique is shown in Figure 3. When control passes from the framework to the application, the concolic executor yields to a static analysis. Similarly, calls from the application back into the framework cause the static analysis itself to yield to the concolic executor. By using the best approach on each part of a program, concolic analyses combine the efficiency of static analysis and the completeness and precision of concolic execution. For example, reflective operations in framework code can be concretely executed, while unbounded loops in application code can be efficiently over-approximated using fixpoint iteration.

Although other authors have examined combining concrete execution and static analyses, these approaches have either used information recorded during executions in a static analysis [13, 24, 73], or used dynamic analysis as a post-processing step to prune false positives or

discharge verification conditions [9, 42, 18]. In contrast, concolic analysis tightly couples different analysis techniques to cooperate concurrently to analyze different parts of a monolithic application.

Automatic Model Synthesis. Recent advances in synthesis technology and techniques have enabled automatic synthesis of complex hardware memory models [14] that were previously hand-axiomatized through multiple iterations of publications [61, 64, 54, 2, 3, 60, 49]. This pattern echoes current work on building models for the Android framework: several papers have been published over the years, each claiming more precise (and sound) models of a single framework. Our community should also focus on automatic synthesis of framework models.

Full specifications of complex frameworks are likely more complicated than those for hardware memory models, so complete specification synthesis is likely intractable. Thus, we envision focusing on synthesizing specifications describing framework behavior for a single, specific application. For example, the input/output behavior of framework methods can be recorded during either directed randomized testing (e.g., [30]) or execution of a program's functional test suite. These traces can be used as inputs to a synthesis procedure that generates specifications (expressed in a DSL) for framework methods. The quality of these generated specifications necessarily depends on the completeness of the observed traces. However, as noted above, frameworks are often driven by static, deterministic configurations and annotations, so we expect only a handful of executions will provide relatively complete set of input/output examples for the framework methods executed by an application.

3.3 Infrastructure for Sharing API Knowledge

Given the overlap in knowledge needs of static analysis writers, the static analysis community would benefit from an open platform to share API knowledge. The high-level API knowledge described in Section 2.3 can often be expressed in a few short English words. Concise tags therefore are a good format to express this API knowledge. We propose the community create and maintain a shared, open database that associates API elements (i.e., classes, methods, etc.) with tags that express the high-level knowledge needed by analysis developers.

Each tag would express a property that is common to multiple methods in different APIs. For example, the `TelephonyManager.getDeviceId()` method of the Android framework returns a unique identifier and is treated as a source of sensitive data for information integrity analyses. This method, and the analogous `UIDevice.uniqueIdentifier()` of the iOS framework could be tagged with the tag `"sensitive-source"`. The collection of methods associated with this tag in the proposed database would replace the hand-curated list of source methods used by many security analyses. Similarly, methods from the reflection API of the Java Class Library (e.g., `Class.newInstance` or `Method.invoke`) would be associated with the tag `"indirect-flow"` indicating that these methods indirectly invoke another method by name. As with the sensitive source example, the methods associated with this tag would replace the hard-coded lists found in many program analysis frameworks' call graph construction facilities.

The implementation and deployment of the tag database poses no major technical hurdles: similar web applications are widely deployed in industry and enjoy extensive library and framework support. We foresee there will be two major challenges. First, as tags are expressed using natural language, different users of the database may interpret the same tag differently. However, we are confident that the community can standardize around a set of tags with widely accepted and understood definitions. Second, although some tags may be assigned automatically (e.g., tags identifying setter and getter methods) other tags require human

knowledge. For this type of information, we envision that after analysis developers manually collect domain knowledge for an API, they then tag the API elements in the shared database.

Although the development and deployment of the shared database does not present core PL research opportunities, this initiative will have an immediate impact on the community. For one, it will free the analysis developers from the tedious, error-prone task of auditing APIs, and improve the soundness of analyses by ensuring no important methods are missed (as in the alias analysis described in Section 2.3). Further, analyses in related domains could be fairly compared as all analyses would consider the same methods of interest (e.g., sources and sinks).

4 Conclusion

Despite advances in tooling and mainstream success, static analysis development is still a painful process. We have outlined our research vision for tackling some of these pain points. Our proposals do not represent the full space of solutions, and there are other difficult aspects of analysis development we have not addressed. Mitigating or eliminating the challenges faced by static analysis writers is a ripe area for research. We believe using static analysis and formal methods techniques to tackle these difficulties (i.e., static analyses for static analyses) is a particularly exciting research direction. In addition, we hope the community will invest in sharing knowledge and results across research projects. Our proposed tag database initiative is a potential first step; there are even more opportunities for community-wide collaboration to ease the burden of constructing static analyses.

Acknowledgments. The authors would like to thank Doug Woos, James Bornholt, Jared Roesch, Zachary Tatlock, and Chandrakana Nandi for feedback on early versions of this work. We would also like to thank our shepherd, Ruzica Piskac, and the anonymous reviewers for their insightful comments.

References

1 Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.

2 Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP*, 2009.

3 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *CAV*, 2010.

4 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

5 Steven Arzt and Eric Bodden. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *ICSE*, 2014.

6 Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *ICSE*, 2016.

7 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

8 Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

9 Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Symposium on Security and Privacy*, 2008.

10 Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, 2015.

11 Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2015.

12 Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *PLDI*, 2013.

13 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

14 James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *PLDI*, 2017.

15 David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *CAV*, 2011.

16 Cristiano Calcagno and Dino Distefano. Infer: an automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, 2011.

17 Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Open-sourcing facebook infer: Identify bugs before you ship. `https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/`, 2015.

18 Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for Javascript. In *PLDI*, 2009.

19 Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

20 Patrick Cousot and Radhia Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, 2002.

21 Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *CAV*, 2015.

22 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

23 Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

24 Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA*, 2007.

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

26 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2), 2008.

27 Apache Foundation. Apache struts 2. `https://struts.apache.org/`.

28 Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android. Technical Report CS-TR-4991, University of Maryland, November 2009.

29 Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE software*, 25(5), 2008.

30 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

31 Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.

32 Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, 2007.

33 Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, 2011.

34 Ben Hardekopf, Ben Wiedermann, Berkeley Churchill, and Vineeth Kashyap. Widening for control-flow. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2014.

35 Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, 2013.

36 Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, 2007.

37 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: a static analysis platform for javascript. In *FSE*, 2014.

38 Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale javascript applications (t). In *ASE*, 2015.

39 Shriram Krishnamurhti. Personal Communication, 2016.

40 Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for c/pthreads. In *ASE*, 2016.

41 Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating program analyses by cross-program training. In *OOPSLA*, 2016.

42 Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *Partial Evaluation and Semantics-based Program Manipulation*, 2008.

43 James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE software*, 21(3), 2004.

44 Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *ASE*, 2015.

45 Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, 2009.

46 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, JoséNelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2), 2015.

47 Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014.

48 Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

49 Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWERmultiprocessors. In *CAV*, 2012.

50 Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *FSE*, 2013.

51 Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient incremental static analysis using path abstraction. In *FASE*, 2014.

52 George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, 2002.

53 Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, 2002.

54 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

55 Lori L. Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12), 1989.

56 Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *PLDI*, 2007.

57 Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *FSE*, 2016.

58 Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, 2008.

59 Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, 2015.

60 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

61 Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, 2009.

62 Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

63 Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

64 Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7), 2010.

65 Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *ICSM*, 2001.

66 Spring framework. `http://spring.io/`.

67 Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA*, 2011.

68 Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep typechecking and refactoring. In *OOPSLA*, 2008.

69 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, 2013.

70 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, 2009.

71 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 1999.

72 WALA – T. J. Watson Libraries for Analysis. `http://wala.sf.net/`.

73 Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *ISSTA*, 2013.

74 John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. Automatically comparing memory consistency models. In *POPL*, 2017.

75 Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *ICSE*, 2015.

76 Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*, 2014.

77 Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

78 Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *ASPLAS*, 2013.