Know Your Analysis: How Instrumentation Aids Understanding Static Analysis

Philipp Dominik Schubert
philipp.schubert@upb.de
Heinz Nixdorf Institut
Paderborn, Germany

Richard Leer
rleer@mail.upb.de
Heinz Nixdorf Institut
Paderborn, Germany

Ben Hermann
ben.hermann@upb.de
Heinz Nixdorf Institut
Paderborn, Germany

Eric Bodden
eric.bodden@upb.de
Heinz Nixdorf Institut
Fraunhofer IEM
Paderborn, Germany

Abstract

The development of a high-quality data-flow analysis—one that is precise and scalable—is a challenging task. A concrete client analysis not only requires data-flow but, in addition, type-hierarchy, points-to, and call-graph information, all of which need to be obtained by wisely chosen and correctly parameterized algorithms. Therefore, many static analysis frameworks have been developed that provide analysis writers with generic data-flow solvers as well as those additional pieces of information. Such frameworks ease the development of an analysis by requiring only a description of the data-flow problem to be solved and a set of framework parameters. Yet, analysis writers often struggle when an analysis does not behave as expected on real-world code. It is usually not apparent what causes a failure due to the complex interplay of the several algorithms and the client analysis code within such frameworks. In this work, we present some of the insights we gained by instrumenting the LLVM-based static analysis framework *PhASAR* for C/C++ code and show the broad area of applications at which flexible instrumentation supports analysis and framework developers. We present five cases in which instrumentation gave us valuable insights to debug and improve both, the concrete analyses and the underlying PhASAR framework.

CCS Concepts • **Theory of computation → Program analysis**.

SOAP '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6720-2/19/06…$15.00
https://doi.org/10.1145/3315568.3329965

Keywords Static analysis, framework, instrumentation, C/C++

ACM Reference Format:
Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. 2019. Know Your Analysis: How Instrumentation Aids Understanding Static Analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3315568.3329965

1 Introduction

There are several reasons why the development of a precise and scalable data-flow analysis is difficult. Concrete client analyses often need additional helper analyses to provide them with type-hierarchy, points-to, and call-graph information in order to provide precise results [5].

However, writing a client analysis and all required helper analyses from scratch is impractical. For this reason, many different static analysis frameworks have been developed to ease that process. Frameworks from academia, among others, include Soot [9], Doop [6], Wala [20], OPAL [7], Soufflé [8], and PhASAR [16]. Those frameworks provide implementations for the helper analyses and generic data-flow solvers that are able to solve a given user-specified data-flow problem in a fully automated manner. Thus, an analysis writer can focus on specifying the actual analysis problem.

Encoding an analysis is still tedious as an analysis developer has to perform a tremendous amount of complex tasks. Encoding an analysis in a general purpose language, for instance, as required by frameworks such as Soot, Wala, or PhASAR still requires an analysis developer to write several hundred to thousand lines of code that comprise the problem description [19]. Choosing the parameters for their analysis is also non-trivial as the parameters have to be chosen according to an analysis's requirements and the target program's characteristics to trade off precision and scalability.

The complexity further increases for frameworks that use analyses encoded in general purpose languages as they often

use distributive frameworks like *inter-procedural finite distributive subset problems* (IFDS) [13], *inter-procedural distributive environments* (IDE) [15], or *weighted pushdown systems* (WPDS) [14] to achieve a decent scalability [5]. Those data-flow frameworks, in turn, solve a problem in a multi-step process. In general, an internal representation, e.g. exploded super-graph or pushdown system, is constructed first and then, the problem is solved on that representation in a second step. A buggy analysis might withstand the construction but still fail the actual solving process.

Eventually, an analysis developer has encoded their analysis successfully with respect to a micro-benchmark that has been used to develop it (c.f. Section 2). Applying the analysis to real-world software, however, they will frequently observe their analysis to fail [19]. The reasons for such a failure can be manifold and oftentimes hide in the complex interplay of the involved algorithms and the complex nature of the analysis description.

Debugging analysis failures is non-trivial as it requires knowledge of algorithms, solvers, executing system, target programs, and intermediate representations. Detecting the cause of the failure with help of a standard debugger is usually a tedious to impossible task as the analysis developer needs to debug through large amounts of non-analysis code; it might be not helpful at all if the analysis is correct but the executing operating system causes an analysis run to fail, e.g. if the system runs out of memory. The work of Nguyen et al. [12] presents a special debugger for static analysis which shows the severity of the problem. The use of logging techniques produces log files that are too large to effectively debug failure on real-world code or slows down the analysis execution to a point that is not acceptable.

In this work, we show how a flexible instrumentation of a static analysis framework is able to aid the understanding of concrete analysis runs. Using an instrumentation in combination with a post-processing step of the recorded data allows to spot anomalies and root causes of analysis failures that would otherwise remain hard to detect when analyzing real-world software. In addition, an extensive instrumentation allows for detailed performance benchmarks which, in turn, allow for spotting bottle-necks and fine-tuning an analysis. Different algorithms can be assessed based on their performance on the target code and a framework user is able to precisely determine how much time of an analysis run is spend in which parts of a framework.

In summary, this paper makes the following contributions:

- It presents our highly flexible instrumentation of the PhASAR framework [16] called *PAMM*,
- and an experience report that presents five cases in which the instrumentation provided us with valuable insights into concrete analysis runs that we used for debugging and optimizations.

2 Analysis Development Process

In this section, we briefly explain a commonly used process to develop a client data-flow analysis.

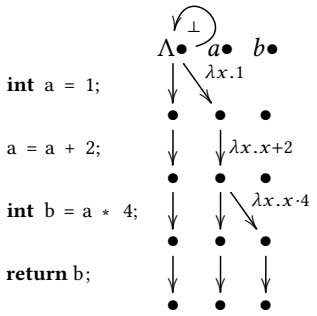
The most labor-intensive task involved in this process is crafting the description of the analysis problem. Depending on the static analysis framework that has been chosen, a developer needs to implement flow functions or specify rules in order to model the interaction of a program's statements with the data-flow facts that the developer is interested in.

The creation of an analysis description is an incremental process. In order to evaluate the correctness and the level of precision, a developer starts specifying their analysis to handle the basic language features and tests it on small example programs. The results reported by the analysis on the example programs are checked and compared to the expected results. Once the quality of the results suffices for the initial example programs, some more advanced example programs are written. These example programs form a micro benchmark that allows a developer to evaluate the quality and completeness of their analysis description. The example programs, that act as test cases, and the analysis code are alternately enhanced until the analysis is able to cope with all common language features and obtains the desired precision. Several micro benchmarks such as DroidBench [2], SecuriBench [3], DaCapo [4], or the Toyota ITC benchmark [17] have been established to evaluate the quality of an analysis which shows that the development process described here is common practice.

When the complexity of the programs of the micro benchmark has risen to a certain level, another part of the development process becomes relevant: the framework's parametrization. Many frameworks allow for the construction of the type hierarchy, call-graph and points-to information which become necessary depending on the complexity of the test programs and the desired precision of the analysis when eventually run on real-world software. For instance, for the construction of call-graph and points-to information developers can choose from a variety of algorithms such as CHA, DTA, VTA, Spark for call-graphs, or Andersen or Steensgard-style algorithms for points-to information. Computations can be chosen to be performed in a full analysis mode or an on-demand manner. Finding *the best* or at least suitable parameters, however, is challenging. While heavy-weight algorithms may produce precise results on the small test programs, they are oftentimes too slow to be used on larger real-world code. Finding the optimal point between scalability and precision is key and an ongoing challenge [5].

3 Distributive Frameworks

If the data-flow problem to be solved is distributive it can be encoded using analysis frameworks such as IFDS [13], IDE [15], or WPDS [14]. The key idea of those frameworks is to create summaries for each procedure p which can then



$$
\begin{array}{ll}
\langle \Lambda, * \rangle \hookrightarrow \langle \Lambda, * \rangle & \lambda x.x \\
\langle \Lambda, 2 \rangle \hookrightarrow \langle a, 3 \rangle & \lambda x.1 \\
\langle a, 3 \rangle \hookrightarrow \langle a, 4 \rangle & \lambda x.x + 2 \\
\langle a, 4 \rangle \hookrightarrow \langle a, 5 \rangle & \lambda x.x \\
\langle a, 5 \rangle \hookrightarrow \langle a \rangle & \lambda x.x \\
\langle a, 4 \rangle \hookrightarrow \langle b, 5 \rangle & \lambda x.x \cdot 4 \\
\langle b, 5 \rangle \hookrightarrow \langle b \rangle & \lambda x.x
\end{array}
$$

Figure 1. Exploded super-graph (left) and rules of a pushdown system (right) for a linear constant propagation performed on program P shown in Listing 1.

be (re)used in each subsequent context p is called. Therefore, analyses encoded within those frameworks turn out to be scalable and precise due to the ∞-context sensitivity that is achieved using the summary mechanism.

The distributive frameworks construct a specific representation of the problem according to the developer's problem specification first, e.g. an exploded super-graph (ESG) in IFDS/IDE or a pushdown system in WPDS. Then, the problem is solved using the internal solver specific representation. The exploded super-graph for IFDS/IDE and the set of rules Δ of a pushdown system for a linear constant propagation performed on program P shown in Listing 1 are shown in Figure 1. A linear constant propagation is an analysis that tracks constant variables and their values, and variables that linearly depend on constant values through the program.

```
int a = 1; a = a + 2; int b = a * 4; return b;
```

Listing 1. Program P

4 Implementation

While designing PAMM we opt for a ready-to-use mechanism to collect different measures related to static analysis. Three basic types of measures turned out to be useful in practice: timer, counter, and histogram. We provide code to start, pause, stop and reset different timers, increase and decrease counters by a given value, and add data points to histograms. All measures used are identified by user-specified IDs and must be registered before use. This allows us to detect and minimize exceptional measurements caused by flawed code instrumentations, e.g. a misspelled ID. We implemented PAMM as a singleton to minimize boilerplate for the construction and destruction of PAMM. Each instrumentation instruction is wrapped into a corresponding preprocessor macro to hide implementation details. This also allows a user to disable PAMM without removing any code instrumentation manually, and thus, guaranteeing zero overhead during non-evaluation runs of PhASAR. However, recompilation is necessary to enable or disable PAMM.

Since code instrumentation is tedious and oftentimes requires a profound knowledge of PhASAR's internal structure, we provide a default instrumentation for all parts of PhASAR relevant to static analysis. Multiple measures can be grouped which allows a user to only collect the data of analysis runs that they are currently interested in. A user is able to instrument their own analysis code and register their instrumentation in a new group to record their client measures without using the default (*full*) framework instrumentation. Our instrumentation of the *core* group, for instance, comprises, among other measures, runtime information for each step of an analysis run and statistics of the analyzed program.

5 Experience Report

In this section, we discuss five cases in which PAMM provided valuable insights for debugging and optimizations.

5.1 Bug Finding and Detection of Anomalies

The GNU coreutil programs [1] are frequently used as a subject for evaluations on real-world C programs. To check the capabilities of the PhASAR framework to handle real-world code, we benchmarked it on the coreutils using several different analyses encoded in IFDS. We found that some of the analysis runs caused a segmentation fault. The backtracing capability of the GNU debugger GDB gave no useful clues what might have caused the segmentation fault. The Valgrind [11] tool for dynamic debugging memory issues was not usable while analyzing the coreutils as it slowed down the execution too much in the order of days. Unfortunately, it also did not report any errors using the micro-benchmarks that have been used to develop the analyses. As we used PAMM to record the analysis runs of the different coreutils and visualized the results, we found a correlation between lines of code, number of call-sites of the programs and the occurrences of segmentation faults. The plot is shown in figure Figure 2. The analysis of coreutils with more than 240k lines of code has led to segmentation faults and with more than 20k call sites have been likely to crash. Based on the recorded data, we assumed that the recursive nature of our IFDS/IDE solver implementation could be troublesome due to the operating system's default stack limit for processes. Increasing the stack limit indeed solved the problem and almost all programs of the coreutils could be successfully analyzed using a larger stack limit. The exceeded stack limit has been confirmed with help of the Linux kernel's ring buffer, too.[1] A small number of coreutils still caused segmentation faults regardless of the chosen analysis. That suggested that either the framework or all analyses did not cope with an infrequently used language feature. The backtracing capabilities of GDB revealed the segmentation fault to be caused by the flow function that handled function calls. A manual inspection uncovered that the failure was caused by C-style

[1]The kernel stores a certain number of (error) log messages in a ring buffer.

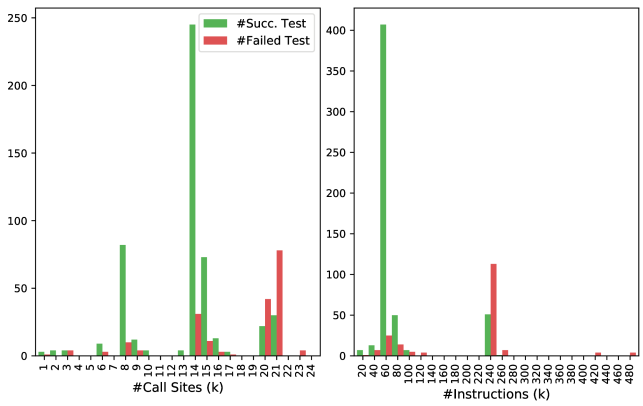


Figure 2. Number of analysis runs that executed (un)successfully and corresponding number of call sites and instructions of the program under analysis.

variadic functions which have not been handled by the analyses yet. At call sites that call variadic functions, the number of actual and formal parameters may not match. After adjusting the responsible flow functions to under-approximate that language feature in the analyses, all analysis runs could be executed successfully. Our handling of variadic functions is unsound. However, it retains an acceptable level of precision whereas a sound handling would lead to impractically imprecise results.

In a different scenario, we inspected the distribution of data-flow facts generated by an analysis. That is, we wanted to know the sizes of the sets of data-flow facts generated by the flow functions. For that reason, we instrumented PhASAR's IFDS/IDE solver to record the number of data-flow facts (ESG edges) generated at each statement. With the help of that information, we aimed at optimizing for the container type used to store the data-flow facts. Our initial implementation used STL's `std::set` which implements a red-black tree. In order to optimize for the container type, we measured the occurrences of different set sizes for an IFDS taint analysis which are shown in Figure 3. Figure 3 confirms that the vast majority of sets only contain very few elements. Therefore, we might wanted to switch to an implementation that is better suited for small sets such as Boost's `flatset` implementation which uses a sorted vector and a binary search to allow for logarithmic lookups. Interestingly, however, some sets contained an exceptionally large amount of facts caused by what is called "overtainting" [18]. We revisited the implementation of the taint analysis and found a place at which all context-insensitive aliases have been accidentally tainted when a tainted value has been stored to a memory address. We could change the responsible flow function to only generate the relevant aliases. We will continue the discussion of the set implementations in terms of performance in Section 5.2.

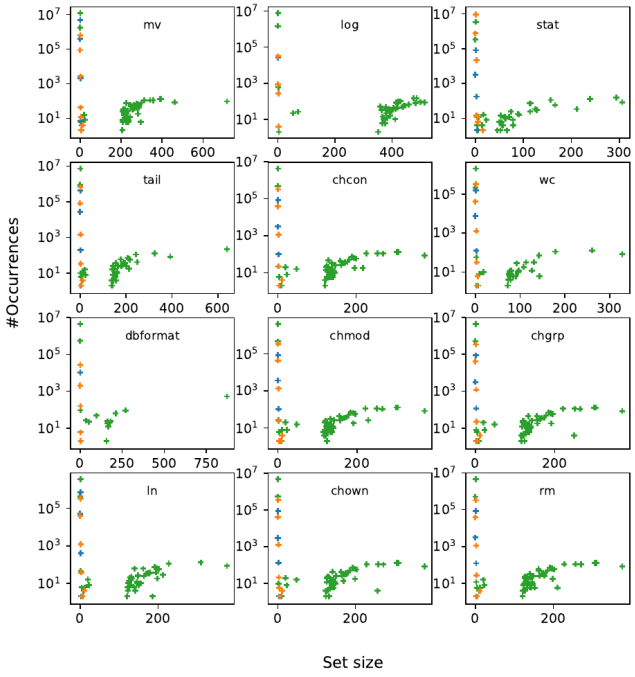


Figure 3. Occurrences of the different sizes of sets generated during ESG construction for several target programs.

5.2 Performance Benchmarking for Optimizations

Let us revise our assumption from Section 5.1 that the more compact `flatset` implementation might be more efficient than the STL implementation in our case. In order to determine which implementation is better suited to hold the flow facts, we created a separate git branch in which we replaced the usages of `std::set` by `boost::flatset`. Since we initially already heavily instrumented PhASAR, we did not need to change any of the code other than specifying the container type to be used. We evaluated the performance by performing some analysis runs on the coreutils and some tools of the LevelDB project using a compile of PhASAR that uses `std::set` and compared the runtimes of various IFDS data-flow analyses with the figures obtained using a compile of the novel branch that uses the `flatset` implementation. Figure 4 shows a plot of the performance figures that we produced. In general, the difference in performance is negligible. The IFDS/IDE solver uses the sets to communicate with the analysis's description only. Much more copying or accessing of those sets would be needed to cause a larger difference in performance. We thus sticked to `std::set` for convienience.

The *C++11* standard introduced novel types for smart pointers that can be used to automatically deallocate heap memory that is no longer in use. `std::unique_ptr` can be used to handle memory that is limited to only one user; it is deallocated when the pointer goes out of scope unless ownership is explicitly transferred to another scope. Another type of smart pointer is `std::shared_ptr` that can be

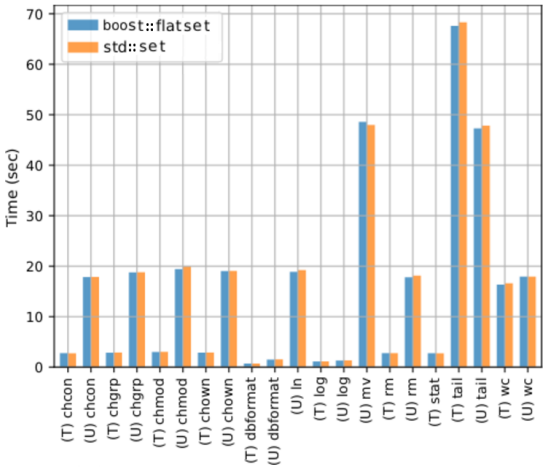


Figure 4. Runtimes using `std::set` vs. `boost::flatset` in seconds for different programs and analysis runs.

used if a piece of heap memory has more than one owner. It uses reference counting to determine at which point the memory can be deallocated. Our IFDS, IDE and WPDS solver implementation query the client analysis's code for flow and edge functions for each statement of the target program. The analysis code provides the respective solver with suitable implementations of these small function objects by returning a shared pointer. Shared pointers entail some amount of overhead due to the additional code that maintains the references and their larger size in memory. Using PAMM we were able to compare the initial implementation of PhASAR using smart pointers to an implementation that uses raw pointers. Figure 5 shows the comparison of smart and raw pointers in terms of runtimes. It can be observed that the use of shared pointers slows down each analysis run. We mitigated the noticeable slowdown due to the use of shared pointers as described in the following. Since the use of raw pointers in application code is considered bad style, we introduced a manager class that exclusively owns the shared pointers to flow and edge function objects following a recommended pattern: the manager class is able to hand out raw pointers to users that are "only looking" at the object; and eventually deallocates all objects it owns once its lifetime ends. Thus, the more expensive copying of shared pointers can be avoided which prevents the slowdown.

In Section 2 we discussed that parameterizing an analysis framework is challenging. Always using the most precise algorithms wherever possible may lead to great precision but also to unsatisfactory performance for larger target programs. We do not want to rely on choosing an algorithms' parameters based on experience only. Depending on the target program under analysis the experience from analyzing one project might lead to false assumptions for another project. Therefore, we used PAMM to instrument all parts of PhASAR that are involved to perform a full analysis run.

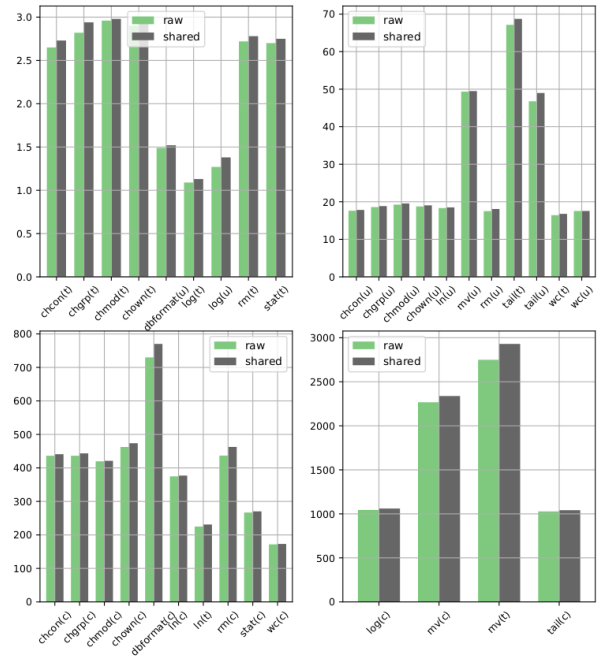


Figure 5. Runtimes using smart vs. raw pointers in seconds for different programs and analysis runs.

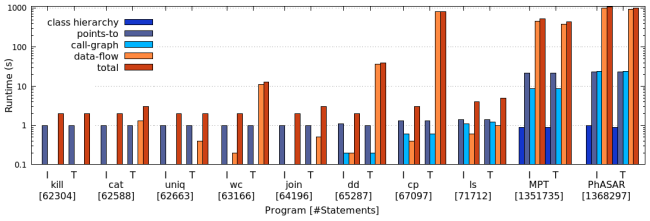


Figure 6. Runtime spend in different parts of an analysis.

Thus, we are able to reveal the analysis runtime distribution of a concrete analysis run. Figure 6 shows such a distribution. Using that knowledge, one can then start adjusting specific parameters to speed up certain computations to cope with larger programs while comparing the precision based on the results obtained for the micro-benchmarks.

6 Related Work

The setup of a static analysis, encoding it in a framework, and finding a suitable parametrization, is a demanding task. Several works have been dedicated to reduce and ease the work that needs to be accomplished by analysis developers.

Lerch et al. developed an approach following the principle of separation of concerns [10]. They propose an approach that effectively separates different aspects and implementations to allow for better maintainability, testability, and reuse of individual components.

A special debugging environment for static analysis called *VISUFLOW* has been developed by Nguyen et al. [12] for

the Java ecosystem. It allows for a direct debugging of the analysis code in Soot without having to step through any of the framework code which makes the process of debugging an analysis feasible in practice.

7 Conclusion

In this paper, we presented the design and implementation of a flexible mechanism for instrumentation called PAMM. We presented five scenarios in which it provides us with valuable insights that help us to understand what a concrete static analysis run on real-world code actually does. In general, we find that PAMM can be used in addition to or whenever standard debugging techniques are unable to track down the cause of an analysis failure. We advocate for integrating ready-to-use mechanisms that aid analysis understanding and debugging into the analysis frameworks to support developers, rather than burdening them with yet additional work. The data collected by the fine-grain instrumentation in combination with a suitable visualization allows for a gaze into concrete analysis runs. Thus, it enables us to spot anomalies and implausible figures. With these insights, we are able to determine how an analysis performs and where it goes wrong helping us to solve issues in a user's analysis code and the PhASAR analysis framework.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre 901 "On-The-Fly Computing" under the project number 160364472-SFB901 and the Heinz Nixdorf Foundation.

References

[1] 2019. coreutils. Retrieved 04/02/2019 from https://www.gnu.org/software/coreutils/coreutils.html

[2] 2019. DroidBench. Retrieved 04/02/2019 from https://github.com/secure-software-engineering/DroidBench

[3] 2019. SecuriBench. Retrieved 04/02/2019 from https://suif.stanford.edu/~livshits/work/securibench/intro.html

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[5] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-based Static Analyses (and How to Master Them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA '18)*. ACM, New York, NY, USA, 85–93. https://doi.org/10.1145/3236454.3236500

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[7] Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614630

[8] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.

[9] Patrick Lam, Eric Bodden, Ondvrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective.

[10] Johannes Lerch and Ben Hermann. 2015. Design Your Analysis: A Case Study on Implementation Reusability of Data-flow Functions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2015)*. ACM, New York, NY, USA, 26–30. https://doi.org/10.1145/2771284.2771289

[11] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[12] Lisa Nguyen, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2018. VISUFLOW, a Debugging Environment for Static Analyses. In *International Conference for Software Engineering (ICSE), Tool Demonstrations Track*.

[13] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[14] Thomas Reps, Stefan Schwoon, and Somesh Jha. 2003. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 189–213. http://dl.acm.org/citation.cfm?id=1760267.1760283

[15] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1-2 (Oct. 1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

[16] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410.

[17] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. 2015. Test Suites for Benchmarks of Static Analysis Tools. In *Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (ISSREW '15)*. IEEE Computer Society, Washington, DC, USA, 12–15. https://doi.org/10.1109/ISSREW.2015.7392027

[18] Asia Slowinska and Herbert Bos. 2009. Pointless Tainting: Evaluating the Practicality of Pointer Tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 61–74. https://doi.org/10.1145/1519065.1519073

[19] John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.18

[20] WALA 2019. WALA. Retrieved 04/02/2019 from http://wala.sourceforge.net/wiki/index.php/Main_Page