Into the Woods: Experiences from Building a Dataflow Analysis Framework for C/C++

Philipp Dominik Schubert*, Richard Leer*, Ben Hermann† and Eric Bodden*‡
*Paderborn University, Paderborn, Germany, {philipp.schubert, eric.bodden}@upb.de, rleer@mail.upb.de
†Technische Universität Dortmund, Dortmund, Germany, ben.hermann@cs.tu-dortmund.de
‡Fraunhofer IEM, Paderborn, Germany

***Abstract*—While traditional static analysis—albeit its complexity—is a topic that is well understood, we especially struggle when it comes to implementing its concepts. Designing and modeling software that implements static analysis is a challenging task. However, developing usable static analysis implementations and providing toolboxes to researchers and practitioners is key to advance the overall progress in this field. This paper reports on the lessons learned from developing the PhASAR and Soot static data-flow analysis frameworks. We present some of the key mistakes of our first implementations of PhASAR and their corrections. From those corrections we distill guidelines that will be helpful to static analysis developers and their users. In our work, we identified modularity as the key guiding principle supported—directly or indirectly—by virtually all other guidelines.**

***Index Terms*—static analysis, guidelines, framework, C/C++, LLVM**

I. Introduction

We implemented the PhASAR framework [1] and open-sourced it in 2018 due to the lack of open-source data-flow analysis implementations for C/C++ that suited our needs. The analysis of C/C++ programs is notoriously hard, as this family of languages presents some unique properties that are seldom found in other languages. These properties include its low-levelness, arbitrary pointers to *memory* (including `void*`), its deliberately unsafe type system, multiple inheritance, possession of a preprocessor, and language features such as `const_cast` and `setjmp`/`longjmp`—to list only a few. Yet, these languages are heavily used in practice making them relevant analysis targets.

While there are relatively lightweight analysis approaches that conduct *syntactic* checks on a given target program, and which are able to analyze even million lines of code in minutes, analysis approaches that compute *semantic* properties of a program are more heavyweight. Many interesting static analysis problems, such as data-flow-, shape, or typestate analysis require detailed, inter-procedural semantic program information. To solve these kinds of analysis problems, detailed abstractions are required that involve complex data-flow solvers, complex analysis domains, and oftentimes multiple different, potentially interleaving, helper analyses, forming an ”analysis blend” that eventually provides useful results. This paper focuses on such semantic analyses.

For many real-world sized target programs, detailed abstractions that are necessary to solve those more heavyweight analysis problems lead to high runtime and memory requirements. This makes it almost impossible to integrate such analyses into software development processes, let alone compilers [2]. Actual solutions to analysis problems are often undecidable, forcing analysis developers to resort to approximations. In addition, the complex concepts and algorithms that are required to solve analyses that reason about semantic properties of a program are one of the (many) reasons that lead to a restricted supply of static-analysis implementations that are able to solve those kinds of analysis problems.

Because there is no reference implementation, guide, or any form of advice on how to build a static data-flow analysis framework for C/C++, we initially borrowed several design decisions from the Soot framework [3], [4], and LLVM [5]. We built PhASAR on top of LLVM as it provides a usable, industrial-strength compiler infrastructure that offers an intermediate representation (IR) and, in addition, provides compilation of target programs into IR and all basic functionalities for inspection and transformation of the IR. Although we were able to use existing, static-analysis toolboxes and compiler infrastructures that allowed us to avoid repeating engineering mistakes others made before, such as (accidentally) introducing tight coupling, we still encountered various difficulties and had to learn many lessons the hard way.

In this work, we thus report on our findings of what makes the development of such frameworks easier. We present the basic concepts of static data-flow analysis in a nutshell and report on the key mistakes and design flaws of early implementations of PhASAR for which we drew several design ideas from Soot [3] and LLVM [5]. From their corrections and PhASAR's partial redesign we elaborate guidelines on how to model and implement static analysis that is usable in practice.

We found that the dominating overall design principle that static analysis implementations must follow is *modularity*. A modular design greatly counters complexity and allows one to build further abstractions on top of basic building blocks. Modularity is involved—directly or indirectly—in six of our eight major guidelines that we distilled from our experience.

In summary, this paper makes the following contributions:

- It presents a report on the key mistakes and their corrections in designing and implementing the concepts of static data-flow analysis within the PhASAR [1] framework,
- and shows guidelines derived from the corrections that will be useful to static analysis developers and their users.

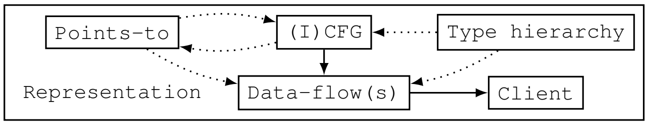

Fig. 1: Dependency model of a concrete client analysis.

II. Background

In this section, we briefly present the basic concepts of static analysis that need to be taken into account when modeling and implementing a static data-flow analysis framework.

A. Client Analyses and Their Dependencies

For presentation here we assume that a concrete client data-flow analysis problem has to be solved.

An analysis problem is solved on a given target program that usually has been translated to some intermediate representation that allows for easier analysis due to a reduced instruction set and an accompanying API to inspect the IR. Depending on the program property that the concrete client analysis is interested in, the client may impose several, possibly interleaving, dependencies on additional helper analyses shown in Figure 1. These also need to be conducted on the given target program.

To check whether a property of interest holds, the client analysis requires data-flow information of one or more data-flow analyses which can be computed by generic solvers that are parameterized with a description of the problem. Depending on the nature of the client problem—distributive or non-distributive—various data-flow solvers may be used [6]. In an intra-procedural setting, *non*-distributive problems can be solved with the monotone framework [7]. In an inter-procedural setting, the call-strings [8] or VASCO [9] approaches are required. Distributive problems can be solved efficiently using the *inter-procedural finite distributive subset* (IFDS) [10], *inter-procedural distributive environments* (IDE) [11] or *(weighted) pushdown systems* ((W)PDS) [12] approaches, all of which compute fine-grain, per-fact, reusable procedure summaries.

Those data-flow solvers are always depending on the target program's (inter-procedural) control-flow graph (ICFG) that guides the solvers through the program, indicated by a solid edge in Figure 1. In addition, they may depend on points-to and type hierarchy information in case variables of pointer types are encountered which is indicated by dotted edges in Figure 1. Depending on the desired precision, the ICFG, in turn, may depend on points-to, type-hierarchy, and virtual-function-table information. A cyclic dependency is introduced due to the fact that precise points-to information also depends on ICFG information [6].

Information on the type hierarchy (and virtual function tables in object oriented languages) have no further dependencies. A client analysis is likely to transitively depend on all of the above information.

B. Parametrization and Configurations

Virtually every algorithm that computes a piece of static analysis information can be heavily parameterized; oftentimes to trade off precision and scalability [13]. Depending on the concrete client analysis and given target program, some parametrizations may be more preferable than others.

In addition to the parametrization of individual analysis algorithms that even affect each others' properties, several configuration options may be applied that can be considered *global*. Those global configuration options apply to entire analysis runs, i.e., they apply to every entity involved in the dependency model presented in Section II-A. Some of the configuration options are implementation independent. For instance, one could model *soundness* as a global option that may carry one of the values *sound*, *soundy* [14], or *unsound*. That option uniformly applies to all analyses required by a client and is independent of any concrete implementation. Other configuration options such as logging, export of results, etc. are global but implementation-dependent.

C. Analysis Styles

On top of the dependencies and setup presented above, various *analysis styles* or *strategies* may be used to conduct an analysis. Those styles include, among others, whole program, incremental, demand-driven, and compositional analysis. All these analysis styles require the same static-analysis information but each style requires them in a slightly different form. Demand-driven analysis, for instance, requires information on forward and backward control flows [15]. Incremental update analysis even requires additional communication between those different pieces of information.

III. Lessons Learned

In this section, we elaborate on implementation mistakes and design flaws we made and had to fix in PhASAR's initial implementations, respectively. We consider it a mistake whenever changes in the code or design have been necessary that required a disproportionate amount of time when building novel (analysis) abstractions on top of existing ones.

A. Modularity and Encapsulation

To allow for efficient inter-procedural analysis, we built our initial implementation of PhASAR starting from a generic and parameterizable IFDS/IDE solver implementation similar to the HEROS [16] data-flow solver frequently used with Soot.

From our experience on Soot we knew that modularity is a key element when it comes to designing an analysis framework. Many of Soot's important data types are implemented as singletons that make it easy to globally access information wherever needed, but also break modularity and local reasoning. When requiring callgraph information, for example, a user sets up an instance of a callgraph type using its constructor. Especially novice users, however, cannot possibly know that there are additional setup possibilities using a singleton configuration object, as there is no direct coupling of the type's interface and its setup. Those singletons also prevent several important use cases as it prevents loading multiple target programs into a single analysis process, for instance.

We borrowed several design decisions regarding the modeling of the solver interfaces from Soot. Thus, the solver operates on a ”problem” interface whose implementations correspond to concrete analysis problems. The problem interface’s constructor takes an implementation of the `ICFG` interface that guides the solver through the program. The client problem is also free to use the information provided by the ICFG. PhASAR manages the underlying target code using the `ProjectIRDB` type. One may thus conclude that providing an ICFG implementation that computes control flows on the code managed by the `ProjectIRDB` to the analysis ”problem” would be sufficient.

While this design allowed us to implement and debug the solver, and generally allows to specify and solve basic inter-procedural client analysis problems, it oversimplifies the concepts of static analysis. For instance, the existence of points-to or type-hierarchy information is not mentioned at all, making it unusable for more complex client analyses.

Because of our modular design and missing capabilities to globally access information, it led to several severe code smells such as passing points-to information via the concrete ICFG implementation to the client problem and eventually prevented us from building further abstractions on top without completely breaking modularity and encapsulation, and thus losing control over the complexity.

Hence, we improved on our model and implemented it according to Section II-A. We modeled each entity as an individual interface that can be separately used from all others.

We employ the type system to match our model shown in Figure 1 and express intent: an analysis problem always needs an ICFG implementation that guides the solver through the program and is passed as a reference, and may uses additional information which we pass as pointers. These pointers can be `nullptr`s to indicate *information not available/used*. As the client analysis is potentially provided with all information on the helper analyses, it can, in turn, spawn additional (helper) data-flow analyses itself, if required.

Modularity and Encapsulation: Modularity and encapsulation are key to keep complexity manageable which has been impressively shown by the LLVM project [5], too. Design a model that is expressive enough to capture all interactions of the different analysis algorithms that are interesting to you, explicitly. Implement each entity of your model such that it can be used (and tested) individually.

B. Accessing Information

Depending on what needs to be computed, the various involved algorithms will need to share lots of information.

Due to our prior experience with stateful singletons in Soot that not only decrease its maintainability but are also particularly bad for thread-parallelism, we could avoid large collections of information that are shared globally. Whenever possible we make information available using uniform parametrization across all entities of a certain type. For instance, we offer uniform constructors for all types of data-flow problems: call strings, IFDS, IDE, etc.—they all accept equal parameter lists.

This allows us to build further abstractions on top of our model shown in Figure 1. We recently started implementing an analysis strategy concept as presented in Section II-C. It enables users to set up entire analysis runs that use one of the presented strategies in only a few lines of code.

To allow for an exchange of information for strategies that not only need to share but also update depending information like incremental update analysis, we use a special `ReviseInfo` type. We modeled the type to carry information on which kind of information needs updates and what pieces of code are affected. The corresponding strategy implementation has been built on top of the `ReviseInfo` type and controls the actual exchange of information using a mediator pattern. By using this model, we can keep a strict modular design and avoid making every piece of information globally available. Similar to the above, Helm et al. present a novel approach that allows for modular, collaborative program analysis by using so-called *blackboard* systems in [17].

Even the information that is general to all respective algorithms, such as the level of soundness that we implemented as suggested in Section II-B, is managed separately for each analysis run. Modeling this information as global variables would forbid us from running individual analyses concurrently: a functionality that is often needed by more complex analyses that need to spawn additional helper analyses.

In our experience, the only information that can be shared globally safely is implementation-specific information that does not affect the semantics of an analysis. Thus, we implemented the constant global implementation-specific information about the system as a special thread-safe singleton configuration type that allows one to access this information.

Accessing Information: Avoid weakening interface boundaries that counteract modularity and encapsulation. If needed, rather than giving individual unrelated components access to each other, exchange information with help of proxy exchange types which are handled by a mediator. Provide unified interfaces to access information to ease building novel abstractions on top of existing ones.

C. Bugs and Debugging

Once we solved analysis problems with a first version of the basic analysis infrastructure, we frequently observed crashes, strange program behavior, and incorrect analysis results.

Finding the root causes of bugs in static analysis is a challenging and time-consuming task, as many different analyses are involved while performing a concrete analysis run [18]. Standard debugging techniques such as debuggers are hard to use, as one needs to step through a tremendous amount of non-related solver code when debugging a client analysis. Complex analysis domains make it hard to even display interesting pieces of information in a meaningful way.

For that reason in a subsequent revision we instrumented the entire framework. Each piece of code involved in solving a concrete analysis run has been instrumented using logging techniques and functionalities to record data that is relevant to static analysis like number of callgraph edges per callsite, data-flow facts generated per statement, etc. After post-processing the recorded data, we are able to gain insights about the undesired program behavior that eventually let us track down bugs. We describe this approach in detail in [18].

Nguyen et al. built a specialized debugger called VisuFlow [19] to ease the debugging process. Unfortunately, this approach is currently only applicable to the Java ecosystem.

Extending on Lerch and Hermann's insights [20], we additionally follow a test-driven approach in PhASAR. We frequently observed that when implementing novel features, other seemingly unrelated parts of the framework broke; bringing us back to the problem described in this section. Those bugs would, without corresponding unit tests, either provoke further undesired behavior causing time-consuming debugging sessions, or—even worse—produce bugs that remain undetected and may corrupt critical analysis users. As a consequence, we now use test-driven development to implement all major parts of the framework.

Bugs and Debugging: Integrate means to allow for debugging especially complex parts, e.g. using instrumentation. Implement individual components using test-driven development to ensure their correctness and retain the ability to check correctness continuously in an automated manner.

D. Parametrization, Configuration and Usability

The large amount of parametrization and setup options decrease overall usability, especially for novices.

In a first implementation of the IR-managing `ProjectIRDB` type, we offered a broad variety of functionalities, many of which could be accessed through public member functions, which in some instances required a distinct order of function calls, e.g. certain IR annotation passes needed to be run before being used by other functionalities. This design turned out to be error prone and difficult to use as it is too easy to introduce mistakes by confusing the order of calls.

To make it more difficult to use the `ProjectIRDB` class in an incorrect manner, we thus revised large parts and moved lots of tasks directly into constructors. Based on the experience gained from Soot we avoided separating a type's interface and its setup. To reduce the amount of configuration needed, for non-essential parameters and configuration options we chose sensible default parameters. Whenever possible we reduced the number of parameters even further. For instance, a callgraph based on points-to information can be constructed by specifying the enumerator option `CallGraphAnalysisType::OTF` in the `LLVMBasedICFG` constructor's parameter list. This specific callgraph option requires additional points-to information. However, if no additional points-to information are provided by a user of that type, the required information is constructed on-the-fly.

Parametrization, Configuration and Usability: Model entities from static analysis as types and couple a type's setup *directly* to its interface. If possible, avoid complex setup mechanisms, use simple constructors instead. For novice users, make it sufficiently hard to misuse a type. Reduce the amount of essential parameters to a minimum by providing suitable default parameters. Compute missing information on-the-fly rather than aborting with an error message.

E. Flexible Usage Modes

Initially, we implemented PhASAR as a command-line tool. However, we received many requests to also allow for further use cases. We extended the framework to allow for the usage of a plugin mechanism. Users can thus ignore most of the framework's infrastructure and focus on specific details they are interested in without the need for modification and costly recompilation of PhASAR's code base. C++ compilation times are typically relatively long compared to C or Java, even for incremental builds. Reasons for that include the hundreds or even thousands of header files that need to be (re)processed for every compilation unit, the monolithic linking process, complex parsing of the complex syntax, code generated by templates, and optimizations. We counteract the compilation times with potent build machines.

Due to the framework's modularity, we could also offer individual functionalities as libraries. Thus, users are free to only choose whatever functionalities they are interested in and can integrate these parts in their own tools. We added full CMake support to PhASAR which eases using it as a library and building tools on top of it.

Since the removal of the aforementioned restrictions of the usages, we noticed that the number of people interested in the framework increased. We could observe a growing number of users and recently received several valuable performance optimizations for our callgraph algorithms from a company that uses our callgraph construction functionalities in their software product.

Flexible Usage Modes: Provide flexible use cases unless you have good reason to apply restrictions. Do not make any assumptions on the users's workflows because people will come up with usages that you did not think of.

F. Build Systems

The earliest versions of PhASAR used Makefile as a build system. This worked as long as PhASAR comprised only a few source files, but after a few months we realized that this harmed the project's maintainability. The monolithic Makefile made it difficult to organize the project in suitable subcomponents, to integrate other libraries, and to allow for cross-platform support.

At the point at which only the initial creator of the Makefile could maintain it, we stopped and replaced the build system with CMake. CMake is an open-source, cross-platform, *modular* tool chain that is designed to build, test and package software. It is now also the de-facto standard for many modern C and C++ open-source projects. Due to its modularity it allows for an easy integration with other software projects—a property that makes it suitable especially for research projects that often need to combine multiple projects to create a prototypical implementation quickly.

Build Systems: Choose a build system that suites the project's needs and integrates well with others in advance. Think ahead and assume that the project will grow not only in terms of its code base but also its number of users.

G. LLVM IR Generation

Following Section III-C, we develop small micro benchmarks comprising several single-file programs that are used to test certain aspects of an analysis implementation.

While LLVM IR can be obtained for individual compilation units by running the clang compiler with the `-emit-llvm` flag, it is difficult to obtain LLVM IR for larger, more complex projects. However, that is exactly what analysis writers wish to do in order to evaluate an analysis' scalability and ability to deal with real-world code. C and C++ neither have a real module system nor a standardized build mechanism. Instead, individual compilation units are compiled into object files that are eventually linked to (hopefully) produce the desired binary. Preprocessor macros and other important flags passed to the compiler can change the semantics and correctness of the final binary. To produce LLVM bitcode for a given real-world project, one needs to extract the exact compile *and* link commands encoded in the build system used by the project. Doing so manually is an infeasible task if one recalls the multitude of different build systems such as Makefile, CMake, Bazel, etc.—if the project uses a build system at all.

Luckily, compiler wrappers such as WLLVM [21] and (a faster implementation in Go) GLLVM [22] have been developed. These tool chains interrupt the compiler and extract the compile command to produce LLVM bitcode for the compilation unit under processing. The path to the LLVM bitcode is stored in an artificial section in the resulting object code. Linker commands are interrupted as well, and, in addition to the ordinary linking job, the bitcode paths of the object codes that are linked are collected and placed in an artificial section of the resulting binary. To produce whole program LLVM IR, the paths to the bitcode files that constitute the binary can be automatically extracted and linked, and finally subjected to a whole program analysis.

LLVM IR Generation: Use WLLVM [21] and GLLVM [22] to build whole program LLVM bitcode files from unmodified C/C++ projects.

H. Contributing Guidelines

We are still affected by having failed to provide contributing guidelines in the early days. Initially, we did not provide suitable contributing guidelines and coding standards, and after we did, we did not enforce them at first. Due to the various contributions from students and practitioners that the project received over time, it has picked up different coding styles and code of varying quality.

A unification of coding styles and overall improvement of code quality using automated analysis and transformation tools such as *clang-tidy* was not directly possible due to various corner cases that those automated approaches cannot handle. Manual unification was very expensive and underwent an incremental process. We updated pieces of code that are adjacent to new features to ensure software evolution over time. It finally allowed us to remove those corner cases and to employ automated tooling.

Contributing Guidelines: Provide contributing guidelines and documentation at the beginning of the project. Assume that the framework has multiple users and developers that provide contributions, which is what eventually will happen. Use tools for automated analysis and transformation to retain a uniform and high-quality code base. Take measures to support community building and communication.

IV. Related Work

Whereas there are several mature program-analysis frameworks from academia like Soot [3], OPAL [23], WALA [24], or Doop [25], there is very little advice on how to actually design and implement the underlying theory.

Some insights on good design of static analysis frameworks, provided by Soot's maintainers [4], refer to avoiding redundant re-computations by using incremental or reactive computation, and quasiquoting for easily generating code from templates. Allowing to independently release framework extensions without having them included in the main distribution also greatly benefits the tool and its community.

Experience reports on applying static analysis tools in commercial context emphasise the importance of low false positive rates and clear error messages to overcome warning blindness of tool users [26]. Additionally, tools must handle real-world code: resilience and robustness is vital when coping with large code bases and peculiar code constructs [27].

An extensive experience report on how to employ distributive, summary-based static analysis to benefit analysis precision and performance is given by Bodden [6]. The report presents practical design tricks for data-flow analysis.

Schubert et al. presents an approach that modularly computes and summarizes all pieces of static analysis information required to answer queries of a concrete client analysis as shown in Figure 1 in [28]. This approach shows that modularity not only eases implementation but also improves flexibility and counters the complexity of static analysis itself.

V. Conclusion

In this paper, we reported on major design flaws and implementation mistakes that we detected in our first implementations of the PhASAR framework. From those incidents, for which we had to provide corrections in order to keep the complexity manageable, we distilled guidelines that we think are useful to static analysis writers and its users.

As shown in this paper, even when using knowledge of the past and falling back to design ideas of existing frameworks one may still suffer from design decisions that turn out to be not advisable. Applying those guidelines helped us to improve PhASAR's overall quality. It reduced complexity, made its usage less error prone and eased building novel abstractions, eventually advancing the progress in this field. We recognized that PhASAR gained more attention from the community as the latest statistics on GitHub suggest.

Acknowledgment

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-SFB901/3 and the Heinz Nixdorf Foundation.

References

[1] P. D. Schubert, B. Hermann, and E. Bodden, “Phasar: An inter-procedural static analysis framework for c/c++,” in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.

[2] “Personal communication with domagoj babic, google,” 2018.

[3] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot - a java bytecode optimization framework,” in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[4] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, “The Soot framework for Java program analysis: a retrospective,” in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011. [Online]. Available: http://www.bodden.de/pubs/lblh11soot.pdf

[5] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[6] E. Bodden, “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*, ser. ISSTA '18. New York, NY, USA: ACM, 2018, pp. 85–93. [Online]. Available: http://doi.acm.org/10.1145/3236454.3236500

[7] J. B. Kam and J. D. Ullman, “Monotone data flow analysis frameworks,” *Acta Inf.*, vol. 7, no. 3, p. 305–317, Sep. 1977. [Online]. Available: https://doi.org/10.1007/BF00290339

[8] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York, NY: New York Univ. Comput. Sci. Dept., 1978. [Online]. Available: https://cds.cern.ch/record/120118

[9] R. Padhye and U. P. Khedker, “Interprocedural data flow analysis in soot using value contexts,” in *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, ser. SOAP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 31–36. [Online]. Available: https://doi.org/10.1145/2487568.2487569

[10] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[11] M. Sagiv, T. Reps, and S. Horwitz, “Precise interprocedural dataflow analysis with applications to constant propagation,” *Theor. Comput. Sci.*, vol. 167, no. 1-2, pp. 131–170, Oct. 1996. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(96)00072-2

[12] T. Reps, S. Schwoon, and S. Jha, “Weighted pushdown systems and their application to interprocedural dataflow analysis,” in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 189–213. [Online]. Available: http://dl.acm.org/citation.cfm?id=1760267.1760283

[13] M. Hind and A. Pioli, “Which pointer analysis should i use?” in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 113–123. [Online]. Available: https://doi.org/10.1145/347324.348916

[14] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, “In defense of soundiness: A manifesto,” *Commun. ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2644805

[15] S. Jaiswal, U. P. Khedker, and S. Chakraborty, “Bidirectionality in flow-sensitive demand-driven analysis,” *Science of Computer Programming*, vol. 190, p. 102391, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642320300022

[16] E. Bodden, “Inter-procedural data-flow analysis with ifds/ide and soot,” in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, ser. SOAP '12. New York, NY, USA: ACM, 2012, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259052

[17] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, “Modular collaborative program analysis in opal,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 184–196. [Online]. Available: https://doi.org/10.1145/3368089.3409765

[18] P. D. Schubert, R. Leer, B. Hermann, and E. Bodden, “Know your analysis: How instrumentation aids understanding static analysis,” in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 8–13. [Online]. Available: https://doi.org/10.1145/3315568.3329965

[19] L. Nguyen, S. Krüger, P. Hill, K. Ali, and E. Bodden, “Visuflow, a debugging environment for static analyses,” in *International Conference for Software Engineering (ICSE), Tool Demonstrations Track*, 1 Jan. 2018.

[20] J. Lerch and B. Hermann, “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*, ser. SOAP 2015. New York, NY, USA: ACM, 2015, pp. 26–30. [Online]. Available: http://doi.acm.org/10.1145/2771284.2771289

[21] (2021, March) Wllvm—whole program llvm. [Online]. Available: https://github.com/travitch/whole-program-llvm

[22] (2021, March) Gllvm—whole program llvm in go. [Online]. Available: https://github.com/SRI-CSL/gllvm

[23] M. Eichberg and B. Hermann, “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*, ser. SOAP '14. New York, NY, USA: ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2614628.2614630

[24] (2019, April) Wala. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page

[25] (2018, August) Doop. [Online]. Available: http://doop.program-analysis.org/

[26] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[27] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: Using static analysis to find bugs in the real world,” *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646374

[28] P. D. Schubert, B. Hermann, and E. Bodden, “Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis,” in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Møller and M. Sridharan, Eds., vol. 194. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 2:1–2:31. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14045