SOUFFLÉ : On Synthesis of Program Analyzers*

Herbert Jordan[1], Bernhard Scholz[2], and Pavle Subotić[3]

[1] University of Innsbruck, Austria
[2] University of Sydney, Australia
[3] University College London, U.K.

Abstract. SOUFFLÉ is an open source programming framework that performs static program analysis expressed in Datalog on very large code bases, including points-to analysis on OpenJDK7 (1.4M program variables, 350K objects, 160K methods) in under a minute. SOUFFLÉ is being successfully used for Java security analyses at Oracle Labs due to (1) its high-performance, (2) support for rapid program analysis development, and (3) customizability. SOUFFLÉ incorporates the highly flexible Datalog-based program analysis paradigm while exhibiting performance results that are on-par with manually developed state-of-the-art tools. In this tool paper, we introduce the SOUFFLÉ architecture, usage and demonstrate its applicability for large-scale code analysis on the OpenJDK7 library as a use case.

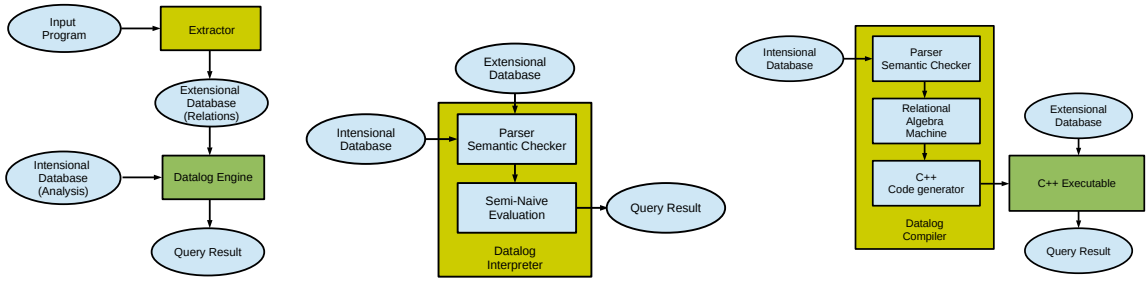
1 Introduction

Among the reasons for the slow industrial adoption of static program analysis is the lack of sufficient customizability and scalability in tools. Recently, the use of Datalog-like languages, has had a resurgence in several computer science communities [9], particularly, in the area of program analysis [3, 2, 16, 4, 12, 18] where tools such as μZ [10], LogicBlox [11] and bddbddb [18] have shown great promise. In these tools, Datalog acts as a domain specific language to express custom program analyses concisely, reducing the complexity of developing program analyzers. The drawback of this approach is that program analyses specified in Datalog typically experience reduced performance compared to manually implemented tools. A notable reason for this decrease in performance appears to be the "one size fits all" approach of evaluating Datalog programs, i.e., Datalog engines generally lack the ability to specialize their evaluation process for a given instance of a program analysis specification.

To close the performance gap, we have developed a tool called SOUFFLÉ that overcomes the performance limitations of standard Datalog evaluation by performing an efficient synthesis of Datalog specifications to executable C++ programs. As a result, SOUFFLÉ is able to perform analyses on-par with state-of-the-art manual tools while retaining the advantages of employing a domain specific language for expressing static program analyses. For example, [6] reports the ground-breaking capability of obtaining points-to analysis results for the OpenJDK library in under a minute. With the same dataset, SOUFFLÉ can obtain a similar performance (35s) using a general purpose analysis infrastructure on a multi-core commodity desktop system.

In this tool paper, we give an overview of the SOUFFLÉ framework; notably its architecture, optimizations and expected performance on very large code bases. We conclude with a summary of on-going developments of the SOUFFLÉ infrastructure.

* Parts of this research was conducted while visiting Oracle Labs, Australia as assistants and visiting professor.



(a) Datalog Setup (b) Conventional Interpreter Scheme (c) SOUFFLÉ Translation Scheme

Fig. 1. Comparison: standard Datalog evaluation versus the architecture of SOUFFLÉ

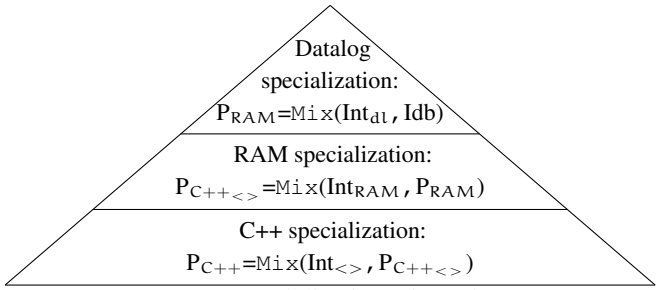
2 How It Works

A Datalog program [1] consists of an extensional database, which is defined by facts, and an intensional database, which is defined by rules. In a setup for static program analysis, the extensional database represents an input program in relational form. The relational representation of an input program is obtained from an extractor [15] describing the relevant semantics of the input program for a given program analysis. The intensional database represents the program analysis specification phrased as Horn clause formulae over finite domains. Fig. 1(a) illustrates the workflow for static program analysis in Datalog. The query result of the Datalog execution represents the actual result of the program analysis. While standard schemes for evaluating Datalog are generally optimized for reducing the amount of redundant computation, e.g., the conventional, interpreter-based semi-naïve evaluation scheme [1] as shown in Fig. 1(b), they lack the ability to specialize their evaluation for a given program analysis specification instance.

SOUFFLÉ takes a different approach: Instead of evaluating a Datalog program on-the-fly, we treat a Datalog program as a specification that is synthesized to a C++ program. The C++ program is compiled, and executed with the extensional database (i.e. facts) as an input. Essentially, the generated executable becomes an analyzer in its own right. Fig. 1(c) depicts our translation scheme, where the Datalog specification is first parsed and semantically checked. The input specification is then translated internally to an imperative Relational Algebra Machine (RAM) program. The RAM program is further translated to a C++ program with OpenMP annotations for parallel execution and C++ template based meta-programming elements. In the last stage, an OpenMP/C++ compiler translates the generated code to a highly optimized, parallel program.

2.1 A Hierarchy of Specializations

To achieve a synthesis of Datalog specifications to C++, we follow a staged specialization hierarchy as depicted in Fig. 2(a). At each stage, a specialization step, as characterized by Futamura projections [8], is applied. The foundation is provided by an abstract transformation `Mix` that, given an interpreter `Int` and a source program `Source`, yields a specialized program amalgamating the interpreter and the source program. The specialized program performs the same computation as the source program (executed by the interpreter) – yet more efficiently. In Fig. 2(b), the semantic equivalence is shown between evaluation under an interpreter `Int` and the program produced by the `Mix` transformation [8]. What is of particular interest, is that at each specialization phase, information is revealed that enables opportunities for code optimizations that were not possible at earlier stages. As a consequence, the binary code produced by our specialization hierarchy is on-par in terms of run-time and memory usage with state-of-the-art hand crafted code.



(a) Specialization Hierarchy

$$\begin{aligned} \texttt{result} &= [\![\texttt{Int}]\!](\texttt{Source}, \texttt{Input}) \\ &= [\![\texttt{Source}]\!]_{\texttt{Int}}(\texttt{Input}) \\ &= [\![\texttt{Mix}(\texttt{Int}, \texttt{Source})]\!]\texttt{Input} \\ &= [\![\texttt{Prog}]\!]\texttt{Input} \end{aligned}$$

(b) Application of Futamura's Projection

Fig. 2. Specialization Hierarchy and First Futamura Projection of Semi-Naïve Evaluation

The first specialization P_{RAM}=Mix(Int_{dl}, Idb) sees the Semi-Naïve Evaluation [1] as the interpreter, Int_{dl}. It is specialized with the intensional database, Idb, corresponding to the analysis specification. As a result, we receive a relational algebra machine program P_{RAM} that expresses the computation of the specified analysis as a series of fix-point computation steps over relational algebra operations. From a high-level viewpoint, the specialization of the Semi-Naïve evaluation is a translation of a declarative Datalog program to an imperative relational algebra program.

The next application of Futamura's projection is performed on the RAM program, i.e., $P_{C++_{<>}}$ = Mix(Int_{RAM}, P_{RAM}), that has been generated by the first stage and the RAM interpreter Int_{RAM}. The conducted specializations target the efficient structuring of loop-based join operations and the identification of optimal index support, in order to reduce the worst-case runtime complexity of the RAM program. However, index management is expensive and a minimal number of indices is desirable. In SOUFFLÉ we employ a novel, optimal and polynomial-time algorithm that is inspired by Dilworth's theorem [7] to compute only necessary indices. The idea of the algorithm is to compute partitions of chains in a lattice of indices. From each chain, a maximum index is computed that subsumes all other indices in the chain. This optimization results in a large run-time improvement in the resulting analyzer. After specializing the relational algebra program, C++ code that makes extensive use of templates is generated.

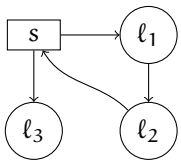
The final specialization step, P_{C++}=Mix($Int_{<>}$, $P_{C++_{<>}}$), is performed while compiling the generated C++ program. This Futamura projection is implemented using template-based meta-programming techniques [17]. With meta-programming techniques, data structures and algorithms are specialized by static information, thereby hoisting computations from run-time to compile-time. E.g., data structure interfaces are realized in form of C++ concepts rather than polymorphic C++ base classes to eliminate virtual-call dispatches and run-time type checks. The generated data structures are highly specialized towards the use of the corresponding relations in the input program. We employ efficient parallel variations of B-trees and Tries, with customized data element, node, and iterator types. Additionally, primary and secondary index support is provided for efficient operations on the represented relation in the program. For example, one of the most time-consuming operations with the use of indices is the comparison of two tuples. For this purpose we instantiate specialized versions of templatized lexicographical order functions in order to removing unnecessary control-flow and memory access overhead from the analysis run-time.

Our staged-translation approach using a specialization hierarchy coupled with standard Datalog optimizations and specialized relational data structures allows SOUFFLÉ to analyze very large code bases, previously considered to be impractical for Datalog-based engines. The generated C++ code is packaged in form of header files for a smooth integration with host applications.

2.2 An Example of the Specialization Process

Fig. 3 illustrates a simple security analysis for an example assuming that there is a low and high security state in a program. The invocation of a security sensitive method `vulnerable` is permitted only in the

```
void m(int i, int j){
 s: while (i < j){
l1:  protect();
l2:  i++;
    }
l3: vulnerable();
}
```




```
// Declarations
E(s:Node,d:Node) input
P(node:Node) input
I(node:Node) output
// Analysis
I("s").
I(y) :- I(x), E(x,y), !P(y).
```

Fig. 3. Java-like input program, a graphical representation of its control-flow, and security specification in SOUFFLÉ

high-security state. A call to the method `protect` transfers the security state from low to high if permitted. The example code of Fig. 3 would not violate the imposed security policy if it can be assumed that $i < j$ whenever `m` is invoked. However, since this can not be ensured, `m` exhibits a security violation which we would like to detect. The control-flow graph of `m` is shown next to the code fragment. It has the start nodes s, and nodes ℓ_1,ℓ_2, and ℓ_3 representing statements in the input program. An edge $(x, y) \in E$ between two nodes represents a potential transfer of control. A statement $x \in P$ raises the security level.

A simple analysis verifying the imposed security policy computes all statements that can be reached without passing the `protect` function. If a call to `vulnerable` is included in this set, the security policy is violated. Such a security analysis is be specified by the SOUFFLÉ code listed next to the control-flow graph in Fig. 3. The first section of the program declares relations used in the SOUFFLÉ program. The relation E is defined as a binary relation between two `Node` elements and the sets P and I contain elements of type `Node`. The qualifier `input` denotes that the relations are an extensional database and are provided as an input when executing the analysis. The set I contains all nodes in the control-flow that are not secure and is denoted as a result of the analysis using the qualifier `output`. In particular, if node ℓ_3 which is a `vulnerable` call is in set I, the method `m` does not fulfill the security policy to be enforced and would thus be identified as insecure.

The analysis always assumes the entry node s to be insecure by adding it to set I via `I("s").` The propagation rule

```
I(y) :- I(x), E(x,y), !P(y).
```

adds node y to the set of insecure nodes if (1) node x is insecure, (2) there is a control-flow from x to y, and (3) the target node y does not raise the security level.

SOUFFLÉ translates the given analysis specification in stages. The specialization hierarchy first fuses the semi-naïve evaluation with rules from the analysis as shown in Fig. 4: For the recursively defined set I code computing a fixed-point is generated. The set I is thereby supported by two auxiliary sets I' and ΔI. The set I' represents the newly gained knowledge within an iteration of the fix-point computation and set ΔI represents the newly gained knowledge of the previous iteration. The fix-point computation is performed in the while loop from line 2 to line 9 of the RAM program listed by Fig. 4(a). The first section of the loop body (lines 3 - 7) computes I' using ΔI as an input. The loop starting in line 4 iterates over all nodes in ΔI and the nested loop starting in line 5 iterates over all edges in the control flow graph. If any of those edges links some node x to a previously discovered insecure node y present in ΔI, where x is not a `protect` call itself and has not been marked as insecure before, it is add to the newly deduced set of insecure nodes I' (lines 6 and 7). In the last two statements of the loop body (i.e. lines 8 and 9) the newly gained knowledge of relation I' is added to relation I and I' becomes ΔI. The fixed-point calculation terminates if no new insecure nodes could be identified.

The pseudo-code of the Futamura projection is not optimal since it might have a worst-case complexity of $\mathcal{O}(n \cdot m)$ where n is the number of nodes in the control-flow graph and m is the number of edges

```
1  I = {s}; ΔI = I
2  while ΔI ≠ ∅
3   I' = ∅
4   for u ∈ ΔI do
5    for (x,y) ∈ E do
6     if ( u = x ∧ y ∉ P ∧ y ∉ I )
7      I' = I' ∪ {y}
8   I = I ∪ I'
9   ΔI = I'
```

(a) Futamura Projection

```
1  I = {s}; ΔI = I
2  while ΔI ≠ ∅
3   I' = ∅
4   for u ∈ ΔI do
5    for (_,y) ∈ E(u,_) do
6     if (y ∉ P ∧ y ∉ I )
7      I' = I' ∪ {y}
8   I = I ∪ I'
9   ΔI = I'
```

(b) Partial Evaluation of RAM Program

Fig. 4. Running Example

in the control-flow graph. To improve the performance of the program, we specialize the loop traversal of the loop in line 5 by employing an index. The index filters out all pairs in the edge relation whose source is not node u, i.e., all the edges are selected which emanate of node u denoted by the set $E(u,\_)$. This specialization requires an index on relation E, yet significantly reduces the runtime complexity. Typical analyses result in potentially hundreds of indices, making index management expensive if performed naively. We therefore employ an optimal, minimal index selection technique based on Dilworth's theorem [7] to select only necessary indices since an index may subsume several indices. Suppose we had another access to relation E on both u and v attributes, i.e., $E(u,v)$. A naive implementation would be to have two indices defined by the lexicographical orders u and $u < v$, however, the minimal solution would be to have only one index, namely, $u < v$ as it subsumes the index with only u. Some information to our solution to this combinatorial problem can be found in [13].

To implement indices from the previous step, we employ templatized B-Trees that require a comparison function for two tuples in the relation. The comparison function is implemented as a lexicographical order in the form of a template as sketched below,

```
template<int...> struct Comperator;
template<int i, int ... tail > struct Comperator<i,tail...> {
 static bool cmp(const tuple& a, const tuple& b){
  return a[i] < b[i] || (a[i] == b[i] && Comperator<tail...>::cmp(a,b));
 }
};
template<> struct Comperator<> {
  static bool cmp(const tuple&, const tuple&){ return true; }
};
```

The variadic template for the struct `Comperator` is parametrized by the columns in order. E.g., the call `Comperator<2,0>::cmp(a,b)` compares the tuples `a` and `b` by checking whether the third element of `a` is less than the third element of `b`. If the comparison results in a tie, the first elements of both tuples are compared to determine the order between the two tuples `a` and `b`. The operator is defined recursively: the base case is given by the `struct Comperator<>` considering every tuple equal, and the inductive case by `struct Comperator<i,tail...>`, comparing the `i`-th components and, if equal, delegating the comparison to `Comperator<tail...>`. The expansion of the template for a given instance such as `Comperator<2,0>` is performed at compile-time and delivers, in combination with function inlining, significant performance gains for index construction and retrieval. Without applying meta-programming techniques that rely on program specializations, i.e., pushing computations from runtime to compile-time, these performance gains would not be achievable.

Tool	CI		CS		Security	
	Δt [hh:mm::ss]	Memory [GB]	Δt [hh:mm::ss]	Memory [GB]	Δt [hh:mm::ss]	Memory [GB]
bddbddb	0:30:00	5.7	DNF	DNF	DNF	DNF
SQLite	6:20:00	40.2	DNF	DNF	DNF	DNF
μZ	DNF	DNF	DNF	DNF	DNF	DNF
SOUFFLÉ	0:00:35	8.5	6:44:08	206.4	14:45:01	75.3

Table 1. Comparison of Datalog evaluation tools for analyses on the OpenJDK7 b147 library, executing on an 8 core Intel Xeon E5-2690 v2 @ 3.0GHz server system. DNF = Did Not Finish within 18h.

3 Case Study: OpenJDK7

In this section we present our experience using SOUFFLÉ as a Java security analysis tool on the Java Development Kit (JDK). We point the reader to [5] for information on the Java vulnerabilities work at Oracle. For more detailed performance data on the techniques used in SOUFFLÉ , we refer the reader to [13].

In Table 1 we present three types of analyses performed on the OpenJDK7-b147. Due to the sheer size of OpenJDK7 (1.4M variables, 350K heap objects, 160K methods, 590K invocations and 17K types) such analyses are typically regarded as either impractical for most tools or at the very least, extremely challenging. The CI column refers to a *context-insensitive* points-to analysis and the CS refers to a *context-sensitive* points-to analysis. Points-to analysis is the main building block of most security analyses performed and are typically dominating the overall execution time. The last column, Security, refers to a large, composite security analysis similar to the *caller sensitive method* analysis in [5].

For our evaluation, we compare the performance of bddbddb [18], Z3's Datalog extension μZ [10], an SQLite based Datalog engine [14], and a 8-core parallel version of SOUFFLÉ . Each analysis has been ported to the respective Datalog variation of the evaluated tools. The resulting specifications typically comprise a few thousand lines of code. For the SOUFFLÉ based specifications, SOUFFLÉ 's module system has been utilized to facilitate the reuse of code among the three analysis, reducing the necessary development effort.

Our experiments reveal the limited capability of pre-existing Datalog-based tools when analyzing very large code bases. The CI analysis represents a very simple points-to analysis that does not construct the call-graph on the fly. A hand-crafted version of this analysis is reported to run under a minute [6]. For the CI analysis, bddbddb performs the analysis in a reasonable amount of time. However SOUFFLÉ outperforms bddbddb in terms of run-time by more than 50× consuming a comparable amount of memory. In the case of the CS and Security analyses, SOUFFLÉ is the only tool capable of performing the analyses within the 18h time limit imposed by the computation resources available to us for our evaluations. The Z3 based versions did not manage to finish any of our evaluated analyses in time.

4 Conclusion and Current Developments

We have presented SOUFFLÉ, a Datalog-based analysis tool that instead of evaluating Datalog, performs several specialization and optimization steps to produce a compiled, binary analyzer that can handle very large code bases. SOUFFLÉ is publicly available[4] and is actively developed by both Oracle and several universities. SOUFFLÉ supports a range of Datalog language extensions to aid in the specification of program analyses and resulting analyzers may be directly included into host applications as a header-only library.

[4] http://souffle-lang.github.io

Acknowledgement

We would like to thank Cristina Cifuentes, Paddy Krishnan, and all our other colleagues from Oracle Labs, Brisbane. We would also like to thank Byron Cook, Yannis Smaragdakis, and our anonymous reviewers.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Allen, N., Krishnan, P., Scholz, B.: Combining type-analysis with points-to analysis for analyzing java library source-code. In: Møller, A., Naik, M. (eds.) Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015. pp. 13–18. ACM (2015)
3. Allen, N., Scholz, B., Krishnan, P.: Staged points-to analysis for large code bases. In: Franke, B. (ed.) Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9031, pp. 131–150. Springer (2015)
4. Alpuente, M., Feliú, M.A., Joubert, C., Villanueva, A.: Datalog-based program analysis with BES and RWL. In: Proceedings of the First International Conference on Datalog Reloaded. pp. 1–20. Springer-Verlag (2011)
5. Cifuentes, C., Gross, A., Keynes, N.: Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In: Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis. pp. 7–12. SOAP 2015, ACM, New York, NY, USA (2015)
6. Dietrich, J., Hollingum, N., Scholz, B.: Giga-scale exhaustive points-to analysis for java in under a minute. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 535–551. OOPSLA 2015, ACM, New York, NY, USA (2015)
7. Dilworth, R.: A decomposition theorem for partially ordered sets. Ann. Math. (2) 51, 161–166 (1950)
8. Futamura, Y.: Partial evaluation of computation process – an approach to a compiler-compiler. Higher Order Symbol. Comput. 12(4), 381–391 (Dec 1999)
9. Green, T.J., Huang, S.S., Loo, B.T., Zhou, W.: Datalog and recursive query processing. Foundations and Trends in Databases 5(2), 105–195 (2013)
10. Hoder, K., Bjørner, N., de Moura, L.M.: μZ – An efficient engine for fixed points with constraints. In: Proceedings of the International Conference on Computer Aided Verification. vol. LNCS 6806, pp. 457–462. Springer (2011)
11. LogicBlox, I.: Declartive cloud platform for applications that combine transactions & analytics. `http://www.logicblox.com`
12. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. SIGPLAN Not. 41(6), 308–319 (Jun 2006)
13. Scholz, B., Jordan, H., Subotic, P., Westmann, T.: On fast large-scale program analysis in datalog. In: Zaks, A., Hermenegildo, M.V. (eds.) Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016. pp. 196–206. ACM (2016)
14. Scholz, B., Vorobyov, K., Krishnan, P., Westmann, T.: A datalog source-to-source translator for static program analysis: An experience report. In: 24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015. pp. 28–37. IEEE Computer Society (2015)
15. Smaragdaiks, Y., Bravenboer, M., Kastrinis, G.: Doop: A framework for java pointer analysis. `http://doop.program-analysis.org/`
16. Smaragdakis, Y., Kastrinis, G., Balatsouras, G.: Introspective analysis: Context-sensitivity, across the board. In: PLDI. pp. 485–495. ACM, New York, NY, USA (2014)
17. Veldhuizen, T.L.: C++ templates as partial evaluation. In: Danvy, O. (ed.) PEPM. pp. 13–18. University of Aarhus (1999), `http://dblp.uni-trier.de/db/conf/pepm/pepm1999.html#Veldhuizen99`
18. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with binary decision diagrams for program analysis. In: APLAS. pp. 97–118 (2005)