PhASAR: An Inter-Procedural Static Analysis Framework for C/C++

Philipp D. Schubert[1][0000−0002−8674−1859], Ben Hermann[1][0000−0001−9848−2017], and Eric Bodden[1,2][0000−0003−3470−3647]

[1] Heinz Nixdorf Institut, Paderborn University, Paderborn 33102, Germany
[2] Fraunhofer IEM, Paderborn 33102, Germany
{philipp.schubert, ben.hermann, eric.bodden}@upb.de

Artifact * AEC * Evaluated * TACAS * Consistent * Complete * Well Documented * Easy to Reuse *

Abstract. Static program analysis is used to automatically determine program properties, or to detect bugs or security vulnerabilities in programs. It can be used as a stand-alone tool or to aid compiler optimization as an intermediary step. Developing precise, inter-procedural static analyses, however, is a challenging task, due to the algorithmic complexity, implementation effort, and the threat of state explosion which leads to unsatisfactory performance. Software written in C and C++ is notoriously hard to analyze because of the deliberately unsafe type system, unrestricted use of pointers, and (for C++) virtual dispatch. In this work, we describe the design and implementation of the LLVM-based static analysis framework PhASAR for C/C++ code. PhASAR allows data-flow problems to be solved in a fully automated manner. It provides class hierarchy, call-graph, points-to, and data-flow information, hence requiring analysis developers only to specify a definition of the data-flow problem. PhASAR thus hides the complexity of static analysis behind a high-level API, making static program analysis more accessible and easy to use. PhASAR is available as an open-source project. We evaluate PhASAR's scalability during whole-program analysis. Analyzing 12 real-world programs using a taint analysis written in PhASAR, we found PhASAR's abstractions and their implementations to provide a whole-program analysis that scales well to real-world programs. Furthermore, we peek into the details of analysis runs, discuss our experience in developing static analyses for C/C++, and present possible future improvements.

Keywords: Inter-procedural static analysis, LLVM, C/C++

1 Introduction

Programming languages from the C/C++ family are chosen as the implementation language in a multitude of projects especially in cases where a direct interface with the operating system or hardware components is of importance. Large portions of any operating system and virtual machine (such as the Java VM) are written in C or C++. The reason for this is oftentimes the amount of control the programmer has over many aspects that allow for the creation of very efficient programs—but also comes with the obligation to use these features correctly to avoid introducing bugs or opening the program to security vulnerabilities.

To aid developers in creating correct and secure software, a multitude of checks have been included into compilers such as GCC [4] and Clang [2]. Various additional tools such as Cppcheck [12], clang-tidy [9], or the Clang Static Analyzer [8] provide additional means to check for unwanted behavior. Compiler-check passes and additional checkers both use static program analysis to provide warnings to their users. However, to create warnings in a timely fashion, these tools use comparatively simple analyses that provide either only checks for simple properties, or suffer from a large number of false or missed warnings, due to the imprecision or unsoundness of the used analysis.

For programs written in Java, program-analysis frameworks like Soot [16], WALA [33], and Doop [13] are available which allow for a more precise data-flow analysis to determine more intricate program problems. Furthermore, algorithmic frameworks such as *Interprocedural Finite Subset (IFDS)* [24], *Interprocedural Distributive Environments (IDE)* [26], or *Weighted Pushdown Systems (WPDS)* [25] can be used to describe dataflow problems and efficiently compute their possible solutions.

So far, such implementations have not been openly available for programs written in C/C++. This work thus presents the novel program-analysis framework PhASAR, an extension to the LLVM compiler infrastructure [17]. In its inception, we used our experience in developing previous such frameworks for JVM-based languages, namely Soot [16] and OPAL [14], to design a flexible framework that can be adapted to several different types of client analyses. Besides solving data-flow problems, PhASAR can be used to achieve other related goals as well, for instance, call-graph construction, or the computation of points-to information. Its features can be used independently and be included into other software. PhASAR's implementation is written entirely in C++ and is available as open source under the permissive MIT license [23].

PhASAR is intended to be used as a static analyzer. Therefore, it does not substitute but complement features from the LLVM toolchain and provides also for analyses which during compilation would be prohibitively expensive.

This paper makes the following contributions:

- It provides a user-centric description of PhASAR's architecture, its infrastructure, and data-flow solvers,
- it presents a case-study that shows PhASAR's overall scalability as well as the precise runtimes of a concrete static analysis, and
- it discusses our experience in developing static analyses for C/C++.

2 Related Work

There are several established and well-maintained tools and frameworks for the Java ecosystems. Frameworks from academia include Soot [16], which is a static analysis framework that allows call-graph construction, computation of points-to information and solving of data-flow problems for Java and Android. Soot does not support inter-procedural data-flow analyses directly. However, a user can solve such problems using the Heros [7] extension that implements an IFDS/IDE solver. The WALA [33] framework provides similar functionalities for Java bytecode, JavaScript and Python. OPAL [14] allows for the implementation of abstract interpretations of Java bytecode. Also the manipulation of bytecode is supported. A declarative approach is implemented by the Doop framework [13]. Doop uses a declarative rule set to encode an analysis and solves it using the logic-based Datalog solver. The framework allows for pointer analysis of Java programs and implements a range of algorithms that can be used for context insensitive, call-site and object sensitive analyses.

Tooling for C/C++ includes Cppcheck [12] which aims for a result without false positives and allows to encode simple rules as well as the development of more powerful add-ons. The clang-tidy tool [9] provides built-in checks for style validation, detection of interface misuse as well as bug-finding using simple rules, but can be extended by a user. Checks can be written on preprocessor level using callbacks or on AST level using AST matchers that can be specified using an embedded domain specific language (EDSL). The Clang Static Analyzer [8] uses symbolic execution and allows custom checks to be written. The SVF [31] framework computes points-to information for constructing sparse value flow and memory static single assignment (SSA). Hence, it can be used for analyses that rely on those information such as memory leak detection or null pointer analysis. Additionally, more precise pointer analysis can be build on top of SVF's results. However, as the computation of memory SSA does require a significant amount of computation, using SVF may not pay off for problems that can be encoded using distributive frameworks, which allow fast, summary-based solutions.

There are also commercial, closed-source tools for static analysis such as CodeSonar [10] and Coverity [11], both of which support analyses for C, C++, Java and other languages. Whereas these products are attractive to industry as they provide polished user interfaces, they are not usable for evaluating novel algorithms and ideas in static-analysis research.

3 Data-flow Analysis

Data-flow analysis is a form of static analysis which works by propagating information about the property of interest—the data-flow facts—through a model of the program, typically a control-flow graph, and captures the interactions of the flow facts with the program. The interaction of a single statement s with a data-flow fact is described by a flow function. There are two orthogonal approaches [27] that can be used in order to solve inter-procedural (whole program) data-flow problems: the call-strings and functional approach. For the call-strings approach we refer the reader to related work [15,27]. In the following we briefly present the functional approach using a linear constant propagation that we apply to a small program shown in Listing 1.1. A linear constant propagation is a data-flow analysis that precisely tracks variables with constant values and variables that linearly ($c = a \cdot x + b$, with a, b constant values) depend on constant values through the program. Non-linear dependencies are over-approximated. In our example, we restrict the analysis to keep track of integer constants only. Such an analysis can be used to perform program optimizations by replacing variables with their constant values, and folding expressions that use constant values, eventually possibly also removing dead code. The analysis would be able to optimize the program shown in Listing 1.1 to `int main() { return 12; }`.

```
int inc(int p) { return ++p; }
int main() {
  int a = 1;
  int b = 2;
  int c = 3;
  a = inc(a); // cs1
  b = inc(b); // cs2
  c = b * 4;
  return c;
}
```

Listing 1.1. Program `P`

If the flow functions of the problem to be solved are monotone and distributive over the merge operator, it can be encoded using *Inter-procedural Finite Distributive Subset* (IFDS) or *Inter-procedural Distributive Environments* (IDE). Unlike the call-string approach which is limited to a certain level of context-sensitivity (commonly denoted as k), IFDS and IDE are fully context-sensitive, i.e., $k = \infty$. In IFDS [24] and its generalization IDE [26], a data-flow problem is transformed into a graph reachability problem. The reachability is computed using the so called exploded super-graph (ESG). If a node (s_i, d_i) in the ESG is reachable from a special tautological node Λ, the data-flow fact represented by d_i holds at statement s_i. The ESG is built according to the flow functions which can be represented as bipartite graphs. Functions for generating (`Gen`) and destoying (`Kill`) data-flow facts can be encoded into flow functions making the framework compatible to more traditional approaches to data-flow analysis. The composition $f \circ g$ of two functions can be computed by composing their corresponding bipartite graphs, i.e., merging the nodes of g with the corresponding nodes of the domain of f. The ESG for the complete program is constructed by replacing every node of the inter-procedural control-flow graph (ICFG) with the graph representation of the corresponding flow function. Scalability issues due to context-sensitivity are mitigated through summaries that are computed by composition of all bipartite graphs of a function for a given input. These summaries are reused for subsequent calls to an already summarized function.

The complexity of the IFDS algorithm is $\mathcal{O}(|N| \cdot |D|^3)$ where $|N|$ is the number of nodes on the ICFG (or number of program statements) and $|D|$ the size of the data-flow domain that is used. To make the analysis scale, the domain D should thus be kept small.

In IDE, a generalization of the IFDS framework, the edges of the ESG are additionally annotated with so-called *edge functions*. With the help of those edge functions, an additional value-computation problem can be encoded, which is solved while performing the reachability computation. The complexity of the IDE algorithm is the same as for the IFDS algorithm. Many problems can be solved more efficiently by encoding them with IDE rather than IFDS, because IDE uses two domains to solve a given problem. In addition to the domain D of the data-flow facts, the value computation problem is formulated over a second value domain V, which can be large, even infinite. Crucially, for a given fixed-size program, the complexity of both IFDS and IDE depends only on the size of D.

Let us consider a linear constant propagation to be performed on the example program shown in Listing 1.1. Using IFDS, the data-flow domain can be encoded by using pairs of $\mathcal{V} \times \mathbb{Z}$ program variables and integer values. However, this strategy leads to a huge domain D and prevents the generation of effective summaries. For each call to `inc()` in Listing 1.1 with a different input value `a`, a new summary must be generated. In the example, we would obtain summaries $\{(\texttt{p}, 1) \mapsto (\texttt{<ret>}, 2)\}$ for call site *cs1* and $\{(\texttt{p}, 2) \mapsto (\texttt{<ret>}, 3)\}$ for *cs2*.

With IDE, the problem can be encoded in a more elegant and efficient way, by using $\mathcal{V}$ as the data-flow domain and $\mathbb{Z}$ as the value domain. The ESG for a linear constant propagation performed on Listing 1.1 using IDE is shown in Figure 1. As the context-dependent part of the analysis is encoded using the edge functions, only one summary is generated for the `inc()` function, $\lambda x.\, x+1$.

Performing a reachability check on the ESG for variable `c` at line 9, one finds that `c` can be replaced by the literal `12`. Because the return statement is the program's only observable effect, all other statements can be safely removed.

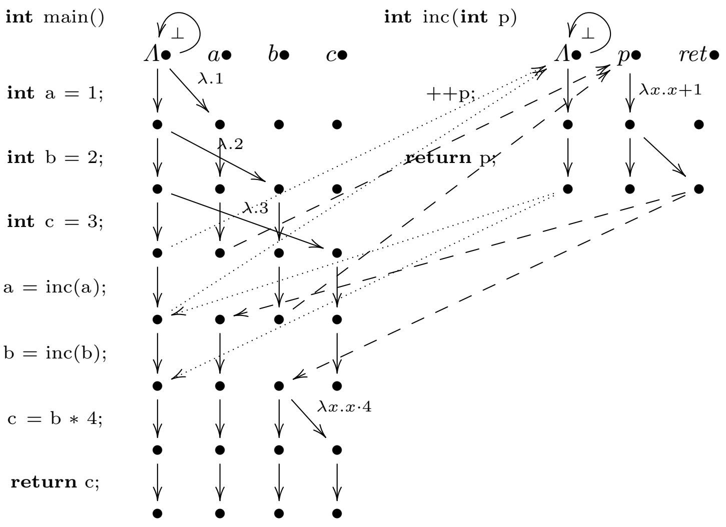

Fig. 1. Exploded super graph for the program `P` in Listing 1.1

4 Architecture

Precise data-flow analysis requires information from multiple supporting analyses which are typically run earlier, such as class-hierarchy, call-graph, and points-to analysis. Algorithmic frameworks like IFDS provide a generalized algorithm that is then parameterized for each individual data-flow problem. The infrastructure provided by these basic analyses and algorithmic frameworks is necessary to allow analysis designers to efficiently concentrate on the goal of a data-flow analysis. PhASAR is the first framework to provide such infrastructure for programs written in the C/C++ language family. Its infrastructure is designed modularly, such that analysis developers can choose the components necessary for their individual goals. In Figure 2 we present the high-level architecture of the framework.

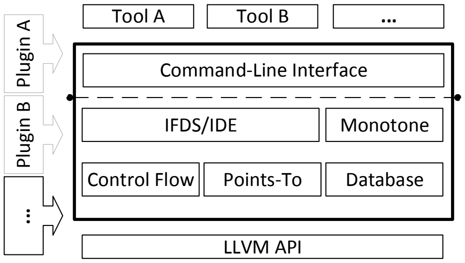

Fig. 2. PhASAR's high-level architecture

We allow PhASAR to be used in multiple ways. The first (and easiest) way is through its command-line interface. Its implementation can be seen as a blueprint to create other tools which use PhASAR. The command-line interface provides a means to execute basic analyses such as call-graph construction or pointer analysis or run pre-defined IFDS/IDE-based analyses. The output of these analyses can then be processed using other tooling or presented to the user directly.

The command-line interface can also be extended with custom analyses, provided as separately compiled plugins. Currently, custom control-flow or call-graph analyses and custom data-flow analyses can be packaged in this way. The command-line interface acts as the runtime for these plugins and delegates control to the plugin at the appropriate times providing necessary information. Plugin providers need to create an implementation of a pre-defined C++ class wrapping their analysis code. The plugin is compiled separately and then provided to PhASAR in form of a shared object library.

PhASAR can also be included into other tools by using it as a library. This way of using PhASAR provides the most flexibility as developers can freely select the components that should be part of an analysis and can reuse even parts of the components provided by the framework.

PhASAR allows analysis developers to specify arbitrary data-flow problems, which are then solved in a fully-automated manner on the specified LLVM IR target code. Solving a static analysis problem on the IR rather than the source language makes the analysis generally easier. This is because it removes the dependency on the concrete source language, as the IR is usually simpler since the IR involves no nesting and has fewer instructions. Various compiler front-ends for a wide range of languages targeting LLVM IR exist. Hence, PhASAR is able to analyze programs written in languages other than C/C++, too. The framework computes all required information to perform an analysis such as points-to, call-graph, type-hierarchy as well as additional parameterizable taint and typestate analyses.

PhASAR provides various capabilities and interfaces to compute data-flow problems or aid other types of analyses. First, the framework contains interfaces and implementations for the computation of an ICFG; we provide some parameterizable implementations for the LLVM IR.

Next, PhASAR currently supports the computation of function-wise points-to information using LLVM's implementations of the *Andersen*-style [6] or *Steensgard*-style [30] algorithms. Points-to information and ICFG computation can be combined to obtain more precise results. We discuss the quality of points-to information and our current efforts to improve their quality in section 8.

To resolve virtual function calls in C++, we provide means to construct a type hierarchy. We construct the type hierarchy for composite types and reconstruct the virtual-method tables from the IR, which together with the hierarchy information allow PhASAR to resolve potential call targets at a given call-site.

PhASAR provides implementations of IDE and IFDS solvers as described by Reps et al. [24] including the extensions of Naeem et al. [20]. We implemented IFDS as a specialization of IDE using a binary lattice only using a top and a bottom element much alike the Heros implementation [7]. Both solvers are accompanied by a corresponding interface for problem definition. To solve a data-flow problem using the IDE or IFDS solver, the data-flow problem must be encoded by implementing this interface. We present this in detail in section 5.

For non-distributive data-flow problems PhASAR provides an implementation of the traditional monotone framework which allows one to solve intra-procedural problems. The framework provides an inter-procedural version as well that uses a user-specified context in order to differentiate calling-contexts. PhASAR provides a context interface and implementations of this interface that realize the call-strings and value-based approach VASCO [22], in which context-sensitivity is achieved by reusing information that has been computed for previous calls under the same context. The framework also implements a version of the context class to represent a *null context*. This context has the same effect as applying the monotone framework directly in an inter-procedural setting. Both solvers are accompanied by corresponding interfaces for problem descriptions which must be implemented to encode the data-flow problem. The details are provided in section 5.

All of PhASAR's data-flow solvers are implemented in a fully generic manner and heavily make use of templates and interfaces. For instance, a solver follows a target program's control-flow that is specified through an implementation of either the `CFG` or the `ICFG` interface. Analysis developers can parameterize a solver with an existing implementation or they can provide their own custom implementation. They can run a forward or backward analysis depending on the direction of the chosen control-flow graph. Moreover, all data-flow related functionality is hidden behind interfaces. A solver queries the required functionality such as flow functions or merge operations for the underlying lattice whenever necessary. We have specified problem interfaces on which the corresponding solver operates. Thus, analysis developers encode their data-flow problem by providing an implementation for the problem interface and provide this implementation to the accompanying solver. PhASAR is able to solve a problem on other IRs when suitable implementations for the IR specific parts such as the control-flow graphs and problem descriptions are provided by the analysis developer.

5 Implementation

Our goal with PhASAR is easing the formulation of a data-flow analysis such that an analysis developer only needs to focus on the implementation of the problem description rather than providing details how the problem is solved.

PhASAR achieves parts of its generalizability through template parameters. These template parameters include, among others, `N`, `D`, `M`. They are consistently used throughout the implementation of PhASAR. `N` denotes the type of a node in the ICFG, i.e., typically an IR statement, `D` denotes the domain of the data-flow facts, and `M` is a placeholder for the type of a method/function. When analyzing LLVM IR, `N` is always of type `const llvm::Instruction*` and `M` is of type `const llvm::Function*`, whereas `D` depends on the specific data-flow analysis that the developer wants to encode. For our example using linear constant propagation described in section 3, `D = pair<const llvm::Value *, int>` could be used to capture the property of interest. LLVM's `Value` type is quite useful as it is a super-type that is located high in the type hierarchy. This allows an analysis developer to use values of all of `Value`'s subtypes in the value domain, which makes it highly flexible.

5.1 Encoding an IFDS Analysis

Listing 1.2 shows the interface for an IFDS problem. An analysis developer has to define a new type — the problem description — implementing the `FlowFunctions` interface.

```
template <typename N, typename D, typename M> struct FlowFunctions {
  virtual ~FlowFunctions() = default;
  virtual FlowFunction<D> *getNormalFlowFunction(N curr, N succ) = 0;
  virtual FlowFunction<D> *getCallFlowFunction(N callStmt,
                                                M destMthd) = 0;
  virtual FlowFunction<D> *getRetFlowFunction(N callSite,
                                               M calleeMthd,
                                               N exitStmt,
                                               N retSite) = 0;
  virtual FlowFunction<D> *
  getCallToRetFlowFunction(N callSite, N retSite, set<M> callees) = 0;
};
```

Listing 1.2. Interface for specifying flow functions in IFDS/IDE

The flow function factories shown in Listing 1.2 handle the different types of flows. The four factory functions each have an individual purpose:

- `getNormalFlowFunction` handles all intra-procedural flows.

- `getCallFlowFunction` handles inter-procedural flows at a call-site. Usually, the task of this flow function factory is to map the data-flow facts that hold at a given call-site into the callee method's scope.
- `getRetFlowFunction` handles inter-procedural flows at an exit statement (e.g. a return statement). This maps the callee's return value, as well as data-flow facts that may leave the function by reference or pointer parameters, back into the caller's context/scope.
- `getCallToRetFlowFunction` propagates all data-flow facts that are not involved in a call along-side the call-site, typically stack-local data not referenced by parameters.

These flow function factories are automatically queried by the solver, based on the inter-procedural control-flow graph.

The functions in Listing 1.2 are factories since they have to return small function objects of type `FlowFunction` which is shown in Listing 1.3. As a `FlowFunction` is itself an interface, an analysis developer has to provide a suitable implementation. The member function `computeTargets()` takes a value of a dataflow fact of type `D` and computes a set of new dataflow facts of the same type. It specificies how the bipartite graph for the statement that represents the flow function is constructed and can be thought of an answer to the question "What edges must be drawn?".

```
template <typename D> struct FlowFunction {
  virtual ~FlowFunction() = default;
  virtual set<D> computeTargets(D source) = 0;
};
```

Listing 1.3. Interface for a flow function in IFDS/IDE

As flow function implementations often follow certain patterns, we provide implementations for the most common patterns as template classes. Many useful flow functions like `Gen`, `GenIf`, `Kill`, `KillAll`, and `Identity` are already implemented and can be directly used. Any number of flow functions can be easily combined using our implementations of the `Compose` and `Union` flow functions. We also provide `MapFactsToCallee` and `MapFactsToCaller` flow functions that automatically map parameters into a callee and back to a caller, since this behavior is frequently desired. Flow functions which are stateless, e.g. `Identity` or `KillAll`, are implemented as a singleton.

5.2 Encoding an IDE Analysis

If an analysis developer wishes to encode their problem within IDE, they have to additionally provide implementations for the edge functions. With help of the edge functions, an analysis developer is able to specify a computation which is performed along the edges of the exploded super-graph leading to the queried node (c.f. Figure 1). The interface for the edge function factories and their responsibilities are analogous to the flow function factories in Listing 1.2.

Each edge function factory must return an edge function implementation: a small function object similar to a flow function which has a `computeTarget()`

function, a compose, a merge, and an equality-check operation. The `EdgeFunction` interface is shown in Listing 1.4.

```
template <typename V> class EdgeFunction {
public:
  virtual ~EdgeFunction() = default;
  virtual V computeTarget(V source) = 0;
  virtual EdgeFunction<V> *
  composeWith(EdgeFunction<V> *secondFunction) = 0;
  virtual EdgeFunction<V> *
  joinWith(EdgeFunction<V> *otherFunction) = 0;
  virtual bool equal_to(EdgeFunction<V> *other) const = 0;
};
```

Listing 1.4. Interface for an edge function in IDE

As this interface is more complex than the flow function interface, we explain the purpose of each function. The `computeTarget()` function describes a computation over the value domain `V` in terms of lambda calculus.

The `composeWith()` function encodes how to compose two edge functions. In most scenarios, this function can be implemented as $(f \circ g)(x) = f(g(x))$. To avoid additional boilerplate code, we provide an `EdgeFunctionComposer` class that performs this job and can be used as a super class.

`joinWith()` encodes how to join two edge functions at statements where two control-flow edges lead to the same successor statement. Depending if a may or a must-analysis is performed, implementations of this function typically check which edge function computes a value that is higher up in the lattice, i.e., a more approximate value, and returns the corresponding edge function. For our linear constant propagation from section 3, this function would return one of the edge functions if both describe the same value computation, the bottom edge function if both of them encode the $\perp$ value and the edge function encoding the top element otherwise. The intuition here is to always pick the element that is higher in the lattice as it represents more information.

The `equal_to()` interface function has to be implemented to return true if both edge functions describe the same value computation, false otherwise.

A complete implementation of the IDE linear constant propagation can be found along with PhASAR's other examples at our website [23].

5.3 Encoding a Monotone Analysis

If an analysis developer wishes to encode a problem that does not satisfy the distributivity property, they have to make use of the monotone-framework implementation or its inter-procedural variant. The interface for specifying an inter-procedural monotone problem is shown in Listing 1.5. Similar to an IFDS/IDE problem, an analysis developer has to specify flow functions for intra- and inter-procedural flows. But in contrast to IFDS/IDE, these flow functions do not operate on single, distributive data-flow facts, but on sets of data-flow facts instead. The solver calls the flow funtions and provides the set of data-flow facts which hold right before the current statement. The return value to be computed in the flow function is a set of data-flow facts that hold after the effects of the current statement. The `join()` function specifies how information is merged when two branches join at a common successor statement. This is typically implemented as set-union or set-intersection depending on whether a may or must-analysis has to be solved. Algorithms from C++'s STL may be used here. Finally, the `sqSubSetEqual()` function must be implemented to determine if the amount of information between two sets has increased in order to check if a fixpoint is reached. The context that is used for the inter-procedural analysis can be specified by the analysis developer using the template parameter. An analysis developer can provide a pre-defined context class in order to parameterize the analysis to be a call-strings approach, a value-based approach, or they can define their own context to be used.

```
template <typename N, typename D, typename M, typename I>
struct InterMonotoneProblem {
  InterMonotoneProblem(I Icfg) : ICFG(Icfg) {}
  virtual ~InterMonotoneProblem() = default;
  virtual set<D> join(const set<D> &Lhs, const set<D> &Rhs) = 0;
  virtual bool sqSubSetEqual(const set<D> &Lhs,
                             const set<D> &Rhs) = 0;
  virtual set<D> normalFlow(N Stmt, const set<D> &In) = 0;
  virtual set<D> callFlow(N CallSite, M Callee, const set<D> &In) = 0;
  virtual set<D> returnFlow(N CallSite, M Callee, N RetStmt,
                            N RetSite, const set<D> &In) = 0;
  virtual set<D> callToRetFlow(N CallSite, N RetSite,
                               const set<D> &In) = 0;
};
```

Listing 1.5. Interface for describing an interprocedural problem for the monotone framework

5.4 Handling of intrinsic and libc Function Calls

LLVM currently has approximately 130 intrinsic functions. These functions are used to describe semantics in the analysis and optimization phase and do not have an actual implementation. Later-on in the compiler pipeline, the back-end is free to replace a call to an intrinsic function with a software or a hardware implementation – if one exists for the target architecture. Introducing new intrinsic functions is preferred over introducing novel instructions to LLVM since, when introducing a new instruction, all optimizations, analyses, and tools built on top of LLVM have to be revisited to make them aware of the new instruction. A call to an intrinsic function can be handled as an ordinary function call.

The functions contained in the libc standard library represent special targets as well as these functions are used by virtually all practical C and C++[3] programs. Moreover, the functions contained in the standard library cannot be analyzed themselves as they are mostly very thin wrappers around system calls and are often not available for the analysis. In many cases, however, it is not necessary to analyze these functions when performing a data-flow analysis. PhASAR models all of them as the identity function. An analysis developer can change the default behavior and model different effects by using special summary functions. The `SpecialSummaries` class can be used to register flow and edge functions other than identity. This class is aware of all intrinsic and libc functions.

[3] The compiler translates many of C++'s features into ordinary calls to libc.

5.5 A Note on Soundness

Livshits et al. have introduced the notion *soundy* analyses [18]. Soundy analyses use sensible underapproximations to cope with certain language features that would otherwise make an analysis impractically imprecise. Analyses in PhASAR are currently *soundy.* For instance, PhASAR's ICFG misses one control-flow edge in the presence of `setjmp()/longjmp()`. Functions that are loaded dynamically from shared object libraries using `dlsym()` cannot be handled either. PhASAR's data-flow solvers treat calls to dynamically loaded libraries and libraries for which function definitions are missing as identity, unless the analysis developer specifies otherwise. A sound handling would be to set all variables involved in such calls to $\top$, which again, may lead to large imprecision.

6 Scalability

In this section, we present the runtime measurements for two concrete static analyses – `IFDSSolverTest` we name $\mathbb{I}$ and `IFDSTaintAnalysis` we name $\mathbb{T}$ – that are both implemented in PhASAR. $\mathbb{I}$ is a trivial IFDS analysis which passes the tautological data-flow fact Λ through the program. The analysis acts as a baseline as it is the most efficient IFDS/IDE analysis that can possibly be implemented. $\mathbb{T}$ implements a taint analysis. A taint analysis tracks values that have been tainted by one or more sources through the program and reports whenever one of the tainted values reaches a sink, which can be functions or instructions. Our taint analysis treats the command-line parameters `argc` and `argv` that are passed into the `main()` function as tainted. Functions that read values from the outside (e.g. `fread()`) are interpreted as sources. Functions that can leak tainted variables to the outside such as `printf()` or `fwrite()` are considered sinks. As a potentially large amount of tainted values have to be tracked through the program, analysis $\mathbb{T}$ will provide insights into the scalability of PhASAR's IFDS/IDE solver implementation.

Table 1 shows the programs that we analyzed. For each program, the IR's lines of code, number of statements, pointers, and allocation sites have been measured with PhASAR. The LLVM IR has been compiled with the Clang compiler using production flags. The figures give an intuition for the program's complexity. The programs that we analyzed comprise some C programs like some of the coreutils [3] as well as two C++ programs like PhASAR itself and a PhASAR-based tool MPT. In addition, it shows the runtimes of the analyses $\mathbb{I}$ and $\mathbb{T}$ separated into different phases (in the format runtime $\mathbb{I}$/runtime $\mathbb{T}$). We measured the runtimes for the construction of points-to information (PT), class hierarchy (CH), call-graph (CG), data-flow information (DF), and the total runtime (Σ). We also measured the number of function summaries $\psi(f)$ that could be reused while solving the analysis. The latter one is a good indicator for the quality of the data-flow domain D, as higher reuse indicates a more efficient analysis. #G and #K denote the number of facts that have been generated or killed in the taint analysis, respectively.

We measured the runtimes by performing 15 runs for each analysis on a virtual machine running on an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz machine with 128GB memory. We removed the minimum and maximum values and computed the average of the remaining 13 values for each of the four analysis steps and the total runtime. We used an on-the-fly call-graph algorithm that uses points-to information for the coreutils. For PhASAR and MPT, we used a declared type-analysis (DTA) call-graph algorithm in order to reduce the amount of memory required to reproduce our results. In addition, we found that DTA performed well enough on our C++ target programs.

With one exception, PhASAR is able to analyze a program from coreutils within a few seconds. Analyzing cp using $\mathbb{T}$ takes around 13 minutes. This is because a large amount of facts is generated which must then be propagated by the solver. This result shows the cubic impact of the number of data-flow facts on IFDS/IDE's complexity. Analyzing the million-line programs PhASAR and MPT ranges from 7 to 18 minutes. As one can observe for PhASAR, an analysis may destroy data-flow facts more often than it generates them. This is caused by C++'s exceptional control-flow where the same fact is destroyed during normal and exceptional flow.

We observed that the DF part of $\mathbb{T}$ actually runs faster than $\mathbb{I}$ for our C++ target programs. This is because $\mathbb{T}$ should behave very similar to the solvertest for the C++ target programs, as only very few facts are actually generated. Furthermore, $\mathbb{T}$ will take shortcuts whenever it plugs in the desired effects at call-sites of source and sink functions. $\mathbb{I}$ in contrast, follows these calls making it slower than $\mathbb{T}$.

Analyzing all of the 97 coreutils, PhASAR, and MPT requires a total analysis time of of 30 minutes for $\mathbb{I}$ and 1 hour and 31 minutes for $\mathbb{T}$. These measurements show that PhASAR is capable of analyzing even a million-line program within minutes, even though PhASAR's algorithms and data structures have not yet undergone manual optimization.

Table 1. Program's characteristics and performance figures for analyses $\mathbb{I}/\mathbb{T}$

Program	kLOC	Stmts	Ptrs	Allocs	CH [ms]	PT [s]	CG [s]	DF [s]	Σ [s]	$\#\psi(f)$	#G	#K
wc	132	63166	10644	396	24/24	1.0/1.0	0.1/0.1	0.2/11	2/13	119/125	10202	6830
ls	152	71712	13200	438	27/27	1.4/1.4	1.1/1.2	0.6/1.0	4/5	836/839	79	74
cat	130	62588	10584	391	24/24	1.0/1.0	0.0/0.0	0.1/1.3	2/3	21/22	2525	1262
cp	141	67097	11722	443	32/30	1.3/1.3	0.6/0.6	0.4/789	3/792	547/737	16999	12839
whoami	129	61860	10433	389	24/23	1.0/1.0	0.0/0.0	0.1/0.3	2/2	8/11	97	92
dd	137	65287	11150	408	25/25	1.1/1.0	0.2/0.2	0.2/37	2/40	164/176	14711	11058
fold	130	62201	10509	390	24/23	1.0/1.0	0.0/0.0	0.1/0.3	2/2	17/22	107	102
join	134	64196	11042	402	24/24	1.0/1.0	0.0/0.0	0.1/0.5	2/3	91/95	104	94
kill	130	62304	10527	394	24/24	1.0/1.0	0.0/0.0	0.1/0.1	2/2	24/24	22	4
uniq	131	62663	10650	396	24/24	1.0/1.0	0.0/0.0	0.1/0.4	2/2	50/53	96	90
MPT	3514	1351735	755567	176540	906/903	22/22	8.8/8.8	458/379	519/439	12531/12532	20	9
PhASAR	3554	1368297	763796	178486	962/946	23/23	24/24	987/917	1064/993	25778/25782	56	77

7 Guidelines for the Analysis on Real-World Code

In this section, we share our experience in analyzing real-world C/C++ programs. Although the LLVM IR is expressive enough to capture arbitrary source languages, we found that the characteristics and complexity of the source language propagate into the IR. Observe the following call-site in LLVM IR: %retval = call i32 %fptr(%class.S* dereferenceable(4) %ptr, i32 5), assuming C to be the source language, a plain function pointer is called. If C++ is the source language, we cannot be sure whether a function pointer or a virtual member function of class `S` is called. This is the reason why we observed that the analysis runtime for C++ target programs is usually much higher than for C programs.

For more complex languages like C++ we have to keep track of special member functions. These functions are mapped into ordinary LLVM IR functions that Clang places in a well-defined order in the generated IR. For some analyses like the declared-type analysis (DTA) call-graph algorithm, we need to be aware of these special member functions in order to preserve high precision.

We also found that even a well-debugged analysis that has been hardened on a large variety of test programs may still fail on production code as some corner cases have not been thought of. The large amount of information available to an analysis run makes debugging errors hard. A standard debugger does not suffice because an analysis writer has to step through a lot of code that is not relevant for them. For Java, a special dedicated debugger for static analysis has been developed [21] which shows the relevance of the problem.

Depending on the optimization passes that have been applied to code in LLVM IR before it is handed over to the analysis, it may have very different characteristics. Although optimization passes are required to have no impact on the semantics, the structure of the IR code changes. In our experience, it is helpful to start developing an analysis on small test programs that are translated into IR without optimization passes, and cover as many cases as the analysis should find. Once an analysis handles these test cases correctly or with the desired precision, optimization passes should be applied to the test cases. After rerunning the analysis the results should be checked against their unoptimized version. When applying an analysis to production code, the code should be compiled using production flags in order to analyze code that is as close as possible to what actually runs on the machine.

We found that the usage of debug symbols is helpful. The Clang compiler's `-g` flag can be added to propagate the debug symbols into the IR. Those can then be queried using LLVM's corresponding API. However, the debug symbols may not always present, which is why an analysis should not rely on them.

8 Future Work

In this section we briefly summarize our plans for future improvements.

It would be interesting to evaluate the use of PhASAR for analyzing a different IR. One type of IR might advantages over others for different analysis problems. We plan to additionally support the GENERIC, GIMPLE and RTL [19,5] IR from the GCC project.

Another interesting framework for data-flow analysis is *Weighted Pushdown Systems* (WPDS) [25,28]. WPDS is able to compute an analysis within a stack automaton. WPDS allows for more compact data structures, the generation of witnesses, as well as precise queries specifying paths of interest using regular expressions. We plan to support WPDS is a future version of PhASAR using the weighted/nested-word automaton library [32].

Checking the correctness of an IFDS/IDE analysis is complex since checking the correctness of the underlying ESG is tedious and time consuming. A high quality visualization may help reduce the amount of time spent debugging an analysis. A graphical user interface will reduce the amount of knowledge that is required to use the framework.

Since the flow and edge functions have to be implemented in a general purpose programming language, they require some amount of boilerplate code. It remains an open question if one could design a non-Turing-complete EDSL with a library like `boost::proto` [1] which simplifies the task of encoding analysis problems.

PhASAR currently uses LLVM's points-to information which is rather imprecise. We plan to integrate a more precise pointer analysis into PhASAR to support more precise call-graph construction and client analyses by adapting the demand-driven Boomerang approach presented in [29] to PhASAR.

9 Conclusion

In this paper, we presented our implementation of a static analysis framework for programs written in C/C++ named PhASAR. We presented its architecture and implementation from a user's perspective to make practical static analysis more accessible. We presented experiments which have shown PhASAR's scalability and discussed the runtimes of the key parts of two concrete client analyses.

With PhASAR we strive toward the goals of providing a framework for static analysis targeting (but not limited to) C/C++, a base for quickly evaluating novel ideas and applications, and a suitable way of handling the complexity. PhASAR is open-source and available online [23] under the permissive MIT licence, and therefore, open for contributions, feedback and use. PhASAR has already received tremendous support in the research community and from practitioners as 223 stars and 26 forks on GitHub show.[4]

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901) and the Heinz Nixdorf Foundation. We would also like to thank Richard Leer for his assistance in developing and improving the framework.

[4] As of 8am February 07, 2019.

References

1. Boost.proto (August 2018), https://www.boost.org/doc/libs/1_68_0/doc/html/proto.html
2. Clang: a c language family frontend for llvm (July 2018), http://clang.llvm.org/
3. coreutils (July 2018), https://www.gnu.org/software/coreutils/coreutils.html
4. Gcc, the gnu compiler collection (July 2018), https://gcc.gnu.org/
5. Gnu compiler collection (gcc) internals (July 2018), https://gcc.gnu.org/onlinedocs/gccint/
6. Andersen, L.O.: Program analysis and specialization for the c programming language. Tech. rep. (1994)
7. Bodden, E.: 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. pp. 3–8. SOAP '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2259051.2259052, http://doi.acm.org/10.1145/2259051.2259052
8. Clangstaticanalyzer (August 2018), https://clang-analyzer.llvm.org/
9. clang-tidy (August 2018), http://clang.llvm.org/extra/clang-tidy/
10. Codesonar (August 2018), https://www.grammatech.com/products/codesonar/
11. Coverity (August 2018), https://scan.coverity.com/
12. Cppcheck (August 2018), http://cppcheck.sourceforge.net/
13. Doop (August 2018), http://doop.program-analysis.org/
14. Eichberg, M., Hermann, B.: 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. pp. 1–6. SOAP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2614628.2614630, http://doi.acm.org/10.1145/2614628.2614630
15. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica **7**(3), 305–317 (Sep 1977). https://doi.org/10.1007/BF00290339, https://doi.org/10.1007/BF00290339
16. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective (10 2011)
17. Lattner, C., Adve, V.: 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. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673
18. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: A manifesto. Commun. ACM **58**(2), 44–46 (Jan 2015). https://doi.org/10.1145/2644805, http://doi.acm.org/10.1145/2644805
19. Merrill, J.: Generic and gimple: A new tree representation for entire functions. In: in Proc. GCC Developers Summit, 2003. pp. 171–180
20. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the ifds algorithm. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction. pp. 124–144. CC'10/ETAPS'10, Springer-Verlag, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11970-5_8, http://dx.doi.org/10.1007/978-3-642-11970-5_8

21. Nguyen, L., Krüger, S., Hill, P., Ali, K., Bodden, E.: Visuflow, a debugging environment for static analyses. In: International Conference for Software Engineering (ICSE), Tool Demonstrations Track (1 Jan 2018)
22. Padhye, R., Khedker, U.P.: 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. pp. 31–36. SOAP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2487568.2487569, http://doi.acm.org/10.1145/2487568.2487569
23. Phasar (July 2018), https://phasar.org
24. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 49–61. POPL '95, ACM, New York, NY, USA (1995). https://doi.org/10.1145/199448.199462, http://doi.acm.org/10.1145/199448.199462
25. Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Proceedings of the 10th International Conference on Static Analysis. pp. 189–213. SAS'03, Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1760267.1760283
26. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. **167**(1-2), 131–170 (Oct 1996). https://doi.org/10.1016/0304-3975(96)00072-2, http://dx.doi.org/10.1016/0304-3975(96)00072-2
27. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. New York Univ. Comput. Sci. Dept., New York, NY (1978), https://cds.cern.ch/record/120118
28. Späth, J., Ali, K., Bodden, E.: Context-, flow- and field-sensitive data-flow analysis using synchronized pushdown systems. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019) (13 - 19 Jan 2019), to appear.
29. Späth, J., Nguyen, L., Ali, K., Bodden, E.: Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In: European Conference on Object-Oriented Programming (ECOOP) (17 - 22 Jul 2016)
30. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 32–41. POPL '96, ACM, New York, NY, USA (1996). https://doi.org/10.1145/237721.237727, http://doi.acm.org/10.1145/237721.237727
31. Svf (August 2018), https://github.com/SVF-tools/SVF/
32. Wali-opennwa (July 2018), https://github.com/WaliDev/WALi-OpenNWA
33. Wala (August 2018), http://wala.sourceforge.net/wiki/index.php/Main_Page