srcPtr: A Framework for Implementing Static Pointer Analysis Approaches

Vlas Zyrianov
Department of Computer Science
Kent State University
Kent, Ohio USA
vzyriano@kent.edu

Christian D. Newman
Dept. of Software Engineering
Rochester Institute of Technology
Rochester, NY USA
cnewman@se.rit.edu

Drew T. Guarnera
Department of Computer Science
Kent State University
Kent, Ohio USA
dguarner@kent.edu

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, Ohio USA
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio USA
jmaletic@kent.edu

***Abstract*— A lightweight pointer-analysis framework, *srcPtr*, is presented to support the implementation and comparison of points-to analysis algorithms. It differentiates itself from existing tools by performing the analysis directly on the abstract syntax tree, as opposed to an intermediate representation (e.g., LLVM IR), by using srcML, an XML representation of source code. Working with srcML and the abstract syntax allows easy access to the actual source code as the programmer views it, thus better supporting comprehension. Currently the framework provides example implementations for both Andersen's and Steensgaard's pointer-analysis algorithms. It also allows for easy integration of other points-to algorithms for comparison of accuracy/speed. The approach is very scalable and can generate pointer dependencies for a 750 KLOC program in less than a minute.**

Keywords*— *Pointer analysis; static program analysis; srcML

I. Introduction

Pointer analysis encompasses a widely-used set of techniques aimed at determining, for a pointer, the possible set of memory locations it will reference at runtime. The idea of pointer analysis is simple: determine what memory locations a given pointer (or perhaps all pointers within a program) will point to. There are a number of characteristics that affect how efficient or precise a given points-to algorithm is [1]. For example, flow-sensitive algorithms consider control-flow information during analysis; context-sensitive algorithms examine the calling context when analyzing functions, respecting call/return semantics to avoid recording interprocedurally-unrealizable paths; algorithms that perform aggregate modeling pay attention to the internals of objects rather than collapsing them into singular objects; and other pointer-analysis algorithms model relationships using either points-to relationships or generalized aliasing [2].

The use of pointers in programs often decrease the understandability of the code. They clearly increase the complexity of conducting static analysis and are often the source of nefarious bugs. Hence, having tools for pointer analysis is of benefit to the programmer in the context of program comprehension.

There are numerous approaches for pointer analysis [3]–[8], each containing some set of the characteristics above [1]. Many of these approaches implement a single pointer-analysis algorithm and work on a specific target language. On its own, this is not a problem. However, if we are interested in comparing these algorithms based on speed/accuracy or how useful they are for comprehension, the inflexibility of these tools is a limitation.

We introduce *srcPtr*, a framework that directly supports the implementation, application, and comparison of pointer-analysis algorithms. *srcPtr* is tested on flow insensitive and context sensitive/insensitive, points-to analysis algorithms. In the future, the framework will also support flow-sensitive analysis. The framework is fast, scalable, and modular; allowing for any static pointer analysis algorithm to be implemented by extending the framework's interface. The tool also works directly with the source code rather than with an intermediate representation or byte code. This facilitates studying how these different algorithms can be used for tracking down errors and comprehending the code.

There are few open-source pointer-analysis tools available to researchers and students that are extensible. With *srcPtr*, developers can easily use the output of multiple pointer-analysis algorithms to solve research or industry problems. For example, one can estimate the impact of a change to a large system using multiple tools to help improve the accuracy of their estimation. This is very important for planning the implementation of new features and understanding how a change is related to other parts of the system. Furthermore, because the *srcPtr* framework is fast, this may be done very efficiently when using an efficient pointer-analysis algorithm; the framework does not significantly impact performance. Finally, we feel a fast pointer-analysis approach can open up new avenues of research in metrics and mining of histories. That is, pointer analysis can now be applied, compared, and synergized on very large systems and even on entire version histories in very practical time frames. This opens the door to a number of experiments and empirical investigations previously too costly to undertake.

II. srcPTR

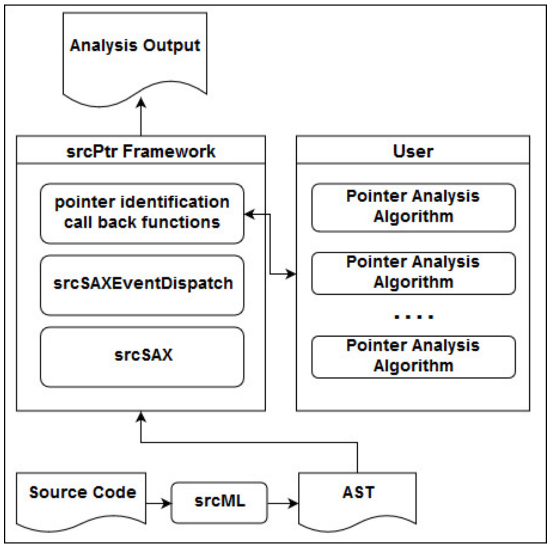
The *srcPtr* framework is built using the srcML [9] infrastructure (www.srcML.org). srcML is an XML format which augments source code with abstract-syntax information. srcML provides direct access to this information to support several activities including static analysis and fact extraction, which are core to *srcPtr*. While srcML supports several languages (i.e., C/C++/C#/Java), the current implementation of srcPtr has been built to operate on C / C++ grammar, but could be extended to support other languages in the future. srcPtr uses the *srcSAXEventDispatcher* (part of the srcML infrastructure) for parsing the generated srcML.

The core responsibility of *srcPtr* is to collect the information necessary to conduct pointer analysis. The framework uses a two-pass algorithm over the srcML document; this document can be one file, an entire system/project, or even a single function. We detail the main activities of this two-pass process within the next three subsections.

A. Initial Data Collection

The first pass of *srcPtr* collects data about class definitions and free-function definitions/declarations. From class definitions, *srcPtr* collects the name, specifiers, and types of public fields and method parameters. It also collects the name and number of parameters of public methods. From free-function definitions, *srcPtr* collects the number of parameters and name/type/specifier for each parameter. Local variables are ignored during the first pass but are resolved in the second pass.

Fig. 1. The architecture of *srcPtr*. Source code is translated into srcML, then it is traversed using srcSAX and pointer information is collected and stored in a dictionary. Pointer-analysis algorithms are then applied to the dictionary before srcPtr outputs the result.


Analysis Output
srcPtr Framework
pointer identification call back functions
srcSAXEventDispatch
srcSAX
User
Pointer Analysis Algorithm
Pointer Analysis Algorithm
....
Pointer Analysis Algorithm
Source Code
srcML
AST

B. Resolving and Analyzing Pointers

The second pass is where *srcPtr* scans for usages of variables collected in the previous steps by re-traversing srcML's AST. That is, it looks for where a variable is assigned or where it is passed into a function or method call. *srcPtr* additionally begins to keep track of local variables, which it skips during the first pass, collecting declaration and usage information. Each time a local declaration is found, *srcPtr* collects the variable's name, specifiers (e.g., constness, whether it is a pointer or reference), type, line number, and containing file. Declarations are stored on a scope stack to differentiate between identifiers with the same name. Every time a new scope is entered, a new frame is pushed onto the stack. When the algorithm exits a scope, the top of the stack is popped. Each frame is composed of two items: a map (i.e., dictionary) from variable names to their collected data (i.e., {{varname_1, vardata_1}, ..., {varname_n, vardata_n}}), and a map from function names to data about the function (i.e., {{funcname_1, funcdata_1}, …, {funcname_n, funcdata_n}}). This way, we can differentiate between variables in differing scopes when collecting data.

When a local variable is found, *srcPtr* checks to see if the type of this variable is that of a known class. If it is, each of the class' public member variables and methods are pushed onto the stack, preceded by the name of the variable. For example, let us assume a class named string with a single public member variable *c_str* and two methods: *print()* and *trim()*. If an object x of type string is declared, we create two variables, x and *x.c_str*, and two functions, *x.print* and *x.trim*. All of this is pushed onto the scope stack. This supports aggregate modeling.

While collecting local-variable information and constructing the stack, *srcPtr* resolves points-to relationships by keeping track of when a pointer is assigned a new memory location through either explicit assignment (i.e., using '=') or calls made to functions/methods (i.e., the pointer is a parameter). When a pointer assignment is detected, *srcPtr* calls one of two special functions. These special functions are how pointer-analysis algorithms interact with *srcPtr*. We discuss these now.

C. Performing Pointer Analysis with srcPtr

srcPtr allows integration of custom pointer-analysis algorithms via a template-method design pattern. The analysis algorithm must implement a class with two methods to handle pointer changes: *AddAssignmentRelationship(lhs,rhs)*, which is called when a pointer is assigned to another pointer, and *AddPointsToRelationship(lhs,rhs)*, which is called when a pointer is assigned to point to a variable. These two methods cover everything that can happen to a pointer in C++. The class containing these two methods is passed via template to *srcPtr*'s main class. These methods are triggered as described above and the algorithm can process/store data as required. A graphical representation of *srcPtr*'s architecture appears in Fig. 1.

srcPtr provides a default algorithm that is used when no custom pointer-analysis algorithm is provided. This default algorithm collects information about pointer assignments, but no analysis is performed. We use it to help measure overhead in *srcPtr*'s performance. Additionally, we have integrated two well-known points-to analysis algorithms, Andersen's [10] and Steensgard's [11], into the *srcPtr* framework.

III. Output

Fig. 2 gives an example of code with pointers. We ran this code through *srcPtr* and used our implementation of Andersen's algorithm to generate results. Output is in Table 1 where the left column gives the line number, data type, alias type and name of each pointer identifier. The right column gives the line number, data type, and name of the identifier pointed to by the corresponding pointer found in the left column. For example, num2 is a pointer to an integer that is declared on line 6. During execution, it might end up pointing to the integer declared on line 6, calc.result, and another integer declared on line 10 called y. In this example, only integers are used as datatypes (and pointers to them), but *srcPtr* supports all datatypes.

This example also highlights a caveat of static pointer analysis. Based on the code, the variable num2 could never end up pointing to calc.result, but since the pointer analysis is done statically, this has to be assumed. *srcPtr* also supports the generation of visualizations as seen in Fig. 3 using graphViz and the pointer-analysis results base on the code in Fig 2.

```
 1 class Calculator { public:
 2    void Add(int& num1, int* num2) {
 3       int sum = num1 + *num2;
 4       result = sum;
 5    }
 6    int result;
 7 };
 8 int main() {
 9    int x = 12;
10    int y = 24;
11    int* ptr = &y;
12    Calculator calc;
13    calc.Add(x, ptr);
14    ptr = &calc.result;
15    std::cout << *ptr;
16 }
```

Fig. 2. Code with pointers and references used directly and as parameters.

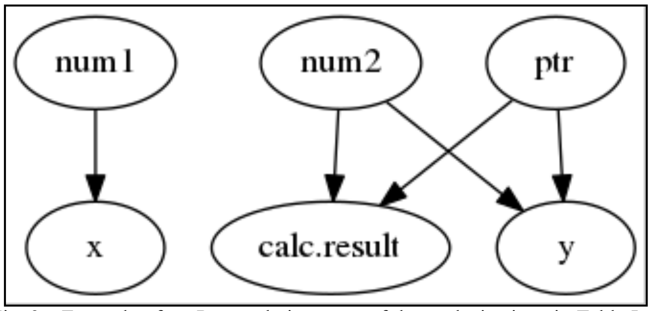


Fig. 3. Example of *srcPtr* graphviz output of the analysis given in Table I of the code example in Fig. 2

Table 1. Points-to relationships for example given in Fig. 2. Line number of the code is given in parentheses.

(Line #) Pointer	(Line #) Points to
`(2)  int & num1`	`(9)  int x`

`(2)  int * num2`	`(6)  int calc.result`
	`(10) int y`

`(11) int * ptr`	`(6)  int calc.result`
	`(10) int y`

IV. Performance

srcPtr is built using the srcML infrastructure and implemented as a SAX parser. Translating source code to the srcML format is very fast with speeds of ~35 KLOC/sec, with translation of the entire Linux kernel in ~7 minutes. The SAX parser is a C++ wrapper around libxml2's SAX interface called srcSAX; it was made specifically to support building tools that use srcML. Since SAX parsers store no data about previously seen nodes (e.g., tags), they are very memory and operation efficient. To take full advantage, we have worked to store as little data as possible and minimize repetition. As a result, *srcPtr* is quite fast.

The blue line in Fig. 4 is baseline performance with no pointer analysis and demonstrates the framework overhead of *srcPtr*. The results are that *srcPtr* is quite scalable as the runtime is linear with respect to the number of lines of code. The orange line in Fig. 4 is pointer analysis performed with Andersen's algorithm using *srcPtr*. The runtime complexity of Andersen's algorithm is O(n^3), which remains consistent with our findings due to *srcPtr*'s low overhead.

To determine if our framework runtime is comparable to current practices, we also ran Andersen's pointer analysis using *srcPtr* on Git to compare runtime performance with the tool PtrTracker [12]. Converting the Git source code to srcML takes less than 8 seconds and running Andersen's algorithm using srcPtr takes 43 seconds. The total runtime from original source code to the output of the points-to analysis is under 1 minute. The time reported for PtrTracker for a similar analysis was 27 minutes. It is worth noting that the speed difference could be due to PtrTracker's use of the Goanna architecture for AST generation and analysis. Accuracy differences between the two implementations was also not considered.

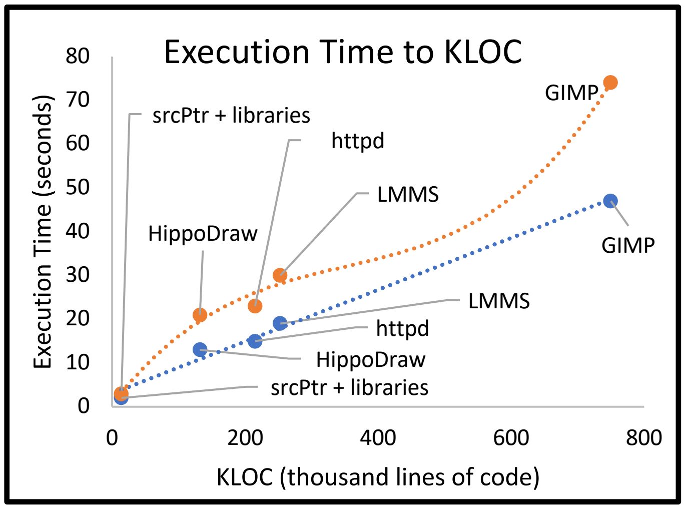


Fig. 4. Comparison of the execution time of the framework to the KLOC of the source code. The blue line is the baseline runtime of data collection versus lines of code, and the results show that the data collection is linear. The orange line shows runtimes with Andersen's algorithm, and demonstrates that the runtime complexity of O(n^3) is preserved with *srcPtr*.

V. Using srcPtr

The example implementations included with *srcPtr* can be run the using the command-line tool srcptr to apply pointer analysis on the srcML format of source code. To convert to srcML, the srcml command-line client is used with the option --*position* so that line numbers are included:

srcml main.cpp --position -o main.cpp.xml

The command to run srcptr on a srcML file is:

srcptr -a main.cpp.xml > pointer.out

where main.cpp.xml is the srcML format of the main.cpp and pointer.out is the output file. The option –a selects Andersen's algorithm and -s for Steensgard's. As other algorithms are implemented, you will be able to use other flags. By default, srcptr outputs the pointer report to stdout.

The client srcptr supports pointer-graph generation with graphViz. To make srcptr generate a dotfile use the flag '-g':

```
srcptr -a -g file.cpp.xml > graph.dot
```

Then, run the following command to generate a graph with graphViz based on the file created earlier (requires graphViz):

```
dot -Tpng graph.dot > out.png
```

You can download *srcPtr* at www.srcML.org under tools. Installation instructions and documentation are also available.

VI. Related Work

Pointer analysis is a heavily-researched topic spanning over a decade of published work. We direct readers interested in a broad overview of the field to surveys [1], [13], [14]. Much of the current work in pointer analysis supports compiler optimizations [3]–[6]. However, SVF can be used more broadly as it a general-purpose analysis framework. These approaches and supporting frameworks utilize compiler technologies, specifically LLVM, to supply the IR (intermediate representation) and core analyses. These IR representations are represented as low-level assembly instructions and require the source code to be complete and compiling.

The goal of PtrTracker [12] is to perform pointer analysis and improve the accuracy of software bugs and vulnerabilities detected by Goanna. PtrTracker is modeled after shape analysis and uses heap graphs for pointer analysis. Goanna is capable of producing an AST of source code in XML and performs interprocedural analysis. PtrTracker integrates with Goanna by enhancing the AST it builds with sets of variables that might be affected by pointer dereferences. Goanna then uses the enhanced AST to complete its bug detection analysis.

srcPtr differentiates itself from the existing approaches by using a much higher-level IR representation. For our work, srcML serves as the IR by inserting AST elements into the source code and allowing the original contents of the source code (whitespace and comments included) to remain intact when moving to and from the srcML format. Parsing with srcML does not require compilation of the source code, which permits *srcPtr* to run analysis on incomplete code, and even code fragments. This allows for better developer support for tasks such as source-code transformation [7] and impact analysis [8]. Using a high-level IR permits *srcPtr* to collect much more code centric pointer information. However, this introduces a trade-off – analyzing C++ code is harder then a lower level representation (LLVM IR or assembly).

VII. Conclusions and Future Work

The framework *srcPtr* supports the use and implementation of pointer-analysis algorithms. *srcPtr* has a number of advantages that make it useful for developers and researchers. There is no need to compile the code or even have a complete system that compiles. This way analysis can be easily done on partial code or only part of a system. This also makes the framework more standalone in that it is not tied to any specific compiler. The framework exhibits low overhead and scales well making experimenting with different analysis algorithms easy.

Currently *srcPtr* is implemented for C/C++ but srcML also supports C# and Java. It is relatively straightforward to extend *srcPtr* to C#/Java. As a research prototype, limitations exist. However, *srcPtr* can be applied to large systems without issue. Because srcML supports multiple languages, pointer analysis can feasibly be done on systems that are written in multiple languages. We also see this as a platform to compare and experiment with pointer-analysis techniques.

There are a number of general limitations. Only simple type-based points-to analysis is supported. This does not address full alias analysis. We do not do any flow analysis. Our type analysis is somewhat limited (but can be improved). Function pointers are not considered at this time; only c-pointers. There is also no support for STL shared pointers. This work was supported in part by a grant from the US National Science Foundation CNS 13-05292/05217.

VIII. References

[1] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?," in *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Snowbird, Utah, 2001, pp. 54–61.

[2] R. O'Callahan, "Generalized aliasing as a basis for program analysis tools," Carnegie-Mellon University Pittsburgh, PA, USA School of Compuer Science, Doctoral Dissertation, 2000.

[3] B. Hardekopf and C. Lin, "Flow-sensitive Pointer Analysis for Millions of Lines of Code," in *9th IEEE/ACM International Symposium on Code Generation and Optimization*, Chamonix, France, 2011, pp. 289–298.

[4] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-flow Analysis in LLVM," in *25th International Conference on Compiler Construction*, Barcelona, Spain, 2016, pp. 265–266.

[5] N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August, "A Collaborative Dependence Analysis Framework," in *2017 International Symposium on Code Generation and Optimization*, Austin, Texas, USA, 2017, pp. 148–159.

[6] M. Maalej, V. Paisante, P. Ramos, L. Gonnord, and F. M. Q. Pereira, "Pointer disambiguation via strict inequalities," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Austin, Texas, USA, 2017, pp. 134–147.

[7] M. Buss, S. A. Edwards, B. Yao, and D. Waddington, "Pointer analysis for source-to-source transformations," in *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, Budapest, Hungary, Hungary, 2005, pp. 139–148.

[8] M. Acharya and B. Robinson, "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems," in *33rd International Conference on Software Engineering*, Waikiki, Honolulu, HI, USA, 2011, pp. 746–755.

[9] Jonathan I. Maletic and Michael L. Collard, "Exploration, Analysis, and Manipulation of Source Code using srcML," presented at the 37th International Conference on Software Engineering - Volume 2, Florence, Italy, 2015.

[10] L. O. Andersen and Københavns Universitet. Datalogisk Institut, *Program Analysis and Specialization for the C Programming Language*. Datalogisk Institut, 1994.

[11] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, USA, 1996, pp. 32–41.

[12] S. Biallas, M. C. Olesen, F. Cassez, and R. Huuck, "PtrTracker: Pragmatic pointer analysis," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Eindhoven, Netherlands, 2013, pp. 69–73.

[13] M. Hind and A. Pioli, "Which Pointer Analysis Should I Use?," in *2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Portland, Oregon, USA, 2000, pp. 113–123.

[14] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias Analysis for Object-Oriented Programs," in *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, vol. 7850, Springer, Berlin, Heidelberg, 2013, pp. 196–232.