SVF: Interprocedural Static Value-Flow Analysis in LLVM

Yulei Sui Jingling Xue

School of Computer Science and Engineering, UNSW Australia

Abstract

This paper presents SVF, a tool that enables scalable and precise interprocedural Static Value-Flow analysis for C programs by leveraging recent advances in sparse analysis. SVF, which is fully implemented in LLVM (version 3.7.0), allows value-flow construction and pointer analysis to be performed in an iterative manner, thereby providing increasingly improved precision for both. SVF accepts points-to information generated by any pointer analysis (e.g., Andersen's analysis) and constructs an interprocedural memory SSA form, in which the def-use chains of both top-level and address-taken variables are captured. Such value-flows can be subsequently exploited to support various forms of program analysis or enable more precise pointer analysis (e.g., flow-sensitive analysis) to be performed sparsely. By dividing a pointer analysis into three loosely coupled components: *Graph*, *Rules* and *Solver*, SVF provides an extensible interface for users to write their own solutions easily. Moreover, our memory SSA design allows users to make scalability and precision trade-offs by defining their own memory partitioning strategies. We discuss some usage scenarios and our previous experiences in using SVF in several client applications.

SVF is available at `http://svf-tools.github.io/SVF`.

1. Introduction

Due to the sheer complexity of modern software systems, finding and fixing software bugs is far cheaper earlier in the software development life cycle (e.g., during the coding stage) than later (e.g., during the testing stage), resulting in higher quality software [9, 39]. Static analysis, which approximates the runtime behaviour of a program at compile time, is a fundamental approach to helping developers catch bugs effectively in early stages of software development.

In static analysis, a fundamental research problem is to resolve program dependencies (aka value-flows). The more precisely the value-flows are resolved, the more effective static analysis will be. By improving the precision and scalability of static value-flow analysis, we can significantly improve the effectiveness of virtually all other forms of program analysis on detecting a variety of bugs, such as memory leaks [11, 36], uninitialised variables [43], security vulnerabilities [28], and tainted information flow [4, 8].

Static value-flow analysis resolves both the data and control dependences of a program. It was initially adopted in software debugging [40, 41] and optimising compilers [17, 33] by providing explicit definition-use relations of program variables. This fundamental technique has subsequently been used widely for program analysis and verification in many open-source and commercial tools. The Wisconsin program-slicing project [22] is a well-known research prototype that supports both forward and backward slicing on its program dependence graph. Later, the tool was integrated into the commercial product CodeSurfer [2]. WALA [23] is an open-source Java analysis framework that provides interprocedural data-flow analysis and a context-sensitive tabulation-based slicer. Recently, Heros [7] also includes an IFDS/IDE [32] solver for analysing single- and multi-threaded code in the Soot framework [25]. Some industry static analysis tools that use program dependence analysis include Coverity [6] from Synopsys, Parfait [13] from Oracle, and SLAM (built on top of Microsoft's in-house compiler) [5].

For the mainstream open-source compilers (e.g., GCC and LLVM), most of program dependence analyses used are intraprocedural with limited alias analysis support, as is the case for LLVM's memory dependence analysis. However, many client applications, such as memory leak detection, require value-flows to be analysed across the procedural boundaries, for which interprocedural analysis is essential.

The traditional iterative approach for computing interprocedural value-flows is costly and unscalable for large programs. Recent progresses in *sparse analysis* [19, 30, 36, 38, 44, 45] provide a promising solution for analysing large programs scalably and precisely. To avoid expensive propagation of data-flow facts across a program's control flow graph, sparse analysis is usually conducted in stages: a pre-analysis is first applied to over-approximate a program's def-use chains, which are then refined by performing a data-flow analysis sparsely, i.e., only along such pre-computed def-use relations.

In this paper, we present SVF, a tool that enables scalable and precise interprocedural analysis for C programs by leveraging recent advances in sparse analysis. SVF allows value-flow construction and pointer analysis to be performed iteratively, thereby providing increasingly improved precision for both. SVF accepts points-to information generated by any pointer analysis (e.g., Andersen's analysis) and builds an interprocedural memory SSA (Static-Single Assignment) form so that the def-use chains of both top-level and address-taken variables are captured. These value-flows can be subsequently exploited to support various forms of program analysis or enable more precise pointer analysis (e.g., flow-sensitive analysis) to be performed sparsely.

SVF is fully implemented in an industry-strength compiler LLVM (in its latest version 3.7.0). The LLVM platform [26] is designed as a set of reusable libraries with a well-defined IR (Intermediate Representation). It has been recognised as a common infrastructure to support analysis and transformation with many front-ends (e.g., C/C++, Objective-C/C++, OpenMP, Java [14] and Javascript [3]). By using the LLVM IR as input, SVF can potentially tap into LLVM's front-ends to handle programs written in other languages (in addition to C).

2. Design Overview

Our SVF framework is depicted in Figure 1. The source code of a program is first compiled by clang into bit-code files, which are merged by LLVM Gold Plugin at link time stage (LTO) to produce a whole-program bc file. Then the "Pointer Analysis" module is invoked. Based on the points-to information obtained, the "Value-Flow Construction" module puts the program in memory SSA form so that the def-use chains for top-level and address-

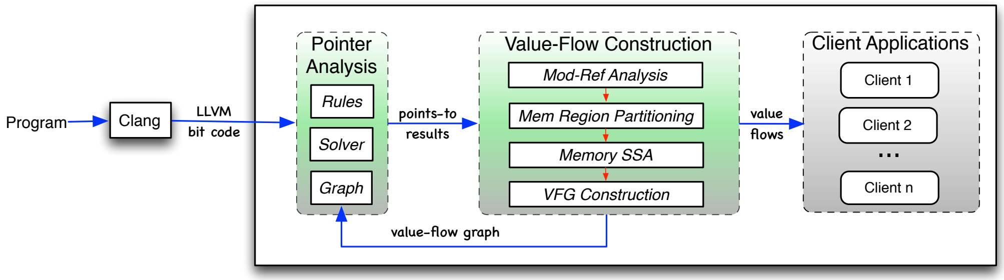


Figure 1. Overview of SVF.

taken variables are identified. These value-flows can be used by a variety of "Client Applications", as discussed in Section 4. If better precision is desired by a client, the current value-flows and points-to information can both be refined iteratively by performing a *sparse pointer analysis* based on the current value-flows [19, 30, 36, 38, 44, 45], as highlighted by the edge directed from "Value-Flow Construction" to "Pointer Analysis".

2.1 Pointer Analysis

Existing implementations for C [19, 27, 45] compute points-to information directly on the LLVM IR of a program, thereby making them difficult to maintain and extend. In contrast, our design consists of three loosely coupled components, Graph, Rules, and Solver. Graph is a higher-level abstraction extracted from the LLVM IR of a program, indicating *where* pointer analysis should be performed. The Rules component defines *how* to derive the points-to information from each statement, i.e., each constraint on the graph. Solver determines in *what* order to resolve all the constraints.

As a result, SVF provides a clean and reusable interface for users to write their own pointer analysis implementations by combining our three components in a flexible way. For example, Andersen's pointer analysis can be written easily by choosing an appropriate solver, e.g., Wave [31] and applying transitive closure rules [1] on an inclusion-based constraint graph. Similarly, a flow-sensitive pointer analysis can be implemented by applying a set of flow-sensitive strong/weak update rules with a points-to propagation solver on a sparse value-flow graph [19, 36].

2.2 Value-Flow Construction

Based on the points-to information obtained, we first perform a lightweight "Mod-Ref Analysis" to capture interprocedural reference and modification side-effects for each variable [37, Section 4.2.1]. Thus, the (alias) set of indirect defs (uses) at a statement s (i.e., a store, load or callsite) in each procedure is obtained and denoted as D_s (U_s). The "Mem Region Partitioning" module allows users to partition memory into a set of (not necessarily disjoint) regions $R_1, \ldots, R_n$ so that scalability and precision trade-offs can be made in analysing large programs. Then statement s is annotated with every R_i, where $D_s \cap R_i \neq \emptyset$ ($U_s \cap R_i \neq \emptyset$), to make explicit the abstract memory objects that may be defined (used) indirectly at s. Note that in Open64 [12] and GCC [29], their memory SSA forms are computed only intraprocedurally, rendering all non-local variables to be placed in one single alias set. Once indirect uses and defs are known, the "Memory SSA" module is invoked to put the program in memory SSA form using a standard SSA conversion algorithm [15]. Finally, the value-flows in the program are captured in a VFG (Value-Flow Graph) by connecting a def of a variable with its uses, accomplished by the "VFG Construction" module.

3. Sparse Value-Flow Representation

We describe below how SVF uses a VFG representation to capture the value-flows in a program. For simplicity, we assume that distinct abstract memory objects are in distinct regions. We adopt the convention of LLVM by separating the set $\mathcal{V}$ of all variables in a program into two subsets, (1) $\mathcal{A}$ containing all possible targets, i.e., *address-taken variables* of pointers and (2) $\mathcal{T}$ containing all *top-level variables* whose addresses are not taken, where $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$.

A sparse VFG for a program is a directed graph that captures the def-use chains of all variables [19, 36]. The def-use chains for top-level variables are readily available once they are in SSA form. Address-taken variables are accessed indirectly at loads and stores. Their def-use chains are built in several steps [12, 36]. First, the points-to information for the program is computed by using, e.g., Andersen's analysis. Second, a load $p = *q$ is annotated with a function $\mu(o)$ for each variable $o \in \mathcal{A}$ that may be pointed to by $q \in \mathcal{T}$ to represent a potential use of o at the load. Similarly, a store $*p = q$ is annotated with $o = \chi(o)$ for each variable $o \in \mathcal{A}$ that may be pointed to by $p \in \mathcal{T}$ to represent a potential def and use of o at the store. If o can be strongly updated, then o receives whatever q points to and the old contents in o are killed. Otherwise, o must also incorporate its old contents, resulting in a weak update to o. A callsite cs is also annotated with $\mu(o)$ and $o = \chi(o)$, where $o \in \mathcal{A}$, to capture interprocedural uses and defs of o. Likewise, $o = \chi(o)$ ($\mu(o)$) is annotated at the entry (exit) of a procedure f to mimic the parameter passing (return) for a non-local variable $o \in \mathcal{A}$. Third, all address-taken variables are converted to SSA form, with each $\mu(o)$ operation being treated as a use of o and each $o = \chi(o)$ operation being treated as both a def and a use of o.

3.1 VFG Construction

Given a program with annotated μ and χ functions after the SSA conversion, its VFG is constructed by connecting the def of each SSA variable $v \in \mathcal{V}$ with its uses. Each node in the VFG represents one of the following:

- A definition of a variable at a non-call statement ℓ:
  - COPY ($\ell : p = q$): $p@\ell$;
  - PHI ($\ell : v_3 = \phi(v_2, v_1)$): $v_3@\ell$;
  - LOAD ($\ell : p = *q\ [\mu(o)]$): $p@\ell$; and
  - STORE ($\ell : *p = q\ [o_2 = \chi(o_1)]$): $o_2@\ell$.
- A variable defined (directly or indirectly) as a return value at a callsite $\ell_{cs} : r = f(\_)\ [\mu(\_)]\ [o = \chi(\_)]$:
  - DRET (Value Directly Returned): $r@\ell_{cs}$; and
  - IRET (Value Indirectly Returned): $o@\ell_{cs}$.
- A parameter defined (directly or indirectly) at the entry of a procedure $f(..., p, ...)\{[o = \chi(\_)]\ ...\ [\mu(\_)]\ return\ \_\}$:
  - DPARA (Parameter Directly Initialised): $p@\ell_f$.

Table 1. Rules for building VFG ($p, q, r, x \in \mathcal{T}, o \in \mathcal{A}, v \in \mathcal{V}$) .

Rule	Statement (SSA)	Value-Flow Edges
COPY	$\ell : p = q$	$p@\ell \hookleftarrow q@\ell'$
PHI	$\ell : v_3 = \phi(v_1, v_2)$	$v_3@\ell \hookleftarrow v_1@\ell'$ $\quad v_3@\ell \hookleftarrow v_2@\ell''$
LOAD	$\ell : p = *q\ [\mu(o)]$	$p@\ell \hookleftarrow o@\ell'$
STORE	$\ell : *p = q\ [o_2 = \chi(o_1)]$	$o_2@\ell \hookleftarrow q@\ell'$ $\quad o_2@\ell \hookleftarrow o_1@\ell''$
CALL	$\ell_{cs} : r = f(..., p, ...)\quad [\mu(o_1)]\quad [o_2 = \chi(\_)]$ $\ell_f : f(..., q, ...)\{[o_3 = \chi(\_)]\quad ...\quad [\mu(o_4)]$ return $x\}$	$q@\ell_f \hookleftarrow p@\ell_1$ $\quad r@\ell_{cs} \hookleftarrow x@\ell_2$ $o_3@\ell_f \hookleftarrow o_1@\ell_3$ $\quad o_2@\ell_{cs} \hookleftarrow o_4@\ell_4$

- IPARA (Parameter Indirectly Initialised): $o@\ell_f$.

An edge between two nodes is added to represent either a *direct value-flow* for a top-level variable or an *indirect value-flow* for an address-taken variable. Table 1 lists the rules used.

For a COPY assignment between two top-level pointers $\ell : p = q$, a direct value-flow edge is added from the def of q at ℓ' to the def of p at ℓ based on the def-use information of q. Instead of linking the def of q to the use of q and then linking this use to the def of p, we add one single edge $p@\ell \hookleftarrow q@\ell'$ directly. We do the same in the other rules.

For a PHI statement $\ell : v_3 = \phi(v_2, v_1)$ at a control-flow joint point, the value-flows are connected from the defs $v_1@\ell'$ and $v_2@\ell''$ of the old SSA instances v_1 and v_2 of variable $v \in \mathcal{V}$ to the def $v_3@\ell$ of the new instance v_3.

A LOAD $\ell : p = *q$ annotated with $\mu(o)$ indicates that o may be used indirectly via $*q$. Thus, the indirect value-flow of o is connected from its def $o@\ell'$ to $p@\ell$. Similarly, a STORE $\ell : *p = q$ annotated with $o_2 = \chi(o_1)$ represents the fact that either the value of o is completely overwritten by the value from $q@\ell'$ (a strong update) or the old value from its previous def $o_1@\ell''$ must be also preserved (a weak update).

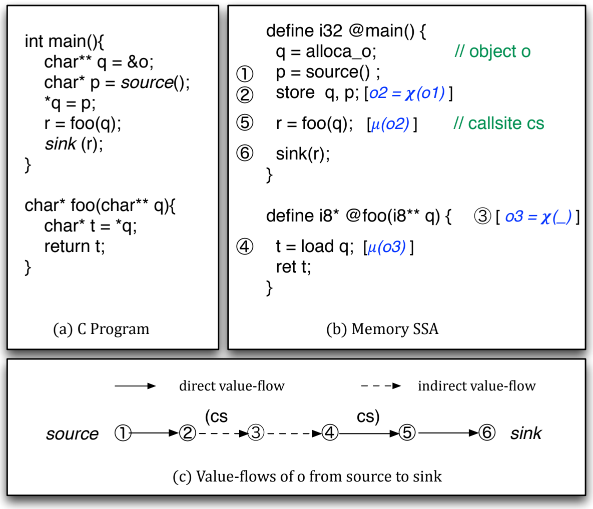
When handling parameter/return passing, the CALL rule shows how to connect interprocedural value-flows for both top-level pointers p, q, r and x as well as an address-taken variable o. A direct value-flow edge $q@\ell_f \hookleftarrow p@\ell_1$ is connected from the def (at ℓ_1) of an actual parameter p at a callsite ℓ_{cs} to its corresponding formal parameter q in a callee f (for `DPara`). Similarly, $r@\ell_{cs} \hookleftarrow x@\ell_2$ captures the direct value-flow of x (with its def at ℓ_2) returned from f to its caller at ℓ_{cs} (for `DRet`).

Handling the indirect value-flows for address-taken variables is conceptually the same as handling the direct value-flows for top-level variables. Given $\mu(o_1)$ annotated at callsite ℓ_{cs}, we know that o may be used in a callee procedure f. As a result, $o_3 = \chi(\_)$ annotated at the entry of f can be understood as an implicit formal parameter receiving o. Thus, an indirect value-flow $o_3@\ell_f \hookleftarrow o_1@\ell_3$ is connected, with the def of o_1 at ℓ_3 (for `IPara`). Likewise, $o_2@\ell_{cs} \hookleftarrow o_4@\ell_4$ captures the indirect value-flow of o returned from f its caller at ℓ_{cs}, with the def of o_4 at ℓ_4 (for `IRet`).

As is standard for supporting context-sensitive analysis, the call and return edges at a callsite cs are labelled with $(_{cs}$ and $)_{cs}$, respectively. Therefore, context sensitivity can be solved as a reachability analysis formulated as a balanced-parentheses problem in the standard manner by matching calls and returns to filter out unrealisable interprocedural paths [32].

3.2 Example

Figure 2 shows a simple C program in Figure 2(a), its memory SSA in Figure 2(b), and the value-flows starting from the def of p at ① (source) to ⑥ (sink) in Figure 2(c). Andersen's pointer analysis is first performed to determine that q points-to an object $o \in \mathcal{A}$. Then χ functions are added at the entries of `main` and `foo` and the store at ② to represent the indirect defs of o, and μ added at the callsite at ⑤ and the load at ④ to represent the indirect uses of o.



```
int main(){
   char** q = &o;
   char* p = source();
   *q = p;
   r = foo(q);
   sink (r);
}

char* foo(char** q){
   char* t = *q;
   return t;
}
```

(a) C Program

```
     define i32 @main() {
       q = alloca_o;              // object o
 ①     p = source() ;
 ②     store  q, p; [o2 = χ(o1) ]
 ⑤     r = foo(q);   [μ(o2) ]       // callsite cs
 ⑥     sink(r);
     }

     define i8* @foo(i8** q) {    ③ [ o3 = χ(_) ]
 ④     t = load q;  [μ(o3) ]
       ret t;
     }
```

(b) Memory SSA

Figure 2. A value-flow example.

Figure 3 shows part of a VFG generated by SVF, illustrating some program dependences of a small program `art` in SPEC2000. Just like in the case of a constraint graph, SVF also highlights different types of nodes in a VFG in different colors, e.g., COPY (in black), LOAD (in red), STORE (in blue) and DPARA/IPARA/DRET/IRET (in yellow). In addition, the direct and indirect value-flow edges are shown as solid and dotted arrows, respectively. The call and return edges are depicted in red and blue, respectively.

Figure 4 shows a screenshot of our eclipse plugin that accepts the output of SVF's analysis results. It shows a typical source-sink related-bug detected by SVF on its VFG. The program is a simple test case from NIST's Juliet Test Suite. When tracing along the program's value-flows, SVF reports a use-after-free error that the pointer variable, `data`, is used in `printf` function at line 37 after it has been freed at line 33. The plugin reads the value-flow traces and maps them to the original source code according to the debugging information in LLVM IR. It provides a simple yet clean reporting interface with a "problem" tab at the bottom of the eclipse IDE for developers to pinpoint the point of the bug easily.

4. Usage Scenarios

We discuss below four usage scenarios as well as our previous experience in using SVF in the past few years.

4.1 Source-Sink Analysis

Many software bug detection methods can be formulated as one of reasoning about some source-sink properties via value-flow reachability. A typical client is memory leak detection, which in-

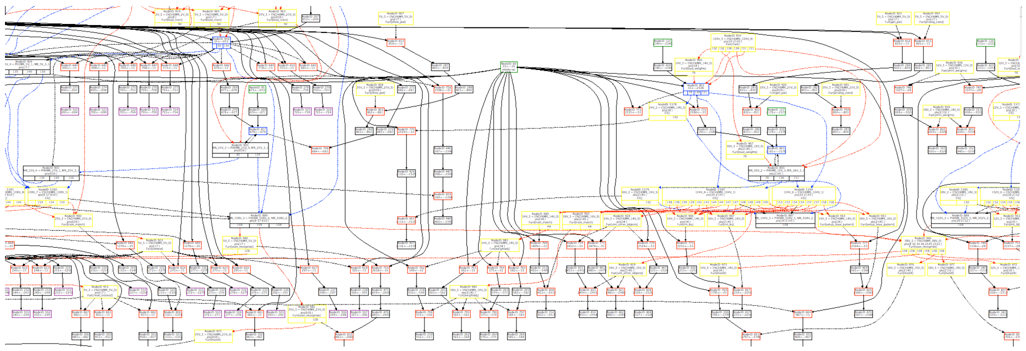
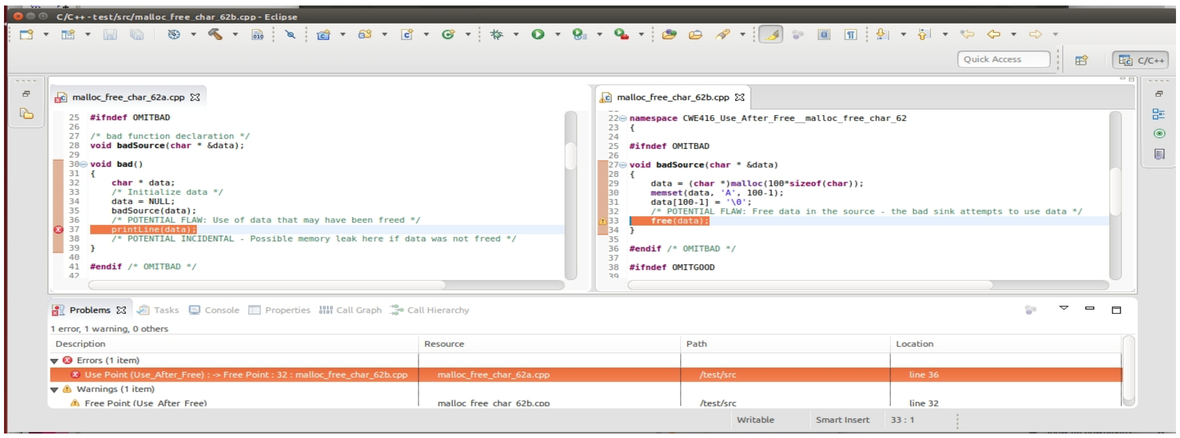


Figure 3. A screenshot for the VFG of `art` in SPEC2000.



C/C++ - test/src/malloc_free_char_62b.cpp - Eclipse

malloc_free_char_62a.cpp

```
25  #ifndef OMITBAD
26
27  /* bad function declaration */
28  void badSource(char * &data);
29
30  void bad()
31  {
32      char * data;
33      /* Initialize data */
34      data = NULL;
35      badSource(data);
36      /* POTENTIAL FLAW: Use of data that may have been freed */
37      printLine(data);
38      /* POTENTIAL INCIDENTAL - Possible memory leak here if data was not freed */
39  }
40
41  #endif /* OMITBAD */
```

malloc_free_char_62b.cpp

```
22  namespace CWE416_Use_After_Free__malloc_free_char_62
23  {
24
25  #ifndef OMITBAD
26
27  void badSource(char * &data)
28  {
29      data = (char *)malloc(100*sizeof(char));
30      memset(data, 'A', 100-1);
31      data[100-1] = '\0';
32      /* POTENTIAL FLAW: Free data in the source - the bad sink attempts to use data */
33      free(data);
34  }
35
36  #endif /* OMITBAD */
37
38  #ifndef OMITGOOD
```

Problems | Tasks | Console | Properties | Call Graph | Call Hierarchy

1 error, 1 warning, 0 others

Description	Resource	Path	Location
Errors (1 item)			
Use Point (Use_After_Free) : -> Free Point : 32 : malloc_free_char_62b.cpp	malloc_free_char_62a.cpp	/test/src	line 36
Warnings (1 item)			
Free Point (Use_After_Free)	malloc_free_char_62b.cpp	/test/src	line 32

Writable | Smart Insert | 33 : 1

Figure 4. A screenshot of use-after-free detection in our eclipse plugin.

volves checking whether a memory allocation (source) must eventually reach a free site (sink) along every execution path of the program. Given the value-flows for both top-level and address-taken variables, our leak detector SABER [36, 37] exploits a sweet spot in the precision/scalability tradeoff. SABER is as accurate as SPARROW [24] (a detector using abstract interpretation) in terms of finding real leaks and suppressing false alarms, while achieving comparable analysis performance as the fastest but less precise detector FASTCHECK [11]. Other bug detectors for checking source-sink properties, such as double-free, file open-close errors, and uses of tainted data, can also be developed easily in the SVF framework.

4.2 Pointer Analysis

Our sparse value-flow framework also opens up more opportunities for the design and implementation of scalable and precise pointer analyses. Our recent solution, SELFS [44], performs selective flow-sensitive pointer analysis on partitioned program regions based on pre-computed value-flow information. SELFS is able to infer the program parts where flow-sensitivity is needed by reasoning about the value-flow properties with load-precision-preserving resolution (implying that a pointer loaded from memory always maintains the same points-to results as the flow-sensitive analysis). Another recent work, FSAM [34] which is built on top of SVF, performs sparse flow-sensitive pointer analysis by applying a series of thread interference analysis phases for multithreaded C programs. Other pointer analysis variants (e.g., adaptive heap cloning [35]) are also made possible in our extensible SVF framework.

4.3 Accelerating Dynamic Analysis

Dynamic analysis, which monitors program execution behaviour using instrumentation, introduces a certain amount of runtime overhead. One possible client of SVF here is to perform selective instrumentation guided by static value-flow information, so that unnecessary instrumentations can be eliminated to reduce runtime overhead. In our previous work [43], we reported a tool called USHER for accelerating dynamic detection of undefined variables. USHER uses interprocedural value-flow analysis to identify the redundant shadow operations where instrumentation can be safely removed. Similar ideas can be applied to detect other bugs such as null dereference and buffer-overflow errors [42]. Furthermore, it is also interesting to see how to incorporate our static value-flow analysis into symbolic execution [10] and dynamic data-flow testing [16] to help both generate meaningful test cases more quickly.

4.4 Program Debugging and Understanding

SVF can also be used as a foundation for software debugging and program understanding [18, 21, 40]. SVF can provide developers with the statements that are potentially the causes of some erroneous behavior efficiently by tracing only the relevant value-flows, thereby bypassing many irrelevant statements in the process of localizing a bug. Scalable and precise interprocedural value-flow analysis is also helpful to software visualisation (e.g., code map [20]) for understanding large code bases.

5. Conclusion

In this paper, we have presented SVF, a scalable and precise interprocedural static value-flow analysis that serves as a foundation for program understanding and software bug detection. We have implemented SVF fully in LLVM with over 40 KLOC in C++ presently.

- The full source code can be downloaded from `https://github.com/unsw-corg/SVF`.

- A micro-benchmark suite including hundreds of synthetic and real test cases for validating the correctness of pointer analysis algorithms can be found at `https://github.com/unsw-corg/PTABen`.
- A working demonstration of SVF in a virtual machine can be found at `https://github.com/unsw-corg/SVF/wiki/Try-SVF-in-VirtualBox`.

In the past few years, SVF has been shown to provide an effective solution to several client applications in handling large real-world C programs. Our future work includes an extension for supporting object-oriented languages such as C++ and Objective-C.

References

[1] L. O. Andersen. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen*, 1994.

[2] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering (WISE '01)*, 2001.

[3] J. R. Andrew Trick. FTL WebKit's LLVM based JIT. In *LLVM Developer Meeting 2014*, 2014.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *PLDI '14*, 49(6):259–269, June 2014.

[5] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. *POPL '02*, 37(1):1–3, Jan. 2002.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[7] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *SOAP '12*, pages 3–8, 2012.

[8] E. Bodden. Position paper: Static flow-sensitive & context-sensitive information-flow analysis for software product lines. In *PLAS '12*, 2012.

[9] B. W. Boehm. Understanding and controlling software costs. *Journal of Parametrics*, 8(1):32–68, 1988.

[10] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08*, pages 209–224, 2008.

[11] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*, pages 480–491, 2007.

[12] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, pages 253–267.

[13] C. Cifuentes, N. Keynes, L. Li, and B. Scholz. Program analysis for bug detection using parfait: Invited talk. In *PEPM '09*, pages 7–8, 2009.

[14] N. K. Cristina Cifuentes Oracle Labs Australia, Oracle. Translating Java into LLVM IR to detect security vulnerabilities. In *LLVM Developer Meeting 2014*, 2014.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89*, pages 25–35.

[16] G. Denaro, A. Margara, M. Pezze, and M. Vivanti. Dynamic data flow testing of object oriented systems. In *ICSE '15*, 2015.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS '87*, 9(3):319–349, July 1987.

[18] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17:751–761, 1991.

[19] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*, pages 289–298.

[20] N. Hawes, B. Barham, and C. Cifuentes. Frappé: Querying the linux kernel dependency graph. In *GRADES '15*, pages 4:1–4:6, 2015.

[21] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ICSE '92*, pages 392–411. ACM, 1992.

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88*, pages 35–46, 1988.

[23] IBM. T.j. watson libraries for analysis (WALA).

[24] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *ISMM '08*, pages 131–140. ACM, 2008.

[25] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *in CETUS '11*, 2011.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–86, 2014.

[27] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16.

[28] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *FSE '03*, pages 317–326.

[29] D. Novillo and R. H. Canada. Memory SSA-a unified approach for sparsely representing memory operations. In *Proc of the GCC Developers' Summit*. Citeseer, 2007.

[30] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI '12*, pages 229–238.

[31] F. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135.

[32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[33] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *ESOP '90*, 1990.

[34] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multithreaded c programs. In *in CGO '16 (To Appear)*, 2016.

[35] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*, pages 1–11.

[36] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264.

[37] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[38] Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS '11*, pages 155–171.

[39] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[40] M. Weiser. Program slicing. In *ICSE '81*, pages 439–449, 1981.

[41] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[42] D. Ye, Y. Su, Y. Sui, and J. Xue. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. *ISSRE '14*, 2014.

[43] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined variables with static value-flow analysis. In *CGO '14*, pages 154–164.

[44] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336. Springer, 2014.

[45] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229.