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 dependencies 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 intra-procedural 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-
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 def’s (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 \( V \) of all variables in a program into two subsets, (1) \( A \) containing all possible targets, i.e., address-taken variables of pointers and (2) \( T \) containing all top-level variables whose addresses are not taken, where \( V = T \cup A \).

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 \( o \in A \) that may be pointed to by \( q \in T \) to represent a potential use of \( o \) at the load. Similarly, a store \( *p = o \) is written easily by choosing an appropriate solver, e.g., Wave [31] and applying transitive closure rules [1] to mimic the parameter passing (return) for a non-local variable \( o \in A \). 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 \( \mu \) and \( \chi \) functions after the SSA conversion, its VFG is constructed by connecting the def of each SSA variable \( v \in V \) with its uses. Each node in the VFG represents one of the following:

- A definition of a variable at a non-call statement \( \ell \):
  - 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 \( (\ell : *p = q [\chi(o_1)]); o_2@\ell. \)
- A variable defined (directly or indirectly) as a return value at a callsite \( \ell_{cs} : r = f(...) [\mu(z)] [\chi(z)]; \)
  - 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 \( f(...) \) \( [\mu(z)] \) return \_:
  - DPARAM (Parameter Directly Initialised): \( p@\ell_f. \)

![Figure 1. Overview of SVF.](image-url)
Table 1. Rules for building VFG (p, q, r, x ∈ T, o ∈ A, v ∈ V).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Statement (SSA)</th>
<th>Value-Flow Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY</td>
<td>ℓ : p = q</td>
<td>p[ℓ] ← q[ℓ']</td>
</tr>
<tr>
<td>PHI</td>
<td>ℓ : v₁ = φ(v₂, v₃)</td>
<td>v₁[ℓ] ← v₁[ℓ'], v₂[ℓ] ← v₂[ℓ'']</td>
</tr>
<tr>
<td>LOAD</td>
<td>ℓ : p = q[μ(o)]</td>
<td>p[ℓ] ← o[ℓ']</td>
</tr>
<tr>
<td>STORE</td>
<td>ℓ : o₂ = q[μ(o)]</td>
<td>o₂[ℓ] ← q[ℓ']</td>
</tr>
<tr>
<td>CALL</td>
<td>ℓ₁ = f[ℓ₂,...,ℓₙ]μ(o₁)</td>
<td>q[ℓ₁] ← p[ℓ₁, q[ℓ₁] = x[ℓ₁] = 2</td>
</tr>
</tbody>
</table>


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-taking variable. Table 1 lists the rules used.

For a COPY assignment between two top-level pointers ℓ : 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[ℓ] ← q[ℓ'] directly. We do the same in the other rules.

For a PHI statement ℓ : v₁ = φ(v₂, v₃) at a control-flow joint point, the value-flows are connected from the defs v₁[ℓ'] and v₂[ℓ'][v₃[ℓ']] of the old SSA instances v₁ and v₂ of variable v ∈ V to the def v₃[ℓ] of the new instance v₃.

A LOAD ℓ : p = q[μ(o)] annotated with μ(o) indicates that o may be used indirectly via q. Thus, the indirect value-flow of o is connected from its def o[ℓ'] to p[ℓ]. Similarly, a STORE ℓ : o₂ = q annotated with o₂ = φ(o₃) represents the fact that either the value of o is completely overwritten by the value from q[ℓ'] (a strong update) or the old value from its previous def o₁[ℓ'] must also be 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-taking variable o. A direct value-flow edge q[ℓ] ← p[ℓ₁] is connected from the def (at ℓ₁) of an actual parameter p at a callsite ℓₙ to its corresponding formal parameter q in a callee f (for DPara). Similarly, r[ℓ₂] ← x[ℓ₃] captures the direct value-flow of x (with its def at ℓ₂) returned from f to its caller at ℓₙ (for DRet).

Handling the indirect value-flows for address-taking variables is conceptually the same as handling the direct value-flows for top-level variables. Given μ(o₁) annotated at callsite ℓₙ, we know that o may be used in a callee procedure f. As a result, o₃ = φ(o₄) annotated at the entry of f can be understood as an implicit formal parameter receiving o. Thus, an indirect value-flow o₃[ℓₙ] ← o₁[ℓ₄] is connected, with the def of o₁ at ℓ₄ (for IPARA). Likewise, o₂[ℓ₆] ← o₄[ℓ₆] captures the indirect value-flow of o returned from f its caller at ℓ₉, with the def of o₄ at ℓ₉ (for DRet).

As is standard for supporting context-sensitive analysis, the call and return edges at a callsite cs are labelled with (c₁ and c₃), 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 (source) to (sink) in Figure 2(c). Andersen’s pointer analysis is first performed to determine that q points-to an object o ∈ A. Then χ functions are added at the entries of main and foo and the store at o to represent the indirect defs of o, and μ added at the callsite at o and the load at o to represent the indirect uses of o.

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

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-
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, SELF3S [44], performs selective flow-sensitive pointer analysis on partitioned program regions based on pre-computed value-flow information. SELF3S 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 analysis algorithms can be found at https://github.com/unswe-corg/FTABen.

A working demonstration of SVF in a virtual machine can be found at https://github.com/unswe-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


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


[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.


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


