SymPas: Symbolic Program Slicing

Ying-Zhou Zhang, Senior Member, CCF, Member, ACM, IEEE

College of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing 210023, China
E-mail: zhangyz@njupt.edu.cn

Received June 5, 2019; accepted January 22, 2020.

Abstract Program slicing is a technique for simplifying programs by focusing on selected aspects of their behavior. Current mainstream static slicing methods operate on dependence graph PDG (program dependence graph) or SDG (system dependence graph), but these friendly graph representations may be a bit expensive for some users. In this paper we attempt to study a light-weight approach of static program slicing, called Symbolic Program Slicing (SymPas), which works as a dataflow analysis on LLVM (low-level virtual machine). In our SymPas approach, slices are stored in symbolic forms, not in procedures being re-analyzed (cf. procedure summaries). Instead of re-analyzing a procedure multiple times to find its slices for each calling context, we calculate a single symbolic slice which can be instantiated at call sites avoiding re-analysis; SymPas is implemented with LLVM to perform slicing on LLVM intermediate representation (IR). For comparison, we systematically adapt IFDS (interprocedural finite distributive subset) analysis and the SDG-based slicing method (SDG-IFDS) to statically slice IR programs. Evaluated on open-source and benchmark programs, our backward SymPas shows a factor-of-6 reduction in time cost and a factor-of-4 reduction in space cost, compared with backward SDG-IFDS, thus being more efficient. In addition, the result shows that after studying slices from 66 programs, ranging up to 336,800 IR instructions in size, SymPas is highly size-scalable.

Keywords data flow analysis, instruction dependency table, low-level virtual machine (LLVM), procedure symbolic slice, program slicing

1 Introduction and Motivation

1.1 Introduction

Program slicing is an effective technique for narrowing the focus on the relevant parts of a program. The basic idea of program slicing is to remove irrelevant statements from source codes while preserving the semantics of the program such that at some program point and/or variable, referred to as a slicing criterion, the variable produces the same value as its original program. A slicing criterion is a pair \((p, V)\), where \(p\) is a program point and \(V\) is a subset of program variables. Program slicing has been widely used in many software activities including software testing and debugging, measurement, re-engineering, program analysis and comprehension, and so on [2-7]. Program slicing can be classified into static slicing and dynamic slicing according to whether they only use static information or dynamic execution information for a specific program input. A static slice does not consider any particular execution, i.e., it works for any possible input data. Program slicing can also be classified into backward slicing and forward slicing according to the traversal direction from the slicing criterion. A backward slice consists of all statements of the program that have some effect on the slicing criterion, whereas a forward slice contains those statements of the program that are affected by the slicing criterion. Backward slicing assists a developer to locate the parts of the program which contain a bug. Forward slicing can be used to predict the parts of a program that will be affected by a modification. In addition, program slicing can be classified into executable slicing and non-executable slicing according to whether a slice of program \(p\) is an executable program whose semantics is a “subset” of the semantics of \(p\) [8]. A significant portion of the non-
executable code is used to support the program hierarchies. In this paper, we focus on a static non-executable and backward slicing method, although it can be used to compute forward slices.

1.2 Motivation

1.2.1 Dependency Structure Representation

The original program slicing method was expressed as a sequence of data flow analysis problems\cite{1}, called Weiser algorithm for short. An alternative approach was based on the reachability problem over dependence graphs such as program dependence graphs (PDGs) for single-procedural programs or system dependence graphs (SDGs) for multi-procedural programs\cite{9-11}, called as PDG/SDG-based algorithms, which are current mainstream slicing methods. In general, program slicing algorithms should select a data structure along with CFG (control flow graph) to store dependencies between its statements/instructions\cite{6}. In the Weiser algorithm\cite{1}, this data structure is the dataflow sets of relevant variables and relevant statements for a slicing criterion, and it is PDG/SDG in graph reachability based slicing methods. The PDG/SDG data structure makes explicit both the data and control dependencies for each operation in a program\cite{6}. But the construction of dependence graphs such as SDG requires a large amount of effort and suffers from blow-up. Moreover, the SDG structure does not answer directly whether there is a dependency between two nodes (statements/instructions). It requires the whole SDG graph to be traversed twice for answering each dependency query.

For example, in an SDG, if we want to query whether a node is dependent on another one (backward slicing), or whether a node influences another one (forward slicing), we need to traverse the SDG graph in two phases, instead of just checking the node reachability between those two nodes. What is more, in each traverse phase we should carefully select different types of edges according to SDG-based algorithms\cite{9}. These may be sophisticated for some users. In fact, most users prefer intuitive data structures such as the instruction dependency table (IDT) with 0/1 values, where 1 indicates the presence of dependency and 0 the absence of dependency. This structure of dependency table can directly give the answer of the dependency query between two instructions. In brief, the list of values in rows corresponds to backward slice result, and the list of values in columns to forward slice result, which are very intuitive from the table. In addition, we do not need to load the whole IDT table for each dependency query.

Of course, IDT (instruction dependency tables) can be computed from SDG, but it requires large extra overhead of computations, i.e., $O(n^2)$, where $n$ is the number of instructions. Can we directly choose above IDT as the data structure along with CFG to store dependencies in slicing algorithms? This is the target of our work.

1.2.2 Calling-Context Problem

SDG-based slicing algorithms have the absolute advantage over the Weiser algorithm because of the calling-context problem\cite{9}. The well-known disadvantage of the Weiser algorithm is that it cannot address the calling-context problem\cite{9} or realizable-path reachability problem\cite{10}, e.g., the example in Fig.1, there is a data flow that does not match. In general, a calling-context problem occurs when a computation descends into a procedure $B$ that is called from a procedure $A$, and the computation will ascend to all procedures that call $B$, not only $A$. This corresponds to execution paths which enter $B$ from $A$ and exit $B$ to a different procedure $A'$; for example, the unrealizable path $x_1 \rightarrow y_1 \rightarrow y_2 \rightarrow z_2$ or $x_2 \rightarrow y_1 \rightarrow y_2 \rightarrow z_1$ in Fig.1. As these execution paths are infeasible, taking them into consideration results in inaccurate slices. Thus the Weiser algorithm may produce less precise slices than the SDG-based algorithms.

SDG-based slicing algorithms can solve the calling-context problem using context-sensitive techniques. The fastest known SDG-based algorithm is based on the interprocedural finite distributive subset (IFDS) analysis\cite{10}, which is an efficient, context-sensitive and flow-sensitive dataflow analysis technique for problems
that satisfy its restrictions. The IFDS-based interprocedural slicing algorithm\textsuperscript{[10, 12]} (the RHS algorithm for short) has been proven to be asymptotically faster than the well-known original SDG-based algorithm given by Horwitz \textit{et al.}\textsuperscript{[9]} (the HRB algorithm for short). Although the extended IFDS algorithm in \textsuperscript{[13]} constructs the exploded supergraph on demand, these reachability algorithms require the whole exploded supergraph to be constructed ahead of time.

Is it necessary to use these heavy-weight interprocedural analysis methods used in SDG-based algorithms to solve the calling-context problem? In fact, the precise interprocedural analysis methods such as IFDS seem a bit expensive for some SSA-form (static single assignment)\textsuperscript{[14]} program languages such as LLVM IR (Low-Level Virtual Machine, Intermediate Representation)\textsuperscript{[15]}, where IR variables can only be assigned once. As LLVM follows a load/store architecture, values are transferred between memory and registers via explicit load or store operations. As usual, SSA form enables most of the benefits from flow-sensitive analysis to be gained from a simple flow-insensitive analysis. Therefore, we try to explore a light-weight analysis method to address calling-context problem in static IR slicing.

\textit{Slice Target Language}. We select LLVM IR as the target language of program slicing, not only because it is suitable for rapid flow-sensitive analysis as mentioned above, but also because IR uses a relatively modest number of instructions (about 50) to describe a program, which is a convenient analysis target. In addition, LLVM provides an extensive infrastructure for program analysis and transformations (which preserve semantics), thus helping us to study IR slicing.

In fact, there are three reasons why we choose LLVM as our analysis infrastructure and its IR as the slice target language. 1) LLVM can help us indirectly slice multiple front-end program languages in the future. LLVM\textsuperscript{1} is a popular and modern compiler framework that aims to support transparent program analysis and optimization for arbitrary programs. Its IR is the common code representation used throughout all phases of the LLVM compilation process. LLVM can be used as a translation of front-end source language, such as C, C++, Object C and Haskell, to LLVM IR (shown in Fig.2) which can then be executed together in a variety of target architectures. In addition, IR slicing results can be easily applied to generate the slices of front-end source languages, by extracting source codes from sliced IRs with source line information in MetaData of each IR instruction. 2) LLVM provides an extensive infrastructure for program analysis and transformations (which preserve semantics), thus allowing us to focus on just studying general IR slicing algorithms. The LLVM project provides large and continuously updated tools for IR-to-IR translations for optimization and static analysis (such as pointer analysis). Therefore, we do not need special consideration of semantic transformations before program slicing done as conditioned program slicing\textsuperscript{[16]} and amorphous program slicing\textsuperscript{[17]}, but directly call the existing LLVM optimization tools. 3) LLVM is suitable for rapid flow-sensitive analysis. LLVM IR is in SSA\textsuperscript{[14]} form, where IR variables can only be assigned once. As LLVM follows a load/store architecture, values are transferred

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{LLVM compiler infrastructure.}
\end{figure}

\textsuperscript{1}http://llvm.org/docs/LangRef.html, Sept. 2019.
between memory and registers via explicit load or store operations. As usual, SSA form enables most of the benefits from flow-sensitive analysis to be gained from a simple flow-insensitive analysis. In addition, LLVM IR uses a relatively modest number of instructions (about 50) to describe a program, and is fairly similar to a RISC architecture so that it is a convenient analysis target without the details of a programming language’s AST (abstract syntax tree). After source files are compiled to LLVM IR (often in *bitcode* files), we can load them and analyze the instruction stream directly.

### 1.2.3 Contributions

In summary, in this paper, we study a light-weight static slicing method, called symbolic program slicing (SymPas), where instruction-dependency structure is the instruction dependency table (IDT), and program slices are stored symbolically rather than in procedure being re-analysed (cf. procedure summaries). We use symbolic slice of procedure parameters to propagate data dependence of parameters in a similar way to parameter passing, thus helping us to solve calling-context problem.

We make the following main contributions.

- We propose a light-weight and context-sensitive static slicing approach, symbolic program slicing (SymPas).
- We implement the novel slicing approach SymPas for LLVM IR, which is publicly available\(^\text{\ref{github}}\), and conduct empirical evaluations on both open-source and benchmark programs. Our results show that the SymPas approach is both efficient and size-scalable.
- For comparison, we implement two other IR slicers, MWeiser and SDG-IFDS, using the Weiser algorithm and the SDG-based algorithm with IFDS, respectively. To the best of our knowledge, this is the first adaptation of the SDG- and IFDS-based approach to statically slicing IR programs. Based on these IR slicers (MWeiser, SDG-IFDS and SymPas), we collect their final slice results for some open-source and benchmark programs.

The rest of the paper is organized as follows. In Section 2, we introduce the data structure of IR-instruction dependency. Based on this structure, we present the backward symbolic slicing algorithm for single-procedural IR programs. In Section 3, we discuss in detail the symbolic slice for parameter dependencies and the method of backward symbolic slicing for multi-procedural IR programs. We introduce forward symbolic slicing of IR programs in Section 4. The implementation and the complexity analysis of our symbolic slicing algorithm are given in Section 5, where we implement two other IR slicers by existing slicing methods to ease comparison with SymPas. In Section 6, we compare our symbolic slicing method with related slicing methods. We, in Section 7, conclude this paper with the discussion about some applications and limitations of SymPas.

### 2 Intra-Procedural Symbolic IR Slicing

#### 2.1 Syntax of LLVM IR

LLVM IR represents a program as a collection of function definitions (shown in Fig.3), each containing a set of basic blocks with an explicit control-flow graph. Each basic block is a sequence of LLVM instructions, ending in exactly one terminator instruction, which explicitly specifies its successor basic blocks\(^{[15]}\). LLVM requires that values start with a prefix: global values (functions, global variables) begin with the `@` character, and local values (register names, types) begin with the `%` character. In addition, named values are represented as a string of characters with their prefix, for example, `@sum` and `@main`. Unnamed values, which are never stored, are represented as numeric values with their prefix, for example, `%10` and `@3`. In Fig.3, “phi” instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Then the non-constant variable assigned by “phi” instruction may have multiple values, and we call “phi” instruction multi-valued. Similarly, “select” instruction is also multi-valued.

In LLVM, virtual registers (local variables) are kept in SSA form, making the def-use chains explicit, i.e., each virtual register is assigned a value in exactly one instruction, and a register used each time is dominated by its definition. Memory locations (such as C arrays and structures) are kept in non-SSA form, making things very simple. Programs transfer values between registers and memory only via RISC-like load and store operations using typed pointers.

#### 2.2 Intraprocedural Symbolic Slicing Algorithm

**Slicing Criterion.** In this subsection, we will show how to compute backward static slices of IR programs. For simplicity, we only consider end slicing for a single variable, i.e., the slicing criterion is \( (p, v) \), where \( v \) is

\(^\text{\ref{github}}\) https://github.com/zhangyz/llvm-slicing, Sept. 2019.
the variable of interest, and $p$ the program end point. One can easily generalize this to a set of points and a set of variables at each point by taking the union of the individual slices [5].

**Instruction Dependency.** In order to get a slice from an IR program, we first compute the control and data dependences between its instructions. We choose $CD(i)$ to denote $i$’s control-dependence instructions; for LLVM IR, we then have $CD(i) \subseteq \text{INFL}(j)$, where $\text{INFL}(j)$ means the range of influence of a branch statement $j$, which can be defined as the set of statements that are dependent on $j$. For LLVM IR, it is the range of influence from a branch instruction, such as “switch”, “indirectbr” or “br” in Fig. 3, which are often used to transfer control flow to one of several different basic blocks in the current function.

The IR CFG makes the control dependences of IR instructions explicit, thereby we will focus on data dependences. As mentioned in Section 1, related information is used to annotate CFGs at each node. In this paper, we try to use intuitive data structure, i.e., IDT (instruction dependency table), to represent the dependences between instructions. In order to avoid the sparse matrix of IDT, we introduce instruction backward-dependence set (IDS), $L(i)$, to store the dependency information for each CFG node $i$, i.e., all instructions/values that may influence $i$’s execution/computation. $L(i)$ can be obtained from (1) as follows, with the initialization $L^0(i) = \{\}$.

$$L^{k+1}(i) \equiv L^k(i) \cup \bigcup_{j \in CD(i)} L^k(j) \cup \bigcup_{x \in \text{REF}(i)} S^k(x), \quad (1)$$

where $k \geq 0$ denotes the number of computation iterations, corresponding to the level of indirect effect; $\text{REF}(i)$ returns the set of variables (LLVM non-constant values) referenced (used) in $i$. In (1), $S(x)$ represents the current backward-static-slice result of $x$ before analyzing $i$. Its computation equation is as follows.

$$S^{k+1}(x) = \begin{cases} L^{k+1}(i), & \text{if } x \in \text{DEF}(i) \text{ and } x \text{ is single-valued,} \\ S^k(x) \cup L^{k+1}(i), & \text{if } x \in \text{DEF}(i) \text{ and } x \text{ is multi-valued,} \\ S^k(x), & \text{otherwise,} \end{cases} \quad (2)$$

where $\text{DEF}(i)$ returns the set of variables (LLVM non-constant values) defined or modified in $i$. For LLVM IR, the non-constant value may be multi-valued such as “phi” or “select” instruction. Thereby for each possible value of multi-value, we extend its slice at the definition node.

**Procedure Symbolic Slice.** Besides the introduction of the IDT structure, another highlight of our work is that we use some symbolic parameters, say $l_x, l_y, \ldots$, to give the initial static slices of the corresponding formal parameters or global variables on entry to a procedure, as shown in (3) below. Here the formal parameters of a procedure $P$ and the global variables used in $P$ are noted as $\text{FRML}(P)$ and $\text{GLOB}(P)$, respectively.

$$S^0(x) = \begin{cases} \{l_x\}, & \text{if } x \in \text{FRML}(P) \cup \text{GLOB}(P), \\ \emptyset, & \text{otherwise.} \end{cases} \quad (3)$$
Formally, a symbolic slice can be represented textually as a lambda abstraction “\( \lambda x:S(x) \)” where \( x \) is a symbolic parameter. The symbolic slicing uses a symbolic parameter (e.g., \( l_x \)) for each global variable or a formal parameter (say, \( x \)) of a procedure to initialize \( x \)’s slice (noted as \( S^0(\text{x}) \) here), i.e., \( S^0(\text{x}) = \{l_x\} \), in the first slice calculation of the procedure. The symbol \( l_x \), referred to as the symbolic parameter of \( x \), is just like a placeholder to be substituted by a dependence set. The slice result of \( x \), \( S(x) \), may subsume some symbolic parameters, for example \( l_x \) and \( l_y \), i.e., \( l_x \subseteq S(\text{x}) \) and \( l_y \subseteq S(\text{x}) \). We then represent this symbolic slice textually as a lambda abstraction “\( \lambda l_xl_y:S(x) \)”.

The initialization approach with symbolic parameters is very useful to pass and obtain the dependencies of procedure parameters in Section 3. In order to facilitate interprocedural analysis in Section 3, here we introduce a table \( T_P \), called procedure symbolic slice, to store a copy of the final symbolic-slice table \( S \) of a procedure \( P \), which combines symbolic slices of named values at exit (termination) instructions (such as “ret”, “unreachable” and “resume”) of the \( P \) procedure. Thus we have:

\[
T_P(x) \equiv S(x),
\]

for each IR named value \( x \) in \( P \).

**Symbolic Slicing Algorithm.** According to (1)–(4) above, an intraprocedural IR slice can be computed by the algorithm IntraIRSlice (shown in Algorithm 1). Concretely, (1) corresponds to lines 9–11 in the algorithm IntraIRSlice, where we compute at each CFG node \( L(i) \) to store instruction dependency information. Lines 13–17, 3–5 and 19–21 implement (2), (3) and (4), respectively.

The key idea of the algorithm IntraIRSlice can be briefly stated as follows: we first use symbolic parameters to give the initial static slices of the corresponding formal parameters or global variables on entry to a procedure (lines 3–5) by (3). At each CFG node, we then compute instruction backward dependences as \( l' \) (line 10) by (1), with storing \( l' \) in the node/instruction dependency table \( L \) (line 11). Now, \( l' \) can be used to extract IR slices. Therefore, if the node analyzed is a memory-modify/definition node, we use \( l' \) to update or extend the IR slice of variables (lines 13–17, according to (2)). The algorithm continues in this manner until the slice table is stable (line 6), i.e., the computation reaches its fixpoint.
A static slice includes the statements that possibly affect the variable in the slicing criterion. Therefore, for capturing these possible instructions, at each CFG join node (which has at least two predecessors), we merge the slice tables of these predecessors as the current slice table, as shown in line 8. After the algorithm finishes, the resulting slice table includes static slices for all variables (LLVM name values) of an IR program. Note that the algorithm produces another output, the IDT table D, which is what we need as mentioned in Section 1, and it is useful for computing forward slices (see Section 4).

Line 11 means that after node i is analyzed/computed, its IDS set D(i) must be updated by the above set l′ (line 10), thus adding corresponding control and data dependencies. In fact, only if instruction i analyzed is a branch IR instruction such as “switch” “indirectbr”, or “br” (excluding unconditional branch), which is often used to transfer control to one of several different basic blocks in the current function, set l′ must be added to the IDS set of each instruction of these basic blocks for propagating control dependences. Therefore, in order to improve the efficiency of Algorithm 1 for backward slicing, we update D(i) with set l′ if i is a branch instruction. In other words, for backward symbolic slicing, we only need to compute l′ by (1), at branch, value-insert or memory-modify instructions rather than at all instructions, thus saving implementation time and space.

Example. Fig. 4 shows a simple C program (adapted from the example program in [9]). As an example of IR slice computation in Algorithm 1, we consider the procedure &add (shaded) in the sample IR program in Fig. 5, which is compiled (by LLVM Clang) from the simple C program shown in Fig. 4.

Table 1 shows the detailed data necessary to calculate the IR symbolic slice table for all single variables (LLVM name value) in &add. Working forwards from the CFG node 24, whose corresponding instruction is the entry of the procedure &add, noted as i24, from (1)–(3), we have

\[ R(i_{24}) = \{%a\}, \text{DEF}(i_{24}) = \{\%1\}, \text{CD}(i_{24}) = \emptyset, \]
\[ L(i_{24}) = L^0(i_{24}) \cup S(\%a) = \{24, b_{2a}\}, \]
\[ S(\%1) = L(i_{24}) = \{24, b_{2a}\}. \]

The data computation of node 25 is similar to that of node 24. At node 26 with i26, since \( R(i_{26}) = \{\%1, \%2\}, \) \( \text{DEF}(i_{26}) = \{\%3\} \) and \( \text{CD}(i_{26}) = \emptyset, \) its IDS set \( L(i_{26}) = L^0(i_{26}) \cup S(\%1) \cup S(\%2) = \{24, 25, 26, b_{2a}, b_{2b}\}, \) and the symbolic slice of its definition variable (non-constant IR value \%3), \( S(\%3) = L(i_{26}) = \{24, 25, 26, b_{2a}, b_{2b}\}. \)

Node 27 (corresponding to instruction i27) is the “store” instruction used to write value %3 to value %a, thereby the slice of %a needs to update with \( L(i_{27}), \) i.e.,

\[ S(\%a) = L(i_{27}) = \{24, 25, 26, 27, b_{2a}, b_{2b}\}. \]

No change of the slice table S occurs at node 28. After finishing the computation/analysis of the last node 28 (the exit instruction of &add), we obtain the final symbolic slice table S for all named values (%a and %b) as follows:

\[ S(\%a) = \{24, 25, 26, 27, b_{2a}, b_{2b}\}, \]
\[ S(\%b) = \emptyset. \]

Here, if we just want to get the final static slice of &add, we simply remove all symbolic parameters in the symbolic slice table, i.e.,

\[ S(\%a) = \{24, 25, 26, 27\}, \]
\[ S(\%b) = \emptyset. \]

The column T in Table 1, which is just a copy of the final symbolic slice table S of named values in &add, is useful for inter-procedural dataflow analysis in Section 3.

Slice Accuracy. For the correctness of IntraIR-Slice in Algorithm 1, we have the following Lemmas

```c
int main()
{
    int n, i, sum;
    void A(int *x, int *y);
    printf("Enter a positive number: ");
    scanf("%d", &n);
    sum = 0;
    i = 1;
    while (i <= n)
    {
        A(&sum, &i);
    }
    printf("sum = %d
", sum);
    printf("i = %d
", i);
    return 0;
}
```

Fig. 4. Simple C source program (adapted from [9]) to be compiled.
define i32 @main ( ) {
    entry:
    1) %n = allocato %32
    2) %i = allocato %32
    3) %sum = allocato %32
    4) %c = call i32 @printf(…)
    5) %s = call i32 @scanf(…, %32, %7)
    6) store %32 0, %32 * %sum
    7) store %32 1, %32 * %i
    8) br label %while.cond
while.while.cond:
    9) %3 = phi %i2 [{%i, pre, %while.body}, [%entry]]
    10) %4 = load %32 * %i
    11) %5 = icmp sge i32 %3, %4
    12) br i1 %5, label %while.body, label %while.end
while.body:
    13) call void @A(i32 * %i, i32 * %v)  
    14) %pre = load i32 * %v
    15) br label %while.body, %while.end
while.end:
    16) %6 = load i32 * %sum
    17) %7 = call i32 @printf(…, %i32 %6)
    18) %8 = load i32 %i
    19) %9 = call i32 @printf(…, %i32 %8)
    20) ret i32 0   
}

define void @A(i32 * %x, i32 * %y)  
{
    entry:
    21) call void @add (i32 * %x, i32 * %y)
    22) call void @inc(i32 * %y)
    23) ret void
}

define void @add(i32 * %a, i32 * %b)  
{
    entry:
    24) %1 = load i32 * %a
    25) %2 = load i32 * %b
    26) %3 = add %a %1, %2
    27) store %i32 %3, %i32 %a
    28) ret void
}

define void @inc (i32 * %y)  
{
    entry:
    29) %tmp = allocato %32
    30) store %i32 1, %i32 %tmp
    31) call void @add(i32 * %y, i32 * %tmp)
    32) ret void
}

Fig.5. Sample IR program, compiled from the C program in Fig.4 by LLVM Clang using LLVM opt passes -basicaa and -gvn.

Table 1. Example IR Slice Computation for the Procedure @add in Fig.5

| Node | Instruction | REF(i) | DEF(i) | CD(i) | $L_i^{k+1}(v)$ | $S$ | $T$ |
|------|-------------|--------|--------|------|----------------|-----|-----|
| 24   | %1 = load i32 * %a | {%a}   | {%a}   | {}   | {24} U S(%a)  | $S^{k+1}(%a) = \{24, l_3\}$ | -   |
| 25   | %2 = load i32 * %b | {%b}   | {%b}   | {}   | {25} U S(%b)  | $S^{k+1}(%b) = \{25, l_2\}$ | -   |
| 26   | %3 = add nsw i32 %1, %2 | {%1, %2} | {%1, %2} | {%3} | {24, 25, 26} U S(%a) U S0(%b) | $S^{k+1}(%a) = \{24, 25, 26, l_3, l_2\}$ | -   |
| 27   | store i32 %3, i32 * %a | {%3}   | {%a}   | {}   | {24, 25, 26, 27} U S0(%a)%S0(%b) | $S^{k+1}(%a) = \{24, 25, 26, l_3, l_2, l_1\}$ | -   |
| 28   | return void | {}     | {}     | {}   | {28} | $S(%a) = \{24, 25, 26, 27\}$ S(%b) = \{24, 25, 26, 27\} | $T_{basicaa}(%a) = \{24, 25, 26, 27, l_3, l_2\}$, $T_{basicaa}(%b) = \{l_2\}$ | -   |

1 and 2, Theorem 1, Corollaries 1 and 2. In fact, the terms $\bigcup_{E \in CD(i)} L_i^k(j)$ and $\bigcup_{E \in REF(i)} S^k(E)$ in (1) can accurately capture control dependences and data dependences, respectively.

Lemma 1. For two instruction vertices, $w$ and $v$, in a PDG, if there is a direct (control or data dependence) edge from $w$ to $v$, noted as $w \rightarrow^{c,v} v$, then we have $L_i^k(w) \subseteq L_i^{k+1}(v)$ for some $k$.

Lemma 2. For two instruction vertices, $w$ and $v$, in a PDG, if $w$ can reach $v$, noted as $w \rightarrow^{*} v$, then we have $L_i^k(w) \subseteq L_i(v)$ for some $k$.

Theorem 2. For two PDG vertices (excluding entry vertices), $w$ and $v$, are reachable, i.e., $w \rightarrow^{*} v$, iff (if and only if), $w \in L(v)$.

Corollary 1. For two PDG vertices (instructions), $w$ and $v$, there is a direct edge from $w$ to $v$, i.e., $w \rightarrow^{c,v} v$, iff (if and only if) $L_i^k(w) \subseteq L_i^{k+1}(v)$ for some $k$.

Corollary 2. The accuracy of the intra-procedural symbolic slicing algorithm (Algorithm 1) is same with that of PDG-based slicing algorithms.

Context sensitivity and pointer analysis are two of the most important issues affecting the final slice precision.[P]. Context-sensitive symbolic slice algorithms, which will be discussed in Subsection 3.3, can address the calling-context problem. The introduction of a pointer will lead to aliasing problems (i.e., multiple variables accessing the same memory location), thereby we need some aliasing analysis (or pointer analysis) to obtain the corresponding data dependence information. Fortunately, LLVM has various optimizations. Some of these aid slicing such as “-basicaa” (basic alias analysis) and “-gvn” (global value numbering), and thus
allows us to just focus on context-sensitive IR-slicing analysis in Section 3. For simplicity, we directly use two LLVMs’ alias analysis passes “-basicaa” and “-gvn” to implement identification, and treat both arrays and all members of a structure as a single entity. The “-basicaa” pass is an aggressive local analysis that knows many important facts such as distinct globals, stack allocations, heap allocations, different fields of a structure. The “-gvn” pass performs global value numbering to eliminate fully and partially redundant instructions. It can eliminate local loads, dead loads and non-local loads (requiring “phi” node insertion).

3 Inter-Procedural Symbolic IR Slicing

3.1 Motivating Example

As mentioned in Section 1, the Weiser algorithm may include non-feasible paths within the program control flow, i.e., it cannot address the calling-context problem. Thus it loses precision when slicing multi-procedural programs [6]. As an example, we consider again the IR program shown in Fig.5. From the Weiser algorithm [1], a static backward slice of this IR program with respect to the slicing criterion \( h_{32} \); \( %z_{i} \) includes all the instructions of the program except the strikethrough and shaded ones shown in Fig.6. However, it is clear that the double-line-through instructions (6 and 21) included in the slice cannot influence the slicing criterion. This less precise problem is due to the fact that the Weiser algorithm does not keep information about the calling context when it traverses procedures up and down. This problem can be solved by interprocedural context-sensitive slicing. In this subsection, we will extend our intraprocedural IR slice algorithm \textit{IntraIRSlice} in Algorithm 1 to produce context-sensitive symbolic slices for multi-procedural IR programs.

3.2 Procedure Dependency Analysis

\textbf{Procedure Call Dependence.} As shown in Section 2, \textit{IntraIRSlice} (Algorithm 1) uses the IDT table \( L \) with \((1)\) to store instruction dependencies. In this subsection, we will show how to extract procedure dependencies from these procedure symbolic slices computed in Algorithm 1, thus extending intraprocedural symbolic IR slicing to interprocedural one.

The dependence relations between procedures can be divided into two categories:

1) control and data dependences caused by the procedure call and parameter passing, and
2) data dependences due to reading and writing global variables.

\begin{quote}
\textbf{define} i32 @main ( ) { 
\textbf{entry:} 
1) \textbf{alloca} i32 
2) \textbf{alloca} i32 
3) \textbf{alloca} i32 
4) \textbf{call} i32 @printf(\ldots) 
5) \%1 = \textbf{load} i32* %2 
6) \textbf{store} i32 %3, i32* %4 
7) \textbf{br} \textbf{label} %while.cond 
\textbf{while.cond:} 
9) \%3 = \textbf{phi} i32 [[\%pre, \%while.body],[1,\%entry]] 
10) \%4 = \textbf{load} i32* %5 
11) \%5 = \textbf{icmp} sge i32 \%3, \%4 
12) \textbf{br} i1 \%5, label %while.body, label %while.end 
\textbf{while.body:} 
13) \textbf{call} void @A (i32* \%sum, i32* \%i) 
14) \%pre = \textbf{load} i32* %1 
15) \textbf{br} \textbf{label} %while.end 
\textbf{while.end:} 
16) \%1 = \textbf{load} i32* %2 
17) \textbf{call} i32 @printf(\ldots,i32 \%1) 
18) \%2 = \textbf{load} i32* %4 
19) \textbf{call} i32 @printf(\ldots,i32 \%2) 
20) \textbf{ret} i32 0 
}\end{quote}

\begin{quote}
\textbf{define} void @A(i32* \%x, i32* \%y) { 
\textbf{entry:} 
21) \textbf{call} void @add (i32* \%x, i32* \%y) 
22) \textbf{call} void @inc(i32* \%y) 
23) \textbf{ret} void 
}\end{quote}

\begin{quote}
\textbf{define} void @add(i32* \%a,i32* \%b) { 
\textbf{entry:} 
24) \%1 = \textbf{load} i32* %2 
25) \%2 = \textbf{load} i32* %4 
26) \%3 = \textbf{add} nsw i32 \%1, %2 
27) \textbf{store} i32 \%3,i32* %a 
28) \textbf{ret} void 
}\end{quote}

\begin{quote}
\textbf{define} void @inc(i32* %z) { 
\textbf{entry:} 
29) \textbf{alloca} i32 
30) \textbf{store} i32 1,i32* %tmp 
31) \textbf{call} void @add(i32* %z, i32* %tmp) 
32) \textbf{ret} void 
}\end{quote}

Fig.6. IR slice result w.r.t. \( (32, %z) \) for the IR program in Fig.5.
A procedure call in essence transfers the program control at call site \( i \) to another procedure (callee), thereby callee’s control dependence can be computed as follows.

\[
L^{k+1}(i) \equiv L^k(i) \cup \bigcup_{j \in CD(i)} L^k(j).
\] (5)

In general, procedure call dependences can be seen as control dependences \([9]\), thereby (5) corresponds to the control-dependence part in (1). Next we try to capture data dependencies among procedures from parameter passing and global variable influence.

**Transitive Data Dependence.** Procedure calls pass some actual parameters to the formal parameters of callee. Here there is a matching process between formal and actual parameters. In general, programming language designers understand three forms of formal parameters: \( \text{in} \), \( \text{out} \), and \( \text{in-out} \), corresponding to call-by-value, call-by-result and call-by-value-result, respectively. We can consider global variables as forms of \( \text{in} \)-values, \( \text{call-by-result} \) and \( \text{call-by-value-result} \), respectively. We can consider global parameters having the same name as the corresponding parameter variables are transitive data dependences.

Furthermore, the introduction of procedure symbolic slices, which is the highlight of our work mentioned in Section 2, can help to obtain parameter-dependent information, thus facilitating the calculation of the dependencies among parameters. For a parameter \( x \) in a procedure \( P \), its parameter-dependent set, noted as \( \text{SUMM}_P(x) \), can be directly obtained as follows.

\[
\text{SUMM}_P(x) \equiv \{ y | y \in T_P(x), y \in \text{FRML}(P) \cup \text{GLOB}(P) \}. (8)
\]

The above \( \text{SUMM}_P \), which is similar to a kind of summary information in SDG-based slicing methods \([9]\), represents transitive interprocedural data dependences. (8) means that for all symbolic parameters (introduced in Section 2) in \( x \)'s symbolic slice \( T_P(x) \), their corresponding parameter variables are transitive data dependences of \( x \). For example, since both \( l_a \) and \( l_b \) are in set \( T_{\text{add}}(\%a) \) shown in Table 1, from (8), we know that parameter \( \%a \) is transitively data-dependent on both \( \%a \) and \( \%b \), i.e., \( \text{SUMM}_{\text{add}}(\%a) = \{ \%a, \%b \} \). In other words, we can directly obtain the parameter summary information from procedure symbolic slices, without extra interprocedural analysis methods such as IFDS.

**Procedure Data Dependence.** In addition to the benefit of obtaining summary information directly from procedure symbolic slices, we now show that procedure symbolic slices can be used to compute data dependences among procedures, without the calling-context problem. There are two cases of interprocedural data dependence at call sites:

Case 1: callees’ data dependence on callers via parameter passing, and

Case 2: callees’ influence on callers from their \( \text{out} \) or \( \text{in-out} \) parameters.

For case 1, we just substitute/backfill symbolic parameters in procedure symbolic slices with corresponding parameter data dependences. In concrete, for a call site (CFG node/instruction) \( i \) in a caller \( P \), which calls a callee \( Q \) with symbolic slice table \( T_Q \), we substitute/backfill and compute as follows the temporary table \( T'_{Q,i} \), from \( T_Q \).

**Parameter Dependency.** Furthermore, the introduction of procedure symbolic slices, which is the highlight of our work mentioned in Section 2, can help to obtain parameter-dependent information, thus facilitating the calculation of the dependencies among parameters.

For a parameter \( x \) in a procedure \( P \), its parameter-dependent set, noted as \( \text{SUMM}_P(x) \), can be directly obtained as follows.
where \( \text{SPEC}_i(l_y) \) specifies how to pass down the corresponding data dependence at call sites to the symbolic parameter \( l_y \), which can be obtained as follows from the current slice table \( S \).

\[
\text{SPEC}_i^k(l_y) = \begin{cases} 
S^k(y), & \text{if } y \in \text{GLOB}(Q), \\
\cup_{z \in \text{REF}(x)} S^k(z), & \text{if } y \in \text{FRML}(Q) \text{ and } x \text{ is } y \text{'s actual parameter.}
\end{cases}
\]

(10)

In (9), the case \( \{x \in \text{OUT}(Q) \wedge \text{SUMM}_Q(x) \neq \emptyset\} \) means that we should substitute each symbolic parameter (say, \( l_y \)) in \( T_Q(x) \) with \( \text{SPEC}_i(l_y) \), since \( x \) has transitive data dependences and may be defined or modified in \( Q \). If \( x \) is a global variable and is not modified in \( Q \), caller \( P \) will not influence \( x \) in \( Q \).

Now above \( T_{Q,i}^{k} \) in (9) includes extra control (by (5)) and data dependencies because of \( P \) calling once \( Q \) at \( i \), on the basis of \( Q \)’s procedural slice result \( T_Q \). Therefore, we can use \( T_{Q,i}^{k} \) to update or extend the current slice table \( S \) of \( Q \), as a result of caller \( P \) influencing callee \( Q \). In summary, for a global value (say, \( x \)) modified in \( Q \), i.e., \( x \in \text{OUT}(Q) \), its current symbolic slice \( S(x) \) needs to be updated with \( T_{Q,i}^{k} \); otherwise, we just extend the symbolic slices of named values in \( Q \), as shown in the first two cases of (11).

\[
S^{k+1}(x) = \begin{cases} 
T_{Q,i}(x), & \text{if } x \in \text{OUT}(Q) \text{ and } x \in \text{GLOB}(Q), \\
S^k(x) \cup T_{Q,i}(x), & \text{if } x \in T_Q \text{ and } x \notin \text{GLOB}(Q), \\
T_{Q,i}(y), & \text{if } x \text{ is an actual parameter in } i, y \text{ is } x \text{'s formal parameter, and } y \in \text{OUT}(Q), \\
S^k(x), & \text{otherwise.}
\end{cases}
\]

(11)

For case 2 above, we need to consider the influence from the callee \( Q \) to the caller \( P \) at their call sites by those parameters that may be modified in \( Q \). With above \( T_{Q,i}^{k} \), the third case of (11) can capture these callees’ influences on callers.

As described above, because of symbolic parameters in procedure symbolic slices, data dependences of parameters among procedures can be propagated in time in a similar way as parameter passing. This timely propagation of data dependence and the strict correspondence of symbolic parameters can effectively guarantee data flowing across realizable paths, thus avoiding the calling-context problem.

3.3 Interprocedural Symbolic Slicing Algorithm

**Context-Sensitive IR Slicing Algorithm.** According to the above interprocedural dependency analysis ((9)–(11)) with procedure symbolic slice, we present a context-sensitive IR slicing algorithm (InterIRSlice) in Algorithm 2 where call nodes/instructions can be handled with the help of procedure symbolic slices of callees, and other instructions can be analyzed in the same way as the intraprocedural IR slicing method in Algorithm 1.

Lines 15–19 in Algorithm 2 generate specific dependences \( \text{SPEC}_i \) (by (10)) at call instruction \( i \) for all symbolic parameters in symbolic slices of callees. Lines 21–26 and 27–32 implement (9) and (11), respectively. In addition, lines 11–13 handle the external call procedure such as “\texttt{llvm.memcpy}” as a definition instruction. Here, all callee procedures must have been analyzed before using Algorithm 2 to analyze their caller procedures. Therefore the procedures must be analyzed in a particular order, for example, strongly connected components (SCC) in procedural call graphs. The analysis starts at the leaves and propagates summary information up the call graphs. For recursive procedures, SCCs are analyzed until a fixed-point is reached. Each SCC is processed as a unit, thereby we can analyze SCCs in parallel. Therefore, Algorithm 2 supports parallelised implementation.

**Example.** As an example, we consider again the symbolic slicing of the sample IR program shown in Fig.5. Based on its procedure call graph, we choose an analysis order (one of the topological sorts) of procedures, i.e., \( \texttt{@add} \rightarrow \texttt{@inc} \rightarrow \texttt{@a} \rightarrow \texttt{@main} \). Since there is no call instruction in the \( \texttt{@add} \) procedure, we can easily compute its symbolic slice table, \( T_{\text{@add}} \), by the \texttt{IntraIRSlice} algorithm as shown in Table 1 in Section 2. From \( T_{\text{@add}} \), the dependency information among the formal parameters of \( \texttt{@add} \) can be obtained by (6)–(8) as follows:

\[
\begin{align*}
\text{OUT}(\texttt{@add}) &= \{\%a\}, \\
\text{IN}(\texttt{@add}) &= \{\%b\}, \\
\text{SUMM}_{\texttt{@add}}(\%a) &= \{\%a, \%b\}, \\
\text{SUMM}_{\texttt{@add}}(\%b) &= \{\%b\}.
\end{align*}
\]
According to above analysis order, we next analyze the \texttt{inc} procedure, which has a call node (the 31st instruction in Fig. 5, noted as $i_{31}$) to callee \texttt{add}. Since neither $\%a$ or $\%b$ in \texttt{add} is a global variable, from line 19 in Algorithm 2 or (10), we have:

\begin{align*}
SPEC_{31}(l_{3a}) &= S(\%z) = \{l_{2a}\}, \\
SPEC_{31}(l_{3b}) &= S(\%tmp) = \{30\}.
\end{align*}

Through lines 21–26 in the \texttt{InterIRSlice} algorithm, the temporary slice table, $T'_{\text{add},31}$, including interprocedural dependencies, needs to be computed as follows.

\begin{align*}
T'_{\text{add},31}(\%a) &= L(i_{31}) \cup (S(\%z) \cup S(\%tmp)) \cup T_{\text{add}}(\%a) - \{l_{3a}, l_{3b}\} \\
&= \{31\} \cup \{(l_{3a}) \cup \{30\} \cup \{24, 25, 26, 27, l_{3a}, l_{3b}\} - \\
&= \{24, 25, 26, 27, 30, 31, l_{3a}, l_{3b}\}, \\
T'_{\text{add},31}(\%b) &= L(i_{31}) = \{31\}.
\end{align*}

Because of \texttt{inc} calling \texttt{add} at $i_{31}$, we extend the symbolic slices of named values (i.e., $\%a$ and $\%b$) in \texttt{add} by using $T'_{\text{add},31}$ above, as line 29 in Algorithm 2. In addition, we use $T'_{\text{add},31}$ to update the symbolic slices of possible modified values (i.e., $\%z$) at $i_{31}$, as line 32 in Algorithm 2.

The algorithm continues in above manner. After analysing all procedures, we can obtain procedure-symbolic-slice table $T$ and the final static slice result shown in Table 2. For example, the static backward slice with respect to the slicing criterion $i_{32};\%z$ includes all the instructions of the IR program except the double-line-through, strikethrough and shaded ones shown in Fig.6.

**Slice Accuracy.** As shown above, in our algorithm \texttt{InterIRSlice}, using procedure symbolic slices helps to obtain accurate parameter-dependent information, so as to facilitate the calculation of the dependencies among parameters ((8)) and the depen-
from w to v, noted as w

formal parameter y of P. Then, there is a summary edge

for connecting PDGs.

P
procedure Q, which calls procedure P, let w be an actual-

respectively

parameter-out edges and summary edges in an SDG

3{5 and Theorem 2. In fact, (5), (10), (11) and (8)

2, (5), and (8){(10), we have the following Lemmas

InterIRSlice, based on Theorem 1, Lemmas 1 and 2,

and (2)—(10), we have the following Lemmas

3—5 and Theorem 2. In fact, (5), (10), (11) and (8)

correspond to call (control) edges, parameter-in edges,

parameter-out edges and summary edges in an SDG

respectively[9], which are four addition kinds of edges

for connecting PDGs.

Lemma 3. In an SDG, for a call-site vertex u in

procedure Q, which calls procedure P, let w be an actual-
in vertex at u, corresponding to formal parameter x of

P, let v be an actual-out vertex at u, corresponding to

formal parameter y of P. Then, there is a summary edge

from w to v, noted as w →sv v, iff x ∈ SUMM_{P}(y).

Lemma 4. For two SDG vertices, w and v, if there is

a direct edge from w to v, i.e., w → v, then we have

Lk (w) ⊆ Lk+1(v) for some k.

Lemma 5. For two instruction vertices, w and v,
in an SDG, if w can reach v in procedure Q by following
control edges, flow edges and summary edges, noted as

w →c, f,sv v, then w ∈ L(v) and Lk (w) ⊆ L(v) for some

k.

Theorem 2. For two instruction nodes, w and v,
in an SDG G, let G/v be the backward slice of v via

SDG-based slicing algorithms[9]. Then, w ∈ G/v iff w

∈ L(v).

In short, our procedure symbolic slices with sym-

bolic parameters can be used to directly answer some

queries such as: whether a parameter variable is mod-

ified/defined in a procedure, and whether two para-

meters have transitive data dependence. In addition,

without extra interprocedural analysis techniques, pro-

cedure symbolic slices can also be used as follows to ob-

tain the set of parameter variables referenced (GREF)

or modified (GMOD) [9] by a procedure.

GMOD(P) = \{ x | T_P(x) ≠ \{l_x\},

x ∈ FRML(P)∪ GLOB(P)\},

GREF(P) = \{ y | (x, S) ∈ T_P, l_y ∈ S_x,

y ∈ FRML(P)∪GLOB(P)\}.

In [9], GMOD and GREF are additionally computed to

eliminate actual-out and formal-out vertices for para-

meters that will never be modified, resulting in more

precise slices. For example,

GMOD(@add) = \{%a\}, GREF(@add) = \{%a, %b\}.

4 Forward Symbolic IR Slicing

For certain cases, we are interested in all those state-

ments that may be influenced by the slicing criterion,

i.e., forward slicing, which can be used to analyze mod-

ification propagation. For example, the forward slice

with respect to (5, %a) for the sample IR program shown

in Fig.5 contains all instructions except those labeled

with 1–4, 6–8, 20, 23, 28 and 32. For simplicity, just

as for backward slicing, we here only consider forward

slicing for a single variable, i.e., the slicing criterion

(p, v), where v is the variable of interest, and p the pro-

gram start point.

The forward slicing algorithm based on our sym-

Table 2. Symbolic Slice Table T and Final Slice Table S of the IR Program in Fig.5

| Proc. | Var. | Symbolic Slice T | SUMM & SPEC | Final Slice Result S |
|-------|------|------------------|-------------|----------------------|
| %a    | 24, 25, 26, 27, %i, %b | SUMM_%a = \{%a, %b\}, SPEC_{21}(%i) = S^4(%i), \quad SPEC_{21}(%b) = S^4(%b) | S%(a) = \{5, 6, 7, 9, 10, 11, 12, 13, 14, 21, 22, 24, 25, 26, 27, 30, 31\} |
| %b    | 24, 25, 26, 27, %i, %b | SUMM_%b = \{%b\}, \quad SPEC_{21}(%i) = S^4(%i), \quad SPEC_{21}(%b) = S^4(%b) | S%(b) = \{5, 7, 9, 10, 11, 12, 13, 14, 21, 22, 24, 25, 26, 27, 30, 31\} |
| %c    | 24, 25, 26, 27, 30, 31, %i | SUMM_%c = \{%c\}, \quad SPEC_{22}(%i) = S^4(%i) | S%(c) = \{5, 7, 9, 10, 11, 12, 13, 14, 22, 24, 25, 26, 27, 30, 31\} |
| %tmp  | (30) | SUMM_%tmp = \{%tmp\} | S%(tmp) = \{5, 7, 9, 10, 11, 12, 13, 14, 22, 24, 25, 26, 27, 30, 31\} |
| %d    | 21, 24, 25, 26, 27, %i, %b | SUMM_%d = \{%d\}, \quad SPEC_{13}(%i) = S^4(%i) | S%(d) = \{5, 6, 7, 9, 10, 11, 12, 13, 14, 21, 22, 24, 25, 26, 27, 30, 31\} |
| %e    | 22, 24, 25, 26, 27, 30, 31, %i | SUMM_%e = \{%e\}, \quad SPEC_{13}(%i) = S^4(%i) | S%(e) = \{5, 7, 9, 10, 11, 12, 13, 14, 22, 24, 25, 26, 27, 30, 31\} |
| %main | 5 | SUMM_%main = \{5\} | S%(main) = \{5\} |
| %f    | 5, 7, 9, 10, 11, 12, 13, 14, 22, 24, 25, 26, 27, 30, 31 | SUMM_%f = \{%f\}, \quad SPEC_{13}(%i) = S^4(%i) | S%(f) = \{5, 6, 7, 9, 10, 11, 12, 13, 14, 22, 24, 25, 26, 27, 30, 31\} |
bolic slicing method, ForwardIRSlice, is shown in Algorithm 3. In fact, the IDS set of node \(i\), \(L(i)\), obtained from the InterIRSlice algorithm (Algorithm 2), includes all instructions/values that may influence \(i\)'s execution. On the other hand, \(i\) must be included in forward slices of all variables (LLVM name values) referenced in \(L(i)\). This is achieved in lines 7 and 8 of the ForwardIRSlice algorithm. As shown in Sections 2 and 3, the IDT table \(L\) includes control and data dependencies between procedures, and thus Algorithm 3 can be used to compute forward IR slices for interprocedural procedures.

5 Implementations

5.1 Implementation of Symbolic Slicing

Based on our symbolic IR-slicing algorithms (Algorithm 1 to Algorithm 3), we implemented a symbolic program slicer based on LLVM, named as SymPas, in Haskell\(^\circ\) using the library llvm-analysis\(^\circ\) for analyzing LLVM bitcodes. All experiments were performed on a desktop PC with four processors (2.5 GHz Intel i5), 8 GB of memory and 64-bit Ubuntu GNU/Linux 12.04.

Data Structures in SymPas. In practice, in order to obtain good performance of set operations on tables such as the IDT table \(L\) and the slice table \(S\) in our algorithms, we choose data structures such as Haskell types IntSet, IntMap and Map\(^\circ\), i.e., \(L::\) IntMap IntSet and \(S::\) Map String IntSet.

The data structures IntSet and IntMap are based on big-Endian Patricia trees\(^\circ\) and use some bit-level coding tricks\(^\circ\) so that they perform well on binary operations like union and intersection. Many of their operations (e.g., lookup and insert) have a worst-case complexity of \(O(\min(m, W))\), where \(m\) is the number of elements, and \(W\) is the number of bits in an Int (32 or 64). The Map data structure is based on size-balanced binary trees\(^\circ\). Its operations such as lookup and insert have a worst-case complexity of \(O(\log k)\), where \(k\) is the number of its keys. What is more, \(S\) can be embedded in \(L\) at those instructions (such as “alloca”) in Fig.3 where slice-criterion variables are initially defined, since the IDSs (instruction backward-dependence sets) of these instructions only contain their own.

In addition, as an implementation trick, we set each symbolic parameter (e.g., \(l_a\)) in the symbolic slice table as a negative number of its unique ID. For example, the symbolic backward slice of \(\%a\) at exit site of \$add in Table 1, \(T_{\text{\$\text{add}}}(\%a)\) or \{24, 25, 26, 27, \(l_a\), \(l_b\)\}, can be represented/stored as \{-100, -101, 24, 25, 26, 27\}, assuming that the unique ID numbers of \%a and \%b are 100 and 101, respectively.

Complexity of SymPas. A bound on the cost of our symbolic slicing can be expressed according to the data structure of tables \(L\) and \(S\), which are effective maps as mentioned above. For the IntraIRSlice algorithm (Algorithm 1), its complexity analysis is restricted to the intermediate set \(l'\) (line 10 of Algorithm 1) by (1). In the case of IR program, the terms \(\cup_{j\epsilon\text{\text{CD}(i)}}L^j(j)\) and \(\cup_{j\epsilon\text{\text{REF}(i)}}S^j(x)\) in (1) will cost \(O(b \times n \times \min(n, W))\) and \(O(r \times n \times \log v)\) time respectively, thereby \(l'\) can be determined in \(O(b \times n \times \min(n, W) + r \times n \times \log v)\) or \(O(b \times n + r \times n \times \log v)\) time, where \(b\) is the number of branch instructions such as “switch”, “indirectbr” or “br”, \(r\) the largest number of (named/unnamed) variables referred in any IR instruction, \(\nu\) the number of single variables in the program, and \(n\) the number of nodes (vertices) in the CFG, which is almost equal to the number of instructions in the IR program. Thus

\[\text{Complexity} = O(b \times n \times \min(n, W) + r \times n \times \log v)\]

\[\text{Complexity} = O(b \times n + r \times n \times \log v)\]

\(\circ\)http://www.haskell.org, Jan. 2021.
\(\circ\)http://hackage.haskell.org/package/llvm-analysis, Jan. 2021.
\(\circ\)http://hackage.haskell.org/package/containers, Jan. 2021.
the intraprocedural symbolic slicing algorithm can determine a static slice in \(O((b + r) \times n \times c)\) time, where \(c\) is the number of edges in CFG. If \(S\) is embedded in \(L\), the time cost of \(\text{IntraIRSlice}\) is \(O((b + r) \times n \times c)\). Below we assume that \(S\) is embedded in \(L\).

For the \(\text{InterIRSlice}\) algorithm (Algorithm 2), at procedural call instructions, besides the cost of \(l\', \) SymPas needs additional time cost of computing non-in parameter dependencies for backfilling the symbolic parameters in the symbolic slice table, i.e., \(SPEC\), (lines 15–19 in Algorithm 2) by (10). Its time cost is \(O(r \times n \times x)\), where \(x = \text{globals} + \text{params}, \) \text{globals} is the number of global variables in the program, and \text{params} the largest number of formal parameters in any procedure. Therefore each call instruction will cost additional time \(O(r \times n \times x^2)\) by (9), and the worst-case running time of \(\text{InterIRSlice}\) can be bounded by

\[
O((b_{\text{max}} + r) \times p \times n_{\text{max}} \times c_{\text{max}} + r \times n_{\text{max}} \times c \times x^2),
\]

where \(c\) is the number of call instructions, \(p\) is the number of procedures in the program, and \(n_{\text{max}}\) and \(e_{\text{max}}\) are the largest number of nodes and edges in any procedure's CFG, respectively.

For forward symbolic slicing in the \(\text{ForwardIRSlice}\) algorithm (Algorithm 3), we need extra time cost \(O(r \times p^2 \times n_{\text{max}}^2)\) on Algorithm 2 to extract forward slice table.

To analyze the space complexity of the algorithms, we focus on the construction tables \(L\) and \(T\). We need space \(O(p^2 \times n_{\text{max}}^2)\) and \(O(p \times n_{\text{max}} \times n_{\text{max}})\) to hold the IDT table \(L\) and the procedure-symbolic-slice table \(T\), respectively. Therefore, the total space cost is \(O(p^2 \times n_{\text{max}}^2)\).

5.2 IR Slicers with Existing Slicing Methods

In order to facilitate comparison of our symbolic IR slicing (SymPas) and existing slicing methods in same running environments such as implementation language, compilation configuration and optimization, we implement two other static slicers for LLVM IR: MWeiser (using the Weiser slicing algorithm [1]) and SDG-IFDS (using the SDG-based slicing algorithm [9] with IFDS [10, 13] analysis).

The IFDS algorithm [10] finds the meet-over-all-valid-paths solution (instead of the meet-over-all-paths solution), through transforming an interprocedural dataflow-analysis problem into a special kind of graph-reachability problem (reachability along interprocedurally realizable paths). The algorithm takes the interprocedural control flow graph (ICFG, also called supergraph in [10], e.g., Fig.7) and transforms it into an exploded supergraph. In order to explicitly track calling-context, the IFDS algorithm uses call strings [21], which are built by labeling the call and return for call-site \(c\) with unique terminal symbols (e.g., “()” and “”) . The call-strings method can ensure that the string of symbols generated by a graph traversal belongs to some context-free language, e.g., each “(“ must be matched with “)”). For example, in the ICFG (shown in Fig.7) of the sample IR program in Fig.5, the call edges (the red dashed line) and the return edges (the blue dotted line) are labelled by the unique symbols that begin with “(" and ")", respectively.

Naem et al. [13] gave some practical extensions to the IFDS algorithm for larger programs and they are suitable for programs in SSA form [14]. The input to the extended algorithm is a dataflow function that, given an ICFG (supergraph) node \(n\), computes all of the edges leaving \(n\). In [13], this dataflow function is split into four separate flow transfer functions. Based on this extended IFDS algorithm, we can compute static IR slices by explicitly defining these flow functions. Therefore we implement the IR slicer SDG-IFDS by providing its corresponding flow transfer functions in LLVM.

As mentioned in Section 1, program slicing algorithms should select a data structure along with CFGs to store dependencies between statements/instructions. In data-flow equations based slicing methods such as the Weiser algorithm, this data structure is the data-flow set such as def-use chains. It is PDG/SDG or ICFG supergraph, in graph reachability based slicing methods such as the HRB algorithm and the RHS algorithm. In our symbolic slicing method (SymPas), we choose the intuitive instruction-dependence table (IDT) as the data structure with CFG for storing instruction dependencies. Most of the existing slicing algorithms rely on relation graphs such as PDG/SDG.

In Table 3, we compare our symbolic slicing method (SymPas) with three existing slicing methods mentioned above: Weiser, HRB (SDG-based) and RHS (SDG-IFDS-based) algorithms. As mentioned before (esp. in Section 3), SymPas is a context-sensitive slicing method via forward dataflow analysis on LLVM. It uses intuitive data structure IDT to represent instruction dependencies, with procedure slices being stored symbolically for precise interprocedural analysis. Theorems 1 and 2 theoretically state that the accuracy of slicing methods based on SymPas and PDG/SDG (or SDG-IFDS) is essentially the same, although the presentation of the final slices is different.
About the efficiency of these static slicing methods, the total time cost of the SDG-based HRB algorithm \cite{9} is \(O(p \times n_{PDG} \times c_{PDG} + x \times c \times x^3)\), where \(c_{max}\) is the largest number of call sites in any procedure, \(c\) the number of call instructions, \(x\) the number of global variables or formal parameters, \(n_{PDG}\) and \(e_{PDG}\) the largest number of nodes and edges in any procedure’s PDG \cite{2}, respectively. Whereas, the SDG-IFDS based RHS algorithm \cite{10} takes time \(O(p \times n_{PDG} \times e_{PDG} + p \times e_{PDG} \times x + c \times x^2)\), which is asymptotically faster than the HRB algorithm (since generally \(p < c\)) \cite{12}. It has one more term (i.e., \(p \times e_{PDG} \times x\)) than the time cost of our backward SynPas, as shown in Table 3. In addition, \(e_{PDG} = O(n_{PDG}^2)\), \(n_{PDG} = O(n_{max})\). Therefore their space costs are asymptotically the same.

Although the extended IFDS algorithm in \cite{13} constructs the exploded supergraph on demand, the reachability algorithms require the whole exploded supergraph to be constructed ahead of time. In other words, SDG-IFDS based slicing exhaustively computes all nodes reachable along valid paths, instead of answering reachability queries for its reachable subgraph. In contrast, our symbolic slicing SynPas constructs the related tables (such as the procedure symbolic slice table.
5.3 Empirical Evaluation

**Benchmarks.** To evaluate the efficiency of our symbolic slicer SymPas and two other IR slicers implemented above (i.e., SDG-IFDS and MWeiser), we automatically generate IR backward (forward) slicing criteria for single variable, to slice on the last (first) instruction of the main procedure for each global variable, and on the last (first) instruction of each procedure for each local variable allocated/declared in the procedure. We use 14 benchmark programs. Five programs are from the Mälardalen WCET (Worst-Case Execution Time) benchmark [22] including lms, compress, ndes, adpcm, and statemate; two from SPECint 2000 benchmark including 181.mcf and 256.bzip2; five from real open-source programs including time-1.7, gitview from gnuit-4.9.5, pkg-config-0.26, barcode-0.99 and byacc; two programs loop50 and loop100 adapted from the while programs in [5] with 50 and 100 while-loop statements, respectively. The basic statistics of these benchmark programs are listed in Table 4, where LOC denotes the number of lines of front-end source code, \( n \) is the number of IR instructions compiled from front-end source code, and \( b, p \) and \( v \) have the same meaning as in Subsection 5.1. For other information such as their source codes (exclude SPECint), call graphs, SDG graphs, detailed forward/backward slice results, please see our SymPas website [6].

**Research Questions.** We want to evaluate the slicing results of our symbolic slicing tool (SymPas) to determine if accurate slices are produced, and are produced efficiently. For the efficiency of these static slicing tools, here we compare our symbolic slicing method (SymPas) with three existing slicing methods mentioned in this paper: Weiser, HRB (SDG-based) and RHS (SDG-IFDS-based) algorithms. Since SDG-IFDS based methods have been proven to be asymptotically faster than original SDG-based methods, we focus on evaluating slices of three slicer tools: SymPas, SDG-IFDS and MWeiser, by taking into consideration time and space efficiency, as shown in research questions RQ1 and RQ2 below.

### Table 4. Efficiency Evaluation of SymPas

| System | LOC | \( n \) | \( b \) | \( p \) | \( v \) | Backward Slicing | Forward Slicing |
|--------|-----|--------|------|------|------|----------------|---------------|
|        |     |        |      |      |      | SymPas SDG-IFDS MWeiser | SymPas SDG-IFDS MWeiser |
|        |     |        |      |      |      | \( T \) \( S \) | \( T \) \( S \) | \( T \) \( S \) | \( T \) \( S \) |
| lms.c  | 271 | 353    | 15   | 8    | 41   | 0.06 2 | 0.56 5 | 4.05 18 | 1.03 6 | 0.65 5 |
| compress.c | 521 | 494    | 36   | 9    | 57   | 0.43 8 | 5.67 59 | 52.28 72 | 3.70 43 | 6.20 59 |
| ndes.c | 238 | 586    | 26   | 5    | 62   | 0.19 5 | 1.44 22 | 16.51 63 | 1.00 20 | 1.90 21 |
| loop50.c | 364 | 819    | 50   | 1    | 6    | 10.52 59 | 273.69 2064 | 322.12 2858 | 90.74 530 | 262.26 2010 |
| time  | 1395 | 953    | 55   | 6    | 66   | 0.64 15 | 18.51 286 | 144.30 374 | 5.16 143 | 17.99 285 |
| adpcm.c | 875 | 1118   | 37   | 17   | 178  | 0.61 14 | 29.96 69 | 487.33 533 | 7.50 37 | 48.88 69 |
| statemate.c | 1276 | 1271 | 179 | 8 | 106 | 3.16 48 | 74.82 659 | \( \infty \) | 37.91 492 | 72.03 661 |
| loop100.c | 714 | 1619   | 100  | 1    | 6    | 105.20 361 | \( \infty \) | \( \infty \) | 1 310.36 4023 | \( \infty \) |
| 181.mcf | 4820 | 2916   | 179  | 26   | 186  | 2.56 30 | 24.47 74 | \( \infty \) | 41.38 178 | 32.07 79 |
| gitview | 976 | 4980   | 421  | 150  | 431  | 115.90 1206 | 1088.90 1266 | \( \infty \) | \( \infty \) | 1 681.87 1281 |
| 256.bzip2 | 8 014 | 6 258 | 479 | 74 | 421 | 59.37 272 | 972.00 2085 | \( \infty \) | \( \infty \) | 1 255.90 3678 | 1 198.00 2086 |
| pkg-config | 5 191 | 7 411 | 702 | 92 | 589 | 172.39 278 | 1238.50 1560 | \( \infty \) | \( \infty \) | 1 571.68 1566 |
| barcode | 30 472 | 8 326 | 655 | 68 | 491 | 57.47 213 | 758.14 1028 | \( \infty \) | 161.71 1154 | 912.86 1025 |
| byacc | 23 906 | 13 953 | 1141 | 208 | 875 | 304.29 280 | \( \infty \) | \( \infty \) | 1 658.60 1640 | \( \infty \) |
| Average | 5 645 | 3647   | 291  | 48   | 251  | 59.49 199 | 373.89 765 | 171.10 653 | 381.25 995 | 483.87 762 |
| GeoMean | 1 717 | 1 981 | 132 | 16 | 113 | 6.68 56 | 57.34 248 | 65.53 190 | 38.07 239 | 69.07 248 |

Note: LOC: #code-lines, \( n \): #IR-instructions, \( b \): #branches, \( p \): #procedures, \( v \): #variables, \( T \): time (sec.), \( S \): space (MB), \( \infty \): timeout (30 min.), and \( \sim \): N/A.

[6] https://github.com/zhangyz/llvm-slicing, Sept. 2019.
For slicer accuracy, as mentioned before, SymPas is a context-sensitive slicing method via dataflow analysis on LLVM. It uses intuitive data structure IDT to represent instruction dependencies, with procedure slices being stored symbolically for precise interprocedural analysis. Theorems 1 and 2 theoretically state that the accuracy of slicing methods based on SymPas and PDG/SDG (or SDG-IFDS) is essentially the same, although the presentations of the final slices are different. In [23], Binkley et al. observed that from 43 programs (ranging up to 136,000 LOC in size), for the most precise slicer, the average slice contains just under one third of the program. In order to check whether our slicing tool also satisfies this rule of precise slicer, we evaluate the slice size, i.e., the number of IR instructions (here possibly including parameter symbols), which is related to slice quality [23], as shown in research question RQ3 below.

In our evaluation, we intend to answer three key research questions.

1) **RQ1.** For backward slicing, is our SymPas more efficient than SDG-IFDS and MWeiser in terms of time and space cost?

2) **RQ2.** For forward slicing, how about the efficiency of SymPas when compared with SDG-IFDS?

3) **RQ3.** Is SymPas highly size-scalable?

**Experimental Results.** The performance statistics of these three static slicers is shown in Table 4, where programs are sorted by the number of IR instructions (column “n”), including results of backward and forward slicing. For each slicing algorithm, we report its running CPU time (“T” columns) and memory cost (“S” columns). In Table 4, we draw the following conclusions.

1) For the Weiser algorithm, which only supports backward slicing, it always needs much more time and space to calculate the slice table of all single variables, since it requires repeated iterations for computing the slice of each variable, without reusing intermediate results as done in SymPas or SDG-IFDS. In addition, MWeiser analyzes each procedure in an arbitrary order, whereas SymPas and SDG-IFDS traverse each SCC in the call graph bottom-up, which makes it easier to share intermediate values between analyses, thus only traversing the call graph once.

2) For static backward slicing, on average, compared with SDG- and IFDS-based slicer SDG-IFDS, our symbolic slicer SymPas reduces 84.0% in running time and 74.0% in space overhead.

3) For forward slicing, SymPas has slight advantage over SDG-IFDS on average, because it needs extra time cost $O(r \times p^2 \times n_{\text{max}})$ to generate forward slice table from the IDT table in SymPas.

4) For the scalability of slice size (size-scalability), we study SymPas slices from 66 programs, ranging up to 336,859 instructions in size. The results (partly shown in Fig.8) show that the average backward-SymPas slice contained 22.4% of the system, and 25.2% in forward SymPas. As expected, SymPas is highly size-scalable.

In summary, for RQ1, we conclude that our symbolic backward slicing performs better than existing slicing methods such as SDG-based and Weiser slicing. For RQ2, SymPas has slight advantage over SDG-IFDS in forward slicing. For RQ3, SymPas has high slice-size scalability.

In addition, as mentioned in Section 2, in order to increase the speed of backward SymPas, we here update $L(i)$ in Algorithm 1 with set $l'$ if $i$ is a branch instruction, i.e., we only compute $l'$ by (1), at branch, value-insert or memory-modify instructions rather than at all instructions. For forward Sympas, we must compute $l'$.

![Fig.8. High size/instruction scalability of SymPas.](image-url)
by (1) at all instructions as shown in Algorithm 1.

6 Related Work

Weiser\cite{1} first introduced program slicing. His interprocedural slicing algorithm is a simple extension of the intraprocedural one, and is easily implemented as it directly makes use of the intraprocedural slicing algorithm. But it cannot address the calling-context problem. Thus, it may produce imprecise slices.

Subsequently, Hwang et al.\cite{24} proposed an iterative solution for interprocedural static slicing based on replacing recursive calls by instances of the procedure body. This approach does not suffer from the calling-context problem because the expansion of recursive calls does not lead to considering infeasible execution paths. However, Reps\cite{25} showed that for a certain family of recursive programs, this algorithm takes exponential time in the length of the program.

Simultaneously, Horwitz et al.\cite{9} proposed an interprocedural slicing algorithm based on the SDG representation. Their algorithm involves two steps: firstly, it constructs the corresponding SDG with summary edges, which represent transitive dependences due to procedure calls; secondly, it computes slices through two-phase traverses on the SDG. This slicing algorithm can address the calling-context problem, but the construction of SDG may be complex, as mentioned in Section 1. Subsequently, Lakhota\cite{26}, and Livadas and Johns\cite{27} improved upon Horwitz’s algorithm. However, the efficiency of the algorithm is not improved essentially, and the complication of constructing SDG and the deficiency of language flexibility remain in these algorithms.

Recently, Alomari et al.\cite{28} presented a highly efficient, light-weight, forward static slicing approach, with no need to build the complex PDG/SDG, but instead dependence and control information is computed as needed while computing the slice on a variable. Their approach relies on an underlying XML representation of the source code. Their slices are special forward slices, called forward decomposition slices, and each with respect to a variable \( v \) is the union of the static forward slices at the set of statements that define \( v \). To avoid building PDGs/SDGs, Mastroni and Zanardini\cite{29} presented abstract program slicing, a formal general notion of slicing with abstract semantics/interpretation\cite{30}, where properties of data are considered instead of their exact value. Similarly, we\cite{31,32} presented monadic slicing, which is the formal slicing method based on modular monadic semantics\cite{33} of the program. These formal slicing methods require users to know esoteric techniques such as abstract semantics, monads\cite{34} and monad transformers\cite{35}, since they should give complete formal semantic descriptions of a program before computing its slices. For the same purpose of avoiding the whole exploded dependence graphs, in this paper, we propose symbolic slicing, where the intuitive data structure IDT (instruction dependency table) is used to represent instruction dependencies, and program slices are stored in symbolic forms, not in procedures being re-analysed. We use symbolic slice of procedure parameters to propagate data dependence of parameters in a similar way as parameter passing, thus alleviating the calling-context problem.

Binkley et al.\cite{36} mentioned that how to slice programs written in multiple languages is still an open long-standing challenge. To solve this challenge for dynamic slicing, Binkley et al. proposed in\cite{36,37} a language-independent slicing technique, Observation-Based Slicing (ORBS), by deleting program statements, executing the candidate slice with some inputs, and observing the behaviour for a given slicing criterion. ORBS can slice multi-language systems through leveraging the existing build tool-chain. This ORBS method may be extended to multi-language static slicing by the union of dynamic slices for many test cases, i.e., union slices\cite{38}. Using the modern compilation framework LLVM, we attempt in this paper to address this challenge for static slicing, by presenting a new context-sensitive slicing technique, called Symbolic Program Slicing. We are slicing in a language-agnostic manner using LLVM IR, thereby one of the benefits is to being able to address Binkley’s challenge for slicing multi-language systems.

Lisper et al.\cite{39} presented a light-weight interprocedural algorithm for backwark static slicing based on a variant of the SLV (strongly live variables) analysis, which is an alternative data flow analysis for computing data dependencies. Their algorithm is context-sensitive, but only works for programs without (direct or indirect) recursive procedures. In addition, along with the progress of SLV slicing analysis, many new slicing criteria will be generated. In comparison, our SymPas does not have the limitation of working only for non-recursive procedures; it eliminates the need to maintain and generate many slicing criterions.

Srinivasan and Reps\cite{40} extended the instruction-level SDG in\cite{9} into a microcode-level SDG (\( \mu\)-SDG)
for slicing machine codes of Intel IA-32 binaries. We in this paper present a new slicing approach for slicing low-level codes of LLVM IR. There are three existing slicing tools for LLVM IR: Dg\textsuperscript{7}, Giri\textsuperscript{8} and LLVMSlicer\textsuperscript{9}. Dg aims to generate a dependence graph for IR programs, which can be used to compute static IR slices via graph reachability. Dg is now under hard development, especially in collecting interprocedural summary information. Giri is actually a dynamic backward slicing compiler pass in LLVM. LLVMSlicer is based on the Weiser algorithm. The LLVMSlicer code is written for the specific purpose (of checking properties described by finite state machines), with turning off the IR optimizations, thereby it is not flexible enough to be used by others.

7 Discussion and Conclusions

In this paper, we proposed Symbolic Program Slicing (SymPas), which is a light-weight context-sensitive slicing method via forward dataflow analysis on LLVM. SymPas uses intuitive data structure IDT to represent instruction dependencies. It works interprocedurally by storing a form of symbolic slice for precise interprocedural analysis, thus alleviating the calling-context problem. Experiments showed that SymPas is both efficient in running time and space usage, and highly scalable in slice size.

The symbolic slicing algorithms are effectively demand-driven. By using a “lazy” computation strategy for symbolic slices (parameterised), only those symbolic slices directly relevant to the slicing criterion are performed. Instead of constructing the whole PDG/SDG or exploded supergraph ahead of time, our symbolic slicing analyzes one procedure at a time in a call-graph dependence order (starting at the leaf-procedures), and instead of repeatedly slicing a procedure called from multiple call sites, we calculate a procedure symbolic slice which can be just used when needed.

SymPas Applications. Program slicing has two main uses: 1) program comprehension, for which slicing at the source-code level is vital, and 2) program analysis and optimization as part of a compiler. We addressed the latter using LLVM as the vehicle, i.e., program slicing of LLVM IR. In fact, IR slice results could be easily applied to generate the slices of source code, by extracting source codes from sliced IRs with line number information in the metadata of each IR instruction.

SymPas has the following key applications.

1) SymPas can be used to generate the slices of LLVM front-end source languages such as C, by extracting source codes from sliced IRs with source line information in MetaData of each IR instruction.

2) SymPas can help statically slicing multi-language systems, which is still an open long-standing challenge for program slicing\textsuperscript{36}, by combining (with the llvm-link tool) their IR instructions generated from existing compiler tools such as Clang\textsuperscript{10} for C/C++, VMKit\textsuperscript{11} for Java, Numba\textsuperscript{12} for Python, and JXcore\textsuperscript{13} for JavaScript.

3) SymPas can be used for IR optimizations as an analysis pass in the LLVM framework.

4) With SymPas, we can measure and evaluate LLVM projects using program slicing metrics\textsuperscript{41}.

SymPas Limitations. The limitations of SymPas includes:

1) fixed slice criterions, $(p, v)$, where $v$ is a single variable of interest, and $p$ the program end point for backward slicing (or the program start point for forward slicing), and

2) restriction on LLVM front-end compilers. For example, the C fragment “\texttt{a = 3; b = a + 2}” will be compiled by LLVM Clang to the IR instruction “\texttt{store i32 5, i32* %b}”, resulting in no dependency of $b$ and $a$.

Future Work. The future work of SymPas includes:

1) more precise symbolic slices by using more advanced alias analyses,

2) more effective data structures such as OBDD (Ordered Binary Decision Diagram)\textsuperscript{42} for related tables (e.g., $L$ and $S$),

3) comparisons with existing C/C++ slicers such as CodeSurfer\textsuperscript{14} or srcSlice\textsuperscript{28} in terms of power and

\textsuperscript{7}https://github.com/mchalupa/dg, Sept. 2019.
\textsuperscript{8}https://github.com/liuml07/giri, Sept. 2019.
\textsuperscript{9}https://github.com/jirislaby/LLVMSlicer/, Sept. 2019.
\textsuperscript{10}http://clang.llvm.org/, Sept. 2019.
\textsuperscript{11}http://vmkit.llvm.org/, Sept. 2019.
\textsuperscript{12}http://numba.pydata.org/, Sept. 2019.
\textsuperscript{13}http://jxcore.com/, Sept. 2019.
\textsuperscript{14}https://www.grammatech.com/products/codesurfer, Sept. 2019.
scalability,
4) dynamic symbolic slicing algorithms, and
5) extension of SymPas to support slicing object-oriented, distributed and concurrent programs.

Acknowledgments We would like to show our gratitude to Prof. Alan Mycroft of Cambridge University, for their helpful and constructive comments that greatly improved the original version of the manuscript. We thank the reviewers for their careful reading of this manuscript and their constructive comments.

References

[1] Weiser M. Program slicing. IEEE Transaction on Software Engineering, 1984, 16(5): 498-509. DOI: 10.1109/TSE.1984.5010248.

[2] Tip F. A survey of program slicing techniques. Journal of Programming Languages, 1995, 3(3): 121-189.

[3] Gallagher K B, Lyle J R. Using program slicing in software maintenance. IEEE Transactions on Software Engineering, 1991, 17(8): 751-761. DOI: 10.1109/32.83912.

[4] Harman M, Hierons R M. An overview of program slicing. Software Focus, 2001, 2(3): 85-92. DOI: 10.1002/swf.41.

[5] Binkley D, Gallagher K B. Program slicing. Advances in Computers, 1996, 43: 1-50. DOI: 10.1016/S0065-2458(08)60641-5.

[6] Josep S. A vocabulary of program slicing-based techniques. ACM Computing Surveys, 2012, 44(3): Article No. 12. DOI: 10.1145/2187671.2187674.

[7] Binkley D, Harman M. A survey of empirical results on program slicing. Advances in Computers, 2004, 62: 105-178. DOI: 10.1016/S0065-2458(03)60203-6.

[8] Binkley D, Danicic S, Glymothy T, Harman M, Kiss A, Ouabrya L. Formalizing executable dynamic and forward slicing. In Proc. the 4th IEEE International Workshop on Source Code Analysis and Manipulation, September 2004, pp.43-52. DOI: 10.1109/SCAM.2004.13.

[9] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 1990, 12(1): 26-60. DOI: 10.1145/77660.77668.

[10] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In Proc. the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1995, pp.49-61. DOI: 10.1145/199448.199462.

[11] Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment. ACM SIGPLAN Notices, 1984, 19(5): 177-184. DOI: 10.1145/990011.808263.

[12] Reps T, Sagiv M, Horwitz S. Interprocedural dataflow analysis via graph reachability. Technical Report, University of Copenhagen, 1994. https://research.cs.wisc.edu/wpis/papers/diku-tr94-14.pdf, Nov. 2020.

[13] Naeem A, Lhotak O, Rodriguez J. Practicalextensions to the IFDS algorithm. In Proc. the 19th International Conference on Compiler Construction, March 2010, pp.124-144. DOI: 10.1007/978-3-642-11970-5_8.

[14] Cytron R, Ferrante J, Rosen B K, Wegman M N, Zadeck F K. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Prog. Langs. and Systs., 1991, 13(4): 451-490. DOI: 10.1145/115372.115320.

[15] Zhao J, Zdancewic S, Nagarakatte S, Martin M. Formalizing the LLVM intermediate representation for verified program transformation. In Proc. the 39th Annual ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, January 2012, pp.427-440. DOI: 10.1145/2103656.2103709.

[16] Canfora G, Cimilite A, De Luca A. Conditioned program slicing. Information and Software Technology, 1998, 40(11/12): 595-607. DOI: 10.1016/S0950-5849(98)00086-X.

[17] Harman M, Binkley D, Danicic S. Amorphous program slicing. Journal of Systems and Software, 2003, 68(1): 45-64. DOI: 10.1016/S0164-1212(02)00135-8.

[18] Morrison D R. PATRICIA-Practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM, 1968, 15(4): 514-534. DOI: 10.1145/321479.321481.

[19] Okaasli C, Gill A. Fast mergeable integer maps. In Proc. the 1998 ACM SIGPLAN Workshop on Machine Learning, September 1998, pp.77-86.

[20] Adams S. Efficient sets: A balancing act. Journal of Functional Programming, 1993, 3(4): 553-562. DOI: 10.1017/S0956796800000885.

[21] Sharir M, Pnueli A. Two approaches to interprocedural data flow analysis. In Program Flow Analysis: Theory and Applications, Muchnick S S, Jones N D (eds.), Prentice-Hall Inc., 1981, pp.189-233.

[22] Gustafsson J, Betts A, Ermedahl A, Lisper B. The mälardalen WCET benchmarks — Past, present and future. In Proc. the 10th Workshop Worst-Case Execution Time Analysis, July 2010, pp.137-147. DOI: 10.4230/OASIcs.WCET.2010.136.

[23] Binkley D, Gold N, Harman M. An empirical study of static program slice size. ACM Transactions on Software Engineering and Methodology, 2007, 16(2): Article No. 8. DOI: 10.1145/1217295.1217297.

[24] Hwang J C, Du M W, Chou C R. Finding program slices for recursive procedures. In Proc. the 19th Annual International Computer Software & Applications Conference, Oct. 1988, pp.220-227. DOI: 10.1109/CMSANC.1988.17176.

[25] Reps T. On the sequential nature of interprocedural program-analysis problems. Acta Informatica, 1996, 33(8): 739-757. DOI: 10.1007/s002360050068.

[26] Lakhotia A. Improved interprocedural slicing algorithm. Technical Report, University of Southwestern Louisiana, 1992. http://citesecrst.ist.psu.edu/viewdoc/download?doi=10.1.1.28.3466&rep=rep1&type=pdf, Nov. 2020.

[27] Livadas P, Johnson T. An optimal algorithm for the construction of the system dependence graph. Information Sciences, 2000, 125(1/2/3/4): 99-131. DOI: 10.1016/S0164-1212(00)000255(99)00030-4.

[28] Alomari H, Collard M, Maletic J, Alhindawi N, Meqdadi O. srcSlice: Very efficient and scalable forward static slicing. In Journal of Software: Evolution and Process, 2014, 26(11): 951-961. DOI: 10.1002/smr.1651.
[29] Mastroeni I, Zanardini D. Abstract program slicing: An abstract interpretation-based approach to program slicing. ACM Trans. Comput. Logic, 2017, 18(1): Article No. 7. DOI: 10.1145/3029052.

[30] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. the 4th ACM Symp. Principles of Programming Languages, January 1977, pp.238-252. DOI: 10.1145/512050.512973.

[31] Zhang Y. A novel formal approach to program slicing. Science in China, Ser. F: Info. Sci., 2007, 50(5): 657-670. DOI: 10.1007/s11432-007-0061-2.

[32] Zhang Y. A precise monadic dynamic slicing method. Knowledge-Based Systems, 2017, 115: 40-54. DOI: 10.1016/j.knosys.2016.10.013.

[33] Liang S. Modular monadic semantics and compilation [Ph.D. Thesis]. University of Yale, 1998.

[34] Wadler P. Comprehending monads. In Proc. the 1990 ACM Conf. LISP and Functional Programming, May 1990, pp.61-78. DOI: 10.1145/91556.91592.

[35] Liang S, Hudak P, Jones M. Monad transformers and modular interpreters. In Proc. the 22nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, January 1995, pp.333-343. DOI: 10.1145/199448.199528.

[36] Binkley D, Gold N, Harman M, Islam S, Krinke J, Yoo S. ORBS: Language-independent program slicing. In Proc. the 22nd Int. Symp. Foundations of Software Engineering, November 2014, pp.109-120. DOI: 10.1145/2635868.2635893.

[37] Binkley D, Gold N, Islam S, Krinke J, Yoo S. A comparison of tree- and line-oriented observational slicing. Empir. Software Eng., 2019, 24(1): 3077-3113. DOI: 10.1007/s10664-018-9675-9.

[38] Beszédes A, Faragó C, Szabó Z, Csirik J, Gyimóthy T. Union slices for program maintenance. In Proc. the 2002 IEEE Int. Conference on Software Maintenance, Oct. 2002, pp.12-21. DOI: 10.1109/ICSM.2002.1167743.

[39] Lisper B, Masud A, Khanfar H. Static backward demand-driven slicing. In Proc. the 2015 Workshop on Partial Evaluation and Program Manipulation, January 2015, pp.115-126. DOI: 10.1145/2678015.2682538.

[40] Srinivasan V, Reps T. An improved algorithm for slicing machine code. In Proc. the 2016 ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications, October 2016, pp.378-393. DOI: 10.1145/2983990.2984003.

[41] Meyers T, Binkley D. An empirical study of slice-based cohesion and coupling metrics. ACM Transactions on Software Methodology, 2007, 17(1): Article No. 2. DOI: 10.1145/1314493.1314495.

[42] Zhang X, Gupta R, Zhang Y. Precise dynamic slicing algorithms. In Proc. the 25th Int. Conf. Software Engineering, May 2003, pp.319-329.

Ying-Zhou Zhang received his Ph.D. degree in computer science and engineering from University of Southeast, Nanjing, in 2005, and his M.S. degree in mathematics from University of Hohai, Nanjing, in 2002. He is currently working as a professor in the School of Computer Science and Technology, Nanjing University of Posts and Communications, Nanjing. He worked with the Hong Kong Polytechnic University, Hong Kong, in 2011, and as a visiting scholar with the University of Cambridge between 2012 and 2013. He has published more than 80 research papers. His research interests include formal methods, software analysis and program slicing, software reliability and security, service computing and functional programming. He is a senior member of CCF, and a member of ACM and IEEE.