Slicing Machine Code *

Venkatesh Srinivasan
University of Wisconsin–Madison, USA
venk@cs.wisc.edu

Thomas Reps
University of Wisconsin–Madison
and GrammaTech, Inc., USA
reps@cs.wisc.edu

Abstract

Machine-code slicing is an important primitive for building binary analysis and rewriting tools, such as taint trackers, fault localizers, and partial evaluators. However, it is not easy to create a machine-code slicer that exhibits a high level of precision. Moreover, the problem of creating such a tool is compounded by the fact that a small amount of local imprecision can be amplified via cascade effects.

Most instructions in instruction sets such as Intel's IA-32 and ARM are multi-assignments: they have several inputs and several outputs (registers, flags, and memory locations). This aspect of the instruction set introduces a granularity issue during slicing: there are often instructions at which we would like the slice to include only a subset of the instruction's multiple assignments, whereas the slice is forced to include the entire instruction. Consequently, the slice computed by state-of-the-art tools is very imprecise, often including essentially the entire program. We present an algorithm to slice machine code more accurately. To counter the granularity issue, our algorithm attempts to include in the slice only the subset of assignments in an instruction's semantics that is relevant to the slicing criterion. Our experiments on IA-32 binaries of FreeBSD utilities show that, in comparison to slices computed by a state-of-the-art tool, our algorithm reduces the number of instructions in backward slices by 36%, and in forward slices by 82%.

1. Introduction

One of the most useful primitives in program analysis is *slicing* [14, 28]. A slice consists of the set of program points that affect (or are affected by) a given program point p and a subset of the variables at p.[1] Slicing has many applications, and is used extensively in program analysis and software-engineering tools (e.g., see pages 64 and 65 in [3]). Binary analysis and rewriting has received an increasing amount of attention from the academic community in the last decade (e.g., see references in [24, §7], [4, §1], [9, §1], [11, §7]), leading to the widespread development and use of binary analysis and rewriting tools. Improvements in machine-code[2] slicing could significantly increase the precision and/or performance of several existing tools, such as partial evaluators [25], taint trackers [8], and fault localizers [29]. Moreover, a machine-code slicer could be used as a black box to build new binary analysis and rewriting tools.

State-of-the-art machine-code-analysis tools [5, 9] recover a *system dependence graph (SDG)* from a binary, and use an existing source-code slicing algorithm [20] to perform slicing on the recovered SDG. (The SDG is an intermediate representation used for slicing; see §2.2) However, the computed slices are extremely imprecise, often including the entire binary. Each node in a recovered SDG is an instruction, and instructions in most Instruction Set Architectures (ISAs) such as IA-32 and ARM are *multi-assignments*: they have several inputs and several outputs (e.g., registers, flags, and memory locations). The multi-assignment nature of instructions introduces a granularity issue during slicing: although we would like the slice to include only a subset of an instruction's assignments, the slice is forced to include the entire instruction. This granularity issue can have a *cascade effect*: irrelevant assignments included at an instruction can cause irrelevant instructions to be included in the slice, and such irrelevant instructions can cause even more irrelevant instructions to be included in the slice, and so on. Consequently, straightforward usage of source-code slicing algorithms at the machine-code level yields imprecise slices.

In this paper, we present an algorithm to perform more precise context-sensitive interprocedural machine-code slicing. Our algorithm is specifically tailored for ISAs and other low-level code that have multi-assignment instructions. Our algorithm works on SDGs recovered by existing tools, and is parameterized by the ISA of the instructions in the binary.

Our improved slicing algorithm should have many potential benefits. More precise machine-code slicing could be used to improve the precision of other existing analyses that work on machine code. For example, more precise forward slicing could improve *binding-time analysis* (BTA) in a machine-code partial evaluator [25]. More precise slicing could also be used to reduce the overhead of taint trackers [8] by excluding from consideration portions of the binary that are not affected by taint sources. Beyond improving existing tools, more precise slicers created by our technique could be used as black boxes for the development of new binary analysis and rewriting tools (e.g., tools for software security, fault localization, program understanding, etc.). Our more precise backward-slicing algorithm could be used in conjunction with a machine-code synthesizer [26] to extract an executable component from a binary, e.g., a word-count program from the `wc` utility. (See §9.) One could construct more accurate dependence models for libraries that lack source code by slicing the library binary. A machine-code slicer is

* Supported, in part, by a gift from Rajiv and Ritu Batra; by DARPA under cooperative agreement HR0011-12-2-0012; by NSF under grant CCF-0904371; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPASTAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from theWisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

[1] In the literature, program point p and the variable set are called the *slicing criterion* [28]. In this paper, when the program point is clear from context, we typically will specify just the variable set and refer to it as the "slicing criterion."

[2] We use the term "machine code" to refer generically to low-level code, and do not distinguish between actual machine-code bits/bytes and assembly code to which it is disassembled.

a useful tool to have when a slicer for a specific source language is unavailable.

We have implemented our algorithm in a tool, called X, which slices Intel IA-32 binaries. X uses the SDG recovered by an existing tool, CodeSurfer/x86 [5]. Given a node n_G and a set of variables used at (defined at) n_G as the slicing criterion, X "grows" the backward (forward) slice from n_G by iterating over the SDG nodes using a worklist. For each worklist item n, X includes only the "sub-portion" of the instruction in n that is relevant to the slicing criterion, and puts only the predecessors (successors) of n that are relevant to the included sub-portion in the worklist. When the worklist iteration terminates, only sub-portions of instructions that are transitively relevant to the slicing criterion are included in the slice. The improved precision of X is due to its method for identifying the sub-portion of the instruction in n that is relevant to the slicing criterion, which addresses the slicing-granularity issue. This concept is formalized in §4.

Contributions. The paper's contributions include the following:

- We identify the granularity issue caused by using source-code slicing algorithms at the machine-code level, and show how the issue can lead to very imprecise slices (§3.1).
- We formalize the notion of a "sub-portion" of an instruction, and state necessary and sufficient conditions for a sub-portion to preserve the soundness of a slice with respect to a slicing criterion, while making an effort to keep the slice as precise as possible (§4).
- We present an algorithm for context-sensitive interprocedural machine-code slicing. Our algorithm works on SDGs recovered by existing tools, and is parameterized by the ISA of the instructions in the binary (§5).
- We provide formal guarantees for the termination and soundness of our slicing algorithm, and for the precision of the slices it computes in comparison to those computed by state-of-the-art tools (Thms. 1, 2, and 3).

Our methods have been implemented in an IA-32 slicer called X. We present experimental results with X, which show that, on average, X reduces the sizes of slices obtained from a state-of-the-art tool by 36% for backward slices, and 82% for forward slices.

2. Background

In this section, we briefly describe a logic to express the semantics of IA-32 instructions, and how state-of-the-art tools recover from a binary an SDG on which to perform machine-code slicing.

2.1 QFBV Formulas for IA-32 Instructions

The operational semantics of IA-32 instructions can be expressed formally by QFBV formulas. In this section, we describe the syntax and semantics of the QFBV formulas that are used in this paper.

Syntax. Consider a quantifier-free bit-vector logic L over finite vocabularies of constant symbols and function symbols. We will be dealing with a specific instantiation of L, denoted by L[IA-32]. (L can also be instantiated for other ISAs.) In L[IA-32], some constants represent IA-32's registers (*EAX*, *ESP*, *EBP*, etc.), and some represent flags (*CF*, *SF*, etc.). L[IA-32] has only one function symbol "*Mem*," which denotes memory. The syntax of L[IA-32] is defined in Fig. 1. A term of the form $ite(\varphi, T_1, T_2)$ represents an if-then-else expression. A *FuncExpr* of the form $FE[T_1 \mapsto T_2]$ denotes a *function-update* expression.

The function $\langle\!\langle \cdot \rangle\!\rangle$ encodes an IA-32 instruction as a QFBV formula. While others have created such encodings by hand (e.g., [21]), we use a method that takes a specification of the concrete operational semantics of IA-32 instructions and creates a QFBV encoder automatically. The method reinterprets each semantic operator as a QFBV formula-constructor or term-constructor (see

$$T \in \mathit{Term},\ \varphi \in \mathit{Formula},\ FE \in \mathit{FuncExpr}$$
$$c \in \mathit{Int32} = \{\ldots, \text{-1}, 0, 1, \ldots\} \quad b \in \mathit{Bool} = \{\mathit{True}, \mathit{False}\}$$
$$I_{\mathit{Int32}} \in \mathit{Int32Id} = \{\mathit{EAX}, \mathit{ESP}, \mathit{EBP}, \ldots\}$$
$$I_{\mathit{Bool}} \in \mathit{BoolId} = \{\mathit{CF}, \mathit{SF}, \ldots\} \quad F \in \mathit{FuncId} = \{\mathit{Mem}\}$$
$$op \in \mathit{BinOp} = \{+, -, \ldots\} \quad bop \in \mathit{BoolOp} = \{\wedge, \vee, \ldots\}$$
$$rop \in \mathit{RelOp} = \{=, \neq, <, >, \ldots\}$$
$$T ::= c \mid I_{\mathit{Int32}} \mid T_1\ op\ T_2 \mid ite(\varphi, T_1, T_2) \mid F(T_1)$$
$$\varphi ::= b \mid I_{\mathit{Bool}} \mid T_1\ rop\ T_2 \mid \neg\varphi_1 \mid \varphi_1\ bop\ \varphi_2 \mid F = FE$$
$$FE ::= F \mid FE_1[T_1 \mapsto T_2]$$

Figure 1: Syntax of L[IA-32].

[16]). To write formulas that express state transitions, all *Int32Id*s, *BoolId*s, and *FuncId*s can be qualified by primes (e.g., *Mem'*). The QFBV formula for an instruction is a restricted 2-vocabulary formula that specifies a state transformation. It has the form

$$\bigwedge_m (I'_m = T_m) \wedge \bigwedge_n (J'_n = \varphi_n) \wedge \mathit{Mem}' = FE, \tag{1}$$

where I'_m and J'_n range over the constant symbols for registers and flags, respectively. The primed vocabulary is the post-state vocabulary, and the unprimed vocabulary is the pre-state vocabulary.

We use *Id* to denote the identity transformation:

$$\mathit{Id} \stackrel{\text{def}}{=} \bigwedge_m (I'_m = I_m) \wedge \bigwedge_n (J'_n = J_n) \wedge \mathit{Mem}' = \mathit{Mem},$$

The QFBV formulas for a few IA-32 instructions are given below. (Note that the IA-32 program counter is register EIP, and the stack pointer is register ESP.)

$$\langle\!\langle \texttt{mov eax, [ebp]} \rangle\!\rangle \equiv \mathit{EAX}' = \mathit{Mem}(\mathit{EBP}) \tag{2}$$
$$\langle\!\langle \texttt{push 0} \rangle\!\rangle \equiv \mathit{ESP}' = \mathit{ESP} - 4 \wedge \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} - 4 \mapsto 0]$$
$$\langle\!\langle \texttt{jz 1000} \rangle\!\rangle \equiv ite(\mathit{ZF} = 0, \mathit{EIP}' = 1000,\ \mathit{EIP}' = \mathit{EIP} + 4)$$
$$\langle\!\langle \texttt{lea esp, [esp-4]} \rangle\!\rangle \equiv \mathit{ESP}' = \mathit{ESP} - 4$$

In this section, and in the rest of the paper, we show only the portions of QFBV formulas that express how the state is *modified*. QFBV formulas actually contain identity conjuncts of the form $I' = I$, $J' = J$, and $\mathit{Mem}' = \mathit{Mem}$ for constants and functions that are *unmodified*.

Semantics. Intuitively, a QFBV formula represents updates made to the machine state by an instruction. QFBV formulas in L[IA-32] are interpreted as follows: elements of *Int32*, *Bool*, *BinOp*, *RelOp*, and *BoolOp* are interpreted in the standard way. An unprimed (primed) constant symbol is interpreted as the value of the corresponding register or flag from the pre-state (post-state). An unprimed (primed) *Mem* symbol is interpreted as the memory array from the pre-state (post-state). (To simplify the presentation in the paper, we pretend that each memory location holds a 32-bit integer; however, in our implementation memory is addressed at the level of individual bytes.)

An IA-32 machine-state is a triple of the form:

$$\langle \mathit{RegMap}, \mathit{FlagMap}, \mathit{MemMap} \rangle$$

RegMap, *FlagMap*, and *MemMap* map each register, flag, and memory location in the state, respectively, to a value. The meaning of a QFBV formula φ in L[IA-32] (denoted by $[\![\varphi]\!]$) is a set of machine-state pairs ($\langle$pre-state, post-state$\rangle$) that satisfy the formula. For instance, a $\langle$pre-state, post-state$\rangle$ pair that satisfies Eqn. (2) is

$$\sigma \equiv \langle [\mathrm{EBP} \mapsto 100],\ [\,],\ [100 \mapsto 1] \rangle$$

$$\sigma' \equiv \langle [\text{EAX} \mapsto 1][\text{EBP} \mapsto 100],\ [\,],\ [100 \mapsto 1] \rangle.$$

Note that the location names in states are not italicized to distinguish them from constant symbols in QFBV formulas. By convention, all locations in a state for which the range value is not shown explicitly have the value 0.

2.2 SDG Recovery and Slicing in State-of-the-Art Tools

Slicing is typically performed using an Intermediate Representation (IR) of the binary called a *system dependence graph* (SDG) [12, 14]. To build an SDG for a program, one needs to know the set of variables that might be used and killed in each statement of the program. However in machine code, there is no explicit notion of variables. In this section, we briefly describe how state-of-the-art tools such as CodeSurfer/x86 [5] recover "variable-like" abstractions from a binary, and use those abstractions to construct an SDG and perform slicing.

CodeSurfer/x86 uses Value-Set Analysis (VSA) [4] to compute the abstract state (σ_{VSA}) that can arise at each program point. σ_{VSA} maps an *abstract location* to an *abstract value*. An abstract location (*a-loc*) is a "variable-like" abstraction recovered by the analysis [4, §4]. (In addition to these variable-like abstractions, a-locs also include IA-32 registers and flags.) An abstract value (*value-set*) is an over-approximation of values that a particular a-loc can have at a given program point. For example, Fig. 2 shows the VSA state before each instruction in a small IA-32 code snippet. In Fig. 2, an a-loc of the form (AR_main, n) denotes the variable-like proxy at offset $-$n in the activation record of function main, and $\top$ denotes any value. In reality, the VSA state before instruction `4` contains value-sets for the flags set by the `sub` instruction. However, to reduce clutter, we have not shown the flag a-locs in the VSA state. For each instruction i in the binary, CodeSurfer/x86 uses the operational semantics of i, along with σ_{VSA} at i to compute $\text{USE}^{\#}(i, \sigma_{VSA})$ ($\text{KILL}^{\#}(i, \sigma_{VSA})$), which is the set of a-locs that might be used (modified) by i. The $\text{USE}^{\#}$ and $\text{KILL}^{\#}$ sets for each instruction in the code snippet are also shown in Fig. 2.

CodeSurfer/x86 uses this information to build a collection of IRs, including an SDG. An SDG consists of a set of *program dependence graphs* (PDGs), one for each procedure in the program. A node in a PDG corresponds to a construct in the program, such as an instruction, a call to a procedure, a procedure entry/exit, an actual parameter of a call, or a formal parameter of a procedure. The edges correspond to data and control dependences between the nodes [12]. For example, in the system-dependence subgraph for the code snippet in Fig. 2, there is a control-dependence edge from the entry of `main` to instructions `1`, `2`, `3`, and `4`; there is a data-dependence edge from instruction `1`, which assigns to register ESP, to instructions `2` and `3`, which use ESP, as well as from instruction `3` to instruction `4`.

In a PDG, a call to a procedure is associated with two nodes: a *call-expression* node, which contains the `call` instruction, and a *call-site* node, which is a control node. The PDGs are connected together with interprocedural control-dependence edges between call-site nodes and procedure entry nodes, and interprocedural data-dependence-edges between actual parameters and formal parameters/return values. (See Fig. 10 for an example SDG snippet with interprocedural edges.)

CodeSurfer/x86 uses an existing interprocedural-slicing algorithm [20] to perform machine-code slicing on the recovered SDG.

3. Overview

In this section, we use two example programs to illustrate the granularity issue involved in slicing binaries using state-of-the-art tools, and the improved slicing technique used in X.

3.1 Granularity Issue in Machine-Code Slicing

Consider the C program diff shown in Fig. 3. The `main` function contains calls to functions `add` and `square`. `main` does not use the

```
int add(int a, int b){
  int c = a + b;
  return c;
}
int square(int a){
  int b = a * a;
  return b;
}
```

```
int main(){
  int a = 10, b = 20;
  int c = add(a, b);
  int d = square(c);
  return a - b;
}
```

Figure 3: Source code for the diff program, and the backward slice with respect to the return value of `main`.

```
add:
push ebp
mov ebp,esp
sub esp,4
mov eax,[ebp+12]
add eax,[ebp+8]
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret

square:
push ebp
mov ebp,esp
sub esp,4
mov eax,[ebp+8]
imul eax,[ebp+8]
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
main:
push ebp
mov ebp,esp
sub esp,16
mov [ebp-16],10
mov [ebp-12],20
push [ebp-12]
push [ebp-16]
call add
add esp,8
mov [ebp-8],eax  *
push [ebp-8]  **
call square
add esp,4
mov [ebp-4],eax
mov eax,[ebp-16]
mov ebx,[ebp-12]
sub eax,ebx
leave
ret
```

Figure 4: Assembly listing for diff with the imprecise backward slice computed by CodeSurfer/x86.

return values of the calls, and simply returns the difference between two local variables `a` and `b`. Suppose that we want to compute the program points that affect `main`'s return value (boxed in Fig. 3). The backward slice with respect to `main`'s return value gets us the desired result. The backward slice is highlighted in gray in Fig. 3. (The slice is computed using CodeSurfer/C [2].)

Let us now slice the same program with respect to the analogous slicing criterion at the machine-code level. The assembly listing for diff is shown in Fig. 4. The corresponding slicing criterion is the register a-loc {EAX} at the boxed instruction in Fig. 4. The backward slice with respect to the slicing criterion includes the lines highlighted in gray in Fig. 4. (The slice is computed using CodeSurfer/x86.) One can see that entire body of the `add` function—which is completely irrelevant to the slicing criterion—is included in the slice. What went wrong?

Machine-code instructions are usually *multi-assignments*: they have several inputs, and several outputs (e.g., registers, flags, and memory locations). This aspect of the language introduces a *granularity* issue during slicing: in some cases, although we would like the slice to include only a subset of an instruction's semantics, the slicing algorithm is forced to include the entire instruction. For example, consider the `push` instruction marked by ** in Fig. 4. The QFBV formula for the instruction is

$$\langle\!\langle \texttt{push [ebp-8]} \rangle\!\rangle \equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto Mem(EBP - 8)]. \quad (3)$$

The instruction updates the stack-pointer register ESP along with a memory location.

Just before ascending back to `main` from the `square` function, the most recent instruction added to the slice is "`push ebp`" in `square`. The QFBV formula for the instruction is given below.

$$\langle\!\langle \texttt{push ebp} \rangle\!\rangle \equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto EBP]$$

```
main:
1: push ebp          [ESP↦(AR_main,-4)][EBP↦ ⊤],  USE#={EBP, ESP},  KILL#={ESP, (AR_main,0)}
2: mov ebp,esp       [ESP↦(AR_main,0)][EBP↦ ⊤][(AR_main,0)↦ ⊤],  USE#={ESP},  KILL#={EBP}
3: sub esp,10        [ESP↦(AR_main,0)][EBP↦(AR_main,0)][(AR_main,0)↦ ⊤],  USE#={ESP},  KILL#={ESP}
4: mov [esp],1       [ESP↦(AR_main,10)][EBP↦(AR_main,0)][(AR_main,0)↦ ⊤],  USE#={ESP},  KILL#={(AR_main,10)}
5: ...               [ESP↦(AR_main,10)][EBP↦(AR_main,0)][(AR_main,0)↦ ⊤][(AR_main,10)↦1]  −,  −
```

Figure 2: A VSA state before each instruction in a small code snippet, and USE# and KILL# sets for each instruction.

```
int add(int a, int b){
  int c = a + b;
  return c;
}
int mult(int a, int b){
  int c = a * b;
  return c;
}
```

```
int main(){
  int a = 1, b = 2;
  int c = add(a, b);
  int d = 3, e = 4;
  int f = mult(d, e);
  return f;
}
```

Figure 5: Source code for the multiply program, and the forward slice with respect to a.

```
add:
push ebp
mov ebp,esp
sub esp,4
mov eax,[ebp+12]
add eax,[ebp+8]
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret

mult:
push ebp
mov ebp,esp
sub esp,4
mov eax,[ebp+12]
imul eax,[ebp+8]
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

```
main:
push ebp
mov ebp,esp
sub esp,32
mov [ebp-24],1
mov [ebp-20],2
push [ebp-20]
push [ebp-24] **
call add
add esp,8
mov [ebp-16],eax
mov [ebp-12],3
mov [ebp-8],4
push [ebp-8]
push [ebp-12]
call mult
add esp,8
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret
```

Figure 6: Assembly listing for multiply with the imprecise forward slice computed by CodeSurfer/x86.

The instruction uses the registers ESP and EBP. When the slice ascends back into `main`, it requires the definition of ESP from the `push` instruction marked by ** in Fig. 4. However, the slice cannot include only a *part* of the instruction, and is forced to include the entire `push` instruction, which also uses the contents of the memory location whose address is EBP − 8. The value in location EBP − 8 is set by the instruction marked by * in Fig. 4. That instruction also uses the value of register EAX, which holds the return value of the `add` function. For this reason, the instruction marked by *, and the entire body of `add`, which are completely irrelevant to the slicing criterion, are included in the slice. This granularity issue also has a cascade effect—the irrelevant instructions cause more irrelevant instructions to be included in the slice.

Consider another C program multiply, shown in Fig. 5. The `main` function contains calls to functions `add` and `mult`. Suppose that we want to compute the program points that are affected by the local variable `a` (boxed in Fig. 5). The forward slice with respect to `a`, highlighted in gray in Fig. 5, gets us the desired result. The machine-code forward slice with respect to the analogous slicing criterion (a-loc $\{(\text{AR\_main}, 24)\}$ at the boxed instruction in Fig. 6) includes the lines highlighted in gray in Fig. 6. One can see that the entire body of the `mult` function—which is completely irrelevant to the slicing criterion—is included in the slice.

The imprecision again creeps in at the `push` instruction marked by ** in Fig. 6. The QFBV formula for the instruction is

$$\langle\!\langle \texttt{push [ebp-24]} \rangle\!\rangle \equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto Mem(EBP - 24)]. \quad (4)$$

The instruction stores the contents of the memory location whose address is EBP − 24 in a new memory location, and updates the stack-pointer register ESP. The slice only requires the update that uses the location EBP − 24. However, because of the granularity issue, the slice also includes the update to ESP. Because all the downstream instructions directly or transitively use ESP, the forward slice includes all the downstream instructions.

3.2 Improved Machine-Code Slicing in X

Given the (i) SDG of a binary, (ii) the SDG node n_G to slice from, and (iii) the global slicing criterion $S^{\#}_G$ after (before) n_G (where $S^{\#}_G$ is a set of a-locs), the improved slicing algorithm in X "grows" the backward (forward) slice from n_G by iterating over the nodes of the SDG using a worklist. X performs the following steps during each iteration.

1. Given a node n from the worklist, the instruction i in n, and the local slicing criterion $S^{\#}$ after (before) n, if X reaches n through a data-dependence edge, X includes in the slice only the "sub-portion" ψ of the semantics φ of i that is just enough to kill (use) $S^{\#}$. If X reaches n through a control-dependence edge, X includes in the slice the entire instruction i(i.e., ψ is φ).
2. X adds to the worklist only the predecessors (successors) that are relevant to ψ. The use (kill) set of ψ becomes the local slicing criterion for the predecessors (successors) that were added to the worklist.
3. For context-sensitive interprocedural slicing, X uses a callstack to match calls and returns.

When worklist iteration terminates, only the instructions (and instruction sub-portions) that are transitively relevant to $S^{\#}_G$ are included in the slice.

This section illustrates the improved slicing algorithm in X on our two examples from §3.1. We first use the program diff to illustrate improved backward slicing. Then we use the program multiply to illustrate improved forward slicing.

The local slicing criterion at instruction `17` of `main` in Fig. 7 is {EAX}. The semantics of instruction `17` can be explicitly represented by the following QFBV formula φ:

$$\begin{aligned}\langle\!\langle \texttt{sub eax,ebx} \rangle\!\rangle \equiv\ & EAX' = EAX - EBX \\ \wedge\ & SF' = ((EAX - EBX) < 0) \\ \wedge\ & \ldots \\ \wedge\ & ZF' = ((EAX - EBX) = 0)\end{aligned}$$

The "sub-portion" ψ of the semantics φ that is just enough to kill the local slicing criterion is given below. More precisely, ψ has the same effect on EAX as the full formula φ. (We formally define what a "sub-portion" of an instruction's semantics is in §4. For now, the reader should think of a "sub-portion" ψ of φ as a sub-formula of φ.)

$$\psi \equiv EAX' = EAX - EBX$$

```
add:
1: push ebp
2: mov ebp,esp
3: sub esp,4
(ESP' = ESP − 4)
4: mov eax,[ebp+12]
5: add eax,[ebp+8]
6: mov [ebp-4],eax
7: mov eax,[ebp-4]
8: leave
9: ret
(ESP' = ESP − 4)

square:
1: push ebp
2: mov ebp,esp
3: sub esp,4
4: mov eax,[ebp+8]
5: imul eax,[ebp+8]
6: mov [ebp-4],eax
7: mov eax,[ebp-4]
8: leave
(EBP' = Mem(EBP))
9: ret
```

```
main:
1: push ebp
(ESP' = ESP − 4)
2: mov ebp,esp
3: sub esp,16
(ESP' = ESP − 16)
4: mov [ebp-16],10
5: mov [ebp-12],20
6: push [ebp-12]
(ESP' = ESP − 4)
7: push [ebp-16]
(ESP' = ESP − 4)
8: call add
(ESP' = ESP − 4)
9: add esp,8
(ESP' = ESP − 8)
10: mov [ebp-8],eax
11: push [ebp-8]
(ESP' = ESP − 4)
12: call square
(ESP' = ESP − 4)
13: add esp,4
14: mov [ebp-4],eax
15: mov eax,[ebp-16]
16: mov ebx,[ebp-12]
17: sub eax,ebx
(EAX' = EAX − EBX)
leave
ret
```

Figure 7: Assembly listing for diff with the backward slice computed by X.

Instead of including the entire instruction in the slice, the improved slicing algorithm in X includes only ψ. (If the slice includes only a sub-portion of an instruction, the QFBV formula for the included sub-portion is shown within parentheses below the instruction, e.g., instruction `17` in Fig. 7.) X uses the VSA state at instruction `17` ($\sigma_{VSA}^{\text{main},17}$) to compute the USE$^{\#}$ set for ψ, and this set becomes the local slicing criterion for the SDG predecessors of instruction `17`.

$$\text{USE}^{\#}(\psi,\, \sigma_{VSA}^{\text{main},17}) = \{\text{EAX}, \text{EBX}\}$$

The slice includes instructions `15` and `16` in `main` because they define a-locs EAX and EBX, respectively. The local slicing criterion before instruction `15` is {EBP, (AR_main, 12), (AR_main, 16)}. (The a-locs (AR_main, 12) and (AR_main, 16) are used by instructions `16` and `15`, respectively.) To include the instructions that define EBP, the slice descends into the `square` function. The QFBV formula for instruction `8` in `square` is

$$\langle\!\langle \texttt{leave} \rangle\!\rangle \equiv EBP' = Mem(EBP) \wedge ESP' = EBP - 4.$$

X includes in the slice only the sub-portion ψ of the formula that is just enough to define EBP, and uses the VSA state before instruction `8` ($\sigma_{VSA}^{\text{square},8}$) to compute the USE$^{\#}$ set for ψ.

$$\psi \equiv EBP' = Mem(EBP)$$
$$\text{USE}^{\#}(\psi,\, \sigma_{VSA}^{\text{square},8}) = \{\text{EBP}, (\text{AR\_square}, 0)\}$$

(AR_square, 0) is the a-loc corresponding to the first slot in the activation record of the `square` function, where the caller's frame pointer is saved by instruction `1` in `square`. The local slicing criterion before instruction `8` in `square` is {EBP, (AR_square, 0), (AR_main, 12), (AR_main, 16)}. X includes in the slice instructions `1` and `2` in `square` because they define a-locs (AR_square, 0) and EBP, respectively. When the slice ascends back into main, the local slicing criterion after instruction `12` in `main` is {EBP, ESP, (AR_main, 12), (AR_main, 16)}.

X includes in the slice only the relevant sub-portion of the `call` instruction (see the QFBV formula below instruction `12` in `main`), and the local slicing criterion after instruction `11` is {EBP, ESP, (AR_main, 12), (AR_main, 16)}. The QFBV formula φ of instruction `11` in `main` is given in Eqn. (3). The sub-portion ψ of φ that is relevant to the local slicing criterion is

$$\psi \equiv ESP' = ESP - 4.$$

(The other sub-portion of φ kills the a-loc (AR_main, 8), which is irrelevant to the local slicing criterion.) X includes only ψ in the slice, and the local slicing criterion before instruction `11` in `main` is {EBP, ESP, (AR_main, 12), (AR_main, 16)}. In contrast, the naïve slicing technique discussed in §3.1 included the entire instruction in the slice, and the local slicing criterion before instruction `11` was {EBP, ESP, (AR_main, 8), (AR_main, 12), (AR_main, 16)}. X does not include in the slice instruction `10` in `main` because it is irrelevant to the local slicing criterion, and consequently, X does not include the entire body of `add` in the slice. The final slice computed by X is shown by the lines highlighted in Fig. 7 in gray. One can see that the slice computed by X is more precise than the backward slice computed by CodeSurfer/x86 (cf. Fig. 4).

X computes forward slices in a similar manner except that it includes the sub-portion of a node's instruction that is just enough to use the local slicing criterion before the node. For example, consider instruction `7` in `main` given in Fig. 8. The local slicing criterion before the instruction is {(AR_main, 24)}. The QFBV formula φ of instruction `7` in `main` is given in Eqn. (4). The sub-portion ψ of φ that is relevant to the local slicing criterion is

$$\psi \equiv Mem' = Mem[ESP - 4 \mapsto EBP - 24].$$

(The other sub-portion of φ kills the a-loc ESP, which is irrelevant to the local slicing criterion.) X includes only ψ in the slice, and the local slicing criterion after instruction `7` in `main` is {(AR_main, 32)}. In contrast, the naïve slicing technique discussed in §3.1 included the entire instruction in the slice, and the local slicing criterion after instruction `7` was {ESP, (AR_main, 32)}. The slice computed by X descends into `add`, and includes instruction `5` in `add` because it uses the a-loc (AR_add, −8). (The a-locs (AR_main, 32) and (AR_add, −8) are aliases of each other, and X binds actual-parameter a-locs to formal-parameter a-locs when it crosses a procedure boundary.) The final forward slice computed by X for the multiply program is shown by the lines highlighted in Fig. 8 in gray. Again, the slice computed by X is more precise than the forward slice computed by CodeSurfer/x86 (cf. Fig. 6).

4. Projection Semantics

In addition to entire instructions, the improved slicing algorithm in X also includes "sub-portions" of instructions in the slice. We would like the slicing algorithm in X to be sound—i.e., it should include in the slice all sub-portions of instructions that are transitively relevant to the *global* slicing criterion—while making an effort to keep the slice as precise as possible. To achieve this goal, at each instruction i, X must include in the slice a sub-portion ψ of i that (i) is relevant to the *local* slicing criterion at i, and (ii) keeps the slice as precise as possible. Let us first look at a few examples that illustrate the desired behavior of the slicing algorithm.

Example 1. *Consider the QFBV formula φ for the "`push eax`" instruction, and the formulas ψ_1 and ψ_2 representing sub-portions of φ.*

$$\varphi \equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto EAX]$$
$$\psi_1 \equiv ESP' = ESP - 4 \quad \psi_2 \equiv Mem' = Mem[ESP - 4 \mapsto EAX]$$

Suppose that we are computing a backward slice, and the local slicing criterion $S^{\#}$ after the `push` instruction is {ESP}. To compute a sound slice, the slicing algorithm can include in the slice either the sub-portion ψ_1 of φ, or the entire formula φ because both ψ_1

```
add:
1: push ebp
2: mov ebp,esp
3: sub esp,4
4: mov eax,[ebp+12]
5: add eax,[ebp+8]
(EAX' = EAX + Mem(EBP +
8))
6: mov [ebp-4],eax
7: mov eax,[ebp-4]
8: leave
9: ret

mult:
1: push ebp
2: mov ebp,esp
3: sub esp,4
4: mov eax,[ebp+12]
5: imul eax,[ebp+8]
6: mov [ebp-4],eax
7: mov eax,[ebp-4]
8: leave
9: ret
```

```
main:
1: push ebp
2: mov ebp,esp
3: sub esp,32
4: mov [ebp-24],1
5: mov [ebp-20],2
6: push [ebp-20]
7: push [ebp-24]
(Mem' = Mem(ESP - 4 ↦
Mem(EBP - 24))
8: call add
9: add esp,8
10: mov [ebp-16],eax
11: mov [ebp-12],3
12: mov [ebp-8],4
13: push [ebp-8]
14: push [ebp-12]
15: call mult
16: add esp,8
17: mov [ebp-4],eax
18: mov eax,[ebp-4]
19: leave
20: ret
```

Figure 8: Assembly listing for multiply with the forward slice computed by X.

and φ kill $S^{\#}$. However, it should not include the sub-portion ψ_2 of φ alone. ψ_2 does not kill the local slicing criterion $S^{\#}$, and is actually irrelevant to $S^{\#}$. If the slicing algorithm included only ψ_2 in the slice, the resulting slice would be unsound.

If the slicing algorithm includes the entire formula φ in the slice, φ would use irrelevant locations, making the slice imprecise. To keep the slice as precise as possible, the slicing algorithm should choose ψ_1—and not the entire φ—to be included in the slice.

Example 2. *Consider the `push` instruction and QFBV formulas from Ex. 1. Suppose that we are computing a forward slice, and the local slicing criterion $S^{\#}$ before the `push` instruction is $\{ESP\}$. The two sub-portions ψ_1 and ψ_2 both use the slicing criterion. However, to be sound, the slicing algorithm should include the entire formula φ in the slice.*

Given a local slicing criterion $S^{\#}$ at an instruction i, the properties of a sub-portion of i that preserves the soundness of the slice with respect to $S^{\#}$, while keeping the slice as precise as possible, are given in Defn. 7. X uses the criteria in Defn. 7 when choosing what sub-portions of instructions to include in a slice. The sequence of definitions in the remainder of this section build up to Defn. 7. First we define what it means for a QFBV formula to represent a "sub-portion" of an instruction's semantics (Defn. 3). Second, we state the properties of a sub-portion that preserves the soundness of the slice with respect to a local slicing criterion (Defn. 6). Finally, we state the criteria for preserving precision (Defn. 7).

4.1 Sub-Portion of an Instruction's Semantics

Given the semantics φ of an instruction, we first provide *semantic criteria* for a QFBV formula ψ to represent a sub-portion of φ (Defn. 2). We then provide a *syntactic* way to obtain a satisfactory formula ψ from φ (Defn. 3).

To simplify matters, we restrict ourselves to the case that arises in the IA-32 instruction set, where the semantics of (most) instructions involves at most one memory update. Exceptions to this rule are x86 string instructions, which have the `rep` prefix. In our implementation, we do not attempt to find sub-portions for such instructions.

We now formulate the concept of when a formula ψ performs less of a state transformation than a formula φ. Recall from §2.1 that the meaning of a QFBV formula is the set of machine-state pairs that satisfy the formula.

Definition 1. ***[Ordering on QFBV formulas via the state transformation performed.]*** *The meaning of ψ **transforms the state less** than the meaning of φ (denoted by $[\![\psi]\!] \sqsubseteq [\![\varphi]\!]$) iff for all $\langle \sigma, \sigma'' \rangle \in [\![\psi]\!]$, $\langle \sigma, \sigma' \rangle \in [\![\varphi]\!]$, $l \in dom(\sigma)$, the following properties hold:*

1. $(\sigma(l) = \sigma'(l)) \Rightarrow (\sigma(l) = \sigma''(l))$
2. $(\sigma(l) \neq \sigma''(l)) \Rightarrow (\sigma''(l) = \sigma'(l))$

The first property ensures that ψ does not modify any location that is unmodified by φ. The second property ensures that ψ produces the same value as φ for each location modified by ψ.

Definition 2. ***Semantic Projection.*** *A QFBV formula ψ is a **semantic projection** of the semantics φ of an instruction i in a binary B iff $[\![\psi]\!] \sqsubseteq [\![\varphi]\!]$.*

Example 3. *Consider the QFBV formulas given below. For this example, let us suppose that the IA-32 ISA has only two registers: EAX and EBX.*

$$\varphi \equiv EAX' = EAX + EAX \wedge EBX' = 1$$
$$\psi \equiv EAX' = EAX * 2 \wedge EBX' = EBX$$

Now consider the following states:

$$\sigma_1 \equiv \langle [EAX \mapsto 1][EBX \mapsto 1] \rangle \quad \sigma_2 \equiv \langle [EAX \mapsto 1][EBX \mapsto 2] \rangle$$
$$\sigma_1' \equiv \langle [EAX \mapsto 2][EBX \mapsto 1] \rangle \quad \sigma_2' \equiv \langle [EAX \mapsto 2][EBX \mapsto 1] \rangle$$
$$\sigma_2'' \equiv \langle [EAX \mapsto 2][EBX \mapsto 2] \rangle$$

ψ always produces the same value as φ for the EAX register. However, ψ never modifies the value in the EBX register, whereas φ may or may not modify the value in EBX, depending on the pre-state. For instance, we have $\langle \sigma_1, \sigma_1' \rangle \models \varphi$ and $\langle \sigma_1, \sigma_1' \rangle \models \psi$. Moreover, $\langle \sigma_2, \sigma_2'' \rangle \models \psi$ and $\langle \sigma_2, \sigma_2' \rangle \models \varphi$. $\sigma_2(EAX) \neq \sigma_2'(EAX)$ and $\sigma_2(EBX) \neq \sigma_2'(EBX)$, so property 1 holds; $\sigma_2(EAX) \neq \sigma_2''(EAX)$ but $\sigma_2''(EAX) = \sigma_2'(EAX)$, so property 2 holds.

Therefore, $[\![\psi]\!] \sqsubseteq [\![\varphi]\!]$, and ψ represents a semantic projection of φ.

A QFBV formula φ can have several semantic projections. Let us see how we can obtain a subset of such ψs syntactically from φ.

Definition 3. ***Projection Sub-Formula.*** *Suppose that φ is a QFBV formula of the form shown in Eqn. (1). A **projection sub-formula** ψ of φ is a QFBV formula (of the form shown in Eqn. (1)) that is obtained from φ by any combination of the following actions:*

- *Replace a conjunct of the form $I_m' = T_m$ with the identity conjunct $I_m' = I_m$.*
- *Replace a conjunct of the form $J_n' = \varphi_n$ with the identity conjunct $J_n' = J_n$.*
- *Replace the conjunct of the form $Mem' = FE$ (where FE is a function-update expression) with the identity conjunct $Mem' = Mem$.*

Trivially, *Id* and φ are projection sub-formulas of φ. (*Id* is obtained by dropping all conjuncts from φ and replacing them by identity conjuncts; φ is obtained by not dropping anything.) In the rest of the paper, we use the terms "sub-portion" and "projection sub-formula" interchangeably.

Example 4. *Consider the `push` instruction and QFBV formulas from Ex. 1.*

$$\varphi \equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto EAX]$$
$$\psi_1 \equiv ESP' = ESP - 4 \quad \psi_2 \equiv Mem' = Mem[ESP - 4 \mapsto EAX]$$

ψ_1 and ψ_2 are both projection sub-formulas of φ.

Observation 1. *If ψ is a projection sub-formula of φ, then ψ is a semantic projection of φ.*

Observation 2. *The projection sub-formula Id transforms the state the least among all projection sub-formulas of φ because Id does not modify any location (i.e., for all $\langle\sigma, \sigma'\rangle \in [\![Id]\!]$, $\sigma = \sigma'$). The projection sub-formula φ of φ transforms the state most among all projection sub-formulas of φ.*

Suppose that φ is a projection sub-formula or a QFBV formula denoting the semantics of an instruction, and σ_{VSA} is a VSA state. In the remainder of the paper, $\text{USE}^{\#}(\varphi, \sigma_{VSA})$ denotes the set of a-locs possibly used by φ, and $\text{KILL}^{\#}(\varphi, \sigma_{VSA})$ denotes the set of a-locs possibly modified by φ.

Observation 3. *Suppose that φ is the QFBV formula of an instruction i in a binary B, and σ_{VSA} is the VSA state at i. If ψ is a projection sub-formula of φ, then $KILL^{\#}(\psi, \sigma_{VSA}) \subseteq KILL^{\#}(\varphi, \sigma_{VSA})$, and $USE^{\#}(\psi, \sigma_{VSA}) \subseteq USE^{\#}(\varphi, \sigma_{VSA})$.*

4.2 Soundness

Now that we have formally defined a "sub-portion" of the semantics of an instruction, we can provide a sufficient condition for a sub-portion that "kills (uses) enough locations" with respect to a local slicing criterion to preserve the soundness of the slice. Before stating the condition, we define the prerequisite notion of *projection complement*. We first provide a *semantic criterion* for a QFBV formula $\overline{\overline{\psi}}^{\varphi}$ to represent the projection complement of a projection sub-formula ψ of φ (Defn. 4). We then provide a way to obtain a suitable formula *syntactically* from ψ and φ (Defn. 5).

Definition 4. ***Semantic Projection Complement.*** *Suppose that ψ is a projection sub-formula of the semantics φ of an instruction i in a binary B. A QFBV formula $\overline{\overline{\psi}}^{\varphi}$ is a **semantic projection complement** of ψ with respect to φ iff $\overline{\overline{\psi}}^{\varphi}$ satisfies the following properties:*

1. *$\overline{\overline{\psi}}^{\varphi}$ is a semantic projection of φ*
2. *for all $\langle\sigma, \sigma'\rangle \in [\![\varphi]\!]$, $\langle\sigma, \sigma''\rangle \in [\![\psi]\!]$, $\langle\sigma, \sigma'''\rangle \in [\![\overline{\overline{\psi}}^{\varphi}]\!]$, $l \in dom(\sigma)$,*
   $(\sigma(l) \neq \sigma'(l) \wedge \sigma(l) = \sigma''(l)) \Rightarrow (\sigma'(l) = \sigma'''(l))$

The first property ensures that the meaning of $\overline{\overline{\psi}}^{\varphi}$ transforms the state less than the meaning of φ. The second property ensures that, for each location l that was modified by φ and not modified by ψ, $\overline{\overline{\psi}}^{\varphi}$ modifies l, producing the same value produced by φ. Together, ψ and $\overline{\overline{\psi}}^{\varphi}$ perform all state transformations performed by φ.

A projection sub-formula ψ of φ can have several semantic projection complements. It always has one, namely, φ itself.

Just as we obtained a projection sub-formula ψ syntactically from φ, we can also obtain a projection complement of ψ syntactically from ψ and φ.

Definition 5. ***Projection Sub-Formula Complement.*** *Suppose that ψ is a projection sub-formula of the semantics φ of an instruction i in a binary B. The **projection sub-formula complement** $\overline{\psi}^{\varphi}$ of ψ with respect to φ is a formula of the form shown in Eqn. (1) that only contains the conjuncts that were dropped from φ to create ψ.*

Note that a projection sub-formula ψ of φ has only one projection sub-formula complement with respect to φ.

Observation 4. *Note that $\overline{\varphi}^{\varphi} = Id$ and $\overline{Id}^{\varphi} = \varphi$.*

Observation 5. *If $\overline{\psi}^{\varphi}$ is the projection sub-formula complement of ψ with respect to φ, then $\overline{\psi}^{\varphi}$ is a semantic projection complement of ψ with respect to φ.*

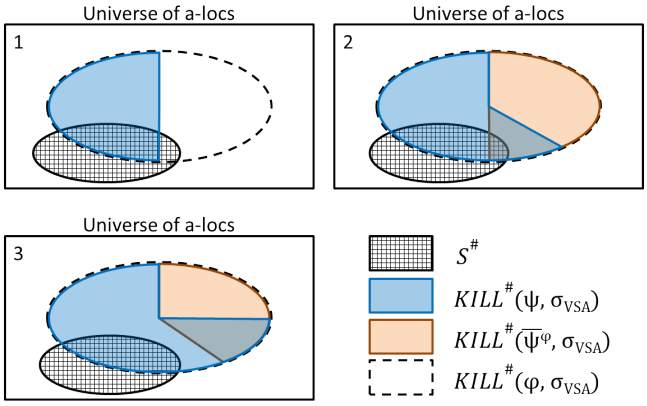


Figure 9: Visualization of sets $S^{\#}$, $\text{KILL}^{\#}(\psi, \sigma_{VSA})$, $\text{KILL}^{\#}(\overline{\psi}^{\varphi}, \sigma_{VSA})$, and $\text{KILL}^{\#}(\varphi, \sigma_{VSA})$.

Example 5. *The projection sub-formula complements of ψ_1 and ψ_2 with respect to φ (from Ex. 4) are given below.*

$$\overline{\psi_1}^{\varphi} \equiv Mem' = Mem[ESP-4 \mapsto EAX] \quad \overline{\psi_2}^{\varphi} \equiv ESP' = ESP-4$$

Given an instruction with semantics φ, and a local slicing criterion $S^{\#}$, the next definition provides a sufficient condition for a sub-portion ψ of φ to preserve the soundness of the slice even if ψ were included in the slice instead of φ.

Definition 6. ***Restricted Projection Sub-Formula.*** *Suppose that φ is the QFBV formula of an instruction i in a binary B, σ_{VSA} is the VSA state at i, and $S^{\#}$ is the local slicing criterion after (before) i. Also suppose that ψ is a projection sub-formula of φ, and $\overline{\psi}^{\varphi}$ is the projection sub-formula complement of ψ with respect to φ. ψ is **kill-restricted (use-restricted) with respect to $S^{\#}$** iff ψ satisfies the following properties:*

Kill-restricted
1. $(S^{\#} \cap KILL^{\#}(\varphi, \sigma_{VSA})) \subseteq KILL^{\#}(\psi, \sigma_{VSA})$
2. $KILL^{\#}(\overline{\psi}^{\varphi}, \sigma_{VSA}) \cap S^{\#} = \emptyset$

Use-restricted
1. $(S^{\#} \cap USE^{\#}(\varphi, \sigma_{VSA})) \subseteq USE^{\#}(\psi, \sigma_{VSA})$
2. $USE^{\#}(\overline{\psi}^{\varphi}, \sigma_{VSA}) \cap S^{\#} = \emptyset$

The first property ensures that the projection sub-formula ψ restricted with respect to $S^{\#}$ includes conjuncts from φ that are just enough to kill (use) as much of $S^{\#}$ as φ. The second property ensures that ψ includes *all* conjuncts of φ that kill (use) as much of $S^{\#}$ as φ. We use the Venn diagrams in Fig. 9 to illustrate the two properties. (The Venn diagrams depict $\text{KILL}^{\#}$ sets. The diagrams for $\text{USE}^{\#}$ sets look similar.) In the first diagram, we have a projection sub-formula ψ that violates property 1—ψ does not kill enough a-locs to meet property 1. In the second diagram, we have a projection sub-formula ψ that kills enough a-locs to meet property 1, but we have not included in ψ all conjuncts of φ that kill $S^{\#}$. (This is evident from the fact that $\text{KILL}^{\#}(\overline{\psi}^{\varphi}, \sigma_{VSA})$ overlaps with $S^{\#}$.) In the third diagram, we have a projection sub-formula ψ that kills enough a-locs to meet property 1, and we have not left out any conjuncts of φ that kill as much of $S^{\#}$ as φ.

Example 6. *Consider the `push` instruction, QFBV formulas, and local slicing criterion $S^{\#} = \{ESP\}$ from Ex. 1.*

$$\varphi \equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto EAX]$$
$$\psi_1 \equiv ESP' = ESP - 4 \quad \psi_2 \equiv Mem' = Mem[ESP - 4 \mapsto EAX]$$

Also suppose that the VSA state σ_{VSA} at the `push` instruction is $[ESP \mapsto (AR\_main, -4)][EAX \mapsto 1]$. The $KILL^{\#}$ sets of φ, Id, ψ_1,

and ψ_2 are given below.

$$KILL^{\#}(\varphi, \sigma_{VSA}) = \{ESP, (AR\_main, 0)\} \quad KILL^{\#}(Id, \sigma_{VSA}) = \emptyset$$
$$KILL^{\#}(\psi_1, \sigma_{VSA}) = \{ESP\} \quad KILL^{\#}(\psi_2, \sigma_{VSA}) = \{(AR\_main, 0)\}$$

ψ_1 and φ are projection sub-formulas of φ kill-restricted with respect to $S^{\#}$, but ψ_2 and Id are not. Both ψ_2 and Id violate property 1 given in Defn. 6.

Example 7. *Consider the* `push` *instruction, QFBV formulas, and VSA state from Ex. 6. Suppose that the local slicing criterion $S^{\#}$ before the* `push` *instruction is $\{ESP\}$, and we want to compute a forward slice. The $USE^{\#}$ sets of φ, Id, ψ_1, and ψ_2 are given below.*

$$USE^{\#}(\varphi, \sigma_{VSA}) = \{ESP, EAX\} \quad USE^{\#}(Id, \sigma_{VSA}) = \emptyset$$
$$USE^{\#}(\psi_1, \sigma_{VSA}) = \{ESP\} \quad USE^{\#}(\psi_2, \sigma_{VSA}) = \{ESP, EAX\}$$

φ is the only projection sub-formula of φ that is use-restricted with respect to $S^{\#}$. Id violates property 1 given in Defn. 6. ψ_1 and ψ_2 satisfy property 1, but not property 2.

4.3 Precision

Given the local slicing criterion $S^{\#}$ after (before) an instruction i with semantics φ, if X includes in the slice any projection sub-formula of φ that is kill-restricted (use-restricted) with respect to $S^{\#}$, the computed backward (forward) slice will be sound. For a given $S^{\#}$, there can be more than one projection sub-formula of φ that is restricted with respect to $S^{\#}$ (e.g., Ex. 6). Under such circumstances, X should choose a projection sub-formula that keeps the slice as precise as possible. What are the properties of a projection sub-formula that maximizes precision? Intuitively, if we are computing a backward slice, and are given a set of projection sub-formulas that are all sound with respect to a local slicing criterion, we would choose the one that uses minimal locations, so that the local slicing criteria for the predecessors will be minimal. (For the precision of a backward slice, we don't really care about the kill sets of the candidate sound projection sub-formulas.) This intuition is formalized by the next definition.

Definition 7. ***Minimal Restricted Projection Sub-Formula.*** *Suppose that φ is the QFBV formula of an instruction i in a binary B, σ_{VSA} is the VSA state at i, and $S^{\#}$ is the local slicing criterion after (before) i. A projection sub-formula ψ of φ is* ***minimally kill-restricted (minimally use-restricted) with respect to $S^{\#}$*** *iff ψ satisfies the following properties:*

1. *ψ is kill-restricted (use-restricted) with respect to $S^{\#}$.*
2. *There does not exist another projection sub-formula ψ' of φ that satisfies property 1 such that $USE^{\#}(\psi', \sigma_{VSA}) \subseteq USE^{\#}(\psi, \sigma_{VSA})$ ($KILL^{\#}(\psi', \sigma_{VSA}) \subseteq KILL^{\#}(\psi, \sigma_{VSA})$).*

A minimal kill-restricted (use-restricted) projection sub-formula gives us a sub-portion of an instruction's semantics that is just enough to kill (use) the local slicing criterion, and also uses (kills) as few locations as possible.

Example 8. *Consider the formulas and the local slicing criterion $S^{\#}$ from Ex. 6. ψ_1 is the projection sub-formula of φ that is minimally kill-restricted with respect to $S^{\#}$, whereas φ is not.*

Example 9. *Consider the formulas and the local slicing criterion $S^{\#}$ from Ex. 7. Because φ is the only projection sub-formula that is use-restricted with respect to $S^{\#}$, φ trivially becomes the projection sub-formula that is minimally use-restricted with respect to $S^{\#}$.*

Given the local slicing criterion $S^{\#}$ at instruction i with semantics φ, the slicing algorithm in X includes in the slice only a projection sub-formula of φ that is minimally use/kill restricted with respect to $S^{\#}$.

Algorithm 1 Strawman backward intraprocedural slicing algorithm

Input: SDG, n_G
Output: Slice

```
1:  worklist ← {n_G}
2:  Slice ← ∅
3:  while worklist ≠ ∅ do
4:      n ← RemoveItem(worklist)
5:      if n ∉ Slice then
6:          Slice ← Slice ∪ {n}
7:          worklist ← worklist ∪ n.predecessors
8:      end if
9:  end while
10: return Slice
```

We now define a primitive called `GrowProjection` that "grows" an existing sub-portion ϕ of the semantics of an instruction, such that the grown sub-portion is just enough to kill (use) the local slicing criterion, while using (killing) as few locations as possible. X uses the `GrowProjection` primitive to grow the sub-portions of instructions included in the slice.

Definition 8. ***`GrowProjection`(φ, ϕ, $S^{\#}$, σ_{VSA}).*** *Suppose that φ is the QFBV formula of an instruction i in a binary B, σ_{VSA} is the VSA state at i, and $S^{\#}$ is the local slicing criterion after (before) i. Given a projection sub-formula ϕ of φ,* `GrowProjection` *returns a QFBV formula ψ that satisfies the following properties:*

1. *ϕ is a projection sub-formula of ψ.*
2. *ψ is a projection sub-formula of φ that is minimally kill-restricted (use-restricted) with respect to $S^{\#}$.*

If there exists more than one ψ that satisfies properties 1 and 2, `GrowProjection` *breaks ties by picking the projection sub-formula whose $USE^{\#}$ ($KILL^{\#}$) set comes earliest in the lexicographic order of the sorted $USE^{\#}$ ($KILL^{\#}$) sets.*

Example 10. *Consider the formulas and the local slicing criterion from Ex. 6. If $\phi \equiv Id$,* `GrowProjection` *returns ψ_1.*

Example 11. *Consider the formulas and the local slicing criterion $S^{\#}$ from Ex. 6. If $\phi \equiv Mem' = Mem[ESP - 4 \mapsto EAX]$,* `GrowProjection` *returns φ because the only projection sub-formula minimally kill-restricted with respect to $S^{\#}$ that can be obtained by "growing" ϕ is φ.*

5. Algorithm

In this section, we describe the slicing algorithm used in X. First, we present X's intraprocedural-slicing algorithm. Then, we present extensions to the algorithm for interprocedural slicing.

5.1 Intraprocedural Slicing

In this sub-section, we present the intraprocedural backward-slicing algorithm used in X. It is straightforward to modify the algorithm to perform forward slicing. We start by presenting the intraprocedural-slicing algorithm used in CodeSurfer/x86; we then present the improved slicing algorithm that is actually used in X.

5.1.1 Base Algorithm

Given the SDG of the binary, and the SDG node n_G from which to slice, the intraprocedural-slicing algorithm in CodeSurfer/x86 includes in the backward slice all nodes that reach n_G by following intraprocedural data-dependence and control-dependence edges. This strawman algorithm is given as Alg. 1. In Alg. 1, `RemoveItem` removes an item from the worklist, and n.predecessors returns the set of intraprocedural data and control predecessors of n in the SDG. The granularity issue inherent in the IA-32 ISA causes Alg. 1 to compute an imprecise backward slice as illustrated in §3.1.

Algorithm 2 Backward intraprocedural slicing algorithm in X

Input: SDG, n_G, $S_G^\#$
Output: Slice

```
 1: for each node n ∈ SDG do
 2:     n.sliceBefore ← ∅
 3:     n.sliceAfter ← ∅
 4:     n.semantics ← Id
 5: end for
 6: n_G.sliceAfter ← S_G^#
 7: worklist ← {n_G}
 8: Slice ← ∅
 9: while worklist ≠ ∅ do
10:     n ← RemoveItem(worklist)
11:     for each s ∈ n.successors do
12:         n.sliceAfter ← n.sliceAfter ∪ s.sliceBefore
13:     end for
14:     prevSemantics ← n.semantics
15:     σ_VSA ← n.VSAState
16:     if IsControlNode(n) then
17:         n.semantics ← ⟨⟨n.instruction⟩⟩
18:     else
19:         φ ← ⟨⟨n.instruction⟩⟩
20:         S^# ← n.sliceAfter ∩ KILL^#(φ, σ_VSA)
21:         if S^# = ∅ then
22:             continue
23:         end if
24:         n.semantics ← GrowProjection(φ, prevSemantics,
            S^#, σ_VSA)
25:     end if
26:     if n.semantics ≠ prevSemantics then
27:         n.sliceBefore ← USE^#(n.semantics, σ_VSA)
28:         Slice ← Slice ∪ {n}
29:         worklist ← worklist ∪ n.predecessors
30:     end if
31: end while
32: return Slice
```

5.1.2 Improved Algorithm

The improved intraprocedural backward-slicing algorithm used in X is given as Alg. 2. The inputs to Alg. 2 are the SDG of the binary, the node n_G from which to slice, and the global slicing criterion $S_G^\#$ given as a set of a-locs. For any given node n in the SDG, n.instruction is the instruction associated with n, and n.semantics denotes the sub-portion of the formula for the semantics of n.instruction that is included in the slice; n.sliceBefore and n.sliceAfter are sets of a-locs denoting the local slicing criteria before and after n, respectively. Alg. 2 initializes n.semantics to *Id* (Line 4) because initially, the slice does not include any sub-portion of any instruction. (Recall from Obs. 2 that *Id* transforms the state the least.) Alg. 2 also initializes n.sliceBefore and n.sliceAfter to empty sets (Lines 2–3).

Alg. 2 uses a worklist to process the nodes in the SDG. Given a worklist item n, and the VSA state at n (obtained using n.VSAState), Alg. 2 performs the following steps during each worklist iteration:

1. Alg. 2 computes n.sliceAfter as the union of the current n.sliceAfter and the sliceBefore sets of n's intraprocedural data and control successors, which are obtained using n.successors (Lines 11–13).
2. Alg. 2 includes entire control nodes (nodes containing jump instructions) when included downstream nodes are control dependent on the control nodes (Lines 16–18).
3. The a-locs that are in both n.sliceAfter and the kill set of n.instruction are added to the local slicing criterion $S^\#$ (Line 20).
4. If $S^\#$ is empty, then n.instruction kills locations that are irrelevant to the local slicing criterion. Alg. 2 does not add such nodes to the slice (Lines 21–23).
5. Alg. 2 uses `GrowProjection` to "grow" n.semantics just enough to kill $S^\#$, while using as few locations as possible (Line 24).
6. If n.semantics "grew" in the previous step, Alg. 2 computes n.sliceBefore as the use set of the new n.semantics, and adds n.predecessors to the worklist (Lines 26–30). A predecessor that kills locations that are irrelevant to the updated n.sliceBefore will be discarded in Step 4 on a subsequent iteration.

One can see that n.semantics transforms the state more each time it gets updated. When the algorithm terminates, only nodes that are transitively relevant to the global slicing criterion $S_G^\#$ are included in the slice, and for each node n included in the slice, n.semantics gives the sub-portion of the semantics of n.instruction that is transitively relevant to $S_G^\#$.

Theorem 1. ***Termination.*** *Alg. 2 terminates.*

Proof. Suppose that φ is the QFBV formula for the instruction associated with a node n in the SDG, and σ_{VSA} is the VSA state at n. By Obs. 3, $\emptyset \subseteq n.\text{sliceBefore} \subseteq \text{USE}^\#(\varphi, \sigma_{VSA})$, and $\emptyset \subseteq n.\text{sliceAfter} \subseteq \text{KILL}^\#(\varphi, \sigma_{VSA})$. $\text{USE}^\#$ and $\text{KILL}^\#$ sets are finite sets of a-locs; thus, the height of the local slicing-criteria lattice is finite. Also, the computations of sliceBefore and sliceAfter sets are monotonic.

- sliceAfter: In Line 12 of Alg. 2, the new n.sliceAfter is computed as the union of the current n.sliceAfter and the sliceBefore sets of n's successors. Because set union in monotonic, if the sliceBefore sets of n's successors grow, n.sliceAfter also grows.
- sliceBefore: If n.sliceAfter grows, $S^\#$ might grow. Thus `GrowProjection` grows the current n.semantics using more conjuncts such that the new n.semantics kills the larger $S^\#$. As a result, the $\text{USE}^\#$ set of the new n.semantics—and thus the new n.sliceBefore—is larger.

sliceBefore and sliceAfter of all SDG nodes are initialized to $\emptyset$ (except n_G.sliceAfter, which is initialized to the global slicing criterion), and they become larger with each iteration until Alg. 2 reaches a fixed point. Consequently, Alg. 2 terminates. □

Theorem 2. ***Soundness.*** *If the SDG of a binary B is sound, Alg. 2 computes a sound slice with respect to the global slicing criterion* $S_G^\#$.

Proof. Suppose that φ is the QFBV formula of the instruction associated with a node n in the SDG. During worklist iterations, Alg. 2 replaces n.semantics with projection sub-formulas of φ that are kill-restricted with respect to the local slicing criterion $S^\#$, and such kill-restricted projection sub-formulas always kill $S^\#$ (by Defn. 3). Consequently, all instruction sub-portions that might transitively affect $S_G^\#$ are included in the final slice computed by Alg. 2, and the slice is sound. □

Theorem 3. ***Precision.*** *Suppose that Slice′(SDG, $S_G^\#$) is the imprecise backward slice of the SDG of binary B with respect to global slicing criterion $S_G^\#$ obtained using CodeSurfer/x86, and Slice(SDG, $S_G^\#$) is the backward slice of the SDG of B with respect to global slicing criterion $S_G^\#$ obtained using Alg. 2. Then Slice(SDG, $S_G^\#$) ⊆ Slice′(SDG, $S_G^\#$).*

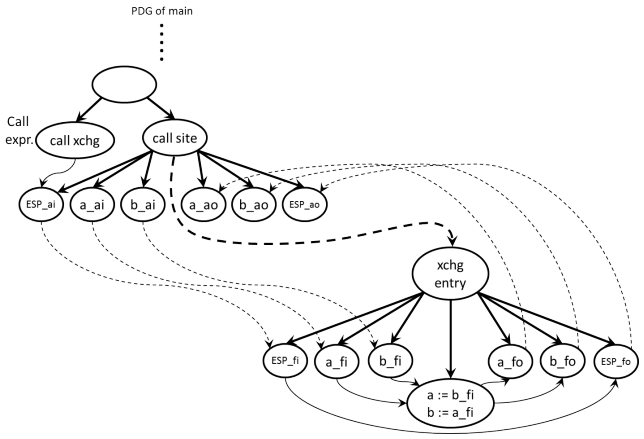


Figure 10: An SDG snippet with the body of xchg, and a call-site to xchg.

Proof. If `GrowProjection` were to return φ in Line 24 of Alg. 2, Alg. 2 would compute Slice$'$(SDG, $S_G^{\#}$). However, because `GrowProjection` returns a projection sub-formula ψ of φ, by Obs. 3, Slice(SDG, $S_G^{\#}$) $\subseteq$ Slice$'$(SDG, $S_G^{\#}$). □

The maximum number of times a node n is processed on lines 27–29 of Alg. 2 is bounded by the number of times n.semantics can grow, which in turn is bounded by the number of conjuncts m in an IA-32 instruction. With the exception of the x86 string instructions (instructions with the `rep` prefix), m is a constant. Thus lines 27–29 execute at most $O(N)$ times. Line 29 involves a constant amount of work per predecessor of n (assuming that sets are implemented with hash tables and the cost of a single lookup, insert, or delete is expected-time $O(1)$). That is, the cost of line 29 is proportional to the number of incoming edges that n has. That number is not bounded because each node in the SDG can have an unbounded number of incoming edges. However, in total, the cost of line 29 is $O(E)$, where E is the number of edges in the SDG. Thus the running time of Alg. 2 is bounded by $O(E)$.

5.2 Interprocedural Slicing

In this sub-section, we describe the extensions to Alg. 2 to perform interprocedural backward slicing. It is straightforward to adapt the extensions to perform interprocedural forward slicing.

Recall from §2.2 that an SDG also has nodes for formal parameters of procedures, and actual parameters of procedure calls. Specifically, each call-site has an *actual-in* node for each actual parameter, and an *actual-out* node for each return value; the PDG of a procedure has a *formal-in* node for each formal parameter, and a *formal-out* node for each return value. Formal-in/out and actual-in/out nodes are also created for global variables and registers. Actual-in/out nodes are control dependent on the call-site node. Formal-in/out nodes are control dependent on the procedure-entry node. Data-dependence edges connect actual-in/out nodes with formal-in/out nodes. (Recall from §2.2 that the SDG also has interprocedural control-dependence edges between call-site nodes and procedure-entry nodes.) An example SDG snippet is shown in Fig. 10. In Fig. 10, control-dependence edges are bold, whereas data-dependence edges are thinner; interprocedural edges are dashed. Fig. 10 shows a call-site with a call to a procedure xchg, which has only one node in its body. Let us assume that the variables a and b are globals. xchg swaps the values in a and b using a single instruction. Except for ESP, we do not show the actual-in/out nodes, formal-in/out nodes, and edges for other registers. To perform interprocedural slicing, X also follows interprocedural edges in addition to intraprocedural edges.

5.2.1 Context Sensitivity

A *context-sensitive* interprocedural-slicing algorithm ensures that infeasible paths involving mismatched calls and returns are not included in the slice. The classical approach is based on *context-free language* (CFL) reachability [19]. Operationally, the SDG is augmented with *summary edges* [14]. Summary edges capture dependence summaries between actual-in and actual-out nodes. Because xchg uses only one instruction to use and kill both global variables a and b, there are summary edges between (i) the actual-ins for a and b, and the actual-out for a, (ii) the actual-ins for a and b, and the actual-out for b, and (iii) the actual-in and actual-out for ESP. Once summary edges are added to the SDG, the slicing algorithm computes a context-sensitive interprocedural slice in two phases:

- In the first phase, the algorithm grows the slice without descending into procedures by "stepping across" call-sites via summary edges. Bodies of called procedures are not included in the slice in the first phase.
- In the second phase, the algorithm descends into procedures, but does not ascend back at call-sites. Bodies of called procedures are incorporated into the slice in the second phase.

Note that the dependences captured by summary edges do not depend on the slicing criterion. Consequently, the set of summary-edge predecessors of an actual-out n includes all the actual-ins that might affect n regardless of the local slicing criterion after n. For example, for the call-site shown in Fig. 10, the summary actual-ins for the actual-outs for a and b are given below.

$$\mathit{SummaryPreds}(\{a\_ao\}) = \{a\_ai,\ b\_ai\}$$
$$\mathit{SummaryPreds}(\{b\_ao\}) = \{a\_ai,\ b\_ai\}$$

To make such an approach work for X, we would need *summary transformers* instead of summary edges. For example, for the call-site in Fig. 10, we would like to have the following transformers:

$$\langle\{a\_ao\}, \{a\}\rangle \rightarrow \langle\{b\_ai\}, \{b\}\rangle$$
$$\langle\{a\_ao\}, \{\mathit{other\ a\text{-}locs}\}\rangle \rightarrow \langle\emptyset, \emptyset\rangle$$
$$\langle\{b\_ao\}, \{b\}\rangle \rightarrow \langle\{a\_ai\}, \{a\}\rangle$$
$$\langle\{b\_ao\}, \{\mathit{other\ a\text{-}locs}\}\rangle \rightarrow \langle\emptyset, \emptyset\rangle$$

For each call-site, one could tabulate the summary transformers (à la [23]) as they get computed. However, one might need such tabulated facts for every subset of a-locs that could arise at a call-site. Given that the number of a-locs in a program can be huge, this approach might incur a lot of overhead.

The approach X uses for context-sensitive interprocedural slicing is, in effect, equivalent to slicing an SDG in which procedures have been expanded inline. (Inline expansion does not work for recursive procedures; see below for how X handles recursion.)

X performs "virtual inlining" by qualifying each node n with the calling context under which X visits n, and maintaining a call-stack to keep track of the current calling context. This way, X can differentiate between the different calling contexts under which it visits n. X reslices the procedures for different calling contexts. Consequently, X records different n.semantics, n.sliceBefore, and n.sliceAfter for different calling contexts of n (i.e., n.semantics from Alg. 2 becomes $\langle n, \mathit{context}\rangle$.semantics, n.sliceBefore becomes $\langle n, \mathit{context}\rangle$.sliceBefore, etc., where *context* is a unique context under which X visits n). Also, when X descends into a procedure body from a call-site C, the stack enables enables X to ascend back to C.

This simple approach of slicing using a callstack does not work for recursive procedures. If X descends from a call-site into the body of a recursive procedure, X could become stuck in an infinite loop. For this reason, at calls to recursive procedures, X "steps

across" the call by following the summary edges computed by CodeSurfer/x86. Following summary edges at calls to recursive procedures causes a slight decrease in slicing precision because summary edges do not take the slicing criterion into account, but allows X to slice programs with recursion. (It is exactly for this reason that we implemented *virtual* inline expansion in X, rather than *actual* inline expansion.)

The context-sensitive interprocedural-slicing algorithm in X has a minor defect—because the algorithm, in effect, is equivalent to slicing an SDG with procedures expanded inline, the algorithm might exhibit exponential behavior in the worst case.[3] However, in our experiments (§7), we did not observe this behavior.

6. Implementation

X uses CodeSurfer/x86 [5] to obtain the SDG for a binary, and the VSA state at each instruction in the binary. X uses Transformer Specification Language (TSL) [15] to obtain QFBV encodings of instructions. The concrete operational-semantics of the integer subset of IA-32 is written in TSL, and the semantics is reinterpreted to produce QFBV formulas [16]. X reinterprets the semantics φ expressed in QFBV logic with respect to a VSA state to compute $\text{USE}^{\#}(\varphi, \sigma_{VSA})$ and $\text{KILL}^{\#}(\varphi, \sigma_{VSA})$.

In CodeSurfer/x86, the abstract transformers for the analyses used to build an SDG are obtained using TSL [15, §4.2]. In principle, if one were to replace the IA-32 semantics written in TSL with the semantics of another ISA, one could instantiate X's toolchain for the new ISA.

7. Experiments

We tested X on binaries of open-source programs. Our experiments were designed to answer the following questions:

- In comparison to CodeSurfer/x86, what is the reduction in slice size caused by X?
- In the slices computed by X, how many entire instructions are included in the slice? (And how many instructions have only a sub-portion of their semantics included in the slice?)
- What instructions have only a portion of their semantics included in slices?
- Does X's algorithm exhibit exponential behavior for our test suite?

All experiments were run on a system with a quad-core, 3GHz Intel Xeon processor running Windows 7; however, X's algorithm is single-threaded. The system has 32 GB of memory.

Our test suite consists of IA-32 binaries of FreeBSD utilities [1]. Table 1 presents the characteristics of the applications. For each application, we selected one slicing criterion for a backward slice and one for a forward slice, respectively.

For backward slices, we selected as the slicing criterion one or more actual parameters of the final call to an output procedure (e.g., `printf`, `fwrite`, or any user-defined output procedure in the application). Only for `pr` did we deviate from this rule and instead chose a variable that gets used toward the end of the application, but is not passed to an output procedure as an actual parameter. Our rationale for choosing these slicing criteria was that variables that are printed toward the end of the application are likely to be important outputs computed by the application, and hence it would be interesting to see which instructions affect these variables.

For forward slices, we selected variables or sets of variables that were initialized near the beginning of the application.

[3] Consider a non-recursive program in which procedure p_k contains two calls to p_{k-1}; p_{k-1} contains two calls to p_{k-2}; etc. Finally, p_1 contains a node n. Because there are 2^{k-1} unique calling contexts for reaching n, node n could be processed 2^{k-1} times.

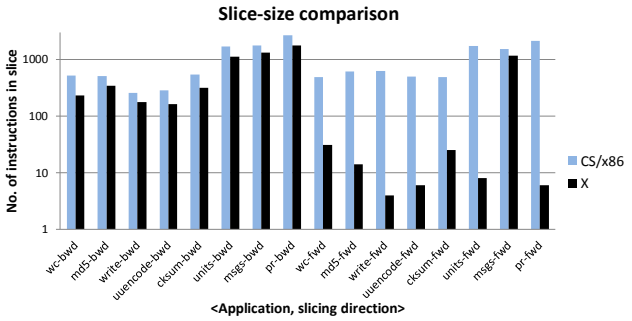
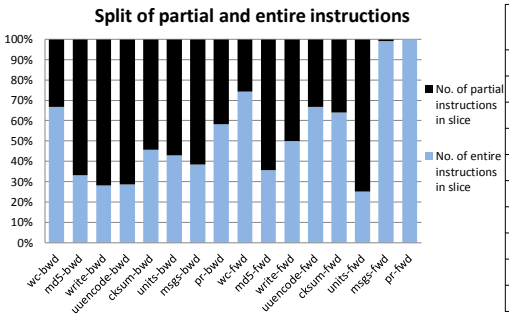


Figure 11: Comparison of sizes of slices computed by CodeSurfer/x86 and X (log-scale).



Opcode Variant	Count
push	780
add	578
sub	481
call	442
cmp	301
test	257
and	580
shl	26
idiv	13
sar	13

Figure 12: Split of partial and entire instructions in slices computed by X.

`units` and `msgs` each had one recursive procedure. The recursive procedure in `units` was included in the backward slice, but not included in the forward slice. The recursive procedure in `msgs` was included in both forward and backward slices.

To answer the first question, we computed the slices using X and CodeSurfer/x86. Fig. 11 shows the number of instructions included in each slice. Note that the y-axis uses a logarithmic scale. The average reduction in the slice sizes, computed as a geometric mean, is 36% for backward slices and 82% for forward slices. For the forward slices of `write`, `units` and `pr`, X reduces the number of instructions in the slice by over two orders of magnitude. The reduction in forward-slice sizes is more pronounced because the imprecision-causing idiom for forward slices (imprecision caused by the instruction marked ** in Fig. 6) occurs much more frequently in practice than the idiom for backward slices (the instruction marked by ** in Fig. 7). For `msgs`, the forward slice computed by X contains many instructions because a procedure call was control dependent on a node that was already in the slice. Because the call contains `push` instructions, updates to the stack pointer were added to the slice because of the control dependence, and consequently, downstream instructions that used the stack pointer were added to the slice.

To answer the second and third questions, for each slice computed by X, we computed the number of instructions in the slice that were included in their entirety for some context, and the number of instructions for which only a sub-portion was included in the slice for all contexts. Fig. 12 shows the results. For backward slices, it was surprising to see that, on an average, only 43% of the slice consisted of entire instructions, and the remaining were sub-portions of instructions. For forward slices, that number was 64.3%. We also identified the top ten opcode variants that constitute the instructions that were not included in their entirety in the slice. For the `push` opcode variant in the table in Fig. 12, either the stack-pointer update or the memory update was included in the slice. For the `call` opcode variant, the stack-pointer update was often the only update that was included in the slice. For the remaining

Table 1: Characteristics of applications in our test suite.

Application	LOC	No. of instructions	Backward-slicing criterion	Forward-slicing criterion
`wc`	295	765	Actual `twordct` of call to `printf` in `main`	Locals `linect`, `wordct`, and `charct` in `cnt`
`md5`	331	796	Actual `p` of final call to `printf` in `main`	Actual `optarg` of call to `MDString` in `main`
`write`	332	932	Actuals of final call to `do_write` in `main`	Local `atime` in `main`
`uuencode`	392	837	Actual `output` of call to `do_write` in `encode`	Initialization of global `mode` in `main`
`cksum`	505	790	Actual `len` in final call to `pcrc` in `main`	Local `lcrc` in `csum1`
`units`	783	2094	Actuals of final call to `showanswer` in `main`	Local `linenum` in `readunits`
`msgs`	951	2245	Actual `nextmsg` of final call to `fprintf` in `main`	Local `blast` in `main`
`pr`	2207	3980	Local `pagecnt` in `vertcol`	Local `eflag` in `vertcol`

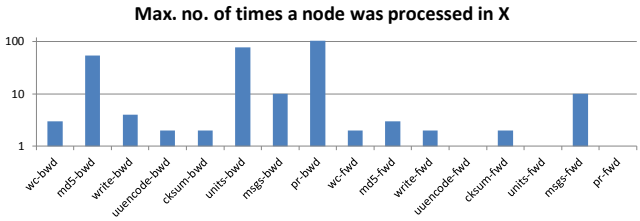


Figure 13: Maximum number of times a node was added to the worklist in X (log-scale).

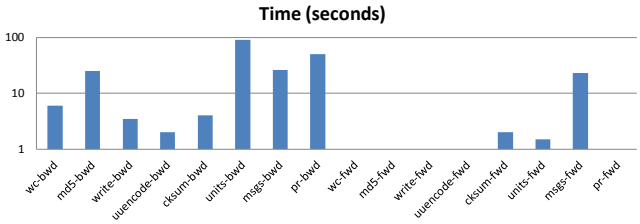


Figure 14: Time taken by X to compute the slice (log-scale).

opcode variants, a subset of flag updates was frequently excluded from the slice.

To answer the fourth question, for the computation of each slice, we counted the maximum number of times a node was processed, and the time taken to compute the slice. The results are shown in Fig. 13 and Fig. 14, respectively. Note that the y-axes use a logarithmic scale. X took the longest time to compute the backward slice for `units`. For computing the slice, the maximum number of times X processed a node on lines 27–29 of Alg. 2 was 78, and the maximum number of unique contexts observed for that node was 42. The number of procedures included in the slice was 17. `pr` was the application that had the second longest slice-computation time. For `pr`, the maximum number of unique contexts observed was 52, and the number of procedures included in the slice was 15. For these examples, one can see that the number of unique contexts is nowhere near exponential in the number of procedures.

8. Related Work

Slicing. The literature on program slicing is extensive [6, 17, 27]. Slicing has been—and continues to be—applied to many software-engineering problems [13]. For instance, recently there has been work on language-independent program slicing [7], which repeatedly creates potential slices through statement deletion, and tests the slices against the original program for semantics preservation. Specialization slicing [3] uses automata-theoretic techniques to produce specialized versions of procedures such that the output slice is an optimal executable slice without any parameter mismatches between procedures and call-sites. The slicing techniques discussed in the literature use an SDG or a suitable IR whose nodes typically contain a single update (and not a multi-assignment instruction). Consequently, the issue of including a "sub-portion" of a node never arises.

IRs for machine-code analysis. Apart from CodeSurfer/x86, the other IRs used for machine-code analysis include Vine [24], REIL [10], and BAP [9]. These IRs use a *Universal Assembly Language* (UAL) to represent the semantics of an instruction. (Typically, an instruction's semantics is a sequence of UAL updates.) The BAP and Vine platforms also support building an SDG for a binary. However, as in the case of CodeSurfer/x86, the nodes of the SDGs in these IRs are *entire* instructions, and thus these IRs also face the granularity issue during slicing.

UAL updates for an instruction i can be thought of as a linearization of the QFBV formula for i, and thus the techniques used by X can be easily adapted to perform more accurate slicing in those platforms. ("Sub-portions" of instructions would then be UAL updates instead of projection sub-formulas.)

Applications of more precise machine-code slicing. WIPER is a machine-code partial evaluator [25] that specializes binaries with respect to certain static inputs. As a first step to partial evaluation, WIPER performs *binding-time analysis* (BTA) to determine which instructions in the binary can be evaluated at specialization time. For BTA, WIPER uses CodeSurfer/x86's forward slicing. To sidestep the granularity issue, WIPER "decouples" the multiple updates performed by instructions that update the stack pointer along with another location (e.g., `push`, `pop`, `leave`, etc.). One can see that instruction decoupling is a sub-optimal solution to address the granularity issue because multi-assignment instructions that do not update the stack pointer also make the forward slice imprecise. X computes more accurate forward slices, and can be used in WIPER's BTA to increase BTA precision.

Taint trackers [18, 22] use dynamic analysis to check if tainted inputs from *taint sources* could affect *taint sinks*. Certain taint trackers such as Minemu [8] rely entirely on dynamic analysis to reduce taint-tracking overhead. X could be used to exclude from consideration portions of the binary that are not affected by taint sources, thereby further reducing taint-tracking overhead.

Conseq [29] is a concurrency-bug detection tool, which uses machine-code slicing to compute the set of critical reads that might affect a failure site. X could be used to compute more accurate backward slices, effectively reducing the number of critical reads that needs to be analyzed by Conseq.

9. Conclusion and Future Work

In this paper, we described an algorithm to slice machine code more accurately. We presented X, a tool that slices IA-32 binaries more precisely than a state-of-the-art tool. Our experiments on binaries of FreeBSD utilities show that, in comparison to slices computed by CodeSurfer/x86, our algorithm reduces the number of instructions in backward slices by 36%, and in forward slices by 82%. For some binaries in our test suite, X reduces the number of instructions in the slice by over two orders of magnitude.

A possible direction for future work is to incorporate X into the BTA of WIPER [25], and see if X increases the precision of BTA, leading to better specialization opportunities. A second direction is to incorporate X into a taint tracker such as Minemu [8] to exclude from consideration portions of the binary that are not affected by taint sources, and measure the reduction in taint-tracking overhead caused by this optimization. A third direction is to generate an executable from a backward slice S computed by X. By using a machine-code synthesizer [26], one can synthesize instruction sequences for the sub-portions of instructions included in S. This approach can be used to extract an executable component from a binary (e.g., a word-count program from the `wc` utility).

References

[1] http://www.opensource.apple.com/source/.

[2] P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *TSE*, 29(8), 2003.

[3] M. Aung, S. Horwitz, R. Joiner, and T. Reps. Specialization slicing. *TOPLAS*, 36(2), 2014.

[4] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 32(6), 2010.

[5] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*, 2005.

[6] D. Binkley and K. Gallagher. Program slicing. In *Advances in Computers, Vol. 43*. 1996.

[7] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In *FSE*, 2014.

[8] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *RAID*, 2011.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. BAP: A Binary Analysis Platform. In *CAV*, 2011.

[10] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*, 2009.

[11] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, 2013.

[12] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3), 1987.

[13] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ICSE*, 1992.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1), 1990.

[15] J. Lim and T. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *TOPLAS*, 35(4), 2013.

[16] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. *Softw. Tools for Tech. Transfer*, 13(1):61–87, 2011.

[17] G. Mund and R. Mall. Program slicing. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter 14. CRC Press, 2nd. edition, 2007.

[18] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[19] T. Reps. Program analysis via graph reachability. *Inf. and Softw. Tech.*, 40(11–12), 1998.

[20] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *FSE*, 1994.

[21] H. Saïdi. Logical foundation for static analysis: Application to binary static analysis for security. *ACM SIGAda Ada Letters*, 28(1):96–102, 2008.

[22] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, 2010.

[23] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[24] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Int. Conf. on Information Systems Security*, 2008.

[25] V. Srinivasan and T. Reps. Partial evaluation of machine code. In *OOPSLA*, 2015.

[26] V. Srinivasan and T. Reps. Synthesis of machine code from semantics. In *PLDI*, 2015.

[27] F. Tip. A survey of program slicing techniques. *JPL*, 3(3), 1995.

[28] M. Weiser. Program slicing. *TSE*, SE-10(4), 1984.

[29] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. Conseq: Detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.