An Improved Algorithm for 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 semantics, 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.

This paper presents an algorithm to slice machine code more accurately. To counter the granularity issue, our algorithm performs slicing at the microcode level, instead of the instruction level, and obtains a more precise microcode slice. To reconstitute a machine-code program from a microcode slice, our algorithm uses machine-code synthesis. 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 size of backward slices by 33%, and forward slices by 70%.

[*] Supported, in part, by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270, and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin 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. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

Keywords Program slicing, machine code, microcode-level SDG, granularity issue, program reconstitution, machine-code synthesis, IA-32 instruction set

1. Introduction

One of the most useful primitives in program analysis is *slicing* [18, 34]. 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] Backward slicing computes the set of program points that might affect the slicing criterion; forward slicing computes the set of program points that might be affected by the slicing criterion. 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 [29, §7], [4, §1], [10, §1], [14, §7]), which has led to the development and wider 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 [30], taint trackers [9], and fault localizers [35]. 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, 10] recover an *instruction-level system dependence graph (SDG)*[3] from a binary, and use an existing source-code slicing algorithm [18, 25] to perform slicing on the recovered SDG. (An instruction-level SDG is an SDG in which nodes are *entire* instructions.) However, the computed slices are extremely imprecise, often including the entire binary. Instructions in most Instruction Set Architectures (ISAs) such as IA-32 [19] and ARM [2] are *multi-assignments*: they have several inputs and several outputs (e.g., registers, flags, and

[1] In the literature, program point p and the variable set are called the *slicing criterion* [34]. In this paper, when we refer to a program point p as the "slicing criterion," we mean p and all the variables used at p.

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

[3] The SDG is an intermediate representation used for slicing; see §2.1.

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 microcode,[4] the slice is forced to include the entire instruction. This granularity issue can have a *cascade effect*: irrelevant microcode 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 on an instruction-level SDG yields imprecise machine-code 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 [30]. More precise slicing could also be used to reduce the overhead of taint trackers [9] 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 to extract an executable component from a binary, e.g., a word-count program from the WC utility. (See §6.1.) One could construct more accurate dependence models for libraries that lack source code by slicing the library binary. A machine-code slicer is a useful tool to have when a slicer for a specific source language is unavailable.

We have implemented our algorithm in a tool, called MCSLICE, which slices Intel IA-32 binaries. MCSLICE uses the instruction-level SDG recovered by an existing tool, CodeSurfer/x86 [5]. MCSLICE performs slicing at the microcode level instead of the instruction level. MCSLICE first converts the instruction-level SDG into a microcode-level SDG (μ-SDG): MCSLICE splits each node containing a multi-assignment instruction into multiple microcode nodes (each new node contains an individual assignment), and recomputes data-dependence edges between the newly created microcode nodes. MCSLICE uses quantifier-free bit-vector (QFBV) logic formulas to explicitly represent the microcode at each node. MCSLICE then uses an existing interprocedural context-sensitive slicing algorithm [18, 25] to slice over the constructed μ-SDG. The final slice includes only the microcode that is relevant to the slicing criterion.

Some clients of the slicing algorithm might require the results to be reported as *executable machine code* instead of a microcode slice (e.g., executable procedure/component extraction). This requirement introduces a new issue: how to *reconstitute* a machine-code program from a microcode slice; the slicing algorithm must now generate machine code, which is at a higher level, from the microcode fragments included in the slice. MCSLICE addresses the program-reconstitution issue via machine-code synthesis: MCSLICE uses an existing machine-code synthesizer [31] to synthesize machine code for the microcode in the slice. By this means, MCSLICE obtains an executable machine-code program from a precise microcode slice.

Contributions. The paper's contributions include the following:

- We identify the granularity issue caused by using source-code slicing algorithms on an instruction-level SDG, and show how the issue can lead to very imprecise machine-code slices (§3.1).
- We present an algorithm for machine-code slicing that is more precise than prior work. Our algorithm overcomes the granularity issue by converting an instruction-level SDG into a microcode-level SDG, and using an existing slicing algorithm over the microcode-level SDG (§4.1).
- We show how machine-code synthesis can be used to reconstitute a machine-code program from a microcode slice (§4.2).
- As a case study of an application of our slicing algorithm, we show how to use our improved slicer to extract an executable component from a binary (§6.1).
- Our algorithm uses QFBV formulas to represent microcode, and thus is not tied to a specific binary-analysis platform. Consequently, our algorithm can be used to improve slicing in other binary-analysis platforms that suffer from the granularity and program-reconstitution issues. (See §2.1.)

Our methods have been implemented in an IA-32 slicer called MCSLICE. We present experimental results with MCSLICE, which show that, on average, MCSLICE reduces the sizes of slices obtained from a state-of-the-art tool by 33% for backward slices, and 70% for forward slices.

2. Background

In this section, we briefly describe how state-of-the-art tools recover from a binary an SDG on which to perform machine-code slicing (§2.1), and a logic to express the semantics of IA-32 instructions (§2.2).

2.1 SDG Recovery and Slicing for Machine Code

Slicing is typically performed using an Intermediate Representation (IR) of the binary called a *system dependence graph* (SDG) [15, 18]. To build an SDG for a program, one needs to know the set of variables that might be used and

[4] In this paper, we use the term "microcode" as a synonym for a specification of an instruction's concrete operational-semantics.

```
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 1: VSA state before each instruction in a small code snippet, and the USE$^{\#}$ and KILL$^{\#}$ sets for each instruction.

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 CodeSurfer/x86 [5] (a state-of-the-art tool for machine-code analysis) recovers "variable-like" abstractions from a binary, and uses 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*) holds an over-approximation of the values that each a-loc can have at a given program point. For example, Fig. 1 shows the VSA state before each instruction in a small IA-32 code snippet. In Fig. 1, 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 abstract state to compute USE$^{\#}$(i, σ_{VSA}) (KILL$^{\#}$(i, σ_{VSA})), which is the set of a-locs that might be used (modified) by *i*. The USE$^{\#}$ and KILL$^{\#}$ sets for each instruction in the code snippet are also shown in Fig. 1.

To perform VSA, use/kill analysis, and other analyses, CodeSurfer/x86 internally uses a specification of the concrete operational-semantics of IA-32 instructions written in the Transformer Specification Language (TSL) [20]. Writing a TSL specification for IA-32 instructions is similar to writing an IA-32 interpreter in first-order ML. (The TSL specification for the instruction `add eax,ebx` is given as Fig. 2.) CodeSurfer/x86 reinterprets the TSL specification of an instruction *i*'s semantics to create different abstract transformers for *i*, which can be used in different analyses.

CodeSurfer/x86 uses the results of VSA and use/kill analysis to build a collection of IRs, including an SDG and a control-flow graph (CFG). 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 [15]. For example, in the system-dependence subgraph for the code snippet in Fig. 1, 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 the stack-pointer register ESP, to instructions `2` and `3`, which use ESP, as well as from instruction `3` to instruction `4`.

```
reg : EAX | EBX | ...
flag : ZF | SF | ...
instruction : ADD(EAX,EBX) | ...
state : State(MAP[reg, INT32],  //registers
              MAP[flag, BOOL],  //flags
              MAP[INT32, INT8]) //memory
state interpInstr(instruction I,state S) {
 with (S) (
  State(regs, flags, memory) :
   with (I) (
    ADD(EAX,EBX) :
     let v1 = regs[EAX];
     v2 = regs[EBX];
     res = v1+v2;
     regs1 = regs[EAX ↦ res];
     flags1 = flags[ZF ↦ res == 0];
     flags2 = flags1[SF ↦ res <s 0];
     flags3 = flags2[CF ↦ −1∗(v1−1) <u v2];
     flags4 = flags3[AF ↦ −1∗(−16 | v1 & 15)
               −1 <u v2 & 15];
     flags5 = flags4[OF ↦ (v1 >=s 0 &&
              v2 >=s 0 || EAX <s 0) &&
              (EAX >=s 0 && res <s 0 ||
              v1 <s 0 && res >=s 0)];
     flags6 = flags5[PF ↦ ((res & 255 ˆ
              (res & 255) >>l 1 ˆ (res & 255 ˆ
              (res & 255) >>l 1) >>l 2 ˆ
              (res & 255 ˆ (res & 255) >>l 1 ˆ
              (res & 255 ˆ (res & 255) >>l 1)
              >>l 2) >>l 4) & 1) == 0];
     in State(regs1, flags6, memory),
   ...
))
}
```

Figure 2: TSL specification for the instruction `add eax,ebx`.

```
addr 0x0 @asm "add %eax,%ebx"
label pc_0x0
t:u32 = R_EBX:u32
R_EBX_74:u32 = R_EBX:u32 + R_EAX:u32
R_CF:bool = R_EBX_74:u32 ¡ t:u32
temp:u32 = R_EBX_74:u32 ∧ t:u32
temp_77:u32 = temp:u32 ∧ R_EAX:u32
temp_78:u32 = 0x10:u32 & temp_77:u32
R_AF:bool = 0x10:u32 == temp_78:u32
temp_80:u32 = ∼R_EAX:u32
temp_81:u32 = t:u32 ∧ temp_80:u32
temp_82:u32 = t:u32 ∧ R_EBX_74:u32
temp_83:u32 = temp_81:u32 & temp_82:u32
R_OF:bool = high:bool(temp_83:u32)
temp_85:u32 = R_EBX_74:u32 ≫ 7:u32
temp_86:u32 = R_EBX_74:u32 ≫ 6:u32
temp_87:u32 = temp_85:u32 ∧ temp_86:u32
temp_88:u32 = R_EBX_74:u32 ≫ 5:u32
temp_89:u32 = temp_87:u32 ∧ temp_88:u32
temp_90:u32 = R_EBX_74:u32 ≫ 4:u32
temp_91:u32 = temp_89:u32 ∧ temp_90:u32
temp_92:u32 = R_EBX_74:u32 ≫ 3:u32
temp_93:u32 = temp_91:u32 ∧ temp_92:u32
temp_94:u32 = R_EBX_74:u32 ≫ 2:u32
temp_95:u32 = temp_93:u32 ∧ temp_94:u32
temp_96:u32 = R_EBX_74:u32 ≫ 1:u32
temp_97:u32 = temp_95:u32 ∧ temp_96:u32
temp_98:u32 = temp_97:u32 ∧ R_EBX_74:u32
temp_99:bool = low:bool(temp_98:u32)
R_PF:bool = ∼temp_99:bool
R_SF:bool = high:bool(R_EBX_74:u32)
R_ZF:bool = 0:u32 == R_EBX_74:u32
```

Figure 3: BIL code for the instruction `add eax,ebx` [11]. BIL is the UAL used in BAP.

In a PDG, a procedure call is associated with two nodes: a *call-expression* node, which contains the `call` instruction, and a *call-site* node, which is a control node. 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. 12 for an example SDG with interprocedural edges.)

CodeSurfer/x86 uses an existing interprocedural-slicing algorithm [18, 25] to perform machine-code slicing on the recovered SDG.

Other platforms for machine-code slicing. Apart from CodeSurfer/x86, there are other machine-code analysis platforms, such as Vine [29], REIL [13], and BAP [10]. Vine and BAP perform VSA, and recover an SDG from a binary, on which slicing can be done.

These platforms use *Universal Assembly Language* (UAL) to represent the semantics of instructions. (Typically, an instruction's microcode is a sequence of UAL updates—see Fig. 3.) The SDG recovered by BAP and Vine is similar to the one recovered by CodeSurfer/x86 in that nodes of the SDG are *entire instructions*, and not individual UAL updates. Because of the program-reconstitution issue—and because the “semantic gap” between instructions and microcode could potentially confuse users—BAP and Vine report information at the entire-instruction level. (If results were reported at the microcode level, it would be a bit like having a source-level slicing tool report its results at the machine-code level.) Consequently, BAP and Vine also face the granularity and program-reconstitution issues during slicing.

One can think of UAL as a flattened variant of the QFBV representation of microcode used in MCSLICE. Consequently, the techniques presented in this paper can be applied in a straightforward manner to other binary-analysis platforms that use UAL. In particular, because our work provides a solution to the program-reconstitution issue, it provides a way to solve the analogous problem—and thereby improve—UAL-based systems.

2.2 QFBV Formulas for IA-32 Instructions

The operational semantics (microcode) of IA-32 instructions can be expressed formally by QFBV formulas. MCSLICE uses QFBV formulas as the explicit representation of microcode in SDG nodes. We chose QFBV formulas to represent microcode because of the following reasons:

- QFBV provides a standard way of specifying microcode that is not tied to a specific binary-analysis platform.
- Conversion of microcode specified in TSL, BIL, etc., to QFBV formulas is straightforward, and encoders that perform this conversion are readily available. Consequently, it is straightforward to use our technique with any binary-analysis platform.
- The use of QFBV allows MCSLICE to be coupled with a machine-code synthesizer [31], which synthesizes an instruction sequence from a QFBV formula. This approach allows MCSLICE to reconstitute machine-code programs from microcode slices.
- Usage of QFBV allows MCSLICE to be extended in the future to use SMT-based techniques for constructing more accurate SDGs.

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. 4. 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., [26]), 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 [21]). 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

$$\bigwedge_m (I'_m = T_m) \wedge \bigwedge_n (J'_n = \varphi_n) \wedge Mem' = FE, \qquad (1)$$

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. The QFBV formulas for the instructions in Fig. 1 are given below. (To reduce clutter, we pretend that the `sub` instruction sets only the zero flag ZF.)

$$\langle\!\langle \texttt{push ebp} \rangle\!\rangle \equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto EBP]$$
$$\langle\!\langle \texttt{mov ebp,esp} \rangle\!\rangle \equiv EBP' = ESP$$
$$\langle\!\langle \texttt{sub esp,10} \rangle\!\rangle \equiv ESP' = ESP - 10 \wedge ZF' = ((ESP - 10) = 0)$$
$$\langle\!\langle \texttt{mov [esp],1} \rangle\!\rangle \equiv Mem' = Mem[ESP \mapsto 1]$$

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 $Mem' = Mem$ for constants and functions that are *unmodified*.

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

Figure 4: Syntax of L[IA-32].

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

Figure 5: Source code for the diff program, and the backward slice with respect to the return value of `main`.

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 MCSLICE.

```
add:
1:  push ebp
2:  mov ebp,esp
3:  sub 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

square:
10: push ebp
11: mov ebp,esp
12: sub esp,4
13: mov eax,[ebp+8]
14: imul eax,[ebp+8]
15: mov [ebp-4],eax
16: mov eax,[ebp-4]
17: leave
18: ret
```

```
main:
19: push ebp
20: mov ebp,esp
21: sub esp,16
22: mov [ebp-16],10
23: mov [ebp-12],20
24: push [ebp-12]
25: push [ebp-16]
26: call add
27: add esp,8
28: mov [ebp-8],eax
29: push [ebp-8]
30: call square
31: add esp,4
32: mov [ebp-4],eax
33: mov eax,[ebp-16]
34: mov ebx,[ebp-12]
35: sub eax,ebx
36: leave
37: ret
```

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

3.1 Granularity Issue in Machine-Code Slicing

Consider the C program diff shown in Fig. 5. The `main` function contains calls to functions `add` and `square`. `main` does not use the 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. 5). The backward slice with respect to `main`'s return value gets us the desired result. The backward slice is highlighted in gray in Fig. 5. (This source-code slice is computed using CodeSurfer/C [1].)

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. 6. The corresponding slicing criterion is the boxed instruction in Fig. 6. (The EAX register holds the return value of `main` at the end of the program. Hence, we slice with respect to the final assignment to EAX, which is performed by the boxed instruction in Fig. 6.) The backward slice with respect to the slicing criterion includes the lines highlighted in gray in Fig. 6. (The slice is computed using CodeSurfer/x86.) One can see that the 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, several outputs (e.g., registers, flags, and memory locations), and several microcode updates. 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 microcode updates, the slicing algorithm is forced to include the entire instruction. For example, consider instruction `29` in Fig. 6 whose QFBV formula is

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

```
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 = 30;
  int e = square(d);
  return e;
}
```

Figure 7: Source code for the square program, and the forward slice with respect to `a`.

```
add:
1:  push ebp
2:  mov ebp,esp
3:  sub 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

square:
10: push ebp
11: mov ebp,esp
12: sub esp,4
13: mov eax,[ebp+8]
14: imul eax,[ebp+8]
15: mov [ebp-4],eax
16: mov eax,[ebp-4]
17: leave
18: ret
```

```
main:
19: push ebp
20: mov ebp,esp
21: sub esp,20
22: mov [ebp-20],10
23: mov [ebp-16],20
24: push [ebp-16]
25: push [ebp-20]
26: call add
27: add esp,8
28: mov [ebp-12],eax
29: mov [ebp-8],30
30: push [ebp-8]
31: call square
32: add esp,4
33: mov [ebp-4],eax
34: mov eax,[ebp-4]
35: leave
38: ret
```

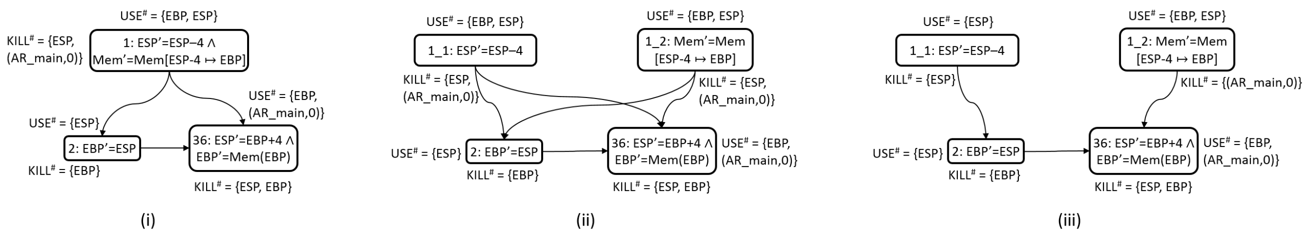
Figure 8: Assembly listing for square with the imprecise forward slice computed by CodeSurfer/x86.

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 instruction `10` in Fig. 6 whose formula is

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

The instruction uses the registers ESP and EBP. When the slice ascends back into `main`, it requires the definition of ESP from instruction `29` in Fig. 6. 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 instruction `28` in Fig. 6. That instruction also uses the value in register EAX, which holds the return value of the `add` function. For this reason, instruction `28`, and the entire body of `add`, which are completely irrelevant to the slicing criterion, are included in the slice. The granularity issue thus has a cascade effect—irrelevant instructions included in the slice cause more irrelevant instructions to be included in the slice.

Consider another C program square, shown in Fig. 7. The `main` function contains calls to functions `add` and `square`. Suppose that we want to compute the program points that are affected by the local variable `a` (boxed in Fig. 7). The forward slice of the source code with respect to `a`, highlighted in


USE# = {EBP, ESP}
KILL# = {ESP, (AR_main,0)}
1: ESP'=ESP–4 ∧ Mem'=Mem[ESP-4 ↦ EBP]
USE# = {EBP, (AR_main,0)}
USE# = {ESP}
2: EBP'=ESP
KILL# = {EBP}
36: ESP'=EBP+4 ∧ EBP'=Mem(EBP)
KILL# = {ESP, EBP}
(i)
USE# = {EBP, ESP}
1_1: ESP'=ESP–4
USE# = {EBP, ESP}
1_2: Mem'=Mem [ESP-4 ↦ EBP]
KILL# = {ESP, (AR_main,0)}
KILL# = {ESP, (AR_main,0)}
USE# = {ESP}
2: EBP'=ESP
KILL# = {EBP}
36: ESP'=EBP+4 ∧ EBP'=Mem(EBP)
USE# = {EBP, (AR_main,0)}
KILL# = {ESP, EBP}
(ii)
USE# = {ESP}
1_1: ESP'=ESP–4
KILL# = {ESP}
USE# = {EBP, ESP}
1_2: Mem'=Mem [ESP-4 ↦ EBP]
KILL# = {(AR_main,0)}
USE# = {ESP}
2: EBP'=ESP
KILL# = {EBP}
36: ESP'=EBP+4 ∧ EBP'=Mem(EBP)
USE# = {EBP, (AR_main,0)}
KILL# = {ESP, EBP}
(iii)
Figure 9: SDG snippet illustrating the construction of μ-SDG.

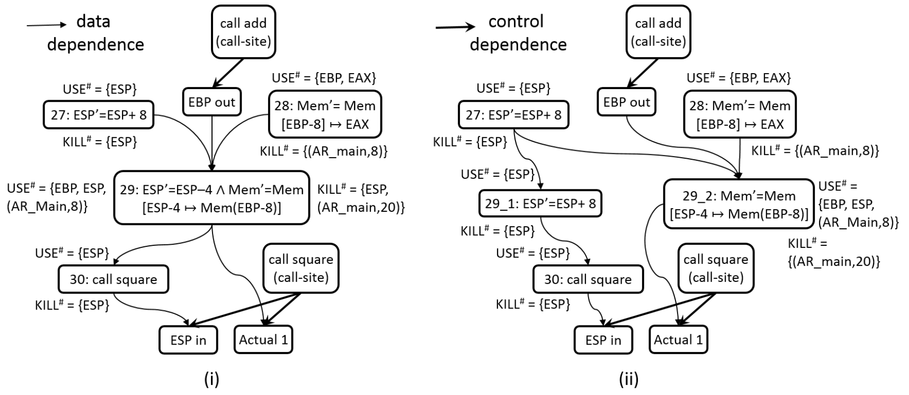


Figure 10: SDG snippet illustrating the construction of μ-SDG.

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

Figure 11: Code generated by MCSLICE from the microcode backward slice for diff. Highlighted instructions are created by machine-code synthesis.

gray in Fig. 7, gets us the desired result. The machine-code forward slice with respect to the analogous slicing criterion (boxed instruction in Fig. 8) includes the lines highlighted in gray in Fig. 8. One can see that the entire body of the `square` function—which is completely irrelevant to the slicing criterion—is included in the slice.

The imprecision creeps in at instruction `25` in Fig. 8. The QFBV formula for the instruction is

$$\langle\!\langle \texttt{push [ebp-20]} \rangle\!\rangle \equiv ESP' = ESP - 4 \;\wedge$$
$$Mem' = Mem[ESP - 4 \mapsto Mem(EBP - 20)].$$

The instruction stores the contents of the memory location whose address is EBP − 20 in a new memory location, and updates the stack-pointer register ESP. The slice only requires the microcode update that uses the location EBP − 20. However, because of the granularity issue, the slice also includes the microcode update to ESP. Because all the downstream instructions directly or transitively use ESP, the forward slice includes all the downstream instructions.

In both examples, the root cause of the imprecision is the `push` instruction. (In our evaluation, we found that the `push` instruction caused the second-highest amount of imprecision in slices computed by CodeSurfer/x86—see Fig. 18.) In both examples, the imprecision creeps in because the slice ends up including both microcode updates that are part of the semantics of `push`, instead of including the only update that actually had to be included in the slice. (In the backward-slicing example, only the update to the stack pointer ESP actually had to be included in the slice; in the forward-slicing example, only the update to a memory location actually had to be included in the slice.)

3.2 Improved Machine-Code Slicing in MCSLICE

Given the (i) instruction-level SDG of a binary, and (ii) the slicing criterion (SDG node to slice from), MCSLICE uses the following steps to compute a more accurate slice:

1. MCSLICE converts the instruction-level SDG into a microcode-level SDG (μ-SDG): MCSLICE splits each SDG node containing a multi-assignment instruction into multiple nodes, each containing a single microcode update, and recomputes data-dependence edges between the newly created nodes. A μ-SDG is just a variant of an SDG in which some instruction nodes are replaced by microcode nodes.
2. MCSLICE uses an existing interprocedural context-sensitive slicing algorithm [18, 25] to compute the slice over the μ-SDG. The final slice includes only the microcode updates that are relevant to the slicing criterion.
3. MCSLICE uses an existing machine-code synthesizer [31] to reconstitute a machine-code program from the microcode slice.

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

To convert the given instruction-level SDG into a μ-SDG, MCSLICE first identifies nodes with non-control-modifying multi-assignment instructions. Fig. 9(i) shows the SDG snippet[5] for the first few instructions in function `main` in the diff

[5] In Fig. 9 and the remaining SDGs and μ-SDGs in the paper, we label each node with its microcode. To facilitate correspondence between SDGs/μ−SDGs and assembly listings, we also include the instruction num-

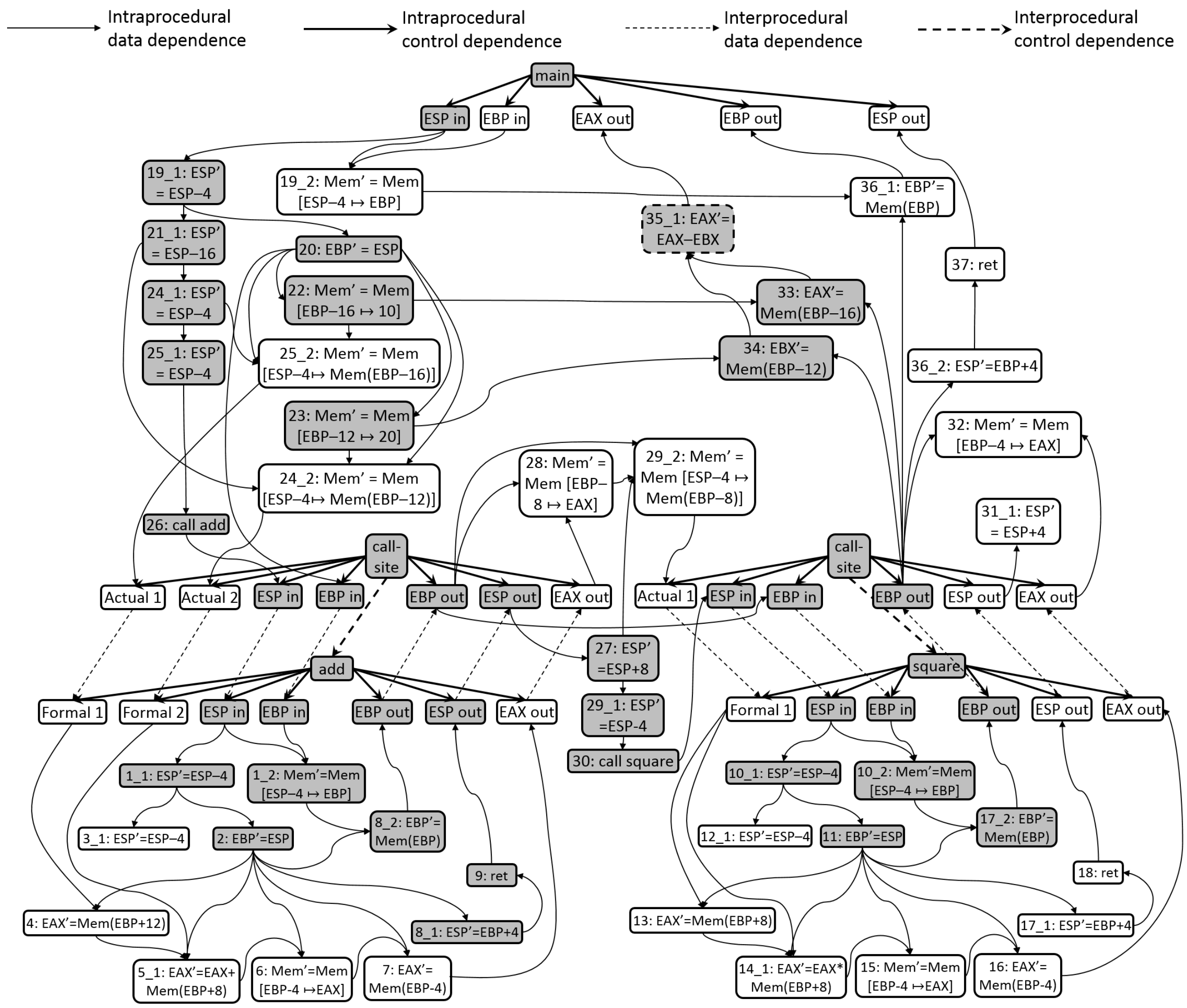


Figure 12: Microcode slice over the μ-SDG for the diff program. The slicing criterion is node `35`, which is indicated by the dashed box.

program, along with their respective USE$^{\#}$ and KILL$^{\#}$ sets. Instruction `1` is a multi-assignment instruction with two independent microcode updates. MCSLICE splits node `1` into two nodes `1_1` and `1_2`, each containing a single microcode update (Fig. 9(ii)). Initially, nodes `1_1` and `1_2` inherit the USE$^{\#}$ and KILL$^{\#}$sets, and the dependence edges of the parent node. MCSLICE then recomputes the USE$^{\#}$ and KILL$^{\#}$ sets of each newly created node n by projecting n's existing sets based on the individual microcode update in n. MCSLICE then recomputes the data-dependence edges based on the updated USE$^{\#}$ and KILL$^{\#}$ sets to create the final μ-SDG. (To recompute data-dependence edges, MCSLICE performs reaching-definitions analysis, for which it also uses the CFG built by CodeSurfer/x86 as an additional input—see §4.1.) The μ-SDG snippet, along with the updated USE$^{\#}$ and KILL$^{\#}$ sets, is shown in Fig. 9(iii).

ber in each node's label. To reduce clutter, we do not show microcode nodes corresponding to flag updates in μ-SDGs.

We illustrate MCSLICE's μ-SDG construction algorithm on another SDG snippet. Fig. 10(i) shows the SDG snippet for the instructions that were used to illustrate the granularity issue for backward slicing in §3.1. Node `29` uses register ESP (defined by the instruction in node `27`) and the value in memory location EBP $-$ 8 (defined by the instruction in node `28`). Node `29` defines register ESP, which flows to the instruction in node `30` (and then to the actual-in node ESP in), and a memory location, which flows to the actual-in node Actual 1. MCSLICE splits node `29` into two nodes `29_1` and `29_2`, computes the new USE$^{\#}$ and KILL$^{\#}$ sets for `29_1` and `29_2`, and recomputes the data-dependence edges. The final μ-SDG snippet, along with the new USE$^{\#}$ and KILL$^{\#}$ sets, is shown in Fig. 10(ii).

MCSLICE computes the remainder of the μ-SDG in a similar manner, and the final μ-SDG for the diff program is given as Fig. 12. (To reduce clutter in Fig. 12 and Fig. 13, some intraprocedural control-dependence edges have been omitted. The omitted edges do not cause additional nodes to

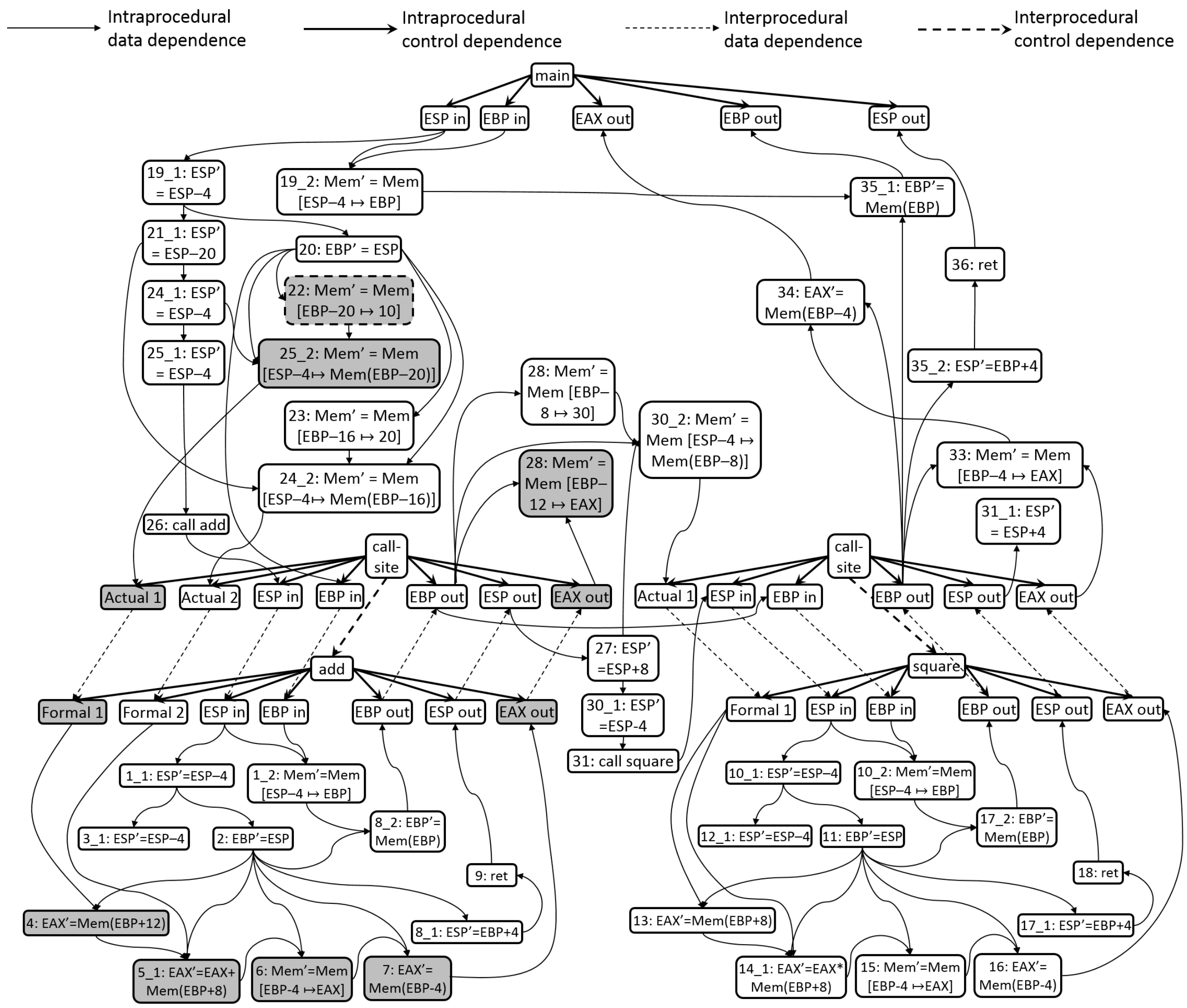


Figure 13: Microcode slice over the μ-SDG for the square program. The slicing criterion is node `12`, which is indicated by the dashed box.

be included in the slice.) McSlice now computes the backward slice over the μ-SDG with node `35_1` as the slicing criterion. (Recall from §3.1 that we are interested in the program points that might affect the return value of `main`, which is available in the register EAX.) The nodes in the backward slice are highlighted in gray in Fig. 12. Among the nodes included in the slice, nodes 22, 23, 33, 34, and 35_1 directly affect the return value of `main`, and the remaining nodes set up the stack-pointer register ESP and frame-pointer register EBP for downstream nodes. One can see that the slice computed by McSlice is more precise than the backward slice computed by CodeSurfer/x86 (cf. Fig. 6).

McSlice reconstitutes a machine-code program from the microcode slice by synthesizing machine-code instructions for each microcode node included in the slice. (If all the microcode nodes corresponding to an old instruction node are included in the microcode slice, McSlice simply reuses the instruction in the old node.) For synthesis purposes, McSlice uses McSynth, a machine-code synthesizer that synthesizes machine-code instructions from a QFBV formula. The machine-code program produced by McSlice from the microcode slice in Fig. 12 is shown in Fig. 11. (To obtain *executable* code from a backward microcode slice, McSlice performs a few additional steps—see §4.2.)

McSlice computes forward slices in a similar manner: McSlice converts the input instruction-level SDG into a μ-SDG, and uses an existing slicing algorithm to compute the forward slice. For example, consider the square program from §3.1. The μ-SDG created by McSlice for the program is given as Fig. 13. McSlice computes the forward slice over the μ-SDG with node `22` as the slicing criterion. (Recall from §3.1 that we are interested in the program points that might be affected by the local variable `a` in `main`.) The nodes in the forward slice are highlighted in gray in Fig. 13. One can see that the slice computed by McSlice is more precise than the forward slice computed by CodeSurfer/x86 (cf. Fig. 8).

4. Algorithm

In this section, we describe the slicing algorithm used in McSlice. First, we describe how McSlice converts an instruction-level SDG into a μ-SDG on which slicing can be done (§4.1). Then, we describe how McSlice reconstitutes an executable machine-code program from a microcode slice (§4.2).

4.1 Construction of μ-SDG and Slicing

In this sub-section, we present the algorithm that McSlice uses to build a μ-SDG for microcode slicing. Apart from working with the SDG, the algorithms in this sub-section also work with the CFGs of the procedures in the binary because the algorithm for μ-SDG construction performs reaching-definitions analysis on the CFGs (described later in this sub-section).

Before presenting the algorithm, we present a primitive called splitNode that splits a node containing a multi-assignment instruction into multiple nodes, each of which contains a single microcode assignment. For a given binary, splitNode takes as input a node m that contains a multi-assignment instruction, and the PDG[6] and CFG of the procedure that contains m. (We assume that the SDG and CFGs share a common set of instruction nodes.) splitNode splits the instruction node m into microcode nodes, updates the PDG and CFG, and returns the updated PDG and CFG. splitnode effectively replaces an instruction node with its microcode nodes in the PDG and CFG, and adds the required edges in the graphs. In the PDG, this step results in more precise, finer-grained data dependences between microcode nodes of different instructions.

To simplify matters, we restrict our presentation of splitNode to the case that arises in the IA-32 instruction set, where the semantics of (most) instructions involves at most one memory access or update. Exceptions to this rule are x86 string instructions, which have the `rep` prefix. In our implementation, splitNode does not attempt to split such instructions.

The algorithm for splitNode is given as Alg. 1. In Alg. 1, for a given node n, n.microcode, n.use, and n.kill are the microcode, USE$^{\#}$ set, and KILL$^{\#}$ set, respectively, associated with n; PDG.nodes and PDG.edges (CFG.nodes and CFG.edges) are the nodes and edges in the input PDG (CFG). First, splitNode splits the microcode in m into individual microcode assignments. Recall from §2.2 that the QFBV formula for an instruction's microcode has the form shown in Eqn. (1). Each conjunct in Eqn. (1) is an individual microcode assignment to a register, flag, or memory location. In Alg. 1, `GetConjuncts` returns the set of conjuncts

[6] We do not split actual-in/out nodes and formal-in/out nodes because they do not have multiple assignments; we do not split call-expression nodes because they contain a control-modifying instruction. Consequently, the effect of splitting an instruction node is localized to the procedure, and therefore, splitNode does not need the entire SDG as input.

Algorithm 1 Algorithm SplitNode

```
Input: PDG, CFG, $m$
Output: Updated PDG, Updated CFG
 1: conjuncts ← GetConjuncts($m$.microcode)
 2: for each conjunct $c \in$ conjuncts do
 3:     $n \leftarrow$ CreateNode( )
 4:     $n$.microcode ← $c$
 5:     $n$.use ← Project($n$.microcode, $m$.use)
 6:     $n$.kill ← Project($n$.microcode, $m$.kill)
 7:     for each edge ← $\langle p, m, \text{data}\rangle \in$ PDG.edges do
 8:         if $p$.kill $\cap$ $n$.use $\neq \emptyset$ then
 9:             PDG.edges ← PDG.edges $\cup \{\langle p, n, \text{data}\rangle\}$
10:         end if
11:     end for
12:     for each edge $\langle m, s, \text{data}\rangle \in$ PDG.edges do
13:         if $n$.kill $\cap$ $s$.use $\neq \emptyset$ then
14:             PDG.edges ← PDG.edges $\cup \{\langle n, s, \text{data}\rangle\}$
15:         end if
16:     end for
17:     for each edge $\langle p, m, \text{control}\rangle \in$ PDG.edges do
18:         PDG.edges ← PDG.edges $\cup \{\langle p, n, \text{control}\rangle\}$
19:     end for
20:     for each edge $\langle m, s, \text{control}\rangle \in$ PDG.edges do
21:         PDG.edges ← PDG.edges $\cup \{\langle n, s, \text{control}\rangle\}$
22:     end for
23:     for each edge $\langle p, m\rangle \in$ CFG.edges do
24:         CFG.edges ← CFG.edges $\cup \{\langle p, n\rangle\}$
25:     end for
26:     for each edge $\langle m, s\rangle \in$ CFG.edges do
27:         CFG.edges ← CFG.edges $\cup \{\langle n, s\rangle\}$
28:     end for
29: end for
30: PDG.edges ← PDG.edges $- \langle *, m, *\rangle - \langle m, *, *\rangle$
31: PDG.nodes ← PDG.nodes $- \{m\}$
32: CFG.edges ← CFG.edges $- \langle *, m\rangle - \langle m, *\rangle$
33: CFG.nodes ← CFG.nodes $- \{m\}$
34: return ⟨PDG, CFG⟩
```

in a QFBV formula (Line 1). splitNode creates a new microcode node for each conjunct using `CreateNode` (Line 3).

For each newly created node n, McSlice computes n.use (n.kill) by projecting m.use (m.kill) with respect to the microcode assignment in n. For example, if n.microcode is $Mem' = Mem[ESP - 4 \mapsto EBP]$, and m.kill = {ESP, (AR_main, 0)} (node 1_2 in Fig. 9), n.kill is {(AR_main, 0)}. Because the microcode in n can only kill a memory location, n.kill gets only the memory a-locs from m.kill. In Alg. 1, `Project` performs this projection operation (Lines 5 and 6).

For each microcode node n created from m, McSlice adds data-dependence edges between n and the data-dependence predecessors and successors of m based on USE$^{\#}$ and KILL$^{\#}$ sets (Lines 7–16). McSlice also adds control-dependence edges between n and the control-

Algorithm 2 Algorithm for μ-SDG construction used in McSlice

```
Input: SDG, CFG set
Output: μ-SDG
 1: for each ⟨PDG, CFG⟩ ∈ ⟨SDG, CFG set⟩ do
 2:     PDG ← RemoveSummaryEdges(PDG)
 3:     for each node $m \in$ PDG do
 4:         if IsMultiUpdateNode($m$) then
 5:             ⟨PDG, CFG⟩ ← splitNode(PDG, CFG, $m$)
 6:         end if
 7:     end for
 8:     RunReachingDefs(CFG)
 9:     for each node $n \in$ PDG do
10:         for each edge $\langle p, n, \text{data}\rangle \in$ PDG do
11:             if $p \notin$ ReachingDefs($n$) then
12:                 PDG.edges ← PDG.edges $- \langle p, n, \text{data}\rangle$
13:             end if
14:         end for
15:     end for
16: end for
17: SDG ← RecomputeSummaryEdges(SDG)
18: return SDG
```

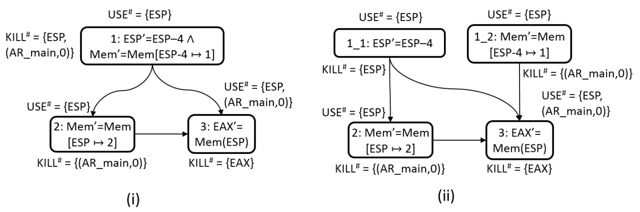


Figure 14: Example SDG and μ-SDG snippets to illustrate removal of data-dependence edges via reaching-definitions analysis.

dependence predecessors and successors of m (Lines 17–22). In a similar manner, n inherits the control-flow edges of m (Lines 23–29). Finally, McSlice removes m and its edges from the PDG and CFG (Lines 30–33).

The algorithm used by McSlice for μ-SDG construction is given as Alg. 2. Alg. 2 takes as input the SDG and CFGs constructed by CodeSurfer/x86, and returns a μ-SDG. For each PDG in the SDG, Alg. 2 first removes the summary edges from the PDG (via `RemoveSummaryEdges`) because the existing summary edges in the SDG reflect imprecise transitive dependences across procedure calls (Line 2). Alg. 2 uses `IsMultiUpdateNode` to identify nodes that contain non-control-modifying, multi-assignment instructions. It then splits each such instruction node into microcode nodes, and updates the PDG and CFG via splitNode (Lines 3–7).

Sometimes, even after splitting instruction nodes via splitNode, the μ-SDG might still have some imprecision that can be eliminated. Consider the code snippet given below

```
push 1
mov [esp], 2
mov eax, [esp],
```

and its corresponding SDG given in Fig. 14(i). After splitNode splits node `1`, and recomputes data dependences, the μ-SDG obtained is given in Fig. 14(ii). On all program paths, node `1_2` will always come before `2`, and one can see that the data-dependence edge from node `1_2` to node `3` can be eliminated. Reaching-definitions analysis tells us that the definition of the memory a-loc in node `1_2` never reaches node `3`, and thus the data-dependence edge between the two nodes can be eliminated.

Alg. 2 performs intraprocedural reaching-definitions analysis on each CFG in the binary via `RunReachingDefs` (Line 8). (For each call-site n that has actual-out nodes o_1, o_2, ... , o_m associated with it, `RunReachingDefs` uses the transformer

$$f_n(S) = S - \langle *, l_1\rangle - \langle *, l_2\rangle - \ \dots \ - \langle *, l_m\rangle \\ \cup \{\langle o_1, l_1\rangle, \langle o_2, l_2\rangle, \ \dots \ , \langle o_m, l_m\rangle\},$$

where $l_1, l_2, \ \dots \ l_m$ are the a-locs defined by the actual-out nodes o_1, o_2, ... o_m, respectively.) If there exists a data-dependence edge e between nodes p and n in the μ-SDG such that no definition at p reaches n, Alg. 2 removes the edge e from the PDG (Lines 9–15). (In Alg. 2, ReachingDefs returns the set of definitions that reach a node.)

At this point, we have a precise, fine-grained, microcode-level PDG for each procedure in the binary. Alg. 2 uses an existing algorithm (Fig. 5 in [25]) to compute summary edges for the SDG to capture context-sensitive transitive dependences across procedure calls (Line 17 via `RecomputeSummaryEdges`), and returns the final SDG.

McSlice uses an existing context-sensitive interprocedural slicing algorithm (Fig. 9 in [18]) to compute a context-sensitive microcode slice over the computed μ-SDG.

4.2 Reconstituting an Executable Machine-Code Program

So far, we have described the algorithms used by McSlice to address the granularity issue, and compute a context-sensitive, fine-grained slice. However, the result of a microcode slice is at a lower level than machine code, and some clients of McSlice might require the results to be reported as *executable machine code*. In this section, we describe how McSlice reconstitutes an executable machine-code program from a microcode slice.

To create executable machine code, McSlice has to resolve three issues: (i) parameter mismatches, (ii) allocation and de-allocation of activation records, and (iii) removal of computations that manipulate uninitialized values. In particular, the latter two issues do not arise in source-code executable slicing. In the remainder of this section, we describe these issues in greater detail, and how McSlice resolves them.

```
                                 int main(){
                                   int a = 10, b = 20;
                                   foo(a, b);
int g1, g2;                        int c = 30, d = g1;
void foo(int a, int b) {           foo(c, d);
  g1 = a;                          return g2;
  g2 = b;                        }
}
```

Figure 15: Example program to illustrate the parameter-mismatch problem in slicing.

Parameter mismatches [6, 18]. Although a backward microcode slice contains all program points that might affect the slicing criterion, it might not be executable because of the *parameter-mismatch* problem: a slice can include multiple calls to the same procedure, with different subsets of actual parameters at different call-sites. However, the slice contains the union of the corresponding formal-parameter sets, which causes a mismatch between the actual parameters at a call-site and the procedure's formal parameters [6, 18]. For example, consider the program shown in Fig. 15. The slicing criterion is boxed in Fig. 15, and the program points included in the slice are highlighted in light gray. On can see that the slice includes only one of the two actual parameters at each call-site, but both formal parameters in procedure `foo`.

To fix parameter mismatches, MCSLICE uses an existing algorithm for monovariant executable slicing [6]. The algorithm fixes call-sites by conservatively including in the slice additional actual parameters to match the formal parameters included in the slice. For example, the additional program points included by the monovariant executable-slicing algorithm are underlined in Fig. 15.

Note that the aforementioned monovariant executable-slicing algorithm is a suboptimal way to fix parameter mismatches: the goal of slicing is to remove as many extraneous program points as possible, but the monovariant executable-slicing algorithm re-introduces removed program points to fix call-sites. Specialization slicing [3] is a polyvariant executable-slicing algorithm that produces optimal executable slices by creating specialized copies of procedures according to the sets of parameters included at various call-sites. One direction for future work is to incorporate the specialization-slicing algorithm in MCSLICE to obtain more precise executable machine code.

Allocation and de-allocation of activation records. To create executable machine-code, activation records should be be correctly allocated and de-allocated in all procedures included in the slice. MCSLICE includes in the slicing criterion the formal-outs corresponding to the stack pointer ESP and frame pointer EBP of the `main` procedure so that all relevant microcode that allocates and de-allocates activation records are included in the microcode slice.

Removal of computations that manipulate uninitialized values. After fixing parameter mismatches and including the relevant microcode that allocate and de-allocate activation records in the microcode slice, the final step creating an executable machine-code program is to generate machine code from the microcode slice. A naïve way to generate machine code from a microcode slice is to add to the output machine-code program each instruction for which *any fragment* of its microcode is included in the microcode slice. For example, if we use this sub-optimal method to generate machine code from the microcode slice shown in Fig. 12, we would obtain the machine-code program shown in Fig. 16. One can see that the code in Fig. 16 performs computations on uninitialized values. For example, the instruction marked by * in Fig. 16 pushes onto the stack the 32-bit value in the memory location EBP − 8, which has not been initialized by upstream instructions.

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

Figure 16: Machine code generated from the microcode slice shown in Fig. 12 by a naïve method.

Moreover, code that performs computations on uninitialized values in the executable machine-code program can sometimes introduce exceptions that otherwise would never occur in the original program. Consider a program that contains the following instruction sequence:

```
1: ...
2: mov eax, 10
3: mov edx, 0
4: mov ebx, 2
5: idiv ebx
6: ...
```

Suppose that EDX:EAX denotes the extended 64-bit register obtained from the 32-bit registers EDX and EAX. The instruction sequence divides the contents of EDX:EAX (10) by the contents of EBX (2), places the quotient (5) in EAX, remainder (0) in EDX, and affects some flags. Suppose that a backward slice only requires the microcode that defines the flags. MCSLICE will compute a precise slice that includes only the microcode that defines the flags in instruction 5; instructions 2, 3, and 4 will not be included in the slice because the microcode corresponding to the division"EDX:EAX / EBX" is not included in the slice.

In contrast, if MCSLICE were to emit the entire instruction 5, the output program might look as follows:

```
1: ... // 2, 3, 4 not included
5: idiv ebx // EBX might be uninitialized here
...
```

When this program executes, if EBX contains 0 when the `idiv` instruction executes, then the program fails with an exception, which does not match the behavior of the original

Algorithm 3 Algorithm used by McSlice for generating machine code from a microcode slice

```
Input: Microcode slice $s$
Output: Machine-code program $s'$
 1: $s' \leftarrow \epsilon$
 2: for each microcode node $n \in s$ do
 3:     $m \leftarrow$ OldNode($n$)
 4:     newNodes $\leftarrow$ NewNodes($m$)
 5:     if newNodes $\subseteq s$ then
 6:         $s' \leftarrow$ AddToProgram($s'$, $m$.instruction)
 7:         $s \leftarrow s -$ newNodes
 8:     else
 9:         I $\leftarrow$ McSynth($n$.microcode)
10:         $s' \leftarrow$ AddToProgram($s'$, I)
11:         $s \leftarrow s - \{n\}$
12:     end if
13: end for
14: return $s'$
```

program. To avoid this issue, we want McSlice to generate code *only* for the microcode included in the backward slice.

To reconstitute a machine-code program that does not contain computations that manipulate uninitialized locations, McSlice uses machine-code synthesis. The algorithm used for reconstitution in McSlice is given as Alg. 3. The input to Alg. 3 is a microcode slice s. (Note that s does not have parameter mismatches, and all relevant microcode that allocate and de-allocate activation records are included in s.) If all newly-created microcode nodes of an instruction node m are included in the slice, McSlice simply generates back the instruction contained in m (Lines 5–7). (In Alg. 3, `OldNode` returns the original instruction node corresponding to a microcode node; `NewNodes` returns the set of microcode nodes that were created by splitting an original instruction node; m.instruction is the instruction contained in the old instruction node m.) However, if only a subset of microcode nodes created from m are included in the slice, then McSlice must generate an instruction (or instruction sequence) that is equivalent to each included microcode node. For this purpose, McSlice uses a machine-code synthesizer. For each microcode node n included in the slice, McSlice supplies the microcode in n as input to the machine-code synthesizer McSynth [31]. McSynth takes a QFBV formula as input, and synthesizes an instruction sequence that is equivalent to the QFBV formula. McSlice inserts the synthesized instruction sequence into the generated code (Lines 8–12). (In Alg. 3, McSynth invokes the synthesizer; we assume that the procedure `AddToProgram` takes care of generating code in the correct order, creating new procedures, etc.) Alg. 3 returns the final executable machine-code program generated from the microcode slice.

For the microcode slice shown in Fig. 12, Alg. 3 produces the machine-code program shown in Fig. 11.

5. Implementation

McSlice uses CodeSurfer/x86 [5] to obtain the SDG for a binary, and the USE$^{\#}$/KILL$^{\#}$ sets at each instruction.

McSlice uses Transformer Specification Language (TSL) [20] to obtain QFBV encodings of instructions. We worked with a specification of the IA-32 instruction set that grouped around 43,000 non-privileged, non-floating point, non-mmx instructions into 164 opcode variants. (These opcode variants do not include the `lock`, `rep`, and `repne` prefixes.) McSlice uses a concrete operational-semantics of these 164 opcode variants, written in TSL, which provides McSlice with a microcode-level specification of each instruction that can be instantiated from one of the 164 opcode variants (i.e., using different registers, addressing modes, etc.). The semantics written in TSL is reinterpreted to produce the QFBV formulas for individual instructions [21]. McSlice's slicing algorithm will split any instruction that

- belongs to one of the 164 opcode variants,
- performs a multi-assignment, and
- does not modify control.

(Instructions that assign to only a single register, flag, or memory location do not need splitting.) Many of the opcode variants are rarely used by compilers, and our benchmark suite of the binaries of 8 FreeBSD utilities compiled with `gcc` (Table 1) included instructions belonging to 35 out of the 164 opcode variants.

McSlice uses the machine-code synthesizer McSynth [31] parameterized for IA-32 to synthesize instruction sequences from QFBV formulas of microcode included in the slice.

In CodeSurfer/x86, the abstract transformers for the analyses used to build an SDG are obtained using TSL [20, §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 McSlice's toolchain for the new ISA.

6. Experiments

We tested McSlice 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 McSlice?
- What percentage of the slice computed by McSlice consists of entire instructions? (And what percentage consists of microcode subsets?)
- Which kinds of instructions have only a subset of their microcode included in slices?

All experiments were run on a system with a quad-core, 3GHz Intel Xeon processor running Windows 7, and 32 GB RAM; however, McSlice's algorithm is single-threaded.

Our test suite consisted of IA-32 binaries of FreeBSD utilities [16]. 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.

Table 1: Characteristics of applications in our test suite.

Application	LOC	No. of instructions	No. of nodes in instruction-level SDG	No. of nodes in μ-SDG	Backward-slicing criterion	Forward-slicing criterion
`wc`	295	790	1105	2173	Actual `twordct` of call to `printf` in `main`	Locals `linect`, `wordct`, and `charct` in `cnt`
`md5`	331	821	2210	3410	Actual `p` of final call to `printf` in `main`	Actual `optarg` of call to `MDString` in `main`
`write`	332	957	1825	3191	Actuals of final call to `do_write` in `main`	Local `atime` in `main`
`uuencode`	392	862	1069	1909	Actual `output` of call to `do_write` in `encode`	Initialization of global `mode` in `main`
`cksum`	505	815	1309	2377	Actual `len` in final call to `pcrc` in `main`	Local `lcrc` in `csum1`
`units`	783	2119	6045	9519	Actuals of final call to `showanswer` in `main`	Local `linenum` in `readunits`
`msgs`	951	2270	4769	8566	Actual `nextmsg` of final call to `fprintf` in `main`	Local `blast` in `main`
`pr`	2207	4005	8548	14746	Local `pagecnt` in `vertcol`	Local `eflag` in `setup`

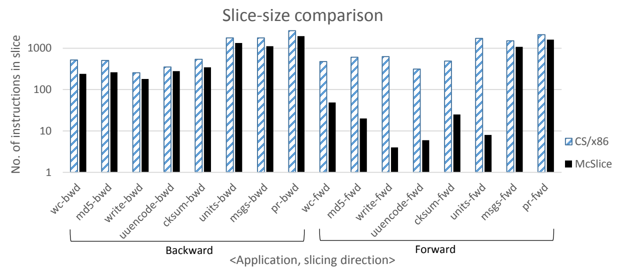
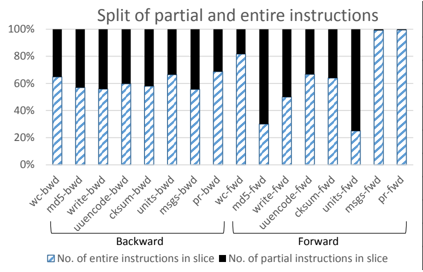


Figure 17: Comparison of sizes of slices computed by CodeSurfer/x86 and McSlice (log-scale).



Opcode variant	Count - bwd	Opcode variant	Count - fwd
ADD	564	PUSH	45
PUSH	410	TEST	3
SUB	445	AND	2
CMP	280		
TEST	265		
AND	39		
SHL	32		
SAR	18		
IMUL	14		
IDIV	13		

Figure 18: Split of partial and entire instructions in slices computed by McSlice.

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 in the beginning of the application.

To answer the first question, we computed the slices using CodeSurfer/x86 and McSlice. CodeSurfer/x86 computes machine-code slices, but McSlice computes microcode slices. To facilitate meaningful comparison between the two slice sizes, we report the number of *instructions* in the binary for which any microcode fragment was included in the slice computed by McSlice as the slice size for McSlice. The results are shown in Fig. 17. Note that the y-axis uses a logarithmic scale. The average reduction in slice size, computed as a geometric mean, is 33% for backward slices and 70% for forward slices. For the forward slices of `write` and `units`, McSlice 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 number of downtream instructions affected by the imprecision-causing idioms for forward slices (e.g., imprecision caused by instruction `25` in Fig. 8) is much higher in practice than their upstream counterparts in backward slices. For `msgs` and `pr`, the forward slice computed by McSlice 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 McSlice, we performed the following computation: for each instruction in the binary whose microcode update was in the microcode slice, we checked if all microcode updates of that instruction were present in the slice, or only a subset of microcode updates were present in the slice. The former case means that the entire instruction is effectively included in the microcode slice, and the latter case means that only a subset of the instruction's semantics is included in the slice. Fig. 18 shows the results. For backward slices 60% of the slice consisted of entire instructions, and the remaining came from microcode subsets of instructions. For forward slices, that number was 84%. (The average percentages were computed as geometric means.) We also identified the top opcode variants that constitute the instructions that were not included in their entirety in the backward and forward slices, respectively. For forward slices, instructions belonging to only three opcode variants caused all the imprecision in CodeSurfer/x86 slices. For the `push` opcode variant in the table in Fig. 18, either the stack-pointer update or the memory update was excluded from the slice. For the remaining opcode variants, a subset of flag updates was generally excluded from the slice.

Table 2: Comparison of sizes of original binary and extracted component.

Application	No. of instructions in binary	No. of instructions in extracted component
`wc-lite`	295	64
`wc`	790	242
`cksum`	815	338

6.1 Extracting Executable Components from Binaries

For two of our benchmark programs and an additional micro-benchmark program, the backward slice computed by McSLICE extracts a meaningful component. For these three programs, we used Alg. 3 to reconstitute a machine-code program for the extracted component, and generated an executable slice (§4.2). Table 2 presents the characteristics of the applications used for component extraction. `wc-lite` is a scaled down version of the `wc` utility.[7] For `wc-lite`, we extracted a component that only counts lines in input files; for `wc`, we extracted a component that only counts the words in input files; for `cksum`, we extracted a component that only computes the length of the input file. Table 2 shows the number of instructions in the original binary, and the number of instructions in the extracted component. For all three programs, there were no parameter mismatches in the slice, and so McSLICE did not have to fix call-sites as described in §4.2.

7. Related Work

Slicing. The literature on program slicing is extensive [7, 22, 33]. Slicing has been—and continues to be—applied to many software-engineering problems [17]. For instance, recently there has been work on language-independent program slicing [8], 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 granularity issue never arises. The ways in which the program-reconstitution issue for machine-code slicing differs from creating executable source-code from source-code slices have been discussed in §4.2.

Applications of more precise machine-code slicing. WIPER is a machine-code partial evaluator [30] 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.). The ad hoc instruction-decoupling algorithm used in WIPER 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. McSLICE computes more accurate forward slices, and could be used in WIPER's BTA to increase BTA precision.

Taint trackers [23, 27, 32] use dynamic analysis to check if tainted inputs from *taint sources* could affect *taint sinks*. Certain taint trackers such as Minemu [9] rely entirely on dynamic analysis to reduce taint-tracking overhead. McSLICE 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 [35] is a concurrency-bug detection tool, which uses machine-code slicing to compute the set of critical reads that might affect a failure site. McSLICE could be used to compute more accurate backward slices, effectively reducing the number of critical reads that needs to be analyzed by Conseq.

Taint explosion in pointer tainting. Taint tracking is commonly used to detect control-diverting attacks, e.g., warn the user if a tainted input flows to the program-counter register, and is about to be used as a jump target. Taint tracking also provides a conservative method to demonstrate the absence of information flow. Pointer tainting [12] is a variant of taint tracking that additionally propagates taint to the dereferenced value whenever a tainted pointer is dereferenced. Pointer tainting is commonly used to detect non-control-diverting attacks such as memory-corruption attacks against non-control data, e.g., a buffer overflow that modifies a user's privilege level.

One of the primary issues with pointer tainting is taint explosion caused by the stack-pointer register ESP and frame-pointer register EBP. If ESP or EBP ever contains a tainted pointer, all the values obtained by dereferencing those registers in later instructions get tainted. Because many instructions dereference ESP and EBP to access values on the stack, if ESP or EBP ever gets falsely tainted, pointer tainting can cause undesired taint explosion, e.g., to the kernel and other unrelated processes. One of the containment techniques used to control taint explosion is to use heuristics to check if the current value in ESP and EBP is a legitimate address or a valid index into a table, and untaint the value if it is [28].

While our work shows how multi-assignment instructions involving ESP and EBP cause undesired explosion in slices in a static context, the study on the effects of pointer tainting [28] show how ESP and EBP registers act as vectors for undesired taint explosion in a dynamic context.

[7] The source code for this micro-benchmark is given as Fig. 2 in [24].

8. Conclusion

In this paper, we described a new algorithm to slice machine code. We presented experiments with McSlice, a tool that incorporates the algorithm, which showed that McSlice slices IA-32 binaries more precisely than CodeSurfer/x86, a state-of-the-art tool for machine-code slicing. Our experiments on binaries of FreeBSD utilities show that, in comparison to slices computed by CodeSurfer/x86, our algorithm reduces the sizes of backward slices by 33%, and forward slices by 70%. For some binaries in our test suite, McSlice reduces the size of the slice by over two orders of magnitude.

Acknowledgments

We thank Tom Johnson, Suan Yong, Brian Alliet, and Junghee Lim for answering several questions during the implementation of McSlice. We thank the anonymous reviewers for their valuable feedback. The first author would like to dedicate this paper to the late Prof. Susan Horwitz and the second author, whose seminal paper "Interprocedural slicing using dependence graphs" inspired him to pursue this work.

References

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

[2] ARM instruction-set manual. `http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf`.

[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. Precise executable interprocedural slices. *LOPLAS*, 2:31–45, 1993.

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

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

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

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

[11] D. Brumley, I. Jager, and E. S. S. Whitman. The BAP handbook, 2014.

[12] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *DSN*, 2005.

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

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

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

[16] FreeBSD utilities. `http://www.opensource.apple.com/source/`.

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

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

[19] IA-32 instruction-set manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

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

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

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

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

[24] T. Reps and T. Turnidge. Program specialization via program slicing. In *Proc. of the Dagstuhl Seminar on Partial Evaluation*, pages 409–429, 1996.

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

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

[27] 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.

[28] A. Slowinska and H. Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *EuroSys*, 2009.

[29] 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.

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

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

[32] E. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

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

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

[35] 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.