Synthesis of Machine Code: Algorithms and Applications

By

Venkatesh Srinivasan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 02/28/2017

The dissertation is approved by the following members of the Final Oral Committee:

Thomas Reps, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Benjamin Liblit, Associate Professor, Computer Sciences
Aws Albarghouthi, Assistant Professor, Computer Sciences
Parameswaran Ramanathan, Professor, Electrical and Computer Engineering

Dedicated to my Amma and Appa

Acknowledgments

My years in graduate school have been the most enlightening, transformative, and adventurous years of my life. I am indebted to so many people for having supported me along the way. This dissertation would not have been possible without them; I would like to express my sincere gratitude to all of them.

First and foremost, I would like to thank Prof. Thomas Reps, my advisor, for his patience, guidance, and support over the course of my graduate studies. Tom was my ideal advisor: he gave me the time to discover my interests, the freedom to choose my projects, the expertise to tackle difficult problems, and the motivation to successfully complete papers. He refined my taste in research: he taught me how to identify important problems, craft elegant solutions, and write succinct papers. The spectrum of things I learned from Tom spans all the way from excellent editing skills to quirky idioms.[1] Tom: you inspired me to be a good researcher, and I assure you that I will sincerely strive to be one.

I would like to thank the members of my committee: Prof. Somesh Jha, Prof. Benjamin Liblit, Prof. Aws Albarghouthi, and Prof. Parameswaran Ramanathan, for their insightful discussions and comments that helped me vastly improve this dissertation.

I would like to thank the late Prof. Susan Horwitz for introducing me to the amazing world of PL research through her CS 701 course. Without her and her offering of CS 701, I would have never gotten the opportunity to work with Tom, and this dissertation would not have existed today.

I would also like to thank Prof. Murali Krishna Ramanathan for being my mentor over the course of my Ph.D. When I was on the verge of quitting, Murali gave me the motivation to toughen up and persist, as well as practical tips to keep progressing in research.

I would like to thank Tushar Sharma and Ara Vartanian for being stellar collaborators. Tushar and Ara shaped my half-baked ideas into elegant solutions with their expertise, and helped me finish the projects in time.

I could not have finished the implementation of my tools without the immense help provided by the folks at GrammaTech. Junghee Lim, Suan Yong, Brian Alliet, and Tom

[1] There were countless times when Tom alerted me before I was about to *throw the baby out along with the bath water*, or when I was *circling the wagons and shooting inward*.

Johnson have promptly answered my millions of questions over the last five years. I would also like to thank Alexey Loginov and David Melski for their helpful comments during the design phases of my tools.

I would like to thank Peter Ohmann for being a superb peer over the course of my Ph.D. Peter was always there by my side, eager to help me; whether it was discussing potential research ideas or "channeling his inner Somesh" (asking difficult big-picture questions) to prepare me for conference talks and exams. I would like to thank my big brother in the group, Tushar Sharma, for always looking out for me, especially during my initial years. I would like to thank my wonderful office mate, Jason Breck, for patiently listening to all the boring unsolicited advice I've been showering upon him for the last three years. I would like to thank the present and past members of the PL group, especially Drew Davidson, Bill Harris, Evan Driscoll, Aditya Thakur, Alisa Maas, and Stephen Lee, for attending my practice talks, and providing useful suggestions for improving my talks.

I would like to thank my friends for bringing fun, excitement, and happiness into my boring and monotonous life. I would like to thank Karthik anna and Madhu for being close by me through good times, and even closer during tough ones; Ramki for being my best mate, my pillar of strength, and my trusted source of cringeworthy dad jokes for the past three years; Srinath for being the sane, logical, and dependable friend in my otherwise insane circle of friends; Ashwin and Smrithi for making sure I have some fun in between projects; Hari for being my therapist, and patiently listening to my countless whines and complaints; Divy for being an amazing roommate and friend; Thanu anna for being my coffee-break pal and job-search counselor; my "kids" (Nivetha, Kausik, and Anshul) for making me sometimes feel young and carefree; Andrea for being my party companion, and teaching me some essential life skills; Srikant, Arvind, Naveen, Sanath, and Venkatanathan for being awesome friends.

I would like to thank my cousin Madhusudhan and his wife Melissa for taking care of me when I arrived as a clueless twenty-one-year-old in the U.S., and providing moral support through some testing times. My stays at their place were the rare instances of my feeling at home in the U.S.

I would like to thank my grandparents for being an integral part of my life and childhood. I would like to thank my *paati* for her abundant affection. She taught me the values of patience and sacrifice, and I hope I can one day be the strong-willed person that she is. I would like to thank my late *thatha* for making my education his number-one priority, and providing all the support and encouragement I needed to excel in my studies. I would also like to thank my *mama* (uncle) and *mami* (aunt) for their love and support, and for taking care of my parents while I was away.

I don't think a few words in the acknowledgements section of this dissertation are even remotely enough to covey my gratitude towards my parents. I have the best parents a son could possibly ask for: they showered me with love, taught me important values, provided me ample support and encouragement, nurtured me to carve out my own identity, and helped me cross extremely difficult times. They strived to provide me the best possible environment to grow up in; even if it meant that they had to fight against all odds and make countless sacrifices. I would like to thank my *amma* for her unconditional love and unfaltering support. She fostered my curiosity and creativity in so many ways, from doing flame-test experiments (for salt analysis) with me in our kitchen to teaching me the basics of Carnatic music and embroidery. She showed me how one can achieve anything one aims for with diligence and determination. I would like to thank my *appa* for being my rock: he was always there for me by my side to offer his support. Observing him as I grew up, I learned what it meant to always be there for one's family. My parents often tell me that they are blessed to have a son like me. However, I think I am twice as blessed to have parents like them. *Amma* and *appa*: thank you for everything. I humbly dedicate this dissertation to you.

I would like to end this section with the following verses my *appa* taught me when I was ten. These verses have given me hope during times when I thought I had none.

विना वेङ्कटेशं न नाथो न नाथः
सदा वेङ्कटेशं स्मरामि स्मरामि ।
हरे वेङ्कटेश प्रसीद प्रसीद
प्रियं वेङ्कटेश प्रयच्छ प्रयच्छ ॥

This dissertation is supported, in part, by a gift from Rajiv and Ritu Batra; by DARPA under cooperative agreement HR0011-12-2-0012; by AFRL under DARPA MUSE award FA8750-14-2- 0270, DARPA STAC award FA8750-15-C-0082, and DARPA CRASH award FA8650-10-C-7088; by NSF under grant CCF-0904371; 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 dissertation are those of the author, and do not necessarily reflect the views of the sponsoring agencies. The author's advisor Thomas Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this dissertation.

Contents

List of Figures, Tables, and Listings

Abstract

The analysis of binaries has gotten an increasing amount of attention from the academic community in the last decade. The results of binary analysis have been predominantly used to answer questions about the properties of binaries. Another potential use of analysis results is to rewrite the binary via semantic transformations. Semantics-based binary-rewriting tools become particularly important when one wishes to modify the functionality of the binary, and the source code and/or compiler toolchain for the binary is unavailable. Semantics-based binary rewriting can be done for the purposes of binary optimization (offline optimization, superoptimization), software reuse (partial evaluation, slicing, binary translation), or software security (binary obfuscation, de-obfuscation).

This dissertation proposes various algorithms for synthesizing a machine-code implementation from a semantic specification of the desired behavior, and describes how one can use a machine-code synthesizer to build different semantics-based binary-rewriting tools.

The first part of this dissertation introduces the problem of machine-code synthesis, proposes a framework for semantics-based binary rewriting via machine-code synthesis, and presents several algorithms for machine-code synthesis. A key challenge in synthesizing machine code is the enormous size of the synthesis search-space. Instruction sets like Intel's IA-32 have around 43,000 unique instruction schemas; this huge instruction pool, along with the exponential cost inherent in enumerative synthesis, results in an enormous search space for the synthesizer: even for relatively small specifications, a naïve synthesizer might take several days to find an implementation. We have developed a suite of techniques—ranging from pruning heuristics to machine learning—that assist an enumerative synthesizer in combating the enormous search space, and speeding up synthesis from the order of days to the order of seconds.

The second part of this dissertation describes how one can instantiate the aforementioned synthesizer-equipped framework to create different semantics-based binary rewriters. In particular, we have instantiated the framework to create two tools that rewrite binaries for purposes of software reuse: a novel machine-code partial evaluator, and an improved machine-code slicer. Among other potential uses, these tools can be used to either modify or extract a component from a binary that lacks source code.

1 Introduction

The analysis of binaries has gotten an increasing amount of attention from the research community in the last decade (e.g., see references in [88, §7], [14, §1], [23, §1]). This research has led to the development of several tools that analyze binaries, and answer questions about their properties. These tools include static-analysis platforms [15, 23, 88], symbolic-execution engines [39, 72], and verifiers [82, 95]. However, when it comes to binary *rewriting*, the suite of state-of-the-art tools is limited to superoptimizers [16, 76], patching tools [63, 77], and de-obfuscators [27, 81, 99]. Currently, there are no tools that can rewrite binaries via *semantic transformations*, e.g., partial evaluators, slicers, program-repair tools, etc. Semantics-based binary-rewriting tools become particularly important when one wishes to modify the functionality of the binary, and the source code and/or compiler toolchain for the binary is unavailable.

To rewrite a binary based on semantic criteria, an important primitive to have is a machine-code synthesizer—a tool that emits machine-code instructions[1] belonging to a specific instruction-set architecture (ISA) for the transformed program semantics.

> **Framework 1.1.** A machine-code synthesizer allows us to create multiple semantics-based binary-rewriting tools that use the following recipe:
>
> 1. Convert instructions in the binary to a semantic representation, e.g., a quantifier-free bit-vector (QFBV) logic formula.
> 2. Use binary-analysis results to transform QFBV formulas.
> 3. Use the synthesizer to produce an instruction sequence that implements each transformed formula.

One simple tool that could be created using Framework 1.1 is an offline binary-optimizer to improve unoptimized binaries. Analyses like value-set analysis (VSA) [14] and def-use analysis (DUA) [53] could be used in Step 2 to optimize QFBV formulas using information about constants, live registers and flags, etc. One can create different binary-rewriting tools

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

by instantiating Framework 1.1 with different binary analyses and formula-transformation mechanisms.

This dissertation advances the state-of-the-art in binary rewriting by applying program-synthesis techniques to binary analysis and rewriting. In particular, the goals of this dissertation are to

- develop algorithms that synthesize machine-code implementations from specifications, and
- instantiate Framework 1.1 to create binary-rewriting tools that can be used to either modify or extract a reusable component from a binary (for purposes of software reuse).

Each of the aforementioned goals is described in the rest of this section.

1.1 Algorithms for Machine-Code Synthesis

Machine-code synthesis is the problem of synthesizing an instruction sequence that implements a semantic specification of the desired behavior, often given as a QFBV formula. Currently, there are no tools that perform machine-code synthesis for a full ISA. Existing approaches either (i) work on small bit-vector languages that do not have all the features of an ISA [41], or (ii) superoptimize instruction sequences [16]. A peephole-superoptimizer has the following type:

$$\mathit{Superoptimize} : \mathit{InstrSequence} \rightarrow \mathit{InstrSequence}$$

A machine-code synthesizer has the following type:

$$\mathit{Synthesize} : \mathit{QFBVFormula} \rightarrow \mathit{InstrSequence}$$

Because an instruction sequence can be converted to a QFBV formula via symbolic execution, a machine-code synthesizer can be used for superoptimization; however, the converse is not possible. (See the paragraph titled "Superoptimization" in §3.5.)

A key challenge in synthesizing machine code is the enormous size of the synthesis search-space. Instruction sets like Intel's IA-32 have around 43,000 unique instruction schemas; this huge instruction pool, along with the exponential cost inherent in enumerative synthesis, results in an enormous search space for the synthesizer: even for relatively small specifications, a naïve synthesizer might take several days to find an implementation. The

first part of this dissertation describes how we addressed this challenge while designing an algorithm for machine-code synthesis and making subsequent improvements to the algorithm. The techniques we developed to combat the size of the synthesis search-space are not restricted to machine code in particular, and can be applied to other program synthesizers as well.

Base algorithm. We developed an enumerative machine-code synthesizer called McSynth [90], which is parameterized by the ISA of the target instruction-sequence, and is easily adaptable to work on other semantic representations, such as a Universal Assembly Language (UAL) [23]. McSynth uses (i) a *divide-and-conquer* strategy to split the input formula into several independent smaller sub-formulas, (ii) *footprint-based* search-space pruning heuristics to prune away candidates during synthesis, and (iii) a novel instantiation of the *counterexample-guided inductive synthesis (CEGIS)* framework as the core synthesis loop. Experiments with the IA-32 instruction set showed that McSynth is 3 to 5 orders of magnitude faster than a baseline enumerative synthesizer.

Optimizations. McSynth brought down synthesis time from days to hours, but it was still not fast enough: McSynth times out for several larger QFBV formulas; even for smaller formulas, McSynth takes several minutes to find an implementation. Consequently, if a binary-rewriter client supplies a formula as input to McSynth, the client has to wait several minutes or hours before McSynth finds an implementation. This delay might not be tolerable for a client that has to invoke the synthesizer multiple times to rewrite an entire binary (e.g., a machine-code partial evaluator). We made several improvements to the synthesis algorithm used in McSynth, and developed McSynth++ [92], which is an improved version of McSynth. In addition to a novel *bits-lost-based* pruning heuristic, the improvements incorporate a number of ideas known from the literature, which we adapted in novel ways for the purpose of speeding up machine-code synthesis. Experiments for IA-32 showed that the improvements enable synthesis of code for 12 out of 14 formulas on which McSynth times out, speeding up the synthesis time by at least 1981X, and for the remaining formulas, speeds up synthesis by 3X.

Model-assisted synthesis. McSynth and McSynth++ perform a linear search over the space of instruction sequences, i.e., after exhausting all one-instruction sequences, they search through all two-instruction sequences, and so on (modulo pruning). One can see that this search strategy is not very efficient because not all k-instruction sequences are equally likely to implement the input specification. (For example, if the specification computes the

sum of the contents of two registers, then an instruction that performs integer multiplication is clearly not relevant.) We converted the linear search in McSynth++ into a *best-first search* over the space of instruction sequences, and developed McSynth-ML [93], which is a model-assisted version of McSynth++. The cost heuristic for the best-first search comes from two models built from a corpus of equivalent ⟨QFBV-formula, instruction-sequence⟩ pairs. (Note that it is straightforward to build such a corpus by harvesting several instruction sequences from binaries, and converting them into QFBV formulas via symbolic execution.) One model is a *language model*, which favors useful instruction sequences (i.e., instruction sequences that occur more frequently in the corpus). The other model correlates features of instruction sequences with features of QFBV formulas, and favors instruction sequences that are more likely to implement the input formula. Experiments for IA-32 showed that McSynth-ML enables synthesis of code for 6 out of 50 formulas on which McSynth++ times out, speeding up the synthesis time by at least 549X, and for the remaining formulas, speeds up synthesis by 4.55X.

In summary, the first part of this dissertation introduces the problem of machine-code synthesis, and describes a suite of approaches—ranging from pruning heuristics to machine learning—that assist an enumerative synthesizer to combat the enormous search space, and quickly find a machine-code implementation for a given specification.

1.2 Binary Rewriting via Synthesis

Our principal motivation for developing algorithms for machine-code synthesis is to develop a general framework for semantics-based binary rewriting (Framework 1.1), and subsequently instantiate Framework 1.1 to create binary-rewriting tools that can be used to either modify or extract a component from a binary. Existing tools can de-obfuscate [27, 81, 99], superoptimize [16, 76], or harden [6, 33, 83] binaries, but they cannot extract a component from a binary. To this end, we developed two binary-rewriting tools by instantiating Framework 1.1: a novel machine-code partial evaluator, and an improved machine-code slicer. Apart from component extraction, these tools have other potential applications (see below). In the remainder of this sub-section, we describe the tools in greater detail.

Machine-code partial evaluation *Partial evaluation* [48] is a program-specialization framework that can be used to specialize a program P with respect to some of its inputs to produce a version of P that is specialized/optimized for those inputs. For example, partially evalu-

ating the `power` program with the value 2 for the exponent produces an optimized `square` program. Currently, there are no tools that can partially evaluate machine code.

We developed the first machine-code partial evaluator.[2] The partial evaluator WiPEr [89] performs off-line partial evaluation of binaries. WiPEr's algorithm follows the classical two-phase approach of *binding-time analysis (BTA)* followed by *specialization*. WiPEr's specializer specializes an explicit representation of the semantics of an instruction, and emits residual code via machine-code synthesis. Moreover, to create code that allows the stack and heap to be at different positions at run-time than at specialization-time, the specializer represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization. WiPEr can be used to specialize binaries with respect to commonly used inputs to produce faster binaries (e.g., a file-write routine optimized for a certain file descriptor), as well as to extract an executable component from a bloated binary (e.g., extract `compress` from the `gzip` binary).

Machine-code slicing. One of the most useful primitives in program analysis is *slicing* [46, 98]. A slice consists of the set of program points that affect (or are affected by) a given program point called the *slicing criterion*. Slicing has many applications, and is used extensively in program analysis and software-engineering tools. A machine-code slicer is often used as a key primitive in several binary analysis and rewriting tools. Improvements in machine-code slicing could significantly increase the precision and/or performance of several existing tools, such as partial evaluators [89], taint trackers [22], and fault localizers [100]. Moreover, a machine-code slicer could be used as a black box to build new binary analysis and rewriting tools.

However, it is not easy to create a machine-code slicer that exhibits a high level of precision. 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. Moreover, small amounts of such local imprecision can be amplified via cascade effects. Consequently, the slice computed by state-of-the-art tools is very imprecise, often including essentially the entire program.

We developed a tool called McSlice [91] that slices machine code more accurately. To counter the granularity issue, McSlice performs slicing at the microcode level, instead of the instruction level, and obtains a more precise microcode slice. McSlice uses QFBV formulas

[2] Confirmed by personal communication with Neil Jones, Robert Glück, and Saumya Debray.

to represent microcode. To reconstitute a machine-code program from a microcode slice, McSlice uses machine-code synthesis.

The granularity issue in machine-code slicing has been addressed in prior work by Bernat et al. in the context of instrumenting x86 binaries [18]. To address the granularity issue and compute precise slices, Bernat et al. use special-case *instruction splitting*: each node in the *system-dependence graph* (SDG)[3] corresponding to an instruction that performs multiple operations is split into multiple nodes, each of which performs a single operation. In contrast, McSlice splits multi-operation instruction nodes into microcode nodes, and performs slicing on a microcode-level SDG. Each microcode node represents a single microcode update, specified by a QFBV formula. Splitting instruction nodes based on microcode has the following advantages over the approach used by Bernat et al.:

- Special-case instruction splitting is tied to a specific ISA and binary-analysis platform: one needs to specify the target single-operation nodes that should be created for each multi-operation instruction in a new ISA, and this mapping is tailored for a specific binary-analysis platform. However, QFBV-based microcode slicing in McSlice is not tied to a specific binary-analysis platform or ISA. (See §7.2.2.)

- If a client of the slicer wishes to render a backward slice as an executable machine-code program, then one needs to provide the slicer—as an additional input—a mapping from each single-operation node to an equivalent instruction-sequence in the target ISA. Again, this mapping needs to be supplied for each binary-analysis platform and target ISA. In contrast, McSlice obtains instruction sequences from microcode nodes automatically via machine-code synthesis.

Experiments on IA-32 binaries of FreeBSD utilities show that, in comparison to slices computed by a state-of-the-art tool (CodeSurfer/x86), McSlice reduces the size of backward slices by 33%, and forward slices by 70%. The backwards executable-slicing functionality of McSlice can also be used to extract executable components from binaries (e.g., extract a line-count program from the `wc` binary).

In summary, the second part of this dissertation describes two instantiations of Framework 1.1—a machine-code partial evaluator and a machine-code slicer—that can be used to extract an executable component from a binary for purposes of software reuse. Note that one can potentially instantiate Framework 1.1 to also create other semantics-based binary-rewriting tools.

[3]The system-dependence graph is an intermediate representation used for slicing. See Chapter 7.

Contributions

The contributions of this dissertation include the following:

1. We have created the first machine-code synthesizer that works on the integer subset of a full ISA (Chapter 3).

2. We have proposed a general framework for semantics-based binary rewriting. Our framework can be used to create novel binary-rewriting tools, and also to improve existing tools.

3. We have proposed several optimizations for speeding up machine-code synthesis. Our optimizations speeds up our baseline synthesis algorithm by several orders of magnitude. Our optimizations are not restricted to machine code in particular, and can be used to speed up other program synthesizers as well (Chapter 4).

4. We have created the first low-level-code synthesizer that uses machine learning to assist synthesis (Chapter 5).

5. We have created the first machine-code partial evaluator, which can be used to optimize binaries with respect to common inputs, and to extract components from bloated binaries (Chapter 6).

6. Our machine-code slicing algorithm significantly improves slicing accuracy with respect to state-of-the-art tools. Our slicer can be used to extract components from binaries, and also to aid other binary analyses, for example, by excluding from consideration portions of the binary that are irrelevant to the analysis (Chapter 7).

The dissertation also makes the following more general contributions that go beyond the realms of binary analysis and rewriting:

1. We have proposed a divide-and-conquer approach to enumerative program synthesis that facilitates breaking a larger synthesis task into a sequence of independent smaller sub-tasks.

2. We have proposed lightweight pruning strategies that use abstractions/overapproximations of enumerated candidates to rapidly prune away useless candidates during synthesis.

3. Our model-assisted synthesis algorithm is the first program-synthesis technique to use models learned from *both* specifications *and* implementations to assist the synthesis search.

4. We have shown, via program synthesis, how one can reconstitute a program that is at a higher level from an intermediate representation (IR) that is at a lower level. (We have demonstrated this synthesis-based approach to program reconstitution in the context of partial evaluation and slicing.) This approach to program reconstitution becomes particularly useful when translation from low-level IR fragments to high-level code constructs is not straightforward, and the search space of high-level code constructs is enormous.

5. We have proposed a symbolic approach to program specialization. This approach to specialization becomes particularly useful when the specializer wants to treat certain inputs as static inputs, but somehow suppress their residuation.

The remaining chapters expand on these contributions. Chapter 2 presents background material required for the remainder of the thesis. Chapter 8 presents conclusions and opportunities for future work.

2 Background

This chapter presents a primer on Intel's IA-32 ISA (§2.1), and a logic in which specifications and semantics of instruction sequences are expressed (§2.2).

2.1 IA-32 Primer

IA-32 is the 32-bit subset of Intel's x86 ISA. This section briefly describes the syntax and concrete operational semantics of IA-32 instructions.

2.1.1 Syntax

IA-32 has some general-purpose registers (EAX, EBX, etc.), and some special-purpose registers: ESP is the stack pointer, EBP is the frame pointer, and EIP is the program counter. Some common IA-32 instructions are given in Fig. 2.1. The `push` instruction pushes an

```
1:  push ebp       3:  sub esp, 16    5:  lea eax, [esp+4]   7:  jmp 1000
2:  mov ebp, esp   4:  mov [esp], 1   6:  cmp eax, ebx       8:  jz 1000
```

Figure 2.1: Some common IA-32 instructions.

operand on top of the stack; the IA-32 stack grows upwards. In many instructions with more than one operand, the operand on the left is the destination. For example, instruction `2` in Fig. 2.1 copies a 32-bit value from ESP to EBP, and instruction `3` decrements ESP by 16 (and sets flags suitably). An exception is the `cmp` instruction, in which both operands are source operands; for example, instruction `6` in Fig. 2.1 compares the values in registers EAX and EBX. Square brackets denote memory operands; for example, instruction `4` in Fig. 2.1 writes the value 1 in the memory location pointed to by ESP. An exception is the `lea` instruction, in which square brackets denote "the effective address of." For example, instruction `5` in Fig. 2.1 loads the effective address of ESP + 4 in the EAX register. The effective address is computed as $\mathit{base} + \mathit{index} * \mathit{scale} + \mathit{offset}$, where *base* and *index* are registers, and *scale* and *offset* are integers. The `jmp` instruction (instruction `7` in Fig. 2.1) jumps to the operand target-address unconditionally, while the `jz` instruction (instruction `8` in Fig. 2.1) jumps to the operand target-address only if the zero flag (ZF) is set.

$$\mathcal{I}[\![\texttt{push ebp}]\!]\,\sigma \equiv \textit{let } \sigma' = \textit{update}_{reg}(\sigma, [\![\texttt{esp}]\!], \textit{access}_{reg}(\sigma, [\![\texttt{esp}]\!]) - \mathbf{4}) \textit{ in } \textit{update}_{mem}(\sigma', \textit{access}_{reg}(\sigma', [\![\texttt{esp}]\!]), \textit{access}_{reg}(\sigma, [\![\texttt{ebp}]\!]))$$

$$\mathcal{I}[\![\texttt{mov ebp, esp}]\!]\,\sigma \equiv \textit{update}_{reg}(\sigma, [\![\texttt{ebp}]\!], \textit{access}_{reg}(\sigma, [\![\texttt{esp}]\!]))$$

$$\mathcal{I}[\![\texttt{sub esp, 16}]\!]\,\sigma \equiv \textit{let } \sigma' = \textit{update}_{reg}(\sigma, [\![\texttt{esp}]\!], \textit{access}_{reg}(\sigma, [\![\texttt{esp}]\!]) - \mathbf{16}) \textit{ in } \textit{update}_{flag}(\sigma', [\![\texttt{ZF}]\!], \textit{access}_{reg}(\sigma', [\![\texttt{esp}]\!]) = \mathbf{0})$$

$$\mathcal{I}[\![\texttt{mov [esp], 1}]\!]\,\sigma \equiv \textit{update}_{mem}(\sigma, \textit{access}_{reg}(\sigma, [\![\texttt{esp}]\!]), \mathbf{1})$$

$$\mathcal{I}[\![\texttt{lea eax, [esp+4]}]\!]\,\sigma \equiv \textit{update}_{reg}(\sigma, [\![\texttt{eax}]\!], \textit{access}_{reg}(\sigma, [\![\texttt{esp}]\!]) + \mathbf{4})$$

$$\mathcal{I}[\![\texttt{cmp eax, ebx}]\!]\,\sigma \equiv \textit{update}_{flag}(\sigma, [\![\texttt{ZF}]\!], (\textit{access}_{reg}(\sigma, [\![\texttt{eax}]\!]) - \textit{access}_{reg}(\sigma, [\![\texttt{ebx}]\!])) = \mathbf{0})$$

$$\mathcal{I}[\![\texttt{jmp 1000}]\!]\,\sigma \equiv \textit{update}_{reg}(\sigma, [\![\texttt{eip}]\!], \mathbf{1000})$$

$$\mathcal{I}[\![\texttt{jz 1000}]\!]\,\sigma \equiv \textit{update}_{reg}(\sigma, [\![\texttt{eip}]\!], \textit{access}_{flag}(\sigma, \mathbf{ZF})\ \mathbf{?}\ \mathbf{1000}\ \mathbf{:}\ \textit{access}_{reg}(\sigma, [\![\texttt{eip}]\!]) + \mathbf{4})$$

Figure 2.2: Valuation functions for instructions in Fig. 2.1.

2.1.2 Semantics

The primitive domains of the IA-32 semantics include 32-bit integers, Booleans, registers, and flags. The primitive domains and their operators are given below. Note that the domains and their elements are in boldface to distinguish them from their syntactic counterparts in the logic that will be defined in §2.2.

$$\mathbf{i} \in \mathbf{INT} = \mathbb{Z}_{32} \qquad \mathbf{b} \in \mathbf{BOOL} = \{\mathbf{True}, \mathbf{False}\}$$

$$\mathbf{r} \in \mathbf{Register} = \{\mathbf{EAX}, \mathbf{ESP}, \ldots\} \quad \mathbf{f} \in \mathbf{Flag} = \{\mathbf{CF}, \mathbf{SF}, \ldots\}$$

$$\mathbf{op} \in \mathbf{ArithOp} = \{+, -, \ldots\} \quad \mathbf{bop} \in \mathbf{BoolOp} = \{\wedge, \vee, \ldots\}$$

$$\mathbf{rop} \in \mathbf{RelOp} = \{=, \neq, <, \ldots\} \quad \mathbf{cop} \in \mathbf{CondOp} = \{\mathbf{?}\ \mathbf{:}\}$$

Store is a compound domain that denotes an IA-32 state. (In all chapters except Chapter 6, the term “state" will be used to refer to an IA-32 *Store*.) *Store* is a triple consisting of three maps: a register map, a flag map, and a memory map. *Store* has operators to access and update the maps. The *Store* domain is defined below.

$$\sigma \in \textit{Store} = \textit{RegMap} \times \textit{FlagMap} \times \textit{MemMap}$$

$$\textit{RegMap} : \mathbf{Register} \to \mathbf{INT} \quad \textit{FlagMap} : \mathbf{Flag} \to \mathbf{BOOL} \quad \textit{MemMap} : \mathbf{INT} \to \mathbf{INT}$$

The valuation function $\mathcal{I}[\![\cdot]\!]$ has the type $\mathcal{I}$: `Instruction` $\to$ *Store* $\to$ *Store*, where `Instruction` is the type for a syntactic instruction. $\mathcal{I}$ takes an instruction i and a pre-state, and returns a post-state that reflects the updates made by the execution of i. The valuation functions for the instructions in Fig. 2.1 are given in Fig. 2.2. In the valuation functions, the overloaded function $[\![\cdot]\!]$ returns primitive semantic objects for their syntactic counterparts.

$$T \in \mathit{Term},\ \varphi \in \mathit{Formula},\ FE \in \mathit{FuncExpr}$$
$$c \in \mathit{Int32} = \{..., \text{-}1, 0, 1, ...\} \quad b \in \mathit{Bool} = \{\mathit{True}, \mathit{False}\}$$
$$I_{\mathit{Int32}} \in \mathit{Int32Id} = \{\mathit{EAX}, \mathit{ESP}, \mathit{EBP}, \ldots, m, n, \ldots\}$$
$$I_{\mathit{Bool}} \in \mathit{BoolId} = \{\mathit{CF}, \mathit{SF}, \ldots, x, y, \ldots\} \quad F \in \mathit{FuncId} = \{\mathit{Mem}\}$$
$$\mathit{op} \in \mathit{BinOp} = \{+, -, ...\} \quad \mathit{bop} \in \mathit{BoolOp} = \{\wedge, \vee, ...\} \quad \mathit{rop} \in \mathit{RelOp} = \{=, \neq, <, >, ...\}$$

$$T ::= c \mid I_{\mathit{Int32}} \mid T_1\, \mathit{op}\, T_2 \mid \mathit{ite}(\varphi, T_1, T_2) \mid F(T_1)$$
$$\varphi ::= b \mid I_{\mathit{Bool}} \mid T_1\, \mathit{rop}\, T_2 \mid \neg\varphi_1 \mid \varphi_1\, \mathit{bop}\, \varphi_2 \mid F = FE$$
$$FE ::= F \mid FE_1[T_1 \mapsto T_2]$$

Figure 2.3: Syntax of L[IA-32].

Instructions `1` through `6` in Fig. 2.1 also increment the program counter EIP by the length of the instruction. For brevity, we do not show this increment explicitly in Fig. 2.2. We also do not show all the flags updates done by the `cmp` and `sub` instructions.

2.2 Quantifier-Free Bit-Vector (QFBV) Logic Formulas

Input specifications to a machine-code synthesizer can be expressed formally by QFBV formulas. In this section, we describe the syntax and semantics of such formulas.

2.2.1 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.), some represent flags (*CF*, *SF*, etc.), and some are free constants (m, n, x, y, etc.). L[IA-32] has only one function symbol "*Mem*," which denotes memory. The syntax of L[IA-32] is defined in Fig. 2.3. A term of the form $\mathit{ite}(\varphi, T_1, T_2)$ represents an if-then-else expression. A *FuncExpr* of the form $FE[T_1 \mapsto T_2]$ denotes a *function-update* expression.

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 a specification is a restricted 2-vocabulary formula that specifies a state transformation. It has the form

$$\bigwedge_m (I'_m = T_m) \wedge \bigwedge_n (J'_n = \varphi_n) \wedge \mathit{Mem}' = FE, \tag{2.1}$$

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. For example, the QFBV formula for the specification "push the 32-bit value in the frame-pointer register EBP onto the stack" is given below.

$$ESP' = ESP - 4 \;\wedge\; Mem' = Mem[ESP - 4 \mapsto EBP] \tag{2.2}$$

In this section, and in the rest of the dissertation, we will 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*. Because we do not want the synthesizer output to be restricted to an instruction sequence that is located at a specific address, specifications do not contain conjuncts of the form $EIP' = T$. (Recall that EIP is the program counter for IA-32.)

Expressing semantics of instruction sequences. In addition to input specifications, one can use L[IA-32] formulas to express the semantics of instruction sequences. The function $\langle\!\langle \cdot \rangle\!\rangle$ encodes a given IA-32 instruction sequence as a QFBV formula. The QFBV formulas for the instructions in Fig. 2.1 are given below. Note that the formulas given below are merely QFBV transcriptions of the valuation functions given in Fig. 2.2.

$$\begin{aligned}
\langle\!\langle \texttt{push ebp} \rangle\!\rangle &\equiv ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto EBP] \\
\langle\!\langle \texttt{mov ebp, esp} \rangle\!\rangle &\equiv EBP' = ESP \\
\langle\!\langle \texttt{sub esp, 16} \rangle\!\rangle &\equiv ESP' = ESP - 16 \wedge ZF' = (ESP - 16 = 0) \\
\langle\!\langle \texttt{mov [esp], 1} \rangle\!\rangle &\equiv Mem' = Mem[ESP \mapsto 1] \\
\langle\!\langle \texttt{lea eax, [esp+4]} \rangle\!\rangle &\equiv EAX' = ESP + 4 \\
\langle\!\langle \texttt{cmp eax, ebx} \rangle\!\rangle &\equiv ZF' = (EAX - EBX = 0) \\
\langle\!\langle \texttt{jmp 1000} \rangle\!\rangle &\equiv EIP' = 1000 \\
\langle\!\langle \texttt{jz 1000} \rangle\!\rangle &\equiv EIP' = ite(ZF, 1000, EIP + 4)
\end{aligned}$$

While others have created such encodings by hand (e.g., [74]), the tools described in this dissertation 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 [54]).

Certain IA-32 string instructions contain an implicit "microcode loop", e.g., instructions with the `rep` prefix, which perform an *a priori* unbounded amount of work determined by the value in the ECX register at the start of the instruction. In other applications that use

the infrastructure on which our tools are built, this implicit microcode loop is converted into an explicit loop whose body is an instruction that performs the actions performed by the body of the microcode loop. (More details about this conversion is available elsewhere [54, §6].) However, the semantics of such instructions cannot be expressed as a single QFBV formula. Because of this expressibility limitation, the synthesis algorithms described in this dissertation do not try to synthesize instructions that use the `rep` prefix.

2.2.2 Semantics

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, we pretend that each memory location holds a 32-bit integer; however in our tools, memory is addressed at the level of individual bytes.) The meaning of a QFBV formula in L[IA-32] is a set of machine-state pairs ($\langle$pre-state, post-state$\rangle$) that satisfy the formula. Recall that an IA-32 machine-state is a triple of the form:

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

where *RegMap*, *FlagMap*, and *MemMap* map each register, flag, and memory location in the state, respectively, to a value. A $\langle$pre-state, post-state$\rangle$ pair that satisfies Eqn. (2.2) is

$$\sigma \equiv \langle [\boldsymbol{ESP} \mapsto \boldsymbol{100}][\boldsymbol{EBP} \mapsto \boldsymbol{200}],\ [\,],\ [\,] \rangle$$
$$\sigma' \equiv \langle [\boldsymbol{ESP} \mapsto \boldsymbol{96}][\boldsymbol{EBP} \mapsto \boldsymbol{200}],\ [\,],\ [\boldsymbol{96} \mapsto \boldsymbol{200}] \rangle.$$

By convention, all locations for which the range value is not shown explicitly in a state have the value 0.

Part I

Algorithms

3 Synthesis of Machine Code from Semantics

This chapter presents (i) a technique to synthesize a straight-line machine-code instruction sequence from a semantic specification of the desired behavior, given as a QFBV formula, and (ii) a tool called McSynth that implements the aforementioned technique. The synthesized instruction-sequence implements the input QFBV formula (i.e., is equivalent to the QFBV formula). McSynth is parameterized by the ISA of the target instruction-sequence, and is easily adaptable to work on other semantic representations, such as a Universal Assembly Language (UAL) [23].

The synthesis algorithm used in McSynth is the baseline machine-code-synthesis algorithm presented in this dissertation. Successive chapters explore improvements to this baseline algorithm.

Contributions

The contributions of this chapter include the following:

- Our machine-code-synthesis technique is the first of its kind to be applied to the integer subset of a full ISA.
- The core synthesis loop of our technique is a new instantiation of the *counterexample-guided inductive synthesis* (CEGIS) framework (§3.2.1).
- We have developed *footprint-based* pruning heuristics to prune away useless candidates, and reduce the synthesis search-space (§3.2.2).
- To counter the exponential cost of enumerative strategies, we have developed a *divide-and-conquer* strategy to divide a QFBV formula into independent sub-formulas, and synthesize instructions for the sub-formulas (§3.2.3). This strategy has been shown to reduce the synthesis time by several orders of magnitude.

Our methods have been implemented in McSynth, a machine-code synthesizer for Intel's IA-32 ISA. We tested McSynth on QFBV formulas obtained from basic blocks in the SPECINT 2006 benchmark suite. We found that, on an average, McSynth's footprint-based search-space-pruning heuristic reduces the synthesis time by a factor of 473, and McSynth's divide-and-conquer strategy reduces synthesis time by a further 3 to 5 orders of magnitude (§3.4).

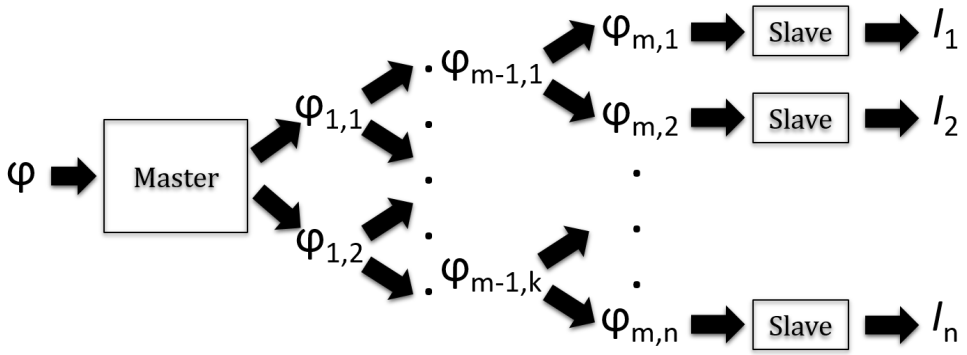


Figure 3.1: Master-slave architecture of McSynth.

In comparison to the x86 peephole superoptimizer [16] (the only tool whose search space is comparable to that of McSynth), which takes several hours to synthesize an instruction sequence of length up to 3, McSynth can synthesize certain instruction sequences of length up to 10 in a few minutes. We have also built an IA-32 partial evaluator (Chapter 6), and an IA-32 slicer (Chapter 7) as clients of McSynth.

3.1 Overview

McSynth uses enumerative strategies for synthesis. However, an ISA like IA-32 has around 43,000 unique instruction schemas, which, when combined with the exponential cost inherent in enumeration, results in an enormous search space for synthesis. McSynth attempts to cope with the enormous search space using a *master-slave* architecture. The design of McSynth is depicted in Fig. 3.1. Given a QFBV formula φ, McSynth synthesizes an instruction sequence for φ in the following way:

1. The master uses a divide-and-conquer strategy to split φ into independent sub-formulas, and hands over each sub-formula to a slave synthesizer.

2. The slave uses enumeration, along with an instantiation of the CEGIS framework and footprint-based search-space pruning heuristics to synthesize code for a sub-formula.

3. If a slave times out, the master uses an alternative split. (For example in Fig. 3.1, if synthesis of code for $\varphi_{m,1}$ times out, the master tries out an alternative split for $\varphi_{m-1,1}$.) If all candidate splits for a sub-formula time out, the master hands over the entire sub-formula to a slave. (For example in Fig. 3.1, if all candidate splits of $\varphi_{m-1,1}$ time out, the master supplies $\varphi_{m-1,1}$ as an input to a slave.)

4. The master concatenates the results produced by the slaves, and returns the final instruction sequence.

In the remainder of this section, we present an example to illustrate McSynth's algorithm.

In procedure calls, a common idiom in the prologue of the callee is to save the frame pointer of the caller, and initialize its own frame pointer. A QFBV formula φ for this idiom is

$$\varphi \equiv ESP' = ESP - 4 \wedge EBP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto EBP]. \tag{3.1}$$

A naïve enumerative synthesizer will take a few days to find an implementation for φ. Using φ as a running example, we will first illustrate how McSynth's master works. We will then illustrate how the slave works.

Master

The divide-and-conquer strategy tries to split the updates in φ across a sequence of sub-formulas $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ such that if one were to synthesize an instruction sequence I_i for each φ_i *independently*, and concatenate the synthesized instruction-sequences in the same order, the result will be equivalent to φ. Such a split is called a *legal* split. A sufficient condition for a legal split is *flow independence*—if we can split the updates in φ across the sequence $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ such that there is no flow dependence from φ_i to any successor sub-formula φ_j ($i < j$), the split is legal. The reason for this sufficiency is as follows: because I_j is equivalent to some sub-formula φ_j of φ, and I_j does not read any of the locations that could be modified by any predecessor instruction-sequence I_i ($i < j$), $I_1; I_2; \ldots; I_k$ performs the same state transformations as φ.

McSynth uses the following one-sided decision procedure to check if a split $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ is flow-independent. The decision procedure returns MAYBE if either (or both) of the following conditions hold:

C1: φ_j might use a register or flag, and a predecessor sub-formula φ_i ($i<j$) might modify the same register or flag

C2: φ_j might use *some* memory location, and a predecessor sub-formula φ_i ($i<j$) might modify *some* memory location

If neither of the conditions hold, the decision procedure returns NO (meaning the split is flow-independent). One can see that the one-sided decision procedure is very conservative

in the way it handles flow dependences through memory. This limitation is addressed in Chapter 4.

One possible way to split φ is as $\langle \varphi_1, \varphi_2, \varphi_3 \rangle$, where

$$\varphi_1 \equiv ESP' = ESP - 4 \quad \varphi_2 \equiv EBP' = ESP - 4 \quad \varphi_3 \equiv Mem' = Mem[ESP - 4 \mapsto EBP].$$

However, McSynth's decision procedure returns MAYBE for the split because φ_2 and φ_3 both use *ESP*, which is killed by φ_1. Consequently McSynth discards this split.

Another possible way to split φ is as $\langle \varphi_1, \varphi_2, \varphi_3 \rangle$, where

$$\varphi_1 \equiv Mem' = Mem[ESP - 4 \mapsto EBP] \quad \varphi_2 \equiv EBP' = ESP - 4 \quad \varphi_3 \equiv ESP' = ESP - 4.$$

This split does not introduce any extraneous flow dependences, and consequently, McSynth's decision procedure returns NO for this split (meaning that the split is flow-independent). McSynth's master supplies the sub-formulas φ_1, φ_2, and φ_3, respectively, to slave synthesizers.

Slave

Given a sub-formula φ_i and—for pragmatic reasons—a timeout value, McSynth's slave synthesizer either synthesizes an instruction sequence that implements φ_i, or returns FAIL if it could not find such an instruction sequence before the timeout expires. The slave synthesizes an instruction sequence for φ_i using the following method:

1. The slave enumerates *templatized* instruction-sequences of increasing length. A templatized instruction-sequence is a sequence of instructions with template operands (or holes) instead of one or more constant values.

2. The slave attempts to find an instantiation of a candidate templatized instruction-sequence that is logically equivalent to φ_i using CEGIS. If an instantiation is found, the slave returns it. Otherwise, the next templatized sequence is considered.

3. The slave uses heuristics based on the *footprints* of QFBV formulas (see below) to prune away useless candidates during enumeration.

In the remainder of this section, we illustrate the working of a slave synthesizer using φ_2 as an example.

The slave starts enumerating templatized one-instruction sequences. To prune away candidates during enumeration, McSynth uses an overapproximation of locations that

could potentially be touched by a formula. We define the *abstract semantic use-footprint* $\text{SFP}^{\#}_{\text{USE}}$ (*kill-footprint* $\text{SFP}^{\#}_{\text{KILL}}$) as an overapproximation of the locations used (modified) by a formula. Concretely, an abstract semantic-footprint of a formula φ is a subset of the set of constant symbols in φ (except symbolic constants) and a special symbol "*Mem*," which denotes the entire memory. The symbols in $\text{SFP}^{\#}_{\text{KILL}}$ are primed.

Let us assume that the first candidate enumerated by the slave is $C_1 \equiv$ "`mov eax, ⟨Imm32⟩`." The formula ψ_1 for C_1 is $EAX' = m$. The abstract semantic footprints for φ_2 and ψ_1, respectively, are given below.

$$\text{SFP}^{\#}_{\text{USE}}(\varphi_1) = \{ESP\} \quad \text{SFP}^{\#}_{\text{KILL}}(\varphi_1) = \{EBP'\}$$
$$\text{SFP}^{\#}_{\text{USE}}(\psi_1) = \{\} \quad \text{SFP}^{\#}_{\text{KILL}}(\psi_1) = \{EAX'\}$$

One can see that the abstract semantic KILL-footprint of ψ_1 is outside that of φ_2, and thus C_1 can never implement φ_2 without possibly modifying a value in a location that is otherwise unmodified by φ_2. Therefore, the slave prunes away C_1 because it is a *useless candidate*. Moreover, regardless of the instruction sequence that is appended to C_1, the resulting instruction sequence would always be discarded at this step. We call instruction sequences such as C_1 *useless prefixes*. By discarding useless prefixes, any future candidate enumerated by the slave has only *useful prefixes* as its prefix.

Suppose that the next candidate is $C_2 \equiv$ "`mov ebp, [esp]`." The formula ψ_2 for C_2 is $EBP' = Mem(ESP)$. The abstract semantic footprints for ψ_2 are given below.

$$\text{SFP}^{\#}_{\text{USE}}(\psi_1) = \{ESP, Mem\} \quad \text{SFP}^{\#}_{\text{KILL}}(\psi_1) = \{EBP'\}$$

The abstract semantic USE-footprint of ψ_2 is outside that of φ_2 because ψ_2 might use some memory location, but φ_2 does not use any memory location. Therefore, the slave also prunes away the useless candidate C_2.

Suppose that the next candidate is $C_3 \equiv$ "`mov ebp, ⟨Imm32⟩`." The formula ψ_3 for C_3 is $EBP' = m$. The abstract semantic USE/KILL-footprints of ψ_3 are within those of φ_2. So the slave uses a CEGIS-based loop to check if there exists an instantiation of the candidate that implements φ_2. The CEGIS loop says that no such instantiation exists, and so the slave discards C_3.

The slave eventually enumerates the candidate $C_4 \equiv$ "`lea ebp, [esp-⟨Imm32⟩]` The CEGIS loop returns the instantiation "`lea ebp, [esp-4]`" of C_4 as the synthesized code for φ_2.

In a similar manner, the slave synthesizes the instruction sequences "`mov [esp-4], ebp`" and "`lea esp, [esp-4]`" for φ_1 and φ_2, respectively. McSynth concatenates the results pro-

duced by the slaves and returns "`mov [esp-4],ebp; lea ebp,[esp-4]; lea esp,[esp-4]`" as the implementation. McSynth completes the overall synthesis task in a few seconds; that is a *four-orders-of-magnitude speedup* compared to a naïve enumerative synthesizer.

3.1.1 The Role of Templatized Instruction-Sequences

In other work on synthesis, "templates" are sometimes used to restrict the set of possible outcomes, and thereby cause synthesis algorithms to be incomplete. In our work, a templatized instruction-sequence is merely a sequence of templatized *instructions*, where the set of templatized instructions spans the full IA-32 instruction set. For example, the templatized instruction "`mov eax, ⟨Imm32⟩`" represents four billion instructions "`mov eax, 0`", "`mov eax, 1`", ... "`mov eax, 4294967295`". Each templatized instruction is created by lifting a single instruction from an immediate operand to a template operand.

Because the templatized instructions still span the full IA-32 instruction set, the templatized instruction-sequences span the full set of IA-32 instruction sequences, hence the use of templates in our work does not cause our algorithms to be incomplete.

3.2 Algorithm

In this section, we describe the algorithms used by McSynth. First, we present the algorithm for McSynth's synthesis loop. Second, we present the heuristics used by McSynth to prune the synthesis search-space. Third, we describe McSynth's divide-and-conquer strategy, and present the full algorithm used by McSynth.

3.2.1 Synthesis Loop

We start by presenting a naïve algorithm for synthesizing machine code from a QFBV formula; we then present a few refinements to obtain the algorithm actually used in McSynth.

Base Algorithm

Given an input QFBV formula φ, a naïve first cut is to enumerate every concrete instruction-sequence in the ISA, convert the instruction sequence into a QFBV formula ψ, and use an SMT solver to check the validity of the formula $\varphi \Leftrightarrow \psi$. The unhighlighted lines of Alg. 3.2 show this strawman algorithm.

```
Input: φ
Output: C_conc
 1: 𝒯 ← ∅
 2: for each concrete instruction-sequence C_conc in the ISA do
 3:   ψ ← ⟨⟨C_conc⟩⟩
 4:   if not TestsPass(ψ, 𝒯) then
 5:     continue
 6:   end if
 7:   model ← SAT(¬(φ ⇔ ψ))
 8:   if model = ⊥ then
 9:     return C_conc
10:   else
11:     𝒯 ← 𝒯 ∪ model
12:   end if
13: end for
```

Algorithm 3.2: Strawman algorithm to synthesize instructions from a QFBV formula

McSynth uses an SMT solver to check the satisfiability of a QFBV formula. (Validity queries are expressed using negated satisfiability queries.) SMT queries are represented in the algorithms by calls to the function `SAT`. If a formula is satisfiable, the SMT solver returns a model. If the query posed to the SMT solver is a satisfiability query, the model is treated as a satisfying assignment. If the query is a validity query, the model is a counterexample to validity.

One optimization is to use the counterexamples produced by the SMT solver as test cases to reduce future calls to the solver. Evaluating a QFBV formula using a test case can be performed much faster than obtaining an answer from an SMT solver. McSynth maintains a finite set of test cases $\mathscr{T}$. (Note that $\langle \sigma , \sigma' \rangle \models \varphi$, for all $\langle \sigma , \sigma' \rangle \in \mathscr{T}$.) McSynth evaluates ψ with respect to each test $\langle \sigma , \sigma' \rangle$ in $\mathscr{T}$ to check if $\langle \sigma , \sigma' \rangle \models \psi$ (i.e., φ and ψ produce identical post-states for each test in $\mathscr{T}$). If all the tests pass, ψ is checked for equivalence with φ (Line 7); otherwise, it is discarded. The strawman algorithm, along with this optimization, is shown in Alg. 3.2. In Alg. 3.2, `TestsPass` evaluates ψ with respect to each test in $\mathscr{T}$.

CEGIS

The search space of Alg. 3.2 is clearly enormous. Almost all ISAs support immediate operands in instructions, and this results in thousands of distinct instructions with the same opcode. To reduce the search space, instead of enumerating concrete instruction-sequences, the synthesizer can enumerate templatized instruction-sequences. A templatized instruction-sequence can be treated as a partial program, or a sketch [85]. CEGIS is a popular synthesis framework that has been widely used in the completion of partial programs. The basic idea of CEGIS is the following: Given (i) a specification φ, (ii) a finite

set of tests $\mathscr{T}$ for the specification, and (iii) a partial program C that needs to be completed, CEGIS tries to find a completion (values for holes in the partial program) C_{conc} that passes the tests. Then, it checks if C_{conc} meets the specification using an SMT solver. If it does, C_{conc} is returned. Otherwise, it adds the counterexample returned by the solver to $\mathscr{T}$, and tries to find another completion. This loop proceeds until no more completions are possible. The rest of this sub-section describes how we have instantiated the CEGIS framework to synthesize machine code in McSynth.

Given φ, McSynth bootstraps its test suite $\mathscr{T}$ with the test $\langle \sigma_0 , \sigma_0' \rangle$. σ_0 is a machine-code state in which all locations are mapped to $\boldsymbol{0}$. McSynth computes σ_0' by substituting σ_0 in φ. The inputs to McSynth's CEGIS loop are φ, the test suite $\mathscr{T}$, a templatized sequence C, and its QFBV formula $\psi \equiv \langle\!\langle C \rangle\!\rangle$.

Checking a candidate against $\mathscr{T}$. Given ψ and $\mathscr{T}$, McSynth simplifies ψ with respect to $\mathscr{T}$ to create $\psi^{\mathscr{T}}$ as follows: Starting with $\psi^{\mathscr{T}} \equiv \mathit{true}$, McSynth iterates through each test $\langle \sigma, \sigma' \rangle \in \mathscr{T}$: McSynth simplifies ψ with respect to $\langle \sigma, \sigma' \rangle$, and conjoins the simplified ψ to $\psi^{\mathscr{T}}$. McSynth then checks the satisfiability of $\psi^{\mathscr{T}}$ using an SMT solver. If $\psi^{\mathscr{T}}$ is unsatisfiable, there exists no instantiation of C that passes all tests in $\mathscr{T}$. If $\psi^{\mathscr{T}}$ is satisfiable, McSynth substitutes the satisfying assignment returned by the SMT solver in C and ψ to obtain C_{conc} and ψ_{conc}, respectively. For each test in $\mathscr{T}$, C_{conc} and ψ_{conc} produce the same post-state as φ.

Because states have memory arrays, simplifying ψ with respect to $\mathscr{T}$ is not straightforward. In the rest of this sub-section, we describe how McSynth simplifies a formula with respect to a set of tests. We present three approaches for simplification: (i) An ideal approach that cannot be implemented for states that have many memory locations, (ii) a naïve approach that produces false-positives (it says that there exists an instantiation of C that is equivalent to φ with respect to $\mathscr{T}$, even when one does not exist), and (iii) the approach used by McSynth, which does not produce false-positives, and can be implemented.

To illustrate these approaches, suppose that φ is

$$\varphi \equiv EAX' = Mem(ESP) \wedge Mem' = Mem[EBP \mapsto EBX].$$

Let us also assume that $\mathscr{T}$ has only one test case.[1]

$$\sigma : [[\boldsymbol{ESP} \mapsto \boldsymbol{100}][\boldsymbol{EBP} \mapsto \boldsymbol{200}][\boldsymbol{EBX} \mapsto \boldsymbol{1}], [\,], [\boldsymbol{100} \mapsto \boldsymbol{2}]]$$
$$\sigma' : [[\boldsymbol{EAX} \mapsto \boldsymbol{2}][\boldsymbol{ESP} \mapsto \boldsymbol{100}][\boldsymbol{EBP} \mapsto \boldsymbol{200}][\boldsymbol{EBX} \mapsto \boldsymbol{1}], [\,], [\boldsymbol{100} \mapsto \boldsymbol{2}][\boldsymbol{200} \mapsto \boldsymbol{1}]]$$

Consider our first candidate $C_1 \equiv$ "`mov eax, [esp]; mov [esp], ebx`." C_1 copies a 32-bit

[1] Recall that any location not shown in a state is mapped to the value 0.

value from the location pointed to by the stack-pointer register ESP to the register EAX, and a 32-bit value from the EBX register to the location pointed to by the frame-pointer register EBP. The QFBV formula ψ_1 for C_1 is

$$\psi_1 \equiv \langle\langle C_1 \rangle\rangle \equiv EAX' = Mem(ESP) \wedge Mem' = Mem[ESP \mapsto EBX].$$

Our goal is to simplify ψ_1 with respect to $\langle \sigma, \sigma' \rangle$ to obtain the simplified formula $\psi_1^{\langle \sigma, \sigma' \rangle}$.

Approach 1. Suppose that we have a function χ that converts a state into a QFBV formula. One way to obtain $\psi_1^{\langle \sigma, \sigma' \rangle}$ is to convert σ and σ' into QFBV formulas (using the function χ), and conjoin the resulting formulas with ψ_1.

$$\psi_1^{\langle \sigma, \sigma' \rangle} \equiv \psi_1 \wedge \chi(\sigma, 0) \wedge \chi(\sigma', 1) \tag{3.2}$$

Note that χ also takes a vocabulary index as an input (the pre-state is vocabulary 0; the post-state is vocabulary 1). The symbols in the QFBV formula produced by χ are in the specified vocabulary. We can define χ as follows:

$$\chi(\sigma, voc) = \chi_{RegFlag}(\sigma, voc) \wedge \chi_{Mem}(\sigma, voc)$$

$\chi_{RegFlag}$ converts the register and flag maps into a QFBV formula; χ_{Mem} converts the memory map into a QFBV formula.

The implementation of $\chi_{RegFlag}$ is straightforward: for each register (flag), generate a constraint using the value of the register (flag) from the argument state. For example,

$$\chi_{RegFlag}(\sigma, 0) \equiv ESP = 100 \wedge EBP = 200 \wedge EBX = 1.$$

One possible way of implementing χ_{Mem} is the following: for every location l in the memory array, generate a constraint on index l of an uninterpreted array symbol Mem.

$$\chi_{Mem}(\sigma, 0) \equiv Mem(0) = 0 \wedge Mem(4) = 0 \wedge \ldots \wedge Mem(100) = 2 \wedge Mem(104) = 0 \wedge \ldots$$

In most ISAs, addressable memory is usually 2^{32} or 2^{64} bytes long. One way to prevent χ_{Mem} from returning enormous formulas is to use a universal quantifier in the formula. However, off-the-shelf SMT solvers cannot be used to check the satisfiability of the resulting formula. Consequently, we need to devise a different approach.

Approach 2. We could use $\chi_{RegFlag}$ in place of χ.

$$\psi_1^{\langle \sigma, \sigma' \rangle} \equiv \psi_1 \wedge \chi_{RegFlag}(\sigma, 0) \wedge \chi_{RegFlag}(\sigma', 1) \tag{3.3}$$

However, Eqn. (3.2) and Eqn. (3.3) are not equisatisfiable. This approach results in false positives. Because Eqn. (3.3) is satisfiable, this approach would conclude that ψ_1 is equivalent to φ with respect to $\langle \sigma, \sigma' \rangle$, even though it is not.

Approach 3. To obtain a simplified formula that is equisatisfiable with the one in Eqn. (3.2), McSynth uses a procedure SimplifyWithTest. SimplifyWithTest generates constraints only for memory locations that are accessed or updated by a QFBV formula for a test case. We illustrate SimplifyWithTest by simplifying ψ_1 with respect to $\langle \sigma, \sigma' \rangle$. First, SimplifyWithTest conjoins ψ_1 with $\chi_{RegFlag}(\sigma, 0)$ and $\chi_{RegFlag}(\sigma', 1)$ to obtain the following formula:

$$2 = Mem(100) \wedge Mem' = Mem[100 \mapsto 1] \tag{3.4}$$

The only memory location that is accessed or updated in Eqn. (3.4) is 100. For this location, SimplifyWithTest generates the following constraints from $\langle \sigma, \sigma' \rangle$.

$$Mem(100) = 2 \wedge Mem'(100) = 2 \tag{3.5}$$

SimplifyWithTest conjoins Eqn. (3.4) and Eqn. (3.5) to obtain

$$\psi_1^{\langle \sigma, \sigma' \rangle} \equiv Mem' = Mem[100 \mapsto 1] \wedge Mem(100) = 2 \wedge Mem'(100) = 2. \tag{3.6}$$

The formulas in Eqn. (3.2) and Eqn. (3.6) are equisatisfiable because any memory location other than 100 is irrelevant to the test case. McSynth checks the satisfiability of $\psi_1^{\langle \sigma, \sigma' \rangle}$ using an SMT solver. The solver says that $\psi_1^{\langle \sigma, \sigma' \rangle}$ is unsatisfiable, which is the desired result.

Consider another candidate $C_2 \equiv$ "`mov eax, [esp]; mov [`$\langle$`Imm32`$\rangle$`], ebx`." C_2 is a template to copy a 32-bit value from the location pointed to by the stack-pointer registerESP to the register EAX, and a 32-bit value from the EBX register to a memory location with a constant address. The QFBV formula ψ_2 for C_2 is

$$\psi_2 \equiv \langle\!\langle C_2 \rangle\!\rangle \equiv EAX' = Mem(ESP) \wedge Mem' = Mem[i \mapsto EBX].$$

After conjoining ψ_2 with $\chi_{RegFlag}(\sigma, 0)$ and $\chi_{RegFlag}(\sigma', 1)$, SimplifyWithTest produces the following formula:

$$2 = Mem(100) \wedge Mem' = Mem[i \mapsto 1] \tag{3.7}$$

Two locations are accessed or updated in the formula. One is the concrete location 100, and another is the symbolic location i. The symbolic location can be any concrete location. To constrain the pre-state value at location i, McSynth generates the following constraint from

Input: ψ, $\langle \sigma, \sigma' \rangle$
Output: $\psi^{\langle \sigma, \sigma' \rangle}$
1: $\psi^{\langle \sigma, \sigma' \rangle} \leftarrow$ `Simplify`$(\psi \wedge \chi_{RegFlag}(\sigma, 0) \wedge \chi_{RegFlag}(\sigma', 1))$
2: concLocs $\leftarrow$ `ConcLocs`$(\psi^{\langle \sigma, \sigma' \rangle})$
3: concMemConstr $\leftarrow$ **true**
4: **for** each a in concLocs **do**
5: val $\leftarrow$ `Lookup`$(\sigma$, a$)$
6: val$'$ $\leftarrow$ `Lookup`$(\sigma'$, a$)$
7: concMemConstr $\leftarrow$ concMemConstr $\wedge$ Mem(a) = val $\wedge$ Mem'(a) = val$'$
8: **end for**
9: symLocs $\leftarrow$ `SymLocs`$(\psi^{\langle \sigma, \sigma' \rangle})$
10: **if** symLocs $= \emptyset$ **then**
11: **return** `Simplify`$(\psi^{\langle \sigma, \sigma' \rangle} \wedge$ concMemConstr$)$
12: **end if**
13: **return** `Simplify`$(\psi^{\langle \sigma, \sigma' \rangle} \wedge$ concMemConstr $\wedge$ `SymMemConstr` (symLocs, σ, Mem) $\wedge$ `SymMemConstr`(symLocs, σ', Mem'))

Algorithm 3.3: Algorithm SimplifyWithTest

the memory map $[\boldsymbol{100} \mapsto \boldsymbol{2}]$ in σ:

$$Mem(100) = 2 \wedge i \neq 100 \Rightarrow Mem(i) = 0 \tag{3.8}$$

To constrain the post-state value at location i, McSynth uses the memory map $[\boldsymbol{100} \mapsto \boldsymbol{2}][\boldsymbol{200} \mapsto \boldsymbol{1}]$ in σ' to generate

$$Mem'(100) = 2 \wedge Mem'(200) = 1 \wedge (i \neq 100 \wedge i \neq 200) \Rightarrow Mem'(i) = 0. \tag{3.9}$$

SimplifyWithTest conjoins Eqns. (3.7)–(3.9), and returns the resulting formula $\psi_2^{\langle \sigma, \sigma' \rangle}$. McSynth checks the satisfiability of $\psi_2^{\langle \sigma, \sigma' \rangle}$. The SMT solver says that $\psi_2^{\langle \sigma, \sigma' \rangle}$ is satisfiable, and produces the satisfying assignment $[i \mapsto 200]$. Indeed, φ and ψ_2 are equivalent with respect to $\langle \sigma, \sigma' \rangle$ when $i = 200$.

The algorithm for SimplifyWithTest is shown in Alg. 3.3. In the algorithm, the function `Simplify` simplifies a formula by removing unnecessary conjuncts; `ConcLocs` identifies the set of concrete memory locations that are accessed or updated by a QFBV formula; `SymLocs` identifies the set of symbolic memory locations that are accessed or updated by a QFBV formula; `Lookup` obtains the value present in a concrete memory location in a state; `SymMemConstr` produces the memory constraint for a set of symbolic locations. Note that `ConcLocs` and `SymLocs` collect concrete and symbolic memory locations, respectively, from all nested terms and sub-formulas (e.g., $Mem' = Mem[Mem(i) \mapsto Mem(0)]$) and not just from those at the top level.

Input: φ, C, $\psi = \langle\langle C \rangle\rangle$, $\mathscr{T}$
Output: Instantiation C_{conc} of C such that $\langle\langle C_{conc} \rangle\rangle \Leftrightarrow \varphi$, or FAIL
1: **while true do**
2: $\psi^{\mathscr{T}} \leftarrow$ **true**
3: **for** each test-case $\langle \sigma, \sigma' \rangle \in \mathscr{T}$ **do**
4: $\psi^{\mathscr{T}} \leftarrow \psi^{\mathscr{T}} \wedge \mathsf{SimplifyWithTest}(\psi, \langle \sigma, \sigma' \rangle)$
5: **end for**
6: $\text{model}_1 = \texttt{SAT}(\psi^{\mathscr{T}})$
7: **if** $\text{model}_1 = \perp$ **then**
8: **return** FAIL
9: **end if**
10: $\psi_{conc} \leftarrow \texttt{Substitute}(\psi, \text{model}_1)$
11: $\text{model}_2 \leftarrow \texttt{SAT}(\neg(\varphi \Leftrightarrow \psi_{conc}))$
12: **if** $\text{model}_2 = \perp$ **then**
13: **return** $\texttt{Substitute}(C, \text{model}_1)$
14: **end if**
15: $\mathscr{T} \leftarrow \mathscr{T} \cup \text{model}_2$
16: **end while**

Algorithm 3.4: Algorithm CEGIS

At this point, McSynth has either determined that no instance of templatized candidate C passes all tests in $\mathscr{T}$, or has a concrete instruction-sequence C_{conc} that passes all tests in $\mathscr{T}$.

The CEGIS loop. Once McSynth obtains C_{conc} (and its corresponding QFBV formula ψ_{conc}) that is equivalent to φ with respect to $\mathscr{T}$, McSynth checks if ψ_{conc} is equivalent to φ using an SMT solver. If they are equivalent, McSynth returns C_{conc}. Otherwise, McSynth adds the counterexample returned by the solver to $\mathscr{T}$, and searches for another concrete instruction-sequence that passes the tests. Alg. 3.4 show McSynth's CEGIS loop. In Alg. 3.4, the overloaded function `Substitute` substitutes a model in a templatized instruction-sequence or QFBV formula.

The full CEGIS-based algorithm to synthesize instructions from a QFBV formula is shown in the unhighlighted lines of Alg. 3.5. In the algorithm, ϵ denotes an instruction sequence with no instructions, `ReadInstrPool` reads the templatized instructions in the ISA from a file, and `Append` appends an instruction to an instruction sequence.

3.2.2 Pruning the Synthesis Search-Space

ISAs such as Intel's IA-32 have around 43,000 unique templatized instructions. For IA-32, Alg. 3.5 will make millions of calls to the SMT solver to synthesize instruction sequences that have length 2 or more. A call to an SMT solver is expensive, and this cost makes

```
Input: φ
Output: C_conc or FAIL
 1: instrPool ← ReadInstrPool( )
 2: 𝒯 ← {⟨σ_0, σ'_0⟩}
 3: prefixes ← {ϵ}
 4: while prefixes ≠ ∅ do
 5:   for each prefix p ∈ prefixes do
 6:     prefixes ← prefixes − {p}
 7:     for each templatized instruction i ∈ instrPool do
 8:       C ← Append(p, i)
 9:       ψ ← ⟨⟨C⟩⟩
10:       if SFP#_USE(ψ) ⊈ SFP#_USE(φ) ∨ SFP#_KILL(ψ) ⊈ SFP#_KILL(φ) then
11:         continue
12:       end if
13:       prefixes ← prefixes ∪ {C}
14:       ret = CEGIS(φ, C, ψ, 𝒯)
15:       if ret ≠ FAIL then
16:         return ret
17:       end if
18:     end for
19:   end for
20: end while
21: return FAIL
```

Algorithm 3.5: Algorithm McSynthSlave

Alg. 3.5 very slow. We have devised heuristics to prune the synthesis search-space, and speed up synthesis. Our heuristics have the guarantee that only useless candidates are pruned away. In this sub-section, we describe our pruning heuristics.

Abstract Semantic-Footprints

First, we define *semantic-footprints* and *abstract semantic-footprints* of QFBV formulas. The *semantic-USE-footprint* (SFP_{USE}) is the set of concrete locations (represented as constant symbols) that *are* used by the QFBV formula for *some* input. The *semantic-KILL-footprint* (SFP_{KILL}) is the set of concrete locations that *are* modified by the QFBV formula for *some* input. For the formula in Eqn. (3.1), SFP_{USE} and SFP_{KILL} are shown below (with a minor abuse of notation).

$$\text{SFP}_{\text{USE}}(\varphi) = \{ESP, EBP\} \qquad \text{SFP}_{\text{KILL}}(\varphi) = \{ESP', EBP', 0', 1', 2', \dots\}$$

$0', 1', 2', \dots$ are in SFP_{KILL} because φ might modify any memory location for some input. If a QFBV formula uses or modifies a memory location, the SFP sets could be large. *Abstract semantic-footprints* are over-approximations of semantic-footprints. The abstract semantic-USE-footprint ($\text{SFP}^{\#}_{\text{USE}}$) is an over-approximation of SFP_{USE}, and the abstract semantic-

KILL-footprint ($\text{SFP}^{\#}_{\text{KILL}}$) is an over-approximation of SFP_{KILL}. We identify $\text{SFP}^{\#}_{\text{USE}}$ and $\text{SFP}^{\#}_{\text{KILL}}$ via a syntax-directed translation over a QFBV formula. In the following definitions, *RF* (*RF′*) is the set of unprimed (primed) constant symbols used for registers and flags, and *T* is the set of QFBV terms.

Definition 3.1.

$$\text{SFP}^{\#}_{\text{USE}}(\text{C}) = \begin{cases} \{\text{C}\} & \textit{if } c \in RF \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\text{SFP}^{\#}_{\text{USE}}(Mem(t)) = \{Mem\} \cup \text{SFP}^{\#}_{\text{USE}}(t), \textit{where } t \in T$$

$$\text{SFP}^{\#}_{\text{USE}}(\text{C}' = \text{C}) = \emptyset, \textit{ where } c' \in RF', c \in RF \tag{3.10}$$

$$\text{SFP}^{\#}_{\text{USE}}(Mem' = Mem) = \emptyset \tag{3.11}$$

$$\text{SFP}^{\#}_{\text{KILL}}(\text{C}') = \begin{cases} \{\text{C}'\} & \textit{if } c' \in RF' \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\text{SFP}^{\#}_{\text{KILL}}(Mem') = \{Mem'\}$$

$$\text{SFP}^{\#}_{\text{KILL}}(\text{C}' = \text{C}) = \emptyset, \textit{ where } c' \in RF', c \in RF \tag{3.12}$$

$$\text{SFP}^{\#}_{\text{KILL}}(Mem' = Mem) = \emptyset \tag{3.13}$$

For all other cases, $\text{SFP}^{\#}_{\text{USE}}$ ($\text{SFP}^{\#}_{\text{KILL}}$) *is the union of* $\text{SFP}^{\#}_{\text{USE}}$ ($\text{SFP}^{\#}_{\text{KILL}}$) *of the constituents.*

Eqns. (3.10)–(3.13) represent the computation of $\text{SFP}^{\#}_{\text{USE}}$ and $\text{SFP}^{\#}_{\text{KILL}}$ for the identify conjuncts in a QFBV formula (representing the unmodified portions of the state). For an input QFBV formula φ, consider the set of instruction sequences $I^{\#}$ that has the following property:

$$\forall \text{C} \in I^{\#}, \text{SFP}^{\#}_{\text{USE}}(\langle\!\langle \text{C} \rangle\!\rangle) \subseteq \text{SFP}^{\#}_{\text{USE}}(\varphi) \wedge \text{SFP}^{\#}_{\text{KILL}}(\langle\!\langle \text{C} \rangle\!\rangle) \subseteq \text{SFP}^{\#}_{\text{KILL}}(\varphi)$$

The set $I^{\#}$ is depicted in Fig. 3.6 as a hexagon. If McSynth restricts the synthesis search-space to $I^{\#}$, McSynth will miss two types of candidates.

1. A candidate that is not equivalent to φ. An example of such a candidate for the QFBV formula in Eqn. (3.1) is "`mov eax, ebx`".

2. A candidate C that satisfies the following properties:

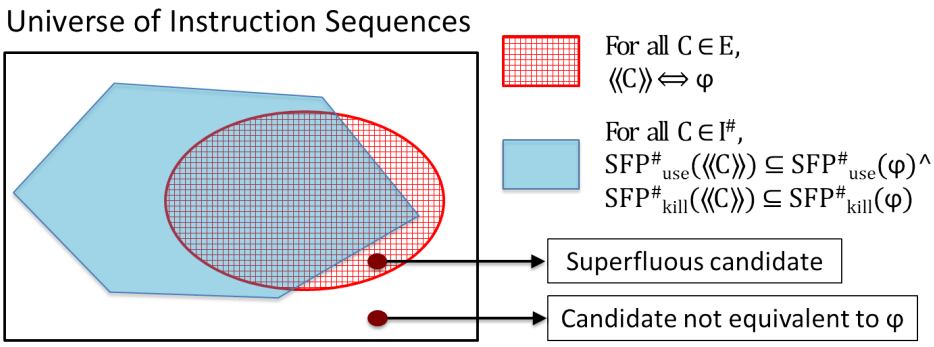


Figure 3.6: Depiction of the set $I^{\#}$.

a) $\langle\!\langle C \rangle\!\rangle \Leftrightarrow \varphi$

b) $SFP^{\#}_{USE}(\langle\!\langle C \rangle\!\rangle) \not\subseteq SFP^{\#}_{USE}(\varphi) \vee SFP^{\#}_{KILL}(\langle\!\langle C \rangle\!\rangle) \not\subseteq SFP^{\#}_{KILL}(\varphi)$

We call such a candidate *superfluous*. Although $\langle\!\langle C \rangle\!\rangle$ semantically uses and modifies the same locations as φ (because $\langle\!\langle C \rangle\!\rangle \Leftrightarrow \varphi$), the syntax of $\langle\!\langle C \rangle\!\rangle$ suggests that it uses (kills) a location that is not used (killed) by φ, and might not implement φ efficiently. Therefore, McSynth prunes away superfluous candidates. For the QFBV formula in Eqn. (3.1), "`push ebp; lea ebp,[esp+eax]; lea ebp,[ebp-eax]`" is an example of a superfluous candidate; the final value of EBP depends on the value of ESP, but does *not* depend on the value of EAX.

Useless-Prefix

Because a location modified (used) by a QFBV formula cannot be "un-modified" ("unused"), if a candidate $C \notin I^{\#}$, no matter what instruction sequence is appended to C, the resulting instruction sequence must lie outside $I^{\#}$. Thus, if McSynth finds that a candidate $C \notin I^{\#}$ during enumeration, it will never enumerate any instruction sequence with C as a prefix. (C is a useless-prefix.)

Theorem 3.7. *For any pair of instruction sequences* C_1, C_2, $C_1 \notin I^{\#}$ *implies* $C_1;C_2 \notin I^{\#}$.

The CEGIS-based synthesis algorithm, along with footprint-based search-space pruning is given in Alg. 3.5. Search-space pruning is carried out in Line 10 of Alg. 3.5. Alg. 3.5 constitutes McSynthSlave, the algorithm used by McSynth's slave synthesizer.

3.2.3 Divide-and-Conquer

The candidate enumeration in Alg. 3.5 has exponential cost. Synthesizing an instruction sequence that consists of a single instruction takes less than a second; synthesizing a two-

```
Input: φ, max
Output: C_conc or FAIL
 1: splits ← EnumerateSplits(φ)
 2: for each split ⟨φ1, φ2⟩ ∈ splits do
 3:   if IsFlowDependent(φ1, φ2) = MAYBE then
 4:     continue
 5:   end if
 6:   ret1 ← McSynthMaster(φ1, max)
 7:   if ret1 = FAIL then
 8:     continue
 9:   end if
10:   ret2 ← McSynthMaster(φ2, max)
11:   if ret2 ≠ FAIL then
12:     ret ← Concat(ret1, ret2)
13:     return ret
14:   end if
15: end for
16: return McSynthSlave_<max>(φ)
```

Algorithm 3.8: Algorithm McSynthMaster

instruction sequence takes a few minutes; synthesizing a three-instruction sequence takes several hours.

Benchmarks previously used to study synthesis of loop-free programs usually consist of a single input (or a few inputs), and a single output. However, machine-code instructions in basic blocks of real programs typically have many inputs and many outputs. An important observation is that the QFBV formulas of such basic blocks often contain many *independent* updates. If a QFBV formula has independent updates, it can be broken into sub-formulas, and the synthesizer can be invoked on the smaller sub-formulas.

McSynth's master uses a recursive procedure (McSynthMaster) that splits φ into two sub-formulas φ_1 and φ_2, and synthesizes instructions for φ_1 and φ_2. For pragmatic reasons, an implementation of the splitting step would typically construct φ_1 and φ_2 from subformulas of φ. The pseudo-code for McSynthMaster is shown in Alg. 3.8. McSynthMaster has an unusual structure because the base case (Line 16) appears after the recursive calls (Lines 6 and 10). The base case is reached if either (i) EnumerateSplits returns an empty set of splits in Line 1, or (ii) for each split, at least one recursive call returns FAIL. Let McSynthSlave$_{<max>}$ be a version of Alg. 3.5 that is parameterized by the maximum length of candidates to consider during enumeration (*max*). McSynthSlave$_{<max>}$ returns FAIL if Alg. 3.5 cannot find an instruction sequence with length $\leqslant$ *max* that implements φ. McSynthMaster uses EnumerateSplits to enumerate all splits of φ, and IsFlowDependent to test if a candidate split is illegal.

Input: $\langle \varphi_1, \varphi_2 \rangle$
Output: NO or MAYBE
1: killed $\leftarrow$ `DropPrimes`($\text{SFP}^{\#}{}_{\text{KILL}}(\varphi_1)$)
2: used $\leftarrow$ $\text{SFP}^{\#}{}_{\text{USE}}(\varphi_2)$
3: **if** killed $\cap$ used $= \emptyset$ **then**
4: **return** NO
5: **else**
6: **return** MAYBE
7: **end if**

Algorithm 3.9: Algorithm IsFlowDependent

Definition 3.2. ***Legal split.*** Suppose that `DropPrimes` removes the primes from constant and function symbols. A split $\langle \varphi_1, \varphi_2 \rangle$ of φ is **legal** iff

$$\texttt{DropPrimes}(\text{SFP}^{\#}{}_{\text{KILL}}(\varphi_1)) \cap \text{SFP}^{\#}{}_{\text{USE}}(\varphi_2) = \emptyset.$$

Observation 1. *Suppose that change_voc*(φ, i, j) *changes the vocabulary of constant and function symbols from* i *to* j *in* φ*, and that the pre-state and post-state vocabularies of* φ *are 0 and 1, respectively. If* $\langle \varphi_1, \varphi_2 \rangle$ *is a legal split of* φ*, then*

The formulas φ *and* $(\textit{change\_voc}(\varphi_1, 1, 2) \wedge \textit{change\_voc}(\varphi_2, 0, 2))$ *are equisatisfiable.*

Note that we use *equisatisfiable* instead of *equivalent* in Obs. 1 because the second formula has an extra vocabulary. Alternatively, one can state Obs. 1 as follows, where voc_2 is the set of vocabulary-2 constant and function symbols:

The formulas φ *and* $\exists\ voc_2\,.\,(\textit{change\_voc}(\varphi_1, 1, 2) \wedge \textit{change\_voc}(\varphi_2, 0, 2))$ *are equivalent*

McSynthMaster uses IsFlowDependent (Alg. 3.9) to check if a split is illegal. IsFlowDependent tests if a candidate split satisfies the property given in Defn. 3.2.

For each legal split $\langle \varphi_1, \varphi_2 \rangle$ of φ, McSynthMaster makes recursive calls to synthesize instructions for φ_1 and φ_2. If the synthesis step succeeds in synthesizing instructions for both φ_1 and φ_2, McSynthMaster concatenates the results (using `Concat`), and returns the resulting instruction sequence. If $\langle \varphi_1, \varphi_2 \rangle$ were an illegal split, φ_1 might kill a location whose pre-state value might be used by φ_2 (and thus, the split might not preserve correctness).

Input: φ
Output: splits
1: splits $\leftarrow \emptyset$
2: killedRegsFlags $\leftarrow$ `KilledRegs`(φ) $\cup$ `KilledFlags`(φ)
3: killedMem $\leftarrow$ `KilledMem`(φ)
4: regFlagSplits $\leftarrow$ `SplitSet`(killedRegsFlags)
5: memSplits $\leftarrow$ `SplitSequence`(killedMem)
6: **for** each $\langle$s1, s2$\rangle \in$ regFlagSplits **do**
7: **for** each $\langle$prefix, suffix$\rangle \in$ memSplits **do**
8: **if** (s1 $= \emptyset \wedge$ prefix $= \langle\,\rangle$) $\vee$ (s2 $= \emptyset \wedge$ suffix $= \langle\,\rangle$) **then**
9: **continue**
10: **end if**
11: $\varphi_1 \leftarrow$ `TruncateFormula`(φ, s1, prefix)
12: $\varphi_2 \leftarrow$ `TruncateFormula`(φ, s2, suffix)
13: splits $\leftarrow$ splits $\cup \langle \varphi_1, \varphi_2 \rangle$
14: **end for**
15: **end for**
16: **return** splits

Algorithm 3.10: Algorithm EnumerateSplits

Pseudo-code for EnumerateSplits is shown in Alg. 3.10. We illustrate Alg. 3.10 with

$$\varphi \equiv ESP' = ESP - 12 \wedge EBP' = ESP - 4 \wedge EAX' = EBX \wedge$$
$$Mem' = Mem[ESP - 12 \mapsto EDI][ESP - 8 \mapsto ESI][ESP - 4 \mapsto EBP].$$

For φ, `KilledRegs` (Line 2) returns $\{ESP', EBP', EAX'\}$, and `KilledMem` (Line 3) returns the sequence $\langle ESP - 4, ESP - 8, ESP - 12\rangle$. Note that the sequence preserves the temporal order of memory updates. `SplitSet` returns the set of disjoint subset pairs for the argument set. For example, $\langle\{EBP'\}, \{ESP', EAX'\}\rangle$ and $\langle\{\}, \{EBP', ESP', EAX'\}\rangle$ are in regFlagSplits (Line 4). `SplitSequence` returns the set of non-overlapping $\langle$prefix, suffix$\rangle$ pairs that partition the argument sequence. For example, $\langle\langle ESP - 4, ESP - 8\rangle, \langle ESP - 12\rangle\rangle$ is in memSplits (Line 5), but $\langle\langle ESP - 4, ESP - 12\rangle, \langle ESP - 8\rangle\rangle$ is not. `TruncateFormula` takes a formula, a set of killed registers and flags, a sequence of memory locations, and returns a formula that has updates only to the provided locations. For example, for $\langle$s1, s2$\rangle = \langle\{ESP', EBP'\}, \{EAX'\}\rangle$ in Line 6 and $\langle$prefix, suffix$\rangle = \langle\langle ESP - 4, ESP - 8\rangle, \langle ESP - 12\rangle\rangle$ in Line 7, φ_1 and φ_2 in Lines 11 and 12, respectively, are

$$\varphi_1 \equiv ESP' = ESP - 12 \wedge EBP' = ESP - 4 \wedge Mem' = Mem[ESP - 8 \mapsto ESI][ESP - 4 \mapsto EBP]$$
$$\varphi_2 \equiv EAX' = EBX \wedge Mem' = Mem[ESP - 12 \mapsto EDI]$$

Note that the split $\langle \varphi_1, \varphi_2 \rangle$ shown above constitutes an illegal split, and is discarded.

Input: φ
Output: C_{conc} or FAIL
1: ret = McSynthMaster(φ, 1)
2: **if** ret $\neq$ FAIL **then**
3: **return** ret
4: **end if**
5: ret = McSynthMaster(φ, 2)
6: **if** ret $\neq$ FAIL **then**
7: **return** ret
8: **end if**
9: **return** McSynthSlave(φ)

Algorithm 3.11: Algorithm McSynth

In addition to divide-and-conquer, our implementation of Alg. 3.8 uses memoization to avoid processing a sub-formula more than once; the result for a sub-formula is either its synthesized code-sequence or FAIL. Consequently, Alg. 3.8 really uses a form of dynamic programming. Practical values for McSynthSlave's parameter *max* are 1 or 2. For these values, McSynthMaster will either return FAIL or the synthesized instruction-sequence in a few minutes or hours (cf. Fig. 3.16). If McSynthMaster returns FAIL, McSynth uses Alg. 3.5 to synthesize instructions for φ. The full synthesis algorithm used by McSynth is given in Alg. 3.11.

3.2.4 Correctness

In this sub-section, we present the soundness and completeness properties of McSynth.

Lemma 1. *Alg. 3.5 is sound. (The instruction sequence returned by Alg. 3.5 is logically equivalent to the input QFBV formula φ.)*

Proof. By lines 11-14 of Alg. 3.4, the returned instruction sequence is logically equivalent to φ. □

Suppose that *sym_exec*(I, i, j) symbolically executes instruction sequence I with respect to the identity state, producing a symbolic state with pre-state vocabulary i and post-state vocabulary j. We overload χ from §3.2.1 to mean the operator that converts a symbolic state into a QFBV formula. $\langle\langle I \rangle\rangle$ can be defined as follows: $\langle\langle I \rangle\rangle \equiv \chi(sym\_exec(I, i, j))$. We assume that *sym_exec* has the following composition property:

$$sym\_exec(I_1; I_2, 0, 1) = sym\_exec(I_2, 2, 1) \circ sym\_exec(I_1, 0, 2)$$

Lemma 2. *For any legal split $\langle \varphi_1, \varphi_2 \rangle$ of φ, if $\varphi_1 \Leftrightarrow \langle\langle I_1 \rangle\rangle$, and $\varphi_2 \Leftrightarrow \langle\langle I_2 \rangle\rangle$, then $\varphi \Leftrightarrow \langle\langle I_1; I_2 \rangle\rangle$.*

Proof. $\langle\langle I_1; I_2 \rangle\rangle$ *iff* $\chi(sym\_exec(I_1; I_2, 0, 1))$

iff $\chi(sym\_exec(I_2, 2, 1) \circ sym\_exec(I_1, 0, 2))$,

iff $\exists voc_2 \,.\, \chi(sym\_exec(I_2, 2, 1)) \wedge \chi(sym\_exec(I_1, 0, 2))$

(because vocabulary 2 acts as an intermediate vocabulary)

iff $\exists voc_2 \,.\, change\_voc(\varphi_1, 1, 2) \wedge change\_voc(\varphi_2, 0, 2)$

(because φ_1 is equivalent to $\langle\langle I_1 \rangle\rangle$, and φ_2 is equivalent to $\langle\langle I_2 \rangle\rangle$)

iff φ (because $\langle \varphi_1, \varphi_2 \rangle$ is a legal split of φ)

Theorem 3.12. ***Soundness.*** *Alg. 3.11 is sound.*

Proof. Follows from Lemmas 1 and 2. □

Theorem 3.13. ***Completeness.*** *Modulo SMT timeouts, if there exists a non-superfluous instruction sequence I that is equivalent to φ, then Alg. 3.11 will find I and terminate.*

Proof. McSynth enumerates templatized instruction-sequences of increasing length. Because the templatized instruction-sequences span the full set of IA-32 instruction sequences (§3.1.1), McSynth searches through all non-superfluous instruction sequences in IA-32 to find an instruction sequence I that is equivalent to φ. □

Note that if such an instruction sequence does not exist (if all instruction sequences that implement φ are superfluous), Alg. 3.11 might not terminate.

3.2.5 Variations on the Basic Algorithm

Scratch registers for synthesis. Certain clients—such as a code-generator client—might want the synthesizer to be able to use "scratch" locations to hold intermediate values. McSynth has the ability to use scratch registers during synthesis. The client can specify a set of registers "Scratch" whose final value is unimportant. (For example, in a code-generator client, Scratch would be the set of dead registers at the point where code is to be generated.)

The set Scratch′ would be added to $\text{SFP}^{\#}{}_{\text{KILL}}(\varphi)$ just before Line 10 of Alg. 3.5. Consequently, instruction sequences that use registers in Scratch to hold temporary computations would not be pruned away. (Note that instruction sequences that have upwards-exposed uses of registers in Scratch would still be pruned away.) The only other change required is that just before line 14 of Alg. 3.5, all conjuncts that update registers in Scratch′ need to be dropped from φ and ψ. (There is one additional minor technical point: to make the Input/Output specification of Alg. 3.4 correct, all conjuncts of the form $S' = T$, for $S' \in$ Scratch′, should be dropped in the two occurrences of $\langle\langle \cdot \rangle\rangle$.)

Input: φ, f, *timeout*
Output: C_{conc} or FAIL
1: seen $\leftarrow \emptyset$
2: min $\leftarrow$ `MaxFn`(f)
3: minSeq $\leftarrow \epsilon$
4: **while not** `TimeoutExpired`(*timeout*) **do**
5: ret = McSynth(φ, seen)
6: **if** ret = FAIL **then**
7: **return** FAIL
8: **end if**
9: val $\leftarrow f$(ret)
10: **if** val $<$ min **then**
11: min $\leftarrow$ val
12: minSeq $\leftarrow$ ret
13: **end if**
14: seen $\leftarrow$ seen $\cup$ ret
15: **end while**
16: **return** minSeq

Algorithm 3.14: Algorithm Biased McSynth

Quality of synthesized code. Certain clients might want the synthesized code to possess a certain "quality" (small size, short runtime, low energy consumption, etc.). For example, a superoptimizer would like the synthesized code to have a short runtime. A client can obtain the desired quality by supplying a quality-evaluation function that the synthesizer can use to bias the search for suitable instruction sequences. For example, a superoptimizer could instruct the synthesizer to bias the choice of instruction sequences to ones with shorter runtimes by supplying an evaluation-function that computes the runtime of an instruction sequence. The algorithm for a biased synthesizer is shown in Alg. 3.14. In Alg. 3.14, the parameter f represents the quality-evaluation function, the parameter *timeout* represents the timeout value for the biased synthesizer, the function `MaxFn` returns the maximum value for a quality-evaluation function, and the call to the function `TimeoutExpired` returns true if *timeout* has expired. Additionally, the following changes have to be made to Algs. 3.5, 3.8, and 3.11 to implement a biased synthesizer:

- Algs. 3.5, 3.8, and 3.11 should take an additional parameter *seen*, which is the set of instruction sequences that have already been synthesized by McSynth.
- The following lines of code should be inserted after line 15 in Alg. 3.5, and after line 12 in Alg. 3.8, respectively:

  if ret $\in$ *seen* **then**
  continue
  end if

Synthesizing code that satisfies properties. Certain clients might want the synthesized code to satisfy a property expressed using a formula φ. For example, consider the formula $\varphi \equiv EAX' + EBX' = EAX + EBX + 4$. Note that φ is not in the form shown in Eqn. (2.1). McSynth can synthesize an instruction sequence that satisfies φ by replacing line 11 in Alg. 3.4 with the following line:

$$\text{model} \leftarrow \texttt{SAT}(\neg(\psi_{conc} \Rightarrow \varphi))$$

Additionally, the output specification of Alg. 3.4 needs to be $\langle\langle C_{conc} \rangle\rangle \Rightarrow \varphi$ instead of $\langle\langle C_{conc} \rangle\rangle \Leftrightarrow \varphi$. With this modification, McSynth synthesizes "`lea eax,[eax+4]`" for φ. The `lea` instruction adds 4 to the contents of the EAX register. ("`lea ebx,[ebx+4]`" is another instruction sequence that would satisfy φ.)

3.3 Implementation

McSynth uses Transformer Specification Language (TSL) [53] to convert instruction sequences into QFBV formulas. The concrete operational semantics of the integer subset of IA-32 is written in TSL, and the semantics is reinterpreted to produce QFBV formulas [54]. McSynth uses ISAL [53, §2.1] to generate the templatized instruction pool for synthesis. McSynth uses Yices [30] as its SMT solver. In the examples presented in this paper, we have treated memory as if each memory location holds a 32-bit integer. However, in our implementation, memory is addressed at the level of individual bytes.

McSynth deviates slightly from the idealized collection of templatized instructions discussed in §3.1.1. It starts from a corpus of around 43,000 IA-32 concrete instructions and creates templatized instructions by identifying each immediate operand in the abstract syntax tree of an instruction in the corpus. For instance, from "`mov eax, 1`," it creates the template "`mov eax, ⟨Imm32⟩`."

The corpus was created using ISAL, a meta-tool similar to SLED [66] for specifying the concrete syntax of ISAs. The corpus was created by running ISAL in a mode in which the input specification of the concrete syntax of the IA-32 instruction set is used to create a randomized instruction *generator*. (Random choices are based on syntactic category, so only a few instructions in the corpus lead to the template "`mov eax,⟨imm32⟩`.") The random generator produces a corpus with a wide variety of instructions ([53], Fig. 19).

In principle, one could have modified ISAL to generate all templates systematically; however, we did not have access to the ISAL source.

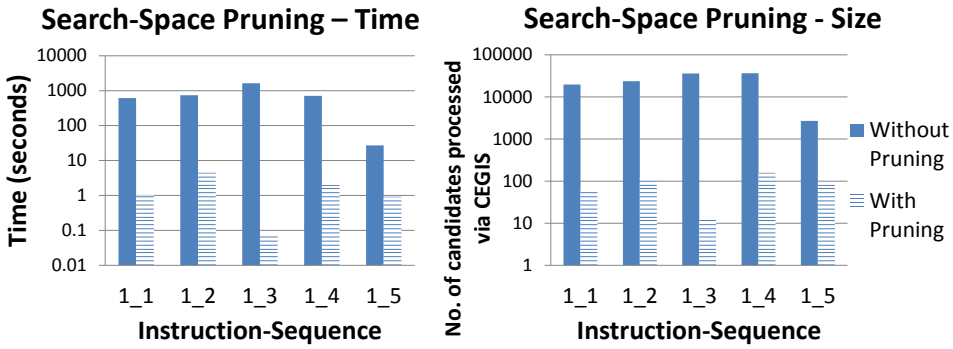


Figure 3.15: Comparison of synthesis time and search-space size with and without footprint-based search-space pruning.

3.4 Experiments

We tested McSynth on QFBV formulas obtained from instruction sequences from the SPECINT 2006 benchmark suite [44]. Our experiments were designed to answer the following questions:

- What is the time taken by McSynth to synthesize instruction sequences of varying length?
- What is the reduction in (i) synthesis time, and (ii) search-space size caused by McSynth's footprint-based search-space pruning heuristic (§3.2.2)?
- What is the reduction in synthesis time caused by McSynth's divide-and-conquer strategy (§3.2.3)?

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

For our experiments, we wanted to obtain a representative set of "important" instruction sequences that occur in real programs. We harvested the five most frequently occurring instruction sequences of lengths 1 through 10 from the SPECINT 2006 benchmark suite (50 instruction sequences in total). We converted each instruction sequence into a QFBV formula and used the resulting formulas as inputs for our experiments. Each instruction sequence in this corpus is identified by an ID of the form *m_n*, where *m* is the length of the instruction sequence, and *n* identifies the specific instruction sequence.

Pruning. The first set of experiments compared (i) the synthesis time, and (ii) the number of candidates processed via CEGIS, with and without McSynth's footprint-based search-space pruning. The results are shown in Fig. 3.15. We have presented such results only for QFBV formulas obtained from instruction sequences of length 1 because synthesis of

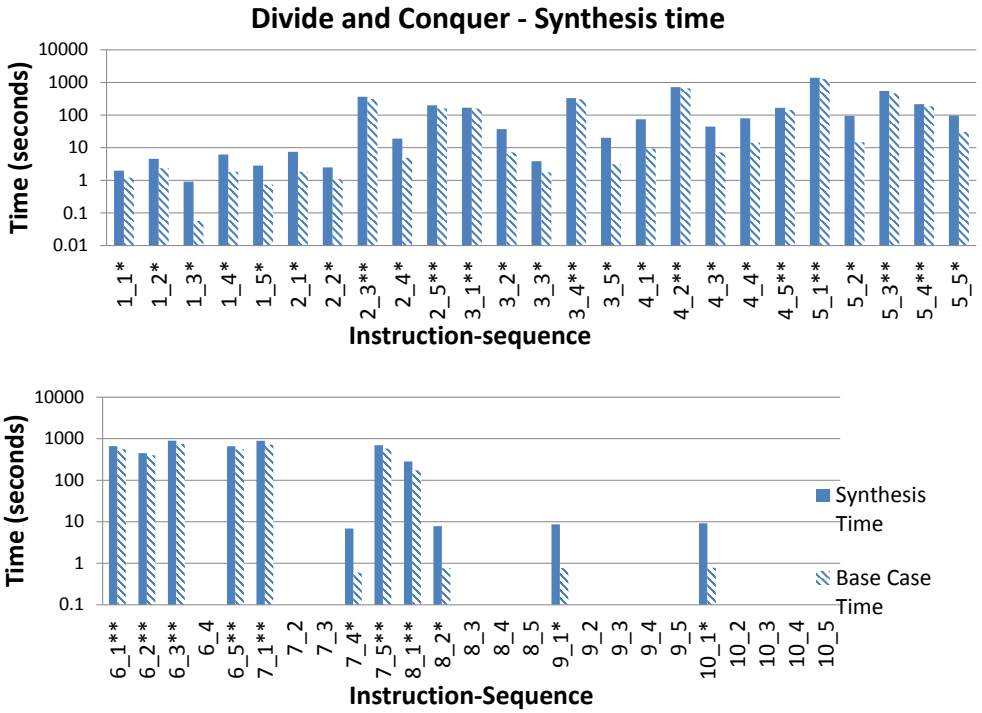


Figure 3.16: Synthesis times using the divide-and-conquer strategy.

longer instruction sequences without footprint-based search-space pruning took longer than 30 hours. For each QFBV formula, the reported time is the CPU time spent by Alg. 3.5. The geometric means of the *without-pruning*/*with-pruning* ratios for (i) synthesis time, and (ii) the number of candidates processed via CEGIS, respectively, are 473 and 273.

Divide-and-Conquer. The second set of experiments measured the synthesis times for formulas created from instruction sequences of lengths 1 through 10 using McSynth's divide-and-conquer strategy (as well as footprint-based pruning). The results are shown in Fig. 3.16. "Synthesis Time" is the total CPU time spent by Alg. 3.11. "Base Case Time" is the time spent in the base case (Line 16 of Alg. 3.8). The QFBV formulas for which FAIL was returned in Lines 1 and 5 of Alg. 3.11 do not have synthesis times reported in Fig. 3.16. The QFBV formulas for which Alg. 3.11 returned a result in Line 3 (i.e., $max = 1$ was sufficient for synthesis) are marked by *, and those for which Alg. 3.11 returned a result in Line 7 (i.e., $max = 2$ was sufficient for synthesis) are marked by **.

To measure the reduction in synthesis time caused by the divide-and-conquer strategy, we measured the synthesis times for QFBV formulas obtained from instruction sequences of lengths 1 and 2, with divide-and-conquer turned off. (We were unable to measure the synthesis times for the other QFBV formulas because synthesis without divide-and-conquer took longer than 4 days for such formulas.) In Fig. 3.17, the total CPU time

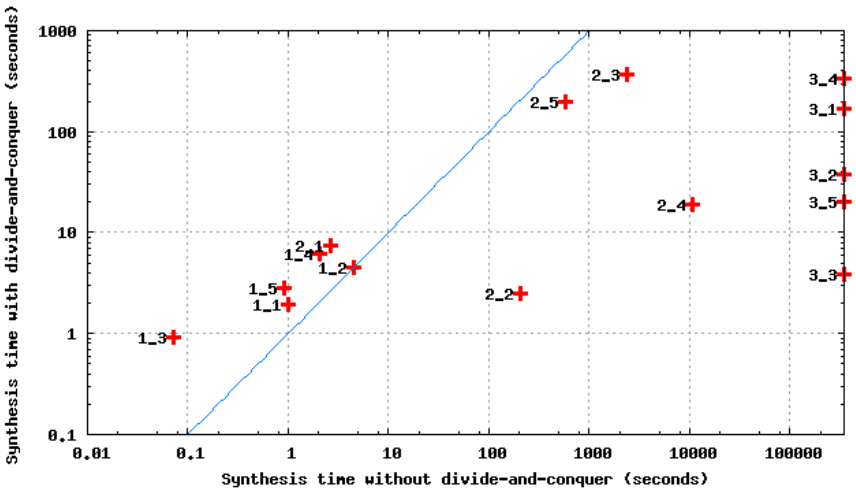


Figure 3.17: Synthesis times with and without divide-and-conquer.

spent by Alg. 3.5 is compared with the total CPU time spent by Alg. 3.11. Points below and to the right of the diagonal line indicate better performance for divide-and-conquer. Synthesis without divide-and-conquer timed out on all QFBV formulas obtained from instruction sequences of length 3. The right boundary of Fig. 3.17 represents 4 days. For instruction sequences of length 1, synthesis with divide-and-conquer takes slightly longer than synthesis without divide-and-conquer because all enumerated splits fail to synthesize instructions. For instruction sequence 2_1, synthesis without divide-and-conquer finds a shorter instruction sequence, leading to a lower synthesis time. For instruction sequences of length 3, divide-and-conquer is 3 to 5 orders of magnitude faster.

McSynth failed to synthesize code for most formulas obtained from instruction sequences of lengths 8 through 10 primarily because of the following reasons:

- The QFBV formula had a "deep" term or sub-formula (i.e., a term/sub-formula with a deep abstract-syntax tree) that can be implemented only by three or more instructions.
- All terms and sub-formulas could be implemented by two instructions or less, but the terms and sub-formulas access or update several independent memory locations. However, because $SFP^{\#}_{USE}$ and $SFP^{\#}_{KILL}$ do not distinguish between memory locations, splits that are actually legal are conservatively disregarded by Line 3 of Alg. 3.8.

These limitations in McSynth's algorithm are addressed in the next chapter.

For the 36 QFBV formulas for which McSynth synthesized code, Table 3.18 compares the length of the synthesized instruction-sequence to the length of the corresponding original instruction-sequence. Table 3.18 shows that our vanilla divide-and-conquer synthesis method often produces longer instruction sequences, but can sometimes produce a

shorter instruction sequence (if the original instruction sequence performed redundant computations).

Table 3.18: Comparison of the lengths of synthesized and original instruction-sequences.

Same	9
Different, but same length	4
Shorter	1
Longer	22

3.5 Related Work

Counter-Example Guided Inductive Synthesis (CEGIS). CEGIS is a synthesis framework that has been widely used in synthesis tools. *Sketching* is a technique that uses CEGIS for completing partial programs, or *sketches* [84–87]. The templatized instruction-sequences enumerated by McSynth can be considered as sketches, with the template operands being the holes. McSynth uses an instantiation of CEGIS for machine code to obtain concrete instruction-sequences.

CEGIS has been used in the component-based synthesis of bit-vector programs in Brahma [41]. Brahma synthesizes bit-vector programs from a library of 14 components. Brahma takes a specification of the desired program, and an upper bound on the number of times each component can be used in the synthesized program, as inputs. Brahma encodes the interconnection between components as a synthesis constraint, and uses CEGIS to solve the constraint. The goals of McSynth and Brahma are the same—namely, to synthesize a straight-line program that is equivalent to a logical specification using a library of components. However, in McSynth, the library is a full ISA, consisting of around 43,000 components. Brahma's approach of offloading the exponential cost of enumerating programs to an SMT solver might not work for an ISA like IA-32 due to the following reasons:

- The inputs and outputs of instructions include registers, flags, and a large memory array. Expressing interconnections between the inputs and outputs of instructions as a synthesis constraint may be nontrivial.
- Because Brahma's synthesis constraint is quadratic in the number of components, the synthesis constraint for a full ISA may be too large for SMT solvers to handle.

CEGIS has also been used in the synthesis of protocols from concolic-execution fragments [97].

Superoptimization. Superoptimization aims to find an optimal instruction-sequence for a target instruction-sequence [16, 17, 49, 55, 64, 65, 76]. Peephole superoptimization [16] uses "peepholes" to harvest target instruction-sequences, and replace them with equivalent instruction-sequences that have a lower cost. A superoptimizer can be implemented by using a machine-code synthesizer that has been enhanced to bias its search toward short instruction sequences. Recall that $\langle\langle\cdot\rangle\rangle$ converts an instruction sequence into a QFBV formula. Suppose that SynthOptimize is a client of the synthesizer that is biased to synthesize short instruction sequences. Then a superoptimizer can be constructed as follows:

$$\mathsf{Superoptimize}(\mathit{InstrSeq}) = \mathsf{SynthOptimize}(\langle\langle \mathit{InstrSeq} \rangle\rangle)$$

However, one cannot construct a synthesizer from a superoptimizer. Moreover, recent superoptimizers sacrifice completeness for reduced superoptimization times by resorting to stochastic search [64, 76]. In contrast, the techniques used in McSynth aim to reduce the synthesis time as much as possible without losing its completeness properties.

From a practical standpoint, certain techniques used in modern superoptimizers like STOKE [76] can be applied to machine-code synthesis. The following points outline how one could potentially adapt STOKE for machine-code synthesis:

- The cost function in STOKE takes into account both correctness and performance. If a client of the synthesizer does not care about performance, one could drop the performance component of the cost function.

- STOKE uses Markov Chain Monte Carlo (MCMC) sampling to search through the space of instruction sequences, and validates candidates by executing the input and candidate instruction-sequences on bare metal using a finite set of test inputs. Executing tests on bare metal allows STOKE to meet the sampling-rate requirements of MCMC sampling. For machine-code synthesis, the specification of the input is a QFBV formula and not an instruction sequence, and for candidate validation, one needs to evaluate the input QFBV formula on test inputs. QFBV evaluation is much slower than executing tests on bare metal, and might not meet the rate requirements of MCMC sampling. However, one could evaluate the input formula using the test inputs a priori, and just use the post-states for comparison inside the MCMC loop.

- While generating candidates, STOKE restricts immediate operands in candidates to take values only from a small set S. If the input formula φ contains values that are not in S, STOKE might not find an implementation for φ. One possible workaround is to add all constants occurring in φ to S, and sample immediate operands from the

updated S in the MCMC loop. However, this strategy might preclude STOKE from finding potentially better implementations. For example, suppose that $\varphi \equiv EAX' = (EAX + 2) + 2$. The aforementioned strategy will not find the implementation "`lea eax, [eax + 4]`" for φ.

It remains for future work to adapt STOKE for machine-code synthesis, evaluate on our set of benchmarks, and compare empirical completeness and performance with McSynth.

Clients of a machine-code synthesizer. Partial evaluation [48] is a program specialization technique that optimizes a program with respect to certain static inputs. A machine-code partial evaluator [89] partially evaluates a binary either to specialize it with respect to certain inputs, or to extract an executable component from a binary. In a machine-code partial evaluator, a synthesizer is used to synthesize residual code from formulas of instructions specialized with respect to static inputs. Machine-code partial evaluation is discussed in detail in Chapter 6.

A machine-code synthesizer also plays a key role in a machine-code slicer [91]. For purposes of precise slicing, a machine-code slicer converts a machine-code program into an intermediate representation (IR) that is at the microcode level (microcode is at an even lower level than machine code), and performs slicing on the microcode IR. However, if a client of the slicer wants a machine-code slice instead of a microcode slice, the slicer has to now reconstitute a machine-code program from the slice. To solve the program-reconstitution issue, a machine-code synthesizer can be used to synthesize instructions from the microcode fragments included in the slice. Machine-code slicing is discussed in detail in Chapter 7.

4 Speeding-up Machine-Code Synthesis

The previous chapter presented the design of the machine-code synthesizer McSynth. To cope with the enormous synthesis search-space, recall that McSynth uses (i) a divide-and-conquer strategy to split the synthesis task into several independent smaller sub-tasks, and (ii) footprint-based search-space pruning heuristics to prune away candidates during synthesis. However as we saw in §3.4, McSynth times out for several larger QFBV formulas; even for smaller formulas, McSynth takes several minutes to find an implementation. Consequently, if a binary-rewriter client supplies a formula as input to McSynth, the client has to wait several minutes or hours before McSynth finds an implementation. This delay might not be tolerable for a client that has to invoke the synthesizer multiple times to rewrite an entire binary. For example, the machine-code partial evaluator WiPEr [89] uses McSynth for the purpose of residual-code generation. For a small binary, WiPEr calls McSynth tens to hundreds of times. If each formula supplied to McSynth is relatively large, partial evaluation of an entire binary might take several hours or days.

This chapter presents several techniques to improve the synthesis algorithm used in McSynth. Some of our techniques improve the "divide" phase of McSynth so that McSynth can better split a synthesis task into smaller sub-tasks; the remaining techniques improve the "conquer" phase of McSynth so that McSynth can find an implementation for a sub-task faster. Our techniques are not restricted to machine code in particular, and can be applied to speed up other enumerative program synthesizers as well. We have implemented our techniques in McSynth to create a newer version of it called McSynth++[92]. Like McSynth, McSynth++ can be parameterized by the ISA of the target instruction-sequence.

Because McSynth++ is much faster than McSynth (see §4.4), McSynth++ should improve the speed of existing binary-rewriting clients that use McSynth [89, 91]. McSynth++ should also allow existing clients to work on larger QFBV formulas for the purpose of obtaining output binaries of better quality. For example, WiPEr currently specializes individual instructions with respect to static inputs. With McSynth++, instead of residuating code for several specialized instructions in a basic block, WiPEr could perform specialization at a basic-block level, and thereby create more optimized code. McSynth++ should also facilitate building of new clients that were impractical to build with McSynth.

Our techniques incorporate a number of ideas known from the literature, including

- alias testing to check for flow independence between elements of an array [40, 56],
- flattening of an abstract-syntax tree (AST) via scratch-register allocation for code-generation purposes [5],
- the observation that during synthesis, it is advantageous to try out sooner the components that are more frequently used in a codebase [67, 68].

McSynth++ uses/adapts these ideas in novel ways for the purpose of speeding up machine-code synthesis.

Contributions.

This chapter's contributions include the following:

- We show how ideas related to array dependence-testing furnish a better method for splitting a formula into independent sub-formulas for the purposes of synthesis (§4.2.1).
- We show how one can use flattening and scratch locations to convert a single large synthesis task involving a "deep" term into multiple smaller synthesis tasks involving "flat" terms (§4.2.1).
- We propose a novel way of pruning candidates and prefixes during synthesis based on information about the pre-state bits lost/destroyed by an instruction sequence when it transforms a state (§4.2.2).
- We show how a simple "move-to-front" heuristic can be used to prioritize instructions that are commonly used to implement operations in an instruction set (§4.2.3).

Our techniques have been implemented in McSynth++, an improved synthesizer for IA-32. Our experiments show that McSynth++ synthesizes code for 12 out of 14 formulas on which McSynth times out, speeding up the synthesis time by at least 1981X, and for the remaining formulas, McSynth++ speeds up synthesis by 3X.

4.1 Overview

In this section, we use an example to highlight the key limitations of McSynth's algorithm, and illustrate the workings of McSynth++.

4.1.1 Limitations of the McSynth Algorithm

Consider the following QFBV formula φ:

$$\varphi \equiv EAX' = Mem(ESP + 4) \wedge Mem' = Mem[ESP \mapsto EAX] \wedge EBX' = ((EAX * 2) \gg 2) + EAX$$

φ performs three updates on an IA-32 state: (i) it copies the 32-bit value in the memory location pointed to by ESP + 4 to the EAX register, (ii) copies the 32-bit value in the EAX register to the memory location pointed to by the stack-pointer register ESP, and (iii) updates the EBX register with a value computed using the value in the EAX register.

McSynth's master tries to find a *legal* split of φ. Recall from §3.1 that a legal split of an input formula φ is a sequence of sub-formulas $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ such that if one were to synthesize an instruction sequence I_i for each φ_i *independently*, and concatenate the synthesized instruction-sequences in the same order, the result will be equivalent to φ. Also recall that a sufficient condition for a legal split is *flow independence*. For our running example, consider the candiate split $\langle \varphi_1, \varphi_2 \rangle$ shown below.

$$\varphi_1 \equiv Mem' = Mem[ESP \mapsto EAX]$$
$$\varphi_2 \equiv EAX' = Mem(ESP + 4) \wedge EBX' = ((EAX * 2) \gg 2) + EAX$$

One can see that there are no flow dependences from φ_1 to φ_2, and the split $\langle \varphi_1, \varphi_2 \rangle$ is actually legal. However, McSynth uses a very conservative one-sided decision procedure for flow dependence: because φ_1 might modify *some* memory location, and φ_2 might use *some* memory location, McSynth deems the split *illegal*, and discards it. Consequently, several such legal splits are missed by McSynth.

McSynth's master identifies the following split $\langle \varphi_3, \varphi_4 \rangle$ as the only possible legal split of φ:

$$\varphi_3 \equiv EBX' = ((EAX * 2) \gg 2) + EAX$$
$$\varphi_4 \equiv EAX' = Mem(ESP + 4) \wedge Mem' = Mem[ESP \mapsto EAX]$$

McSynth cannot split either φ_3 or φ_4 any further, so it hands over φ_3 and φ_4, respectively, to slave synthesizers.

McSynth's slave enumerates templatized instruction-sequences of increasing length, and uses an instantiation of the CEGIS framework along with footprint-based search-space pruning heuristics to synthesize code for a sub-formula. Consider the sub-formula φ_3. The slave exhausts all templatized one-instruction and two-instruction sequences, and

eventually enumerates the candidate C ≡ "`imul ebx,eax,`⟨`Imm32`⟩; `shr ebx,`⟨`Imm32`⟩; `lea ebx,[ebx+eax]`." (For this example, we pretend that the `shr` instruction does not set flags.) The slave returns the instantiation "`imul ebx,eax,2; shr ebx,2; lea ebx,[ebx+eax]`" of C as the synthesized code for φ_3. Because of the depth of the AST of φ_3, and the large number of templatized instructions in IA-32 (around 43,000), the slave takes a few days to synthesize this instruction sequence for φ_3 via enumerative synthesis.

In a similar manner, the slave synthesizes the instruction sequence "`mov [esp],eax; mov eax,[esp+4]`" for φ_4. McSynth concatenates the results produced by the slaves and returns the resulting instruction sequence. McSynth takes a few days to complete the overall synthesis task.

In summary, McSynth suffers from the following limitations:

1. McSynth's one-sided decision procedure for flow independence treats memory conservatively. Consequently, McSynth loses opportunities to find legal splits.

2. If a synthesis task involves a "deep" term (a term whose AST is deep), McSynth does not attempt to split the task into smaller sub-tasks.

The overall effect of these limitations is the high synthesis time for even relatively small QFBV formulas.

4.1.2 Overview of McSynth++

At a high level, McSynth++ has the same design as McSynth: McSynth++'s master splits the input QFBV formula into a sequence of independent sub-formulas, and hands over each sub-formula to a slave synthesizer. However, McSynth++'s master and slave are improved versions of those of McSynth. McSynth++'s master uses an improved divide-and-conquer strategy that addresses the limitations of McSynth in the following ways:

1. McSynth++ uses an improved one-sided decision procedure for flow independence. McSynth++'s decision procedure is capable of reasoning about flow independence between different memory locations.

2. If the input formula contains a conjunct with a "deep" term/sub-formula, McSynth++ flattens the deep term/sub-formula into a sequence of sub-formulas.

Compared to the master used in McSynth, the improved master identifies more and finer-grained legal splits. Each of the sub-formulas in a split is given to a slave synthesizer, which synthesizes an instruction sequence for the sub-formula. McSynth++'s improved slave uses

an additional pruner based on the bits lost/destroyed by a candidate instruction-sequence to prune away candidates during synthesis. Additionally, the slave uses a "move-to-front" heuristic to boost the priority of instructions that have already been used in synthesized code. McSynth++'s master concatenates the results produced by the slaves, and returns the concatenated instruction-sequence as the synthesized code. Because of the aforementioned improvements, McSynth++ typically finishes the synthesis task much faster than McSynth.[1]

This section illustrates the workings of McSynth++ using our running example φ from §4.1.1.

$$\varphi \equiv EAX' = Mem(ESP + 4) \wedge Mem' = Mem[ESP \mapsto EAX] \wedge EBX' = ((EAX * 2) \gg 2) + EAX$$

McSynth++'s master first flattens the $EBX' = ...$ conjunct into a sequence of conjuncts using *interface* constants. An interface constant is a symbolic constant that facilitates the flow of data between conjuncts created by McSynth++. Each interface constant will be replaced by a concrete location (register, flag, or memory location) in a later step. For example, McSynth++ rewrites φ as φ' by adding interface constants m and n.

$$\begin{aligned} \varphi' \equiv\ & EAX' = Mem(ESP + 4)\ \wedge Mem' = Mem[ESP \mapsto EAX] \wedge \\ & m = EAX * 2 \wedge n = m \gg 2 \wedge EBX' = n + EAX \end{aligned} \tag{4.1}$$

Note that φ' and φ are equisatisfiable. (They are *equisatisfiable* instead of *equivalent* because the vocabulary of φ' contains extra constant symbols. However, if we disregard these constants in the meaning of φ', then $\varphi' \Leftrightarrow \varphi$.)

McSynth++'s master now uses the improved divide-and-conquer strategy to identify legal splits of φ'. Because the updates in φ' contain interface constant-symbols, flow independence is no longer a sufficient condition for a legal split. If the following two conditions hold, then a split $\langle \varphi_1, \varphi_2, \ldots, \varphi_k \rangle$ is legal:

- *Flow independence* for registers, flags, and memory locations: there is no flow dependence through registers, flags, or memory locations from a sub-formula φ_i to any successor sub-formula φ_j $(i < j)$
- *Mandatory flow dependence* for interface constants: each interface constant that gets used in a sub-formula φ_j is defined in some predecessor sub-formula φ_i $(i < j)$

Given a candidate split, it is straightforward to check the second property. To check the first property, McSynth++ uses an improved one-sided decision procedure. We illustrate

[1] Note that McSynth++ is not guaranteed to be faster because it has more legal splits to consider.

the improved decision-procedure using examples. Consider the candidate split $\langle \varphi_1, \varphi_2 \rangle$ given below.

$$\varphi_1 \equiv m = EAX * 2 \wedge n = m \gg 2 \wedge EBX' = n + EAX \wedge EAX' = Mem(ESP + 4)$$
$$\varphi_2 \equiv Mem' = Mem[ESP \mapsto EAX]$$

The decision procedure for flow independence first checks whether condition C1 from §3.1 holds. If it does, the decision procedure returns MAYBE; otherwise, it moves to the next step. C1 holds for $\langle \varphi_1, \varphi_2 \rangle$ because of register EAX, and so the decision procedure returns MAYBE. (Note that the one-sided decision procedure in McSynth also returns MAYBE for this split.)

Consider another candidate split $\langle \varphi_3, \varphi_4 \rangle$ given below.

$$\varphi_3 \equiv Mem' = Mem[ESP \mapsto EAX]$$
$$\varphi_4 \equiv m = EAX * 2 \wedge n = m \gg 2 \wedge EBX' = n + EAX \wedge EAX' = Mem(ESP + 4)$$

C1 does not hold for $\langle \varphi_3, \varphi_4 \rangle$. The improved decision procedure now collects the terms that denote the addresses of memory locations that might be modified by φ_3 (denoted by MemUpdateTerms(φ_3)), and the set of terms that denote the addresses of memory locations that might be used by φ_4 (denoted by MemAccessTerms(φ_4)). The sets are given below.

$$\text{MemUpdateTerms}(\varphi_3) = \{ESP\} \qquad \text{MemAccessTerms}(\varphi_4) = \{ESP + 4\}$$

If any term T_1 in MemUpdateTerms might alias with any term T_2 in MemAccessTerms, the decision procedure returns MAYBE; otherwise, it returns NO. The decision procedure checks if T_1 and T_2 might be aliases of each other by testing the satisfiability of $T_1 = T_2$. In our example, $ESP = ESP + 4$ is UNSAT, so the decision procedure returns NO, meaning that there is no flow dependence from φ_3 to φ_4. (Note that the one-sided decision procedure in McSynth returns MAYBE for this split.)

The master recursively splits φ_4 in a similar manner. Ultimately, the master splits φ into the following sequence of sub-formulas:

$$\varphi_3 \equiv Mem' = Mem[ESP \mapsto EAX] \qquad \varphi_5 \equiv m' = EAX * 2 \qquad \varphi_6 \equiv n' = m \gg 2$$
$$\varphi_7 \equiv EBX' = n + EAX \qquad \varphi_8 \equiv EAX' = Mem(ESP + 4)$$

Note that when the two occurences of an interface constant are put in different sub-formulas of a split, the occurence on the left of the $=$ operator is primed. The master adds the primes

when it enumerates the candidate splits of a formula.

Before giving the sub-formulas in a legal split to slave synthesizers, McSynth++'s master assigns concrete locations to the interface constants. We illustrate how McSynth++ assigns concrete locations to the split $\langle \varphi_3, \varphi_5, \varphi_6, \varphi_7, \varphi_8 \rangle$. (Note that concrete-location assignment is done on a per-split basis.) Suppose that the registers {EAX, EBX} are dead at the point where code is to be synthesized, and these registers are supplied as scratch registers to McSynth++. McSynth++ can assign an interface constant any scratch register that is definitely not used in a downstream sub-formula. For example, McSynth++ cannot assign m the register EAX because EAX might be used in φ_7. If McSynth++ were to assign m the register EAX, then it introduces a flow dependence that was originally not present in the split, making the split illegal. So McSynth++ assigns m the register EBX. McSynth++ also assigns the interface constant n the register EBX. The sub-formulas φ_5, φ_6, and φ_7 after register assignment are given below.

$$\varphi_5 \equiv EBX' = EAX * 2 \qquad \varphi_6 \equiv EBX' = EBX \gg 2 \qquad \varphi_7 \equiv EBX' = EBX + EAX$$

McSynth++ supplies the sub-formulas as inputs to the improved slave-synthesizers. (Note that, as in McSynth, if synthesis for any sub-formula in a split times out, McSynth++'s master tries an alternative split; if McSynth++ fails to synthesize code for all candidate splits, the entire formula is given to a slave.) In the remainder of this sub-section, we illustrate the working of the improved slave-synthesizer using φ_7 as an example.

McSynth++'s slave—just like McSynth's slave—enumerates templatized instruction-sequences of increasing length, and prunes away candidates based on abstract semantic-footprints. However unlike McSynth, candidates enumerated by McSynth++ pass through an additional *bits-lost-based* pruner before reaching the CEGIS loop.

The role of the bits-lost-based pruner is illustrated by the following example: suppose that the slave is currently enumerating templatized one-instruction candidates, and the current candidate is $C_1 \equiv$ "`mov ebx,eax`." The formula ψ_1 for C_1 is $EBX' = EAX$. (Note that the abstract semantic-footprints of C_1 are within those of φ_7, and so C_1 does not get pruned away by the footprint-based pruner.) φ_7 requires the pre-state bits in register EBX for the computation it performs. However, when C_1 transforms a pre-state to a post-state, C_1 loses the pre-state bits that were in EBX because they were overwritten by the pre-state bits that were in register EAX. (Note that this overwrite is explicitly shown in the QFBV formula ψ_1.) This observation has two implications: (i) C_1 cannot implement φ_7 because it has lost some of the pre-state bits that are required to implement φ_7, and (ii) no matter what instruction sequence we append to C_1, we can never get back the pre-state bits in

register EBX. Consequently, McSynth++ discards C_1.

Suppose that the next candidate enumerated by the slave is $C_2 \equiv$ "`sub ebx,eax`." The formula ψ_2 for C_2 is $EBX' = EBX - EAX$. (To simplify the presentation of this example, we pretend that the `sub` instruction does not set any flags.) One requires the pre-state bits in registers EAX and EBX to implement φ_7. When C_2 transforms a pre-state to a post-state, C_2 overwrites the pre-state bits in register EBX with a value computed from the pre-state bits in registers EAX and EBX. Even though C_2 has overwritten the pre-state bits in register EBX per se, those bits are latent in the post-state value in EBX (denoted by the term "$EBX - EAX$"), and could be recovered from that post-state value. In this specific example, one can restore the pre-state bits in register EBX by appending the instruction "`add ebx,eax`" to C_2. However, recovery of the pre-state bits is not always possible, e.g., consider $C_3 \equiv$ "`and ebx,eax`." When the pre-state bits required to implement φ_7 are *possibly* latent in the current candidate, such as C_2 or C_3, McSynth++ *conservatively* assumes that they can be recovered by additional instructions, and does not prune the candidate. (The algorithm used in the bits-lost-based pruner is described in §4.2.2.)

Eventually, the slave enumerates $C_4 \equiv$ "`add ebx,eax`." The formula ψ_4 for C_4 is $EBX' = EBX + EAX$. (To simplify the presentation of this example, we pretend that the `add` instruction does not set any flags.) C_4 is not discarded by the footprint-based pruner and the bits-lost-based pruner, and enters the CEGIS loop. CEGIS tests equivalence and returns C_4 as the implementation of φ_7. McSynth++ also moves C_4 to the front of the instruction pool for the next synthesis task. In clients like partial evaluators where McSynth++ is invoked multiple times, this heuristic allows McSynth++ to try out sooner the instructions that are commonly used in synthesized code.

McSynth++'s slaves synthesize code for the remaining sub-formulas in a similar manner. Each slave finishes the synthesis task in a few seconds, and McSynth++'s master returns the concatenated instruction-sequence given below as the synthesized code.

```
mov [esp], eax; imul ebx, eax, 2;
shr ebx, 2; add ebx, eax;
mov eax, [esp + 4]
```

The entire synthesis task finishes in under a minute. For this example, in comparison to McSynth, McSynth++ speeds up synthesis by *over four orders of magnitude*.

4.2 Algorithm

In this section, we describe the algorithms used by McSynth++. First, we present the algorithms for the improvements to the "divide" phase. Second, we present the algorithm for the improvement to the "conquer" phase. Third, we describe the "move-to-front" heuristic. Finally, we provide the correctness guarantees for McSynth++'s algorithm.

4.2.1 Divide Phase

McSynth++'s master implements the "divide" phase of synthesis. The goal of the "divide" phase is to split the input formula into as many smaller independent sub-formulas as possible. We first briefly summarize the base algorithm used by McSynth's master, and we then present the algorithms for the improvements to the "divide" phase in McSynth++.

Recall that the algorithm McSynthMaster used by McSynth's master is given as Alg. 3.8. This paragraph briefly summarizes the key aspects of McSynthMaster. McSynthMaster takes a formula φ and a timeout value as inputs, and either returns an implementation C_{conc} or FAIL. McSynthMaster first enumerates all possible splits $\langle \varphi_1, \varphi_2 \rangle$ of φ via EnumerateSplits (Line 1), and then uses the one-sided decision procedure IsFlowDependent to test if a split is legal (Lines 3–5). (Recall from §4.1.1 that a sufficient condition for legality of a split in McSynth is flow independence.) McSynthMaster discards illegal splits; for a given legal split $\langle \varphi_1, \varphi_2 \rangle$, McSynthMaster tries to synthesize code for φ_1 and φ_2, respectively, by recursively calling McSynthMaster (Lines 6–14). If McSynthMaster synthesizes code for all sub-formulas in a split, it returns the concatenated instruction-sequence (Line 13). If none of the splits work out, McSynthMaster hands over the entire formula to a slave synthesizer (Line 16). In McSynthMaster, McSynthSlave invokes the slave synthesizer.

McSynth++'s master improves upon McSynth's master in two ways:

- McSynth++ uses an improved one-sided decision procedure for flow dependence, and
- McSynth++ flattens "deep" terms before splitting a formula into subformulas.

In the remainder of this section, we describe these improvements in greater detail.

Improved one-sided decision procedure for flow dependence. One can see that IsFlowDependent (Alg. 3.9) is overly conservative in its treatment of memory. Because IsFlowDependent uses abstract semantic-footprints, which in turn overapproximate all memory locations by a single symbol "*Mem*," IsFlowDependent can return MAYBE for splits that are actually flow-independent.

Input: $\langle \varphi_1, \varphi_2 \rangle$
Output: NO or MAYBE
1: killed $\leftarrow$ `DropPrimes`($\text{SFP}^{\#}_{\text{KILL}}(\varphi_1)$)
2: used $\leftarrow$ $\text{SFP}^{\#}_{\text{USE}}(\varphi_2)$
3: killedRegsFlags $\leftarrow$ killed $- \{Mem\}$
4: usedRegsFlags $\leftarrow$ used $- \{Mem\}$
5: **if** killedRegsFlags $\cap$ usedRegsFlags $\neq \emptyset$ **then**
6: **return** MAYBE
7: **end if**
8: killedMem $\leftarrow$ `CollectMemUpdateTerms`(φ_1)
9: usedMem $\leftarrow$ `CollectMemAccessTerms`(φ_2)
10: **for** $T_1 \in$ killedMem **do**
11: **for** $T_2 \in$ usedMem **do**
12: **if** `SAT`($T_1 = T_2$) **then**
13: **return** MAYBE
14: **end if**
15: **end for**
16: **end for**
17: **return** NO

Algorithm 4.1: Algorithm ImprovedIsFlowDependent

McSynth++ uses a more precise one-sided decision procedure (ImprovedIsFlowDependent) to test split $\langle \varphi_1, \varphi_2 \rangle$ for possible flow dependence. The decision procedure is given as Alg. 4.1. To check for the absence of flow dependences via registers and flags, Alg. 4.1 uses abstract semantic-footprints (Lines 1–7). If there are no flow dependences introduced through registers or flags, Alg. 4.1 collects the terms denoting memory locations that might be modified by φ_1 via `CollectMemUpdateTerms` (Line 8), and the terms denoting memory locations that might be accessed by φ_2 via `CollectMemAccessTerms` (Line 9). Alg. 4.1 then uses an SMT solver to test if any modified location might overlap with any used location, and returns MAYBE if that is the case; otherwise, it returns NO (Lines 10–17). Consequently, ImprovedIsFlowDependent returns NO (meaning that the split is flow independent) only when each memory location possibly modified by φ_1 is guaranteed to not overlap with any memory location that might be used by φ_2.

Flattening "deep" terms. During the "divide" phase, McSynth splits the conjuncts (updates to registers, flags, and memory locations) in φ between φ_1 and φ_2. However, the master does not attempt to split the terms *within* a conjunct, and the smallest formula that a McSynth slave can receive as input is an entire conjunct in φ. If such a conjunct contains a term/sub-formula with a deep AST that can only be implemented by three or more (IA-32) instructions, searching for an implementation by a slave might take days. Consequently to speed up synthesis, the master must attempt to split such "deep" terms/sub-formulas in φ. (In the remainder of this sub-section, we use "deep term" as shorthand for "deep

Input: Flattened φ, ρ, L, I_{def}, *timeout*
Output: C_{conc} or FAIL, updated ρ

```
 1: φ ← Substitute(φ, ρ)
 2: splits ← EnumerateSplits(φ)
 3: for each split ⟨φ1, φ2⟩ ∈ splits do
 4:   if ImprovedIsFlowDependent(⟨φ1, φ2⟩) = MAYBE then
 5:     continue
 6:   end if
 7:   I^1_use ← UsedInterfaceConsts(φ1)
 8:   I^1_def ← DefinedInterfaceConsts(φ1)
 9:   I^2_use ← UsedInterfaceConsts(φ2)
10:   if I^1_use − I_def ≠ ∅ or I^2_use − I_def − I^1_def ≠ ∅ then
11:     continue
12:   end if
13:   L^1 ← L ∪ UsedRegs(φ2)
14:   ⟨ret1, ρ1⟩ ← McSynth++Master(φ1, ρ, L^1, I_def, timeout)
15:   if ret1 = FAIL then
16:     continue
17:   end if
18:   ⟨ret2,ρ2⟩←McSynth++Master(φ2,ρ1,L,I_def ∪ I^1_def,timeout)
19:   if ret2 = FAIL then
20:     continue
21:   end if
22:   ret ← Concat(ret1, ret2)
23:   return ⟨ret, ρ2⟩
24: end for
25: for each interface constant c ∈ φ do
26:   r ← PickRegister(L)
27:   ρ[c] = r
28: end for
29: φ ← Substitute(φ, ρ)
30: return ⟨McSynth++Slave(φ, timeout), ρ⟩
```

Algorithm 4.2: Algorithm McSynth++Master

term/sub-formula.")

McSynth++ flattens such "deep" terms using interface constants. An interface constant is an extra symbolic constant that McSynth++ adds to the vocabulary of φ. Given a "deep" term T, McSynth++ iteratively picks a term/sub-formula t in T, replaces t by a fresh interface constant n in T, and appends the conjunct $n = t$ to φ. After flattening all such "deep" terms in φ, the resultant formula φ' and φ are equisatisfiable. (In fact, φ and φ' have identical sets of models if we disregard interface constants in models.) An example of a flattened formula φ' is given in Eqn. (4.1) in §4.1.2. McSynth++ supplies φ' as input to its master.

The algorithm for McSynth++'s master is given as Alg. 4.2. Because the input formula now contains interface constants, McSynth++'s master must (i) take into account interface

constants while checking if a split is legal (Lines 7–12), and (ii) assign interface constants concrete locations before invoking slave synthesizers (Lines 25–28). (To simplify the presentation of Alg. 4.2, we assume that McSynth++'s master assigns interface constants scratch registers. It is straightforward to use memory locations as scratch locations in Alg. 4.2.) To solve the aforementioned tasks, Alg. 4.2 has additional inputs and outputs in comparison to Alg. 3.8. To solve task (i), Alg. 4.2 takes as input the set of interface constants defined in predecessor sub-formulas (I_{def}). To solve task (ii), Alg. 4.2 takes as input a map ρ that maps interface constants to assigned scratch registers, and the set of registers L that are live at the point where code is to be synthesized. Alg. 4.2 also returns an updated version of ρ as an additional output.

For each interface constant c that has been assigned a register r in predecessor sub-formulas, Alg. 4.2 first substitutes r for c in φ via `Substitute` (Line 1). This step ensures that each interface constant is uniformly replaced by the same register in all sub-formulas of a split. Alg. 4.2 then enumerates the candidate splits of φ, and uses ImprovedIsFlowDependent (Alg. 4.1) to check if a split $\langle \varphi_1, \varphi_2 \rangle$ of φ is flow-independent (Line 4). Once Alg. 4.2 finds a flow-independent split, it checks if each use of an interface constant is preceded by its definition: each interface constant used in φ_1 must be defined in a predecessor sub-formula, and each interface constant used in φ_2 must be defined either in a predecessor sub-formula or φ_1 (Line 10). (In Alg. 4.2, `UsedInterfaceConsts` returns the set of interface constants used in a formula; `DefinedInterfaceConstants` returns the set of interface constants that are defined in a formula.) At this point, Alg. 4.2 has found a legal split of φ.

Alg. 4.2 computes the set of registers L^1 that are live after φ_1 (Line 13). (In Line 13, `UsedRegs` returns the set of unprimed registers in a formula.) Alg. 4.2 then recursively calls itself on the sub-formulas φ_1 and φ_2, respectively, while passing the suitable sets of live-after registers and defined interface-constants (Lines 14–21). If the algorithm fails to synthesize code for all candidate splits of φ, Alg. 4.2 assigns dead registers to interface constants via `PickRegister`, and records the assignments in ρ (Lines 25–28). (Note that any register not in the live-after set L is dead at the point where code is to be synthesized.) Finally, Alg. 4.2 replaces interface constants in φ with the assigned registers via `Substitute`, and invokes the slave synthesizer (Lines 29 and 30).

In summary Alg. 4.2 recursively splits the input formula into sub-formulas, while assigning dead registers to interface constants, and ensuring legality of splits. If there are sufficient dead registers available at the point where code is to be synthesized, one can also use a more naïve register-assignment technique in Alg. 4.2, e.g., each interface constant gets a unique dead register.

4.2.2 Conquer Phase

McSynth++'s slave implements the "conquer" phase of synthesis. The goal of the "conquer" phase is to synthesize an instruction sequence for a given sub-formula. We first briefly summarize the base synthesis-algorithm used by McSynth's slave, and we then present the algorithm for the improvement to the "conquer" phase in McSynth++.

Recall that the algorithm used by McSynth's slave is given as Alg. 3.5. This paragraph briefly summarizes the key aspects of McSynthSlave. Given a formula φ, the McSynthSlave enumerates templatized instruction-sequences of increasing length, and uses CEGIS to check if there exists an instantiation of a candidate instruction-sequence that implements φ (Lines 14–17). Before the CEGIS loop, the slave tries to prune away candidates via abstract semantic-footprints: if the abstract semantic-footprints of the candidate are outside those of φ, the slave prunes the candidate away (Lines 10–12). The slave prunes away only useless candidates (candidates that either do not implement φ, or implement φ but superfluously use or modify locations that are otherwise unused/unmodified by φ). If an instance of any of the remaining candidates implements φ, the slave returns that instance as the implementation of φ.

McSynth++'s slave improves upon McSynth's slave by using an additional pruner based on the pre-state bits lost by an instruction sequence when it transforms a pre-state to a post-state.

McSynth uses abstract semantic-footprints to prune away candidates that might either use/modify a location that is otherwise unused/unmodified by the specification φ. While this pruning heuristic prunes away candidates that might use *extra information* in comparison to φ, it does not prune away candidates that *do not have enough information* to implement φ. We now present a pruning heuristic that discovers when candidates have insufficient information to implement φ.

Suppose that $\text{BITS}_{\textit{required}}(\varphi)$ represents the set of registers and flags that are required to implement a specification φ. (The bits-lost-based pruner in McSynth++ currently handles only registers and flags. Extending this pruner to handle memory locations is a possible direction for future work.) For example, consider the formula $\varphi_1 \equiv EAX' = EAX + EBX$. To implement φ_1, one needs the pre-state bits in registers EAX and EBX, i.e., $\text{BITS}_{\textit{required}}(\varphi_1)$ = {EAX, EBX}. Consider another formula $\varphi_2 \equiv EAX' = EAX \,\&\, \texttt{0x0000ffff}$. To implement φ_2, one needs only the least-significant 16 pre-state bits in register EAX, i.e., $\text{BITS}_{\textit{required}}(\varphi_1)$ = {AX}. (In IA-32, register AX denotes the least-significant half of register EAX.) One can semantically characterize $\text{BITS}_{\textit{required}}$ as follows:

Definition 4.1. A bit $b \notin \text{BITS}_{\textit{required}}(\varphi)$ iff $\forall m, m \models \varphi \Leftrightarrow m_{\overline{b}} \models \varphi$, where $m_{\overline{b}}$ is m with bit b flipped.

Recall from §2.2 that the QFBV formula that specifies the state transformation performed by an instruction sequence is a restricted 2-vocabulary formula of the form

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

where FE is a function-update expression of the form $Mem[L_1 \mapsto V_1][L_2 \mapsto V_2] \ldots [L_p \mapsto V_p]$. (Each L_i is a term that denotes a memory location l, and each V_i is a term that denotes the value in l in the post-state.) In such a 2-vocabulary formula, we call each term/formula T_m, φ_n, and V_p an r-*value* term[2] because they denote the post-state r-values. Given a candidate C whose state transformation is represented by the formula ψ_c, let $\text{BITS}^{\#}_{\textit{available}}(\psi_c)$ represent the set of unprimed register and flag constant-symbols that appear in r-value terms in ψ_c. $\text{BITS}^{\#}_{\textit{available}}(\psi_c)$ is really an over-approximation of the registers and flags whose pre-state bits can be recovered from ψ_c. For example, consider the candidate $C_1 \equiv$ "`mov eax, ebx`" whose formula ψ_1 is $EAX' = EBX \wedge EBX' = EBX \wedge \ldots \wedge CF' = CF \wedge SF' = SF \wedge \ldots \wedge Mem' = Mem$. (Note that we have explicitly shown in ψ_1 the identity conjuncts for parts of the state that are unmodified by C_1.) Then $\text{BITS}^{\#}_{\textit{available}}(\psi_1) = \{$EBX, …, CF, SF, …$\}$. (Note that EAX is *not* in $\text{BITS}^{\#}_{\textit{available}}(\psi_1)$.) McSynth++ computes $\text{BITS}^{\#}_{\textit{available}}$ for a formula ψ via a syntax-directed translation over ψ. In the following definitions, RF is the set of unprimed constant symbols used for registers and flags, T is the set of QFBV terms, and FE is the set of function-update expressions over the function symbol Mem.

Definition 4.2.

$$\text{BITS}^{\#}_{\textit{available}}(c) = \begin{cases} \{c\} & \textit{if } c \in RF \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\text{BITS}^{\#}_{\textit{available}}(Mem(t)) = \emptyset, \textit{ where } t \in T$$

$$\text{BITS}^{\#}_{\textit{available}}(Mem) = \emptyset$$

$$\text{BITS}^{\#}_{\textit{available}}(fe[l \mapsto v]) = \text{BITS}^{\#}_{\textit{available}}(v) \cup \text{BITS}^{\#}_{\textit{available}}(fe), \textit{where } l, v \in T, \textit{ and } fe \in FE$$

For all other cases, $\text{BITS}^{\#}_{\textit{available}}$ *is the union of* $\text{BITS}^{\#}_{\textit{available}}$ *of the constituents.*

[2] In compiler parlance, the l-value of an assignment denotes the location that is assigned to by the assignment; the r-value denotes the value that is assigned to the location.

In the implementation, if a register R is in $\text{BITS}^{\#}_{available}$, McSynth++ adds the smaller registers enclosed by R to $\text{BITS}^{\#}_{available}$. (For example, if EAX is in $\text{BITS}^{\#}_{available}$, McSynth++ adds AX, AL, and AH to $\text{BITS}^{\#}_{available}$.)

McSynth++ prunes away any candidate C (with QFBV formula ψ_c) that satisfies the following property:

$$\text{BITS}^{\#}_{available}(\psi_c) \not\supseteq \text{BITS}_{required}(\varphi).$$

Suppose that one can obtain $\text{BITS}_{required}(\varphi)$ precisely for an input formula φ. Even when $\text{BITS}^{\#}_{available}$ overapproximates the registers/flags whose pre-state bits are available in ψ_c, if a candidate C satisfies the above property, then it could never implement φ, and thus McSynth++ prunes away C. Moreover, no matter what instruction sequence we append to C, we can never get back the lost bits. Consequently, McSynth++ also does not retain C as a prefix for enumerating future candidates. If $\text{BITS}_{required}$ can be obtained precisely for an input formula, then the bits-lost-based pruner does not affect the completeness properties of the synthesizer.

Because it is difficult to obtain $\text{BITS}_{required}(\varphi)$ precisely for an input formula φ, McSynth++ uses an overapproximation of $\text{BITS}_{required}(\varphi)$: McSynth++ computes $\text{SFP}^{\#}{}_{\text{USE}}(\varphi)$, and disregards the symbol "*Mem*," which denotes the entire memory in abstract semantic-footprints. (i.e., $\text{BITS}^{\#}_{required}(\varphi) = \text{SFP}^{\#}{}_{\text{USE}}(\varphi) - \{Mem\}$). Even though this overapproximation makes McSynth++'s synthesis algorithm incomplete (Thm. 4.4), the bits-lost-based pruner provides an additional 7–14% improvement on top of the improvements obtained by our other techniques (see §6.5).

The algorithm used by McSynth++'s slave is identical to McSynthSlave given in Alg. 3.5, except for the following pseudo-code apprearing after Line 12 in Alg. 3.5:

if $\text{BITS}^{\#}_{available}(\psi_c) \not\supseteq \text{BITS}^{\#}_{required}(\varphi)$ **then**
  continue
end if

4.2.3 Pragmatics

The principal use case of McSynth++ is that of a code generator in semantics-based binary-rewriting clients (e.g., the residual-code generator in the machine-code partial evaluator WiPEr [89]). Binary rewriters would typically convert instructions in a program into a formula, modify the formula based on various semantic criteria, and supply the modified formula as input to McSynth++. In programs, certain operations tend to occur more frequently than others (e.g., increment/decrement the stack pointer, write to the stack, etc.)

and certain instructions tend to be used more frequently than others to implement such operations. It is beneficial to prioritize during synthesis the instructions that are used to implement common operations in input formulas.

To prioritize useful instructions, McSynth++ uses a "move-to-front" heuristic: whenever a slave finds an implementation, McSynth++ moves the templatized instructions that occur in that implementation to the front of the list that serves as the instruction pool in the next synthesis task; the next synthesis task could be the invocation of a slave on a different sub-formula in the same input formula, or the invocation of a slave on a sub-formula in a new input formula. This heuristic is implemented in McSynth++'s slave via the following lines of pseudo-code appreaing after Line 15 in McSynthSlave (Alg. 3.5):

instrPool ← `MoveToFront`(ret, instrPool)
`WriteInstrPool`(instrPool)

`MoveToFront` moves the instruction templates appearing in ret to the front of the instruction pool (instrPool); `WriteInstrPool` dumps the instruction pool to a file, which would be read by the next synthesis task (Line 1 of Alg. 3.5).

Both McSynth and McSynth++ also incorporate function caching to reuse implementations of previously seen formulas.

4.2.4 Correctness

In this sub-section, we present the soundness and completeness properties of McSynth++.

Lemma 3. *McSynth++'s slave is sound. (The formula $\langle\langle I \rangle\rangle$ for instruction sequence I returned by McSynth++'s slave is logically equivalent to the input QFBV formula φ.)*

Proof. The CEGIS loop of the slave (Line 14 in Alg. 3.5) returns an instruction sequence I only if $\langle\langle I \rangle\rangle$ is equivalent to φ. □

Lemma 4. *For any legal split $\langle \varphi_1, \varphi_2 \rangle$ of φ, if $\varphi_1 \Leftrightarrow \langle\langle I_1 \rangle\rangle$, $\varphi_2 \Leftrightarrow \langle\langle I_2 \rangle\rangle$, and $\mathfrak{l}$ is a set of scratch locations that appear in I_1 and I_2 but not in φ, then $\varphi \Leftrightarrow \langle\langle I_1; I_2 \rangle\rangle$ disregarding scratch locations $\mathfrak{l}$.*

Proof. If there are no interface constants in φ, then flow independence is the only criterion for a legal split, and the proof for this lemma under the aforementioned case is available elsewhere (Lemma 2 in [90]). Thus, we only have to prove that the lemma still holds when interface constants are present in a flow-independent split $\langle \varphi_1, \varphi_2 \rangle$. Without loss of generality, let us assume that there is only one interface constant $\mathfrak{n}$, which replaces term T in φ. Because $\langle \varphi_1, \varphi_2 \rangle$ is a legal split, "$\mathfrak{n}' = T$" would appear in φ_1, and $\mathfrak{n}$ would be used in φ_2. McSynth++ assigns $\mathfrak{n}$ a location $\mathfrak{l}$ that is dead at the point where code is to be synthesized, and I_1 initializes the location $\mathfrak{l}$ according to T. Because I_2 follows I_1, I_2 always

reads the value written by I_1 in $\mathfrak{l}$. Consequently, φ and $\langle\langle I_1; I_2 \rangle\rangle$ are equisatisfiable, and $\varphi \Leftrightarrow \langle\langle I_1; I_2 \rangle\rangle$ disregarding $\mathfrak{l}$. □

Theorem 4.3. ***Soundness.*** *McSynth++ is sound.*

Proof. Follows from Lemmas 3 and 4. □

With the exception of candidates pruned away because of the imprecision introduced while computing $\text{BITS}^{\#}_{required}$ for an input formula, McSynth++ has the same completeness guarantees as McSynth (Thm. 2 in [90]).

Theorem 4.4. ***Completeness.*** *Modulo SMT timeouts and candidates that are pruned away because of imprecision in $\text{BITS}^{\#}_{required}(\varphi)$, if there exists an instruction sequence I that (i) is equivalent to φ, and (ii) does not superfluously use/modify locations that are otherwise unused/unmodified by φ, then McSynth will find I and terminate.*

Proof. In comparison with McSynth, the only source of incompleteness in McSynth++ is the bits-lost-based pruner. If $\text{BITS}^{\#}_{required}(\varphi)$ is imprecise, then a candidate that actually has enough information to implement φ might be pruned away. Apart from the bits-lost-based pruner there are no other sources of incompleteness: the "move-to-front" heuristic does not prune any candidate; McSynth++'s master tries out all candidate splits of φ, and if McSynth++ fails to synthesize code for all candidate splits, McSynth++ supplies the entire φ as input to a slave synthesizer. □

4.3 Implementation

McSynth++ uses Transformer Specification Language (TSL) [53] to convert instruction sequences into QFBV formulas. The concrete operational semantics of the integer subset of IA-32 is written in TSL, and the semantics is reinterpreted to produce QFBV formulas [54]. McSynth++ uses ISAL [53, §2.1] to generate the templatized instruction pool for synthesis. McSynth++ uses Yices [30] as its SMT solver. In the examples presented in this paper, we have treated memory as if each memory location holds a 32-bit integer. However, in our implementation, memory is addressed at the level of individual bytes. McSynth++, just like McSynth, is capable of accepting scratch registers for synthesis [90, §4.4].

4.4 Experiments

We tested McSynth++ on QFBV formulas obtained from instruction sequences from the SPECINT 2006 benchmark suite [44], and also in the context of a client—the machine-code

partial evaluator WiPEr [89]. Our experiments were designed to answer the following questions:

1. In comparison with McSynth, what is the speedup in synthesis time caused by each individual improvement to the "divide" phase? What is the speedup when both improvements are used together?

2. In comparison with McSynth, how many timeouts remain with each individual improvement to the "divide" phase? How many timeouts remain with both improvements together?

3. With both improvements to the "divide" phase turned on, what is the speedup in synthesis time caused by the bits-lost-based pruner and the "move-to-front" heuristic, respectively? What is the speedup when both improvements are used together?

4. What is the speedup in residual-code-synthesis time when McSynth++ is used in the place of McSynth in WiPEr?

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

To answer the first three questions, we used the same benchmark suite that was used to test McSynth: QFBV formulas obtained from a representative set of "important" instruction sequences that occur in real programs. We harvested the five most frequently occurring instruction sequences of lengths 1 through 10 from the SPECINT 2006 benchmark suite (50 instruction sequences in total). We converted each instruction sequence into a QFBV formula and used the resulting formulas as inputs for our experiments.

Note that in general there is no restriction on the source of the input formula, and the formula can come from any client; we simply chose to obtain the input formulas from instruction sequences for experimental purposes.

To answer the first two questions, we measured the synthesis time with (i) only the improved decision-procedure for flow independence turned on, (ii) only flattening of "deep" terms turned on, and (iii) both improvements turned on. We compared the numbers against the baseline synthesis-time numbers obtained from McSynth.

The results are shown in Fig. 4.5(i), (ii), and (iii), respectively. In Fig. 4.5, the blue lines represent the diagonals of the scatter plots. If a point lies below and to the right of the diagonal, the baseline performs worse. All axes in Fig. 4.5 use logarithmic scales. McSynth timed out on 14 formulas. (The timeout value was three days.) With only the improved decision-procedure turned on, the number of timeouts was 10; with only flattening turned on, the number of timeouts was 6; with both improvements to the "divide" phase turned

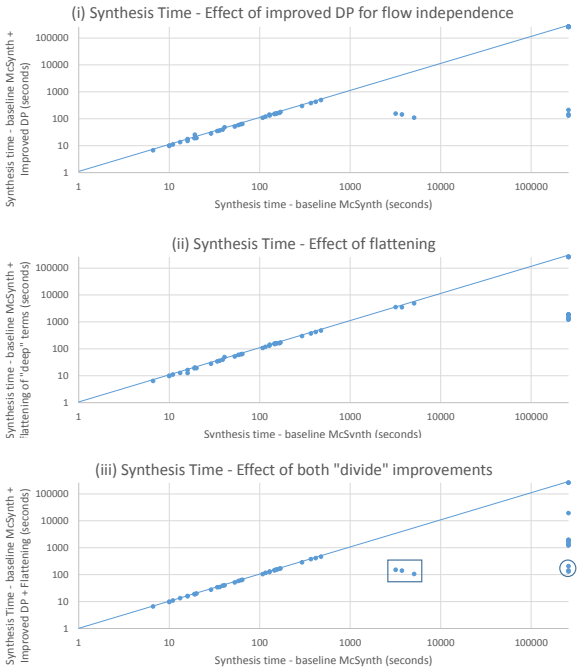


Figure 4.5: Synthesis times obtained via improvements to the "divide" phase in McSynth++ for the corpus of 50 QFBV formulas.

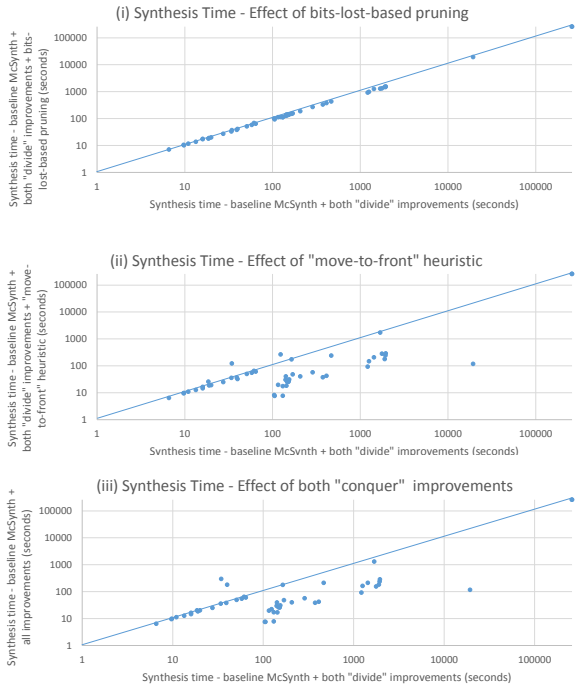


Figure 4.6: Synthesis times obtained via improvements to the "conquer" phase in McSynth++ for the corpus of 50 QFBV formulas.

on, the number of timeouts was 2. For the formulas that timed out in McSynth but did not timeout in McSynth++, the average speedup in synthesis time was over 233X (computed as a geometric mean). For 3 formulas, the speedup was over three orders of magnitude (surrounded by a circle in Fig. 4.5(iii)). Among the formulas that did not timeout in McSynth, the two improvements reduced the synthesis time considerably only for three formulas (surrounded by a square in Fig. 4.5(iii)). For the formulas that did not timeout, the average speedup in synthesis time caused by the two improvements was 1.34X.

To answer the third question, we turned on both improvements to the "divide" phase, and we measured the synthesis time with (i) only the bits-lost-based pruner turned on, (ii) only the "move-to-front" heuristic turned on, and (iii) both improvements turned on. We compared the numbers against the synthesis times obtained when both improvements to the "divide" phase were turned on. The results are shown in Fig. 4.6(i), (ii), and (iii), respectively. The average speedup in synthesis time caused by the bits-lost-based pruner was 1.07X (computed as the geometric mean). If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is slightly higher: 1.14X. We believe that the speedup is relatively small because McSynth's footprint-based pruner already prunes away most of the candidates that could potentially be pruned away by the

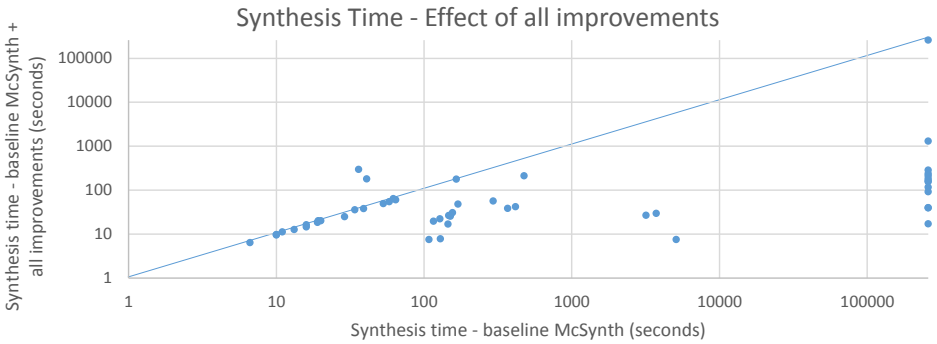


Figure 4.7: Effect of all improvements in McSynth++ for the corpus of 50 QFBV formulas.

bits-lost-based pruner. For example, consider the input formula $\varphi \equiv EBX' = EAX + EBX$, and the candidate C $\equiv$ "`mov eax, ebx`" with the QFBV formula $\psi \equiv EAX' = EBX$. The candidate loses the pre-state bits in register EAX when it transforms a state. Because φ requires the pre-state bits in EAX for the computation it performs, C will be potentially pruned away by the bits-lost-based pruner. However, $\text{SFP}^{\#}{}_{\text{KILL}}(\varphi) = \{EBX'\}$ and $\text{SFP}^{\#}{}_{\text{KILL}}(\psi) = \{EAX'\}$, and the abstract semantic kill-footprint of ψ is outside that of φ. Consequently, C will be pruned away by the footprint-based pruner, and will never reach the bits-lost-based pruner.

The "move-to-front" heuristic caused more pronounced speedup. The average speedup in synthesis time caused by the heuristic was 3X (computed as a geometric mean). If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is 6X. Our corpus consists of formulas obtained from the most frequently-occurring instruction sequences, and programs tend to perform certain computations more frequently than others (e.g., increment/decrement the stack pointer, write to a stack location, etc.). Consequently, many formulas in our corpus contain specifications for such frequently performed computations. (Note that this would be the case even if a binary-rewriter client like a partial evaluator were producing the corpus of formulas because the rewriter originally obtains the base formulas from instructions in the binary.) The "move-to-front" heuristic produced a sizeable speedup because it prioritizes instructions that are used to implement common operations in the input formulas. The heuristic caused a slowdown in two formulas because it moved some instructions to the front of the instruction list, pushing later some more-infrequently-used instructions required to implement the formulas.

The comparison of synthesis-time numbers produced by McSynth and McSynth++ is shown in Fig. 4.7. In summary, with all the improvements turned on and in comparison with McSynth, McSynth++ speeds up the synthesis time by over 1981X for formulas that

Table 4.8: Comparison of residual-code-synthesis time using McSynth and McSynth++, respectively, in WiPEr.

Application	No. of calls to the synthesizer	Synthesis time using McSynth (seconds)	Synthesis time using McSynth++ (seconds)	Speedup
power	6	16	13.5	1.19
interpreter	19	30	22.8	1.32
sha1	23	25.4	21	1.21
filter	212	241	177	1.36
dotproduct	306	312	267	1.17

timed out in McSynth but did not timeout in McSynth++. For the formulas that did not timeout in McSynth, McSynth++ speeds up the synthesis time by 3X. If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is 11X.

To answer the fourth question, we measured the total time taken to synthesize residual code using McSynth and McSynth++, respectively, while partially evaluating the microbenchmarks used in [89] with WiPEr. The results are shown in Table 4.8. The average speedup in residual-code-synthesis time caused by McSynth++ is 1.25X (computed as a geometric mean). Note that the microbenchmarks are fairly small programs (see Table 1 in [89]); the specialized formulas given to the synthesizer are also small, and are often implemented by one instruction.

4.5 Related Work

Dependence testing in arrays. A parallelizing compiler employs a series of tests to check for flow dependences between array references [40, 56]. The tests are often ordered as a sequence, ranging from cheapest (but approximate) to most expensive (but exact). If the tests say that an ⟨array-update, array-access⟩ pair is flow-independent, the parallelizing compiler proceeds to parallelize the sequential code.

The one-sided decision procedure in McSynth++ aims to solve the same problem as the aforementioned work: test if an ⟨array-update, array-access⟩ pair is flow-independent. However, instead of array variables in programs, McSynth++ deals with the memory array in QFBV formulas. Also, because state-of-the-art SAT solvers are quite efficient, McSynth++ uses SAT for alias testing instead of the series of tests used in the aforementioned work.

Alternatives to machine-code synthesis: QFBV-to-IA-32 compiler. One could try to build a QFBV-to-IA-32 compiler by (i) defining a set of patterns $\mathscr{P}$ that map basic QFBV

fragments to IA-32 instruction sequences, and (ii) using a bottom-up rewriting system [7, 36] that computes the least-cost IA-32 cover for the input QFBV formula. However, one finds that the QFBV-to-IA-32 translation problem has some subtleties that make it difficult to create such a compiler.

- Not all QFBV formulas would be straightforward to handle. The easy formulas would be ones that specify a pre-state to post-state transformation (e.g., $\varphi_1 \equiv EAX' = EAX + 2 \wedge EBX' = EBX + 2$). Such a formula is said to be in *explicit form*. However, a client can supply a formula that expresses a property over pre-states and post-states (e.g., $\varphi_2 \equiv EAX'+EBX' = EAX+EBX+4$). Such a formula is said to be in *implicit form*. In such cases, the client would expect an instruction sequence whose pre-state-to-post-state transformation satisfies the input formula. For example, "`lea eax, [eax + 4]`" is one instruction sequence that satisfies φ_2. Both McSynth and McSynth++ are capable of searching for an instruction sequence that satisfies a formula in implicit form. (See [90, §4.4], "Synthesizing code that satisfies properties.") However, it would be difficult to create a compiler that handles formulas in implicit form.

- Formulas use operators that are associative and commutative, and consequently different elements in a formula that are relevant for selecting a given instruction can be arbitrarily far apart. This situation prevents one from creating a compiler for formulas based on a bottom-up rewrite system (e.g., iburg [7], Twig [36], etc.).

- Certain clients might want the output instruction-sequence to possess a certain "quality" (small size, short runtime, low energy consumption, etc.). For example, a superoptimizer would like the synthesized code to have a short runtime. Because a QFBV-to-IA-32 compiler would have a fixed set of patterns $\mathscr{P}$, the compiler would not be able to produce instruction sequences with varying qualities: given input formula φ, it would always return the instruction sequence specified by $\mathscr{P}$. In contrast, because a synthesizer searches over the space of instruction sequences, it can find different implementations of φ with varying qualities. The algorithm used by McSynth and McSynth++ to find an implementation with a certain quality is given in [90, §4.4], "Quality of synthesized code."

5 Model-Assisted Machine-Code Synthesis

The previous chapter presented the optimizations that went into the improved machine-code synthesizer McSynth++. McSynth++ was able to synthesize code for the formulas on which McSynth timed out, speeding up synthesis by over *three orders of magnitude*; for the remaining formulas, McSynth++ could speed up synthesis by 3X on an average. However, for roughly one third of the formulas in our corpus of 50 formulas, McSynth++ takes more than 180 seconds to synthesize an implementation. Further, McSynth++ times out for two formulas, and takes more than 1000 seconds to find an implementation for one formula. Such synthesis-time numbers are still quite high, and would become a bottleneck for binary rewriters such as a machine-code partial evaluator [89] or a machine-code slicer [91], which make multiple calls to a synthesizer.

McSynth++'s master incorporates several techniques to split the input formula as finely as possible into subformulas. However when it comes to McSynth++'s slave, there is still room for improvement. McSynth++'s slave performs a *linear search* over the space of instruction sequences to find an implementation for a subformula φ: it first exhausts one-instruction sequences, then moves to two-instruction sequences, and so on. The slave tests every enumerated candidate for equivalence with φ via CEGIS, and does not attempt to prioritize certain candidates. For example, it would be advantageous to prioritize instruction sequences that are commonly used to implement idioms in programs; it might also be advantageous to prioritize instruction sequences that contain instructions that are highly likely to implement φ (e.g., if $\varphi \equiv EAX' = EAX + 10$, it would be advantageous to prioritize instruction sequences that contain the `add` instruction).

This chapter describes how we use *machine learning* to make the search in McSynth++'s slaves smarter. Given a huge corpus of specifications and implementations—in our case, a corpus of ⟨QFBV-formula, instruction-sequence⟩ pairs—we learn (i) a *language model* for instruction sequences in the instruction set, and (ii) a model that correlates features of a QFBV formula with features of its equivalent instruction sequence. In particular, we use an *n-gram* model for the former, and *k-nearest-neighbor (k-NN) regression* for the latter. We have equipped McSynth++'s slaves with these models to build an improved machine-code synthesizer, called McSynth-ML.

First, McSynth-ML uses features of the input formula φ along with k-NN regression to

restrict the slave's instruction pool to contain only instructions that are highly likely to implement φ. McSynth-ML then performs a *best-first search* over the truncated instruction-sequence space to find an implementation for φ. The cost heuristic for the search comes from both the n-gram language model and the k-NN-regression model: the former allows the search to prioritize useful instruction sequences that are commonly used to implement idioms in binaries; the latter allows the search to prioritize instruction sequences that contain instructions most likely to implement φ, and steer the search away from instruction sequences with instructions that are irrelevant to φ. The effect of the model-assisted best-first search is potentially faster synthesis in slave synthesizers.

Contributions

This chapter's contributions include the following:

- Our technique is the first of its kind to employ machine learning for the synthesis of low-level code.
- While existing approaches for low-level-code synthesis employ enumerative, symbolic, and stochastic search strategies [64, 76, 90, 92], we employ a novel best-first search assisted by models learned from a corpus of specifications and implementations.
- Our technique makes novel usage of language models to steer the search towards useful instruction sequences; existing approaches have used language models only for purposes of finding most likely completions of partial programs [42, 67].
- Ours is the first synthesis technique that employs a model (specifically, k-NN regression) that correlates features of implementations with features of specifications; we use the model to steer the search towards instruction sequences that are highly likely to implement the input specification.

Our techniques have been implemented in McSynth-ML, a model-assisted synthesizer for IA-32. Our experiments show that McSynth-ML synthesizes code for 6 out of 50 formulas on which McSynth++ times out, speeding up synthesis by at least 549X, and for the remaining formulas, McSynth-ML speeds up synthesis by 4.55X.

5.1 Overview

In this section, we use an example to highlight the key limitations of McSynth++'s algorithm, and illustrate the workings of McSynth-ML.

5.1.1 Limitations of McSynth++

The limitations of McSynth++ we present in this sub-section lie in McSynth++'s slave synthesizers. Consider the following QFBV formula φ:

$$\varphi \equiv EAX' = Mem(ESP + 10) \wedge ESP' = ESP + 18 \wedge ZF' = (ESP + 10 = 0)$$

φ performs three updates on an IA-32 state: (i) it copies the 32-bit value in the memory location pointed to by ESP + 10 to the EAX register, (ii) increments the stack-pointer register ESP by 18, and (iii) sets the zero flag ZF according to the test $ESP + 10 = 0$.

McSynth++'s master uses a combination of *divide-and-conquer* (§3.2.3) and *flattening* (§4.2.1) strategies to split φ into a sequence of independent sub-formulas $\langle \varphi_1, \varphi_2 \rangle$ shown below. Note that $\langle \varphi_1, \varphi_2 \rangle$ is a *legal* split of φ (§4.1.2).

$$\varphi_1 \equiv EAX' = Mem(ESP + 10) \qquad \varphi_2 \equiv ESP' = ESP + 18 \wedge ZF' = (ESP + 10 = 0)$$

McSynth++'s master then hands over φ_1 and φ_2, respectively, to slave synthesizers.

We now illustrate the limitations in McSynth++'s slave synthesizer using φ_2 as an example. Note that one implementation of φ_2 is "`add esp, 10; lea esp, [esp+8]`." McSynth++'s slave enumerates templatized instruction-sequences of increasing length, and uses an instantiation of the CEGIS framework along with two pruners—a *footprint-based* pruner (§3.2.2) and a *bits-lost-based* pruner (§4.2.2)—to synthesize code for a sub-formula. Additionally for pragmatic purposes, McSynth++ uses a *"move-to-front"* heuristic to prioritize useful instructions[1] in the instruction pool. However, the slave suffers from the following limitations:

Retaining instructions that are irrelevant to φ in the instruction pool. The slave does not attempt to prune its instruction pool based on the QFBV operators present in the input formula φ. The existing pruners in McSynth++ prune from the instruction pool instructions whose *operands* are inconsistent with those of φ. (For example, the existing pruners will

[1] In the remainder of this chapter, when we use the terms "instruction" or "instruction-sequence," we refer to a templatized instruction or templatized instruction-sequence, respectively. If we refer to a concrete instruction-sequence, we will explicitly say so.

prune the instruction "`add ebp, ⟨Imm32⟩`" because the instruction uses and modifies the EBP register, which is untouched by φ_2.) However, instructions whose *opcode variants* (defined in §5.1.2) are highly unlikely to implement the input formula are left unpruned. (For example, instructions belonging to the IMUL opcode variant could never be used to implement φ_2; yet existing pruners do not prune instructions such as "`imul esp, ⟨Imm32⟩`" from the instruction pool.) Consequently, candidate instruction-sequences containing such instructions are wastefully enumerated and subsequently discarded by the slave.

Linear search. The basic search strategy used by McSynth++'s slave is *linear search*: the slave first exhausts all one-instruction sequences, followed by two-instruction sequences, and so on. During the search, it might be better to prioritize (i) commonly used instruction-sequence prefixes (e.g., instruction sequences that implement common idioms), and (ii) prefixes with instructions that are highly likely to implement φ. However, McSynth++'s slave does not attempt to perform any prioritization. For example, when a McSynth++ slave searches for an implementation for φ_2, let us assume that the slave encounters the prefix $P_1 \equiv$ "`ror esp, ⟨Imm32⟩`" (rotate right instruction) before the prefix $P_2 \equiv$ "`add esp, ⟨Imm32⟩`" during its linear search; consequently, the slave expands P_1 before P_2, and takes a much longer time to find the implementation "`add esp, 10; lea esp, [esp+8]`" for φ_2. However in comparison to P_1, P_2 is much more frequently found in binaries; furthermore, P_2 contains an `add` instruction, which is much more likely to implement φ_2 than the `ror` instruction in P_1. The slave could have found the implementation much faster had it expanded P_2 first.

The slave's "move-to-front" heuristic suboptimally performs some prioritization by moving instructions that occurred in previously synthesized implementations to the front of the instruction pool in the current synthesis task. However, the heuristic would not always guarantee faster synthesis, e.g., if there were no prior synthesis tasks.

In effect, McSynth++ takes around 10 minutes to synthesize an implementation for our example φ. Given the enormous size of the search space that McSynth++ has to deal with, this number is not high by itself. (Note that a naïve enumerative synthesizer would have taken a few days to find an implementation for φ.) However, in the context of a binary-rewriting client that makes several calls to the synthesizer (e.g., the machine-code partial evaluator WiPEr [89]), such synthesis times would cause the client to take hours or days to rewrite an entire binary.

5.1.2 Overview of McSynth-ML

This section presents an example to illustrate the workings of McSynth-ML. Along the way, we also provide necessary background on the models used by McSynth-ML.

At a high level, McSynth-ML has the same design as McSynth++: McSynth-ML's master splits the input QFBV formula into a sequence of independent sub-formulas, and hands over each sub-formula to a slave synthesizer. In fact, McSynth-ML uses the same master as McSynth++. However, McSynth-ML's slave is an improved version of that of McSynth++. (McSynth-ML's slave inherits the footprint-based and bits-lost-based pruners from McSynth++.) McSynth-ML's slave uses machine learning to address the limitations of McSynth++'s slave in the following ways:

1. McSynth-ML uses features of the input formula φ along with k-NN regression to prune from the instruction pool instruction templates that are least likely to implement φ.

2. McSynth-ML then uses best-first search instead of linear search as its core search strategy. To prioritize instruction-sequence prefixes in the search, McSynth-ML uses a cost heuristic that combines scores supplied by (i) an n-gram-based language model , and (ii) the aforementioned k-NN-regression model. The former prioritizes common/useful IA-32 instruction sequences, and the latter prioritizes instruction sequences containing instructions that are highly likely to implement φ.

Step 1 is more of an optimization: instead of first enumerating a candidate C and then postpone processing C because of C's low value of the cost heuristic, step 1 eagerly truncates the instruction pool so that C never gets enumerated.[2] Because of the smarter model-assisted search, McSynth-ML's slave is typically much faster than McSynth++'s slave.[3]

Beacause McSynth-ML uses models to assist synthesis, the models need to be trained on training inputs before the actual synthesis. In the remainder of this section, we first describe how the models are trained, and then the actual synthesis using McSynth-ML.

Training Phase

To train models that assist machine-code synthesis, one needs a huge corpus of specifications (QFBV formulas) and their corresponding implementations (instruction sequences). However, creating such a huge corpus via synthesis can take a very long time. (Recall from

[2]To preserve the McSynth-ML's completeness guarantees (Thm. 5.5), if synthesis with the truncated instruction pool fails to find an implementation, McSynth-ML subsequently attempts synthesis with the untruncated instruction pool. (See §5.2.2.)

[3]Note that McSynth-ML's slave is not guaranteed to be faster because its heuristics are sensitive to training data.

§5.1.1 that finding an implementation with McSynth++ can take a few minutes. Finding implementations for hundreds of thousands of specifications can take several days.) An important observation is that, while finding an instruction sequence for a QFBV formula involves search/synthesis and is slow, converting an instruction sequence into a QFBV formula can be done very quickly via *symbolic execution*. There are many readily available tools that can perform this conversion [54]. We used this observation to create a corpus of hundreds of thousands of equivalent ⟨QFBV-formula, instruction-sequence⟩ pairs as follows: (i) we harvested several straight-line instruction sequences from binaries, (ii) converted each instruction sequence I into a QFBV formula φ via symbolic execution, and (iii) added $\langle \varphi, I \rangle$ to the corpus. It took almost 12 hours to create a corpus of 612,000 pairs by this method. (This was the total time spent to harvest and canonicalize instruction sequences from binaries, and perform symbolic execution.)

k-NN regression. Our intended use case for k-NN regression during synthesis is to predict how likely it is for an IA-32 instruction to occur in an implementation of the input formula. k-NN regression is a non-parametric regression technique that stores all available training data and predicts a numerical target based on a *metric*[4] over its input space. In the next few paragraphs, we describe a specific instantiation of k-NN regression that we use in our context.

Each pair $\langle \varphi, I \rangle$ in the corpus corresponds to a point in the input space. Each point in the input space is identified by m binary features (features that take either 0 or 1 as their value), and is associated with an output label, which is a numeric value. The input space is the following mapping:

$$\langle f_1, f_2, \ldots, f_m \rangle \mapsto l,$$

where f_i is the entry corresponding to the i^{th} feature, and l is a numeric value.

In our case, the presence or absence of QFBV operators in the specification φ constitute the features of the input space: an entry in the input feature-vector is 1 if φ contains the QFBV operator corresponding to that entry; it is 0 otherwise. There are 70 QFBV operators in the logic described in §2.2, and so the length of each input vector is 70 (i.e., $m = 70$).

For a training-input pair $\langle \varphi, I \rangle$, the output label's value denotes the probability that instruction sequence I contains an instruction belonging to a specific opcode variant. (Note that the label's value is either 0 or 1.) An *opcode variant* is a conjunction of (i) the opcode category (e.g., ADD, IMUL, OR, etc.), and (ii) the operands sizes. Some examples of opcode variants include ADD_32_32, ADD_32_8, and IMUL_32_32_8.

[4] A metric or distance function is a function that defines a distance between two points in the input space. An example of a metric commonly used in k-NN regression is Euclidean distance.

For a given test-input formula, we are interested in predicting how likely it is for each of the 87 opcode variants in IA-32 to implement that formula. So we associate each output label with an opcode variant, i.e., the input space is now the following mapping:

$$\langle f_1, f_2, \ldots, f_m \rangle \mapsto \langle l_1, l_2, \ldots, l_n \rangle,$$

where f_i is the entry corresponding to the i^{th} feature, l_j is the output label for the j^{th} opcode variant, and n is the number of opcode variants in IA-32. In the remainder of this chapter, we use the term "label vector" to refer to the vector $\langle l_1, l_2, \ldots, l_n \rangle$. For example, if I of some training input $\langle \varphi, I \rangle$ is "`push ebp; add esp, 1`," the output labels corresponding to opcode variants PUSH_32 and ADD_32_8 will be 1 in the label vector, and the remaining labels will be 0. However, we emphasize that the labels are all independent, and our procedure is equivalent to training 87 separate k-NN-regression models, one for each opcode variant. In a discrete domain, there may be many training-set items that lie on the same point in input space (e.g., two different formulas can contain the same set of QFBV operators). In this case, we use the mean label count over all instances at that point.

The training phase of k-NN regression simply involves obtaining an input feature-vector and label vector for each $\langle \varphi, I \rangle$ pair in the corpus, and adding the pair of vectors to the input space of the model. Later on in this section, we describe how the model is actually used to make predictions during synthesis.

n-gram model. Our intended use case for the n-gram model is to estimate the commonness of an IA-32 instruction sequence. Given a sequence of words $X_1, X_2, \ldots, X_m$ whose probability we would like to model, an n-gram model is a language model satisfying the Markov assumption

$$P(X_1, X_2, \ldots, X_m) = \prod_{i=1}^{m} P(X_i | X_{i-1}, X_{i-2}, \ldots, X_1) \approx \prod_{i=1}^{m} P(X_i | X_{i-1}, X_{i-2}, \ldots, X_{i-(n-1)}). \quad (5.1)$$

The assumption is that the probability of observing the i^{th} word X_i in the context history of the preceding $i-1$ words can be approximated by the probability of observing it in the shortened context history of the preceding $n-1$ words. The conditional probabilities in the above equation can be calculated from frequency counts as follows:

$$P(X_i | X_{i-1}, X_{i-2}, \ldots, X_{i-(n-1)}) = \frac{count(X_i, X_{i-1}, X_{i-2}, \ldots, X_{i-(n-1)})}{count(X_{i-1}, X_{i-2}, \ldots, X_{i-(n-1)})}$$

Typically, n-gram probabilities are not directly derived from frequency counts, and incorporate some *smoothing* mechanism to account for unseen words or n-grams.[5]

In our context, an n-gram is an instruction subsequence of length n (e.g., a bigram is a two-instruction subsequence, a trigram is a three-instruction subsequence, etc.), and a word in an n-gram is an individual instruction. Training the model only requires the instruction-sequence component I of the $\langle \varphi, \mathrm{I} \rangle$ pairs in the corpus, and just involves recording frequency counts for various n-grams.

Synthesis Phase

We now illustrate the workings of McSynth-ML using the same example that was used in §5.1.1.

$$\varphi \equiv EAX' = Mem(ESP + 10) \wedge ESP' = ESP + 18 \wedge ZF' = (ESP + 10 = 0)$$

McSynth-ML uses the same master as McSynth++, and so it splits φ into the same subformulas φ_1 and φ_2 from §5.1.1.

$$\varphi_1 \equiv \varphi \equiv EAX' = Mem(ESP + 10) \qquad \varphi_2 \equiv ESP' = ESP + 18 \wedge ZF' = (ESP + 10 = 0)$$

McSynth-ML's master then hands over φ_1 and φ_2, respectively, to slave synthesizers.

We now illustrate how McSynth-ML's model-assisted slave synthesizes code for φ_2. McSynth-ML first obtains a feature vector for φ_2 based on the QFBV operators that appear in φ_2. Some QFBV operators that would have a 1 in their corresponding entries in the feature vector for φ_2 are $+$ (QFBV_PLUS), $\wedge$ (QFBV_AND), and $=$ (QFBV_EQUALS). McSynth-ML then gives the feature vector as input to the k-NN model, and obtains as output the label vector, which contains a k-NN-regression probability for every opcode variant in IA-32. Given a query point q (input feature-vector), k-NN predicts the probability distribution over the opcode variants as follows: k-NN finds the k training-set items nearest to q using Euclidean distance as the metric, averages their label values, and returns the resulting label vector as the output. Intuitively, the probability value for an opcode variant represents the likelihood of an implementation of φ_2 to contain an instruction belonging to that opcode variant. McSynth-ML discards opcode variants whose probabilities are below a certain threshold (say, 0.1 for this example). For our running example, this step reduces the size of the instruction pool from 240 to 20 instructions. (Note that the pruners inherited from

[5]n-gram models that estimate probabilities directly from frequency counts encounter problems when confronted with any n-grams that have not explicitly been seen before. In practice it is necessary to smooth the probability distributions by also assigning non-zero probabilities to unseen words or n-grams.

McSynth++ had already reduced the size of the instruction pool from around 43,000 to 240.) Let us use $\mathscr{P}$ to denote this remnant instruction pool.

Once the instruction pool has been truncated, McSynth-ML starts best-first search. The search maintains a fringe of instruction-sequence prefixes of varying lengths. The search starts with the empty prefix (ϵ). At any given point during the search, the prefix p that has the highest score according to the cost heuristic c gets expanded, i.e., McSynth-ML appends every instruction in $\mathscr{P}$ to p, thus expanding the contour of the fringe. The cost heuristic c for a prefix p is computed via a combination of (i) p's n-gram probability according to Eqn. (5.1), and (ii) the combined k-NN-regression probability for the opcode variants of individual instructions in p. (c is defined in Line 25 in Alg. 5.3.) Because ϵ is the only prefix in the initial fringe, McSynth-ML expands it by appending every instruction in $\mathscr{P}$ to it. McSynth-ML then checks if any of the newly created candidates implements φ_2 via CEGIS. (Note that the pruners inherited from McSynth++ attempt to prune away a newly created candidate before testing it via CEGIS.) None of the one-instruction sequences in the fringe implement φ_2, and so McSynth-ML looks for the next prefix to expand. Suppose that the one-instruction prefix "`mov esp,` ⟨`Imm32`⟩" has the highest score according to the cost heuristic. McSynth-ML expands it, and tests if any of the newly created two-instruction candidates implements φ_2; none of them do. Note that now the fringe of the search contains both one-instruction and two-instruction prefixes. Suppose that the one-instruction prefix "`add esp,` ⟨`Imm32`⟩" now has the highest score according to the cost heuristic. McSynth-ML expands it, and tests the resulting candidates via CEGIS. The instantiation "`add esp, 10; lea esp, [esp+8]`" of the candidate C ≡ "`add esp,` ⟨`Imm32`⟩`; lea esp, [esp +` ⟨`Imm32`⟩`]`" implements φ_2, and so the slave returns that concrete instruction-sequence as the implementation for φ_2.

The slave similarly finds the implementation "`mov eax, [esp+10]`" for φ_1. McSynth-ML's master concatenates the two instruction sequences and returns the final implementation. The entire synthesis task finishes in a few seconds.

5.2 Algorithm

In this section, we describe the algorithms used by McSynth-ML. First, we present the algorithms for the training phase. Second, we present the algorithms for the synthesis phase. Third, we provide correctness guarantees for McSynth-ML's algorithms. Finally, we present the threats to the validity of our algorithms.

5.2.1 Training Phase

Input: Corpus, k, n
Output: ⟨kNNModel, nGramModel⟩
1: kNNModel ← `InitKNNModel`(k)
2: nGramModel ← `InitNGramModel`(n)
3: **for** each ⟨φ, I⟩ ∈ Corpus **do**
4: ipVector ← `GetQFBVOperators`(φ)
5: labelVector ← `GetOpcodeVariants`(I)
6: kNNModel.`AddToModel`(ipvector, labelVector)
7: nGramModel.`UpdateCounts`(I)
8: **end for**
9: **return** ⟨kNNModel, nGramModel⟩

Algorithm 5.1: Algorithm TrainModels

During the training phase, one needs to train the k-NN regression model and the n-gram language model. Recall from §5.1.2 that the corpus for the training phase consists of hundreds of thousands of equivalent ⟨QFBV-formula, instruction-sequence⟩ pairs produced via symbolic execution. The algorithm for training the models is given as Alg. 5.1. The algorithm takes as input the corpus, the hyperparameter k for k-NN regression, and the length n of n-grams up to which the model should maintain n-gram counts. (For example, if $n = 3$, the model maintains counts for unigrams, bigrams, and trigrams.) The output is a pair of models: kNNModel and nGramModel.

In Alg. 5.1, `InitKNNModel` and `InitNGramModel` initialize parameters of the k-NN model and the n-gram model, respectively (Lines 1 and 2). `GetQFBVOperators`(φ) returns an ordered list of ⟨opr, isPresent⟩ pairs, where isPresent is 1 if operator opr is present in φ; it is 0 otherwise (Line 4). `GetQFBVOperators` traverses the AST of φ, and collects QFBV operators. `GetQFBVOperators` orders its output by QFBV operator. `GetOpcodeVariants`(I) returns an ordered list of ⟨opc, isPresent⟩ pairs, where isPresent is 1 if instruction sequence I contains an instruction belonging to the opcode variant opc; it is 0 otherwise (Line 5). Recall from §5.1.2 that the opcode variant is a conjunction of opcode category and operand sizes. `GetOpcodeVariants` inspects the ASTs of instructions in I to produce its output, which is ordered by opcode variant. `AddToModel` adds an ⟨input-vector, label-vector⟩ pair to the input space of the k-NN-regression model (Line 6). `UpdateCounts` teases apart the various n-grams of length up to n from the instruction sequence I, and updates their counts in the model (Line 7). Recall from §5.1.2 that in the context of McSynth-ML, an n-gram is an instruction subsequence of length n. Alg. 5.1 finally returns the two models (Line 9).

5.2.2 Synthesis Phase

Recall from §5.1.2 that McSynth-ML uses the same master as McSynth++, and so in this sub-section, we describe only the algorithm used by McSynth-ML's slave. In this sub-section,

Input: φ, instrPool, kNNModel, t_p
Output: instrPool′, labelVector
1: ipVector ← `GetQFBVOperators`(φ)
2: labelVector ← kNNModel.`GetLabelVector`(ipVector)
3: instrPool′ ← ϵ
4: **for** each ⟨opc, prob⟩ ∈ labelVector **do**
5: **if** prob < t_p **then**
6: **continue**
7: **end if**
8: instructions ← instrPool.`GetInstrByOpcodeVariant`(opc)
9: instrPool′.`Concat`(instructions)
10: **end for**
11: **return** ⟨instrPool′, labelVector⟩

Algorithm 5.2: Algorithm TruncateInstrPool

we use φ to refer to a sub-formula given as input to a slave synthesizer.

Before presenting the algorithm used by the slave, we present a procedure called TruncateInstrPool that uses features of φ along with k-NN regression to retain in the instruction pool only those instructions that are highly likely to implement φ. The algorithm for TruncateInstrPool is given as Alg. 5.2. The algorithm takes as inputs a formula φ, the instruction pool, and the k-NN model built during the training phase. The role of the remaining input (t_p) will be explained in the next paragraph. TruncateInstrPool returns the remnant instruction pool, along with the label vector for φ, as the output.

Alg. 5.2 first obtains the QFBV operators present in φ via `GetQFBVOperators` (Line 1). It supplies that set of operators as the test vector to the k-NN-regression model, and queries for the opcode-variant probabilities via `GetLabelVector` (Line 2). `GetLabelVector` produces as output an ordered list of ⟨opc, prob⟩ pairs (ordered by opc), where opc is an opcode variant, and prob is its corresponding k-NN-regression probability. Recall from §5.1.2 that prob represents the likelihood of an implementation of φ to contain an instruction belonging to the opcode variant opc. Starting with an empty instruction pool instrPool′ (Line 3), TruncateInstrPool repeatedly adds to instrPool′ instructions from instrPool belonging to opcode variants whose k-NN-regression probabilities are greater than a threshold value t_p (Lines 4–10). TruncateInstrPool finally returns the remnant instruction pool instrPool′, along with the label vector for φ (Line 11). instrPool′ contains only the instructions that, according to k-NN regression, are most likely to implement the input φ.

Now we are ready to present the algorithm used by McSynth-ML's slave. Recall from §5.1.2 that the slave performs the actual enumerative synthesis in McSynth-ML. The algorithm used by the slave is given as Alg. 5.3. Alg. 5.3 takes the following inputs: (i) the input formula φ, (ii) a k-NN-regression model, (iii) the threshold probability t_p used in TruncateInstrPool, (iv) an n-gram-based language model, and for pragmatic purposes (v)

Input: φ, kNNModel, t_p, nGramModel, *max*
Output: C_{conc} or FAIL

```
 1: instrPool ← ReadInstrPool( )
 2: for each instruction i ∈ instrPool do
 3:   if PruneFootprint(φ, i) or PruneBitsLost(φ, i) then
 4:     instrPool ← instrPool − {i}
 5:   end if
 6: end for
 7: ⟨instrPool′, labelVector⟩ ← TruncInstrPool(φ, instrPool, kNNModel, t_p)
 8: prefixes ← new PriorityQueue( )
 9: prefixes.insert(⟨ϵ, 0⟩)
10: while prefixes ≠ ∅ do
11:   ⟨p, prob⟩ ← prefixes.PopMax( )
12:   for each i ∈ instrPool′ do
13:     C ← Append(p, i)
14:     ψ_C ← ⟪C⟫
15:     if PruneFootprint(φ, C) or PruneBitsLost(φ, C) then
16:       continue
17:     end if
18:     ret = CEGIS(φ, C, ψ_C)
19:     if ret ≠ FAIL then
20:       return ret
21:     end if
22:     if C.length = max then
23:       continue
24:     end if
25:     c ← (1 − λ)∗ nGramModel.GetProb(p) + λ ∗ ∏_{i∈p} labelVector.GetProb(OpcodeVariant(i))
26:     prefixes.insert(C, c)
27:   end for
28: end while
29: if t_p = 0 then
30:   return FAIL
31: else
32:   return McSynthMLSlave(φ, kNNModel, 0, nGramModel, max)
33: end if
```

Algorithm 5.3: Algorithm McSynthMLSlave

the maximum length *max* of instruction-sequence prefixes to enumerate during synthesis. The slave also takes a timeout value as an additional input (not shown in Alg. 5.3). The output of Alg. 5.3 is either a concrete instruction-sequence C_{conc} that implements φ, or FAIL if the slave could not find an implementation before the timeout expires.

The slave first reads the list of instructions from a file (via `ReadInstrPool`) (Line 1), and uses the footprint-based pruner and the bits-lost-based pruner inherited from McSynth++ to prune away useless instructions from the instruction pool via `PruneFootprint` and `PruneBitsLost`, respectively (Lines 2–6). The slave then calls TruncateInstrPool, and obtains

a smaller pool of instructions (Line 7), and subsequently begins its best-first search. The fringe of the search is implemented as a priority queue, with instruction-sequence prefix as key and the cost heuristic c (defined in Line 25) as priority. Recall from §5.1.2 that the cost-heuristic value c for a prefix p is a combination of (i) the n-gram probability of p according to the language model (obtained via nGramModel.`GetProb`), and (ii) the combined k-NN-regression probability of the opcode variant of each instruction i in p (obtained via labelVector.`GetProb`(`OpcodeVariant`(i))). In Line 25, λ denotes the weighting parameter for the scores obtained from the two models. The slave initially inserts the empty prefix into the queue (Line 9). At any given point in the search, the slave picks the prefix with the highest priority, and expands it by appending to it every instruction in the instruction pool (Lines 11–13). The slave then checks if a candidate instruction-sequence obtained via expansion is useless, and thus can be pruned away using the footprint-based and bits-lost-based pruners (Lines 14–17). Candidates that escape pruning are tested for equivalence with φ via CEGIS (Line 18). If CEGIS finds an instantiation C_{conc} of a candidate C that implements φ, the slave returns that instantiation as the implementation (Lines 19–21). Otherwise, the slave adds the (useful) candidate as a prefix in the priority queue (Line 26) provided that the length of the candidate is not greater than *max* (Lines 22–24).

To preserve the completeness guarantees of McSynth-ML (see Thm. 5.5), if the slave fails to find an implementation for an input φ with the truncated instruction pool, then the slave attempts to find an implementation without truncating the instruction pool. In Alg. 5.3, if an implementation could not be found when $t_p > 0$, Alg. 5.3 recursively calls itself with $t_p = 0$ (Line 32). Consequently in that recursive call, TruncateInstrPool will default to merely returning the input instruction pool. If an implementation could not be found even in this second attempt, the slave returns FAIL (Line 30).

5.2.3 Correctness

In this sub-section, we present the soundness and completeness guarantees of McSynth-ML. Because McSynth-ML uses the same master as McSynth++, and correctness properties of McSynth++ have been proven elsewhere [92], we only discuss correctness of McSynthMLSlave (Alg. 5.3) in this section.

Theorem 5.4. ***Soundness.*** *Alg. 5.3 is sound. (The formula $\langle\langle I \rangle\rangle$ for instruction sequence I returned by Alg. 5.3 is logically equivalent to the input QFBV formula φ.)*

Proof. The CEGIS loop of the slave (Line 18 in Alg. 5.3) returns an instruction sequence *I* only if $\langle\langle I \rangle\rangle$ is equivalent to φ. □

McSynth-ML's slave has the same completeness guarantees as that of McSynth++ (Thm. 2 in [92]).

Theorem 5.5. ***Completeness.*** *Modulo SMT timeouts and candidates that are pruned away because of imprecision in $BITS^{\#}_{required}(\varphi)$ ([92, §4.2.2]), if there exists an instruction sequence I that (i) is equivalent to φ, and (ii) does not superfluously use/modify locations that are otherwise unused/unmodified by φ, then Alg. 5.3 will find I and terminate.*

Proof. McSynth-ML's best-first search equipped with the n-gram-based and kNN-based cost heuristic does not prune away instruction-sequence prefixes; it merely prioritizes them. Although McSynth-ML initially tries to synthesize an implementation using a truncated instruction pool produced by k-NN regression, if synthesis with the truncated pool fails, McSynth-ML attempts synthesis with the untruncated instruction pool (Line 32 in Alg. 5.3). Consequently, the only sources of incompleteness in McSynthMLSlave are the pruners inherited from McSynth++ (Lines 3–5 and 15–17 in Alg. 5.3), and McSynth-ML's slave has the same completeness guarantees as that of McSynth++. □

5.2.4 Threats to Validity

There are three threats to the validity of our algorithms.

1. The parameters of our models and algorithms need to be tuned for McSynth-ML to synthesize code effectively. If the parameters are significantly off, one cannot guarantee low synthesis times using McSynth-ML. (§5.3 describes how we did the tuning for our experiments.)

2. The models used by McSynth-ML should be trained with sufficient training data; failure to do so might result in higher synthesis times.

3. If one wishes to synthesize code that possesses a certain "quality" using McSynth-ML, the implementations in the training data should also possess that "quality." For example, if one wishes to find optimal implementations using McSynth-ML, one should generate training examples using a superoptimizer [16, 17, 49, 55, 64, 65, 76]. For our experiments, the instruction sequences in our training data come from code sequences generated by a standard compiler (`gcc -O2`), and so the implementations produced by McSynth-ML resemble those produced by a compiler.

5.3 Implementation

Because McSynth-ML is an extension of McSynth++, McSynth-ML has the same underlying components as McSynth++: Transformer Specification Language (TSL) [53] to convert instruction sequences into QFBV formulas; ISAL [53, §2.1] to generate the templatized instruction pool for synthesis; and Yices [30] as its SMT solver. Just like McSynth++, in McSynth-ML, memory is addressed at the level of individual bytes; McSynth-ML is also capable of accepting scratch registers for synthesis [90, §4.4]. To implement k-NN regression, we used the scikit-learn toolkit [4]. We used the MIT Language Modeling Toolkit (MITLM) [3] for our n-gram model.

For our n-gram model, we chose via cross-validation (i) the value 3 for the maximum length n up to which the n-gram model has to maintain n-gram counts (see Alg. 5.1), and (ii) Kneser-Ney smoothing. We set as 2 the hyperparameter k of k-NN regression (see Alg. 5.1) through cross-validation. We determined the weighting parameter λ (see Alg. 5.3) as follows: we created a separate test suite $\mathscr{T}$ of 50 formulas (different from the test formulas used in our experiments, and the formulas in our training corpus), and for different values of λ, we measured the average synthesis time obtained via McSynth-ML for $\mathscr{T}$. We were able to obtain the lowest average synthesis-time for the value $\lambda = 0.25$; so we set $\lambda = 0.25$ in our experiments. We set the truncation threshold parameter $t_p = 0.1$ in our experiments (see Alg. 5.2).

5.4 Experiments

We tested McSynth-ML on QFBV formulas obtained from instruction sequences from the SPECINT 2006 benchmark suite [44]. Our experiments were designed to answer the following questions:

1. In comparison with McSynth++, what is the speedup in synthesis time caused by McSynth-ML's best-first search assisted only by the n-gram model ($\lambda = 0.0$)? What is the speedup when McSynth-ML's best-first search is assisted only by k-NN regression ($\lambda = 1.0$)? What is the speedup when the search is assisted by both models ($\lambda = 0.25$)?
2. In comparison with McSynth++, on how many formulas does McSynth-ML timeout?
3. In comparison with McSynth++, what is the reduction in the size of the instruction pool caused by McSynth-ML's k-NN-based instruction-pool truncation?
4. As a limit study, how does McSynth++ perform when k-NN regression is used to truncate instruction pools in its slaves?

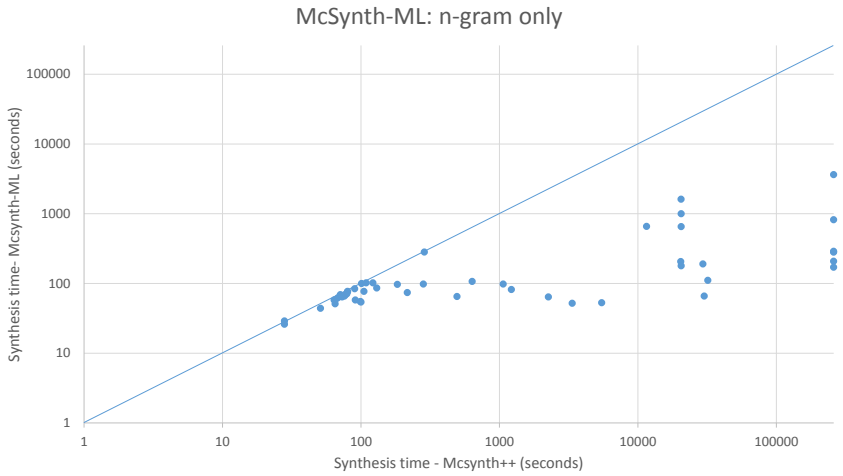


Figure 5.6: Effect of only the n-gram model assisting McSynth-ML's best-first search for the corpus of 50 QFBV formulas.

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

To answer the questions, we used a benchmark suite that is slightly different from the suite that was used in the previous chapters. We used QFBV formulas obtained from a representative set of "important" instruction sequences that occur in real programs. We harvested the most frequently occurring instruction sequence of length 6 through 10 from each of the 10 binaries in the SPECINT 2006 benchmark suite (50 instruction sequences in total). We converted each instruction sequence into a QFBV formula and used the resulting formulas as inputs for our experiments.

Note that in general there is no restriction on the source of the input formula, and the formula can come from any client; we simply chose to obtain the input formulas from instruction sequences for experimental purposes.

In the remainder of this section, when we use the term "synthesis time," we refer to the time spent only by the slave synthesizers in McSynth++ and McSynth-ML, respectively; we do not include the time spent by the masters because McSynth++ and McSynth-ML have identical masters. However for all formulas in our test suite, the time spent by the master is negligible: < 2% of the total synthesis time.

To answer the first two questions, we measured the synthesis time with (i) only the n-gram-based language model supplying the cost heuristic for McSynth-ML's best-first search ($\lambda = 0.0$), (ii) only k-NN regression supplying the cost heuristic for McSynth-ML's best-first search ($\lambda = 1.0$), and (ii) both models supplying the cost heuristic for McSynth-ML's best-first search ($\lambda = 0.25$, determined by the method outlined in §5.3). We compared the numbers against the baseline synthesis-time numbers obtained from McSynth++.

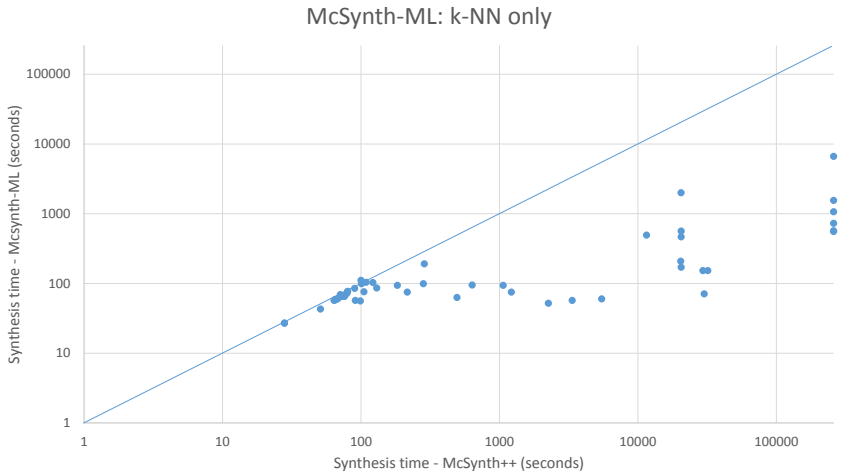


Figure 5.7: Effect of only k-NN regression assisting McSynth-ML's best-first search for the corpus of 50 QFBV formulas.

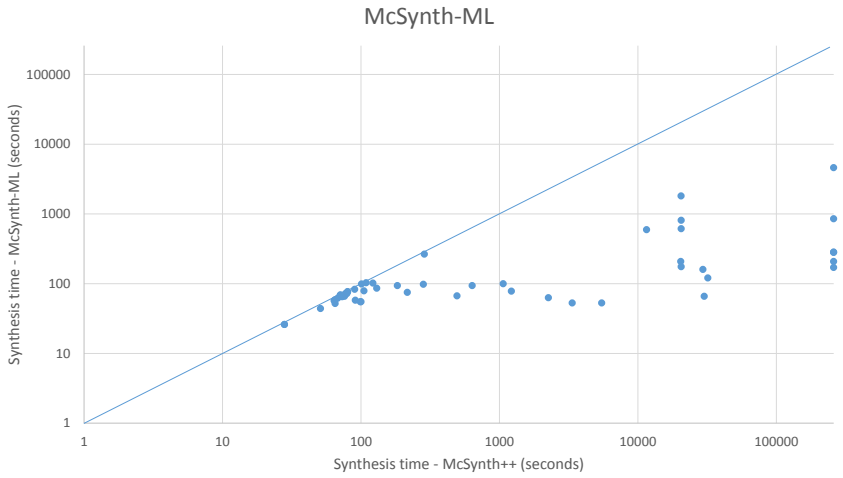


Figure 5.8: Effect of both models assisting McSynth-ML's best-first search for the corpus of 50 QFBV formulas.

The results are shown in Figs. 5.6, 5.7, and 5.8, respectively. In the figures, the blue lines represent the diagonals of the scatter plots. If a point lies below and to the right of the diagonal, the baseline performs worse. All axes use logarithmic scales. McSynth++ timed out on 6 formulas. (The timeout value was three days.) McSynth-ML did not timeout on any formula in the test suite. For the formulas that timed out in McSynth++ but did not timeout in McSynth-ML, the average speedup in synthesis time caused by improvements (i), (ii), and (iii) are over 573X, 223X, and 549X, respectively (computed as a geometric mean). For the formulas that did not timeout, the average speedup in synthesis time caused by improvements (i), (ii), and (iii) are 4.5X, 4.57X, and 4.55X, respectively. If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedups are more pronounced: 12.3X, 13.2X, and 12.7X, respectively. One can see that

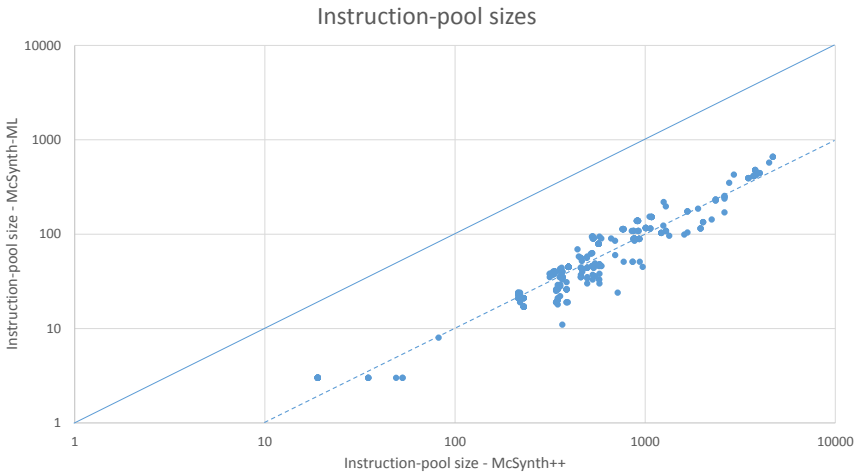


Figure 5.9: Reduction in instruction-pool sizes caused by McSynth-ML's k-NN-based truncation.

the n-gram model has a more pronounced effect on the speedup for formulas that timed out in McSynth++, whereas k-NN regression has a more pronounced effect on the speedup for the other formulas.

To answer the third question, we measured the sizes of the instruction pools used by McSynth++ and McSynth-ML, respectively, for each formula in the test suite. The results are shown in Fig. 5.9. The blue line represents the diagonal of the scatter plot. If a point lies below and to the right of the diagonal, the baseline has a larger instruction pool. The axes use logarithmic scales. Synthesis of code for each formula in the test suite requires several slave invocations, and each slave invocation creates a new instruction pool, which is represented by a single data point in Fig. 5.9. Consequently in Fig. 5.9, there are more than 50 data points (369 exactly). The average reduction in instruction-pool size caused by truncation based on k-NN-regression probabilities is 9.7X (computed as a geometric mean). One can notice this order-of-magnitude improvement in Fig. 5.9: all the data points roughly lie along the dotted line that represents an order-of-magnitude improvement over the baseline.

To answer the fourth question, we performed a control experiment in which we used k-NN regression to truncate the instruction pools in McSynth++'s slaves. The goal of this experiment is to demonstrate how well the linear search in McSynth++ can perform when it is presented with the same instruction pools as McSynth-ML. We compared the numbers against the synthesis-time numbers obtained from McSynth++ and McSynth-ML. The results are shown in Fig. 5.10 and Fig. 5.11, respectively. The blue lines represent the diagonals of the scatter plot. If a point lies below and to the right of the diagonal, the baseline performs worse. All axes use logarithmic scales. For the formulas that timed out in McSynth++, the average speedup in synthesis time caused by the truncated instruction

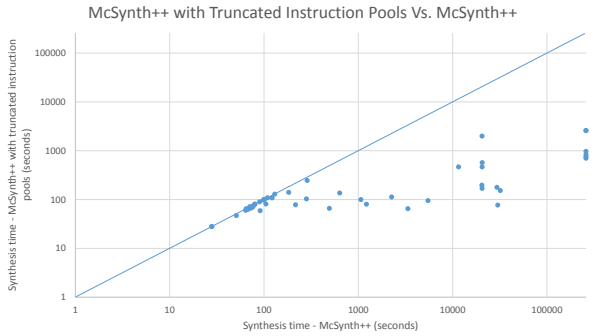


Figure 5.10: Effect of k-NN-based instruction-pool truncation on McSynth++ for the corpus of 50 QFBV formulas.

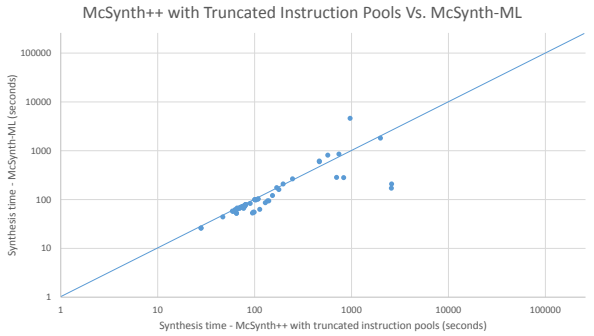


Figure 5.11: Comparison of McSynth-ML against McSynth++ with truncated instruction pools.

pool is over 218X (computed as a geometric mean), versus 549X speedup for McSynth-ML. For 2 out of these 6 formulas, linear search performed better than model-assisted best-first search. For the formulas that did not timeout, the average speedup in synthesis time caused by the truncated instruction pool is 4X (computed as a geometric mean), versus 4.55X for McSynth-ML. For 6 out of these 44 formulas, linear search performed better than model-assisted best-first search.

In summary, in comparison with McSynth++, McSynth-ML speeds up the synthesis time by over 549X for the 6 formulas that timed out in McSynth++ but did not timeout in McSynth-ML. Moreover, McSynth-ML does not timeout on any formula in our test suite. For the formulas that did not timeout in McSynth++, McSynth++ speeds up the synthesis time by 4.55X. If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is 12.7X.

5.5 Related Work

Search strategies in superoptimization. Superoptimization aims to find an optimal instruction sequence for a target instruction-sequence. While early attempts at superoptimization used linear search as the core search strategy [16, 17, 49, 55], modern superoptimizers use other approaches.

The stochastic superoptimizer STOKE [76] uses stochastic techniques for finding an optimal instruction sequence. STOKE uses Markov Chain Monte Carlo (MCMC) sampling to search through the space of instruction sequences, and encodes its cost heuristic in terms of correctness and performance. To find implementations that are algorithmically different from the input instruction-sequence, STOKE begins its search from random points in the space of instruction sequences (instead of starting the search from the input instruction-

sequence). The stochastic search strategy, along with the fast MCMC sampling allows STOKE to quickly synthesize larger programs (10 - 15 x86 instructions).

The GreenThumb superoptimizer framework [64] employs a cooperative search strategy: complementary enumerative, symbolic, and stochastic search strategies work in conjunction by exchanging the best programs each search instance has discovered so far. The cooperative strategy, along with other improvements, allow instantiations of the GreenThumb framework to perform better than STOKE: the GreenThumb instantiations optimize the benchmarks faster, and obtain better (faster) implementations.

STOKE and GreenThumb employ stochastic strategies for the search, which renders the search incomplete. In comparison, McSynth-ML uses a model-assisted best-first search, which merely prioritizes (reorders) the candidates during search, while maintaining completeness guarantees. Moreover, unlike McSynth-ML, superoptimizers only use information obtained from the input instruction-sequence to guide the search; they do not use models learned from superoptimized code-sequences to guide the search.

Model-assisted synthesis. Recent works in program synthesis have employed models learned from a corpus of programs to assist synthesis.

- Slang is a code-completion tool that uses API calls to fill holes in a partial program [67]. Slang learns two language models from a corpus of API-call sequences harvested from programs: an n-gram model and a recurrent neural-network (RNN) model. Slang fills the holes in a test program with the most likely API calls according to the models.

- *anyCode* synthesizes well-formed Java expressions from free-form queries containing a mixture of English and Java [42]. *anyCode* uses a conjunction of the following to synthesize and rank expressions: (i) natural-language processing (NLP) tools to process the input English text, and (ii) language models and a probabilistic context-free-grammar model built from corpus of Java programs.

- JSNice predicts names and types for identifiers in JavaScript programs [68]. JSNice learns a probabilistic model (employing conditional random fields) for program properties from a huge corpus of existing programs, and uses the model to predict names and type annotations in a test program.

McSynth-ML advances the state-of-the-art in model-assisted synthesis in the following ways:

- While existing approaches commonly use language models to find most likely code completions, McSynth-ML uses a language model to assist in synthesizing an entire low-level code sequence.
- None of the existing model-assisted-synthesis techniques employ a model that correlates features of implementations with features of specifications, and subsequently use that model to find the most likely implementation for a test specification. To the best of our knowledge, McSynth-ML is the first model-assisted synthesizer that learns such a model to assist synthesis.

Part II

Applications

6 Partial Evaluation of Machine Code

This chapter presents the first instantiation of the semantics-based binary rewriting framework (Framework 1.1) described in Chapter 1: a partial evaluator for IA-32.

One of the potential applications of machine-code analysis and rewriting is to facilitate reuse of software binaries in the absence of source code. In particular, it would be desirable to have a binary-rewriting tool that can (i) optimize a binary for the common case (e.g., a file-write routine optimized for a certain file descriptor), or (ii) extract an executable component from a bloated binary (e.g., a small text encoder embedded in a web server).

Partial evaluation [48] is a program-specialization framework that can perform the aforementioned tasks. However, there are no tools that partially evaluate machine code. Existing binary-rewriting tools either de-obfuscate [27, 81, 99], superoptimize [16, 76], or harden [6, 33, 83] binaries, but do not perform partial evaluation. Moreover, machine-code partial evaluation presents a number of new challenges, and source-code partial-evaluation algorithms [9, 26, 50] would not produce satisfactory results if they were applied to machine code in a straightforward manner (see §6.2).

This chapter presents an algorithm for off-line partial evaluation of machine code. The inputs to the algorithm are a binary B, and a division of B's inputs as S (*static inputs*) and D (*dynamic inputs*). Values for static inputs are known at specialization time; values for dynamic inputs are unknown at specialization time. Our partial-evaluation algorithm produces a binary B_S that satisfies the equation

$$[\![B]\!](S, D) = [\![B_S]\!](D),$$

where $[\![\cdot]\!]$ denotes the meaning function for the instruction set in which the binary is written. Executing B_S with input D produces the same results as executing B with inputs S and D. However, B_S is specialized with respect to S, and can be significantly faster than B.

Partially evaluating a binary with respect to commonly used inputs produces specialized versions of the binary that are optimized for the common inputs. (See §6.5.1.) Partially evaluating with respect to a fixed value of an input flag can extract a smaller executable component from a bloated binary. (See §6.5.2.) The extracted component can be used in embedded systems where executable size matters, or be linked against other programs that require the component. (See the experiment with `lzfx` in §6.5.2.)

We have implemented our algorithm in a tool, called WiPEr (an acronym for the "Wisconsin Partial Evaluator"), that partially evaluates Intel IA-32 binaries. WiPEr works on the Intermediate Representations (IRs) recovered from machine code by an existing tool, CodeSurfer/x86 [15]. WiPEr follows the classical two-phase approach of *binding-time analysis* (BTA) followed by *specialization*. WiPEr's BTA uses *forward slicing* to determine a priori which instructions can be specialized away (static instructions), and which cannot (dynamic instructions). Given the values for static inputs in the form of a partial state, WiPEr's specializer evaluates static instructions, residuates code for static values that might be used by dynamic instructions, and emits unmodified dynamic instructions.

As mentioned earlier, there are several new challenges that arise in the context of machine-code partial evaluation. These issues, along with the strategies that WiPEr uses to overcome them, distinguish WiPEr from other partial evaluators that have been described in the literature.

- 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 follow only a subset of an instruction's semantics, the slicing algorithm is forced to include the entire instruction. This effect can cascade, and cause BTA to be very imprecise. The BTA algorithm in WiPEr makes use of a special-case instruction-splitting method that *decouples* multiple updates performed by a single instruction by splitting the assignments across multiple instructions. The decoupling transformation increases the precision of WiPEr's BTA significantly.

- An instruction's abstract-syntax tree (AST) is often not parameterized by all of the operands of the instruction: some operands are "baked into" the semantics. In contrast with source-code partial evaluators that specialize the AST of a dynamic statement, WiPEr specializes an explicit representation of the *semantics* of a dynamic instruction, and uses a machine-code synthesizer [90] to produce residual code.

- In machine code, values in memory are accessed by explicit address computations, followed by loading. The residual code produced by a naïve machine-code partial evaluator will contain specialization-time addresses, and consequently would not allow the stack and heap to be at different positions at run-time than at specialization-time. WiPEr represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization. The resulting code produced by WiPEr is

independent of the layout of the specialization-time address space, and can be run with, e.g., the stack base at a different address.[1]

Contributions

The contributions of this chapter include the following:

- We present an algorithm for machine-code partial evaluation. To the best of our knowledge, WiPEr is the first partial evaluator that works on machine code.[2]
- We have developed an instruction-decoupling transformation to counter cascading imprecision caused by the granularity issue in machine-code slicing. Chapter 7 presents a more principled approach to counter the granularity issue in machine-code slicing.
- We have introduced an instruction-specialization technique that specializes an explicit representation of the semantics of an instruction with respect to a partial state, and residuates code via machine-code synthesis.
- We have developed a specialization algorithm that represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization. The residual code produced by our specializer is independent of the specialization-time memory layout, and allows the stack and heap to be at different positions at run-time than at specialization-time.

Our methods have been implemented in WiPEr, a partial evaluator for Intel IA-32. We partially evaluated seven test applications using WiPEr, and found that, on average (computed via geometric mean), the specialized binaries have a speedup of 1.3. We also used WiPEr to extract executable components from bloated binaries such as `bzip2`.

6.1 Background

In this section, we briefly describe source-code partial evaluation, and slicing.

[1] Note that we are referring here to the ability to reposition the stack at the beginning of run-time, not the relocatability of the residual code per se. The residual code produced WiPEr is also relocatable.

[2] Confirmed by personal communication with Neil Jones, Robert Glück, and Saumya Debray.

```
int power(int a, int b, int n) {
1: int prod = 1;
2: while (n--) {
3:   prod *= (a + b);
4: }
5: return prod;
}
```

Figure 6.1: Input power program—computes $(a + b)^n$.

```
int power_r(int a) {
   int prod = 1;
   prod *= (a + 1);
   prod *= (a + 1);
   prod *= (a + 1);
   prod *= (a + 1);
   return prod;
}
```

Figure 6.2: Residual power_R program—computes $(a + 1)^4$.

6.1.1 Partial Evaluation

Partial evaluation is a framework for specializing and optimizing programs [48]. Given a program that takes some inputs, and given values for some of the inputs, the goal of partial evaluation is to produce a program that is specialized (optimized) with respect to the input values provided. Fig. 6.1 shows a program power that computes $(a + b)^n$, where a, b, and n are the inputs to the program. Suppose that power is frequently called with the values $b = 1$, and $n = 4$. Given power, and the input values $b = 1$ and $n = 4$, a partial evaluator produces the residual program power_R shown in Fig. 6.2. As we can see in power_R, partial evaluation has compiled data ($n = 4$) into control. (See Eqn. (6.1).) In effect, partial evaluation performs optimizations such as loop unrolling, constant propagation and folding, function in-lining, etc. to produce the residual program—although a partial evaluator does not have such optimizations built into it *per se*. In this section, and in the rest of this chapter, when we say "source-code partial evaluator," we refer to a partial evaluator for an imperative language, such as C [9].

An off-line source-code partial evaluator works in two phases: *binding-time analysis* and *specialization*.

The input to BTA is a division (or classification) of the input variables as *static inputs* and *dynamic inputs*. Given a division of the input variables, the goal of BTA is to extend the division to all program variables—i.e., classify each occurrence of a variable as either static or dynamic. A division computed by BTA must be *congruent*, i.e., any variable that depends on a dynamic variable must be classified dynamic. Using the computed division, expressions and statements are classified as either static or dynamic [8]. For the power program, BTA classifies variables a and prod, and statements 1, 3, and 5 as dynamic.

Given values for static inputs, in the form of a partial static-state σ, the specializer specializes the input program with respect to σ, and produces the residual program. A source-code specializer uses the following approach:

```
int foo() {
1: int a = 1, b = 2;
2: int c, d, e;
3: scanf(''%d'', &c);
4: d = add(a, b);
5: e = add(c, c);
6: return d + e;
}
```

Figure 6.3: Example program foo to illustrate lifting.

```
int foo_r() {
1: int c, d, e;
2: scanf(''%d'', &c);
3: d = add(1, 2);
4: e = add(c, c);
5: return d + e;
}
```

Figure 6.4: Residual foo_r program.

Approach $\mathbf{PE}_S$

- Evaluate static expressions and statements, and
- Specialize the AST of dynamic expressions and statements with respect to σ by substituting values from σ for variables, simplifying, and emitting the resulting AST.

The second step is sometimes called *residuation*.

For the power program, σ is $[n \mapsto 4, b \mapsto 1]$. Because the loop condition (Line 2 in Fig. 6.1) is static, the specializer unrolls the loop four times. The specializer residuates the expression `a + 1` by specializing the dynamic expression `a + b` with respect to σ.

Sometimes a static expression could be used in a dynamic context. For example, consider the program foo given in Fig. 6.3. Because `c`'s value is established dynamically in line 3, both of the formal parameters of `add` are classified dynamic by BTA. Although both actual parameters of the call to `add` in line 4 of Fig. 6.3 are static, they appear in a dynamic context—the static actuals will be bound to dynamic formals in `add`. To produce a residual program that compiles and/or does not access uninitialized locations, the partial evaluator must residuate code for the static actuals. Thus the partial evaluator "lifts" static expressions that appear in dynamic contexts, and the specializer residuates code for lifted expressions (e.g., line 3 in Fig. 6.4). The term "lifting" refers to the process of changing the binding time of an expression from static to dynamic.

Suppose that each statement in the original program is uniquely identified by a label l, and that each concrete state σ that arises during an execution is partitioned into σ_S and σ_D according to the division established by BTA. Then an execution trace of the original program is a sequence of elements of the form $l : (\sigma_S, \sigma_D)$. (Each element of the trace records the state (σ_S, σ_D) observed at an execution of a statement l.) The effect of partial evaluation on an execution trace of the specialized program can be expressed by the following reparenthesization:

$$l : (\sigma_S, \sigma_D) \longrightarrow (l, \sigma_S) : \sigma_D \tag{6.1}$$

Operationally, the partial evaluator creates a new residual statement for every unique (l, σ_S) pair it observes at specialization time. Eqn. (6.1) shows how each element $(l, \sigma_S) : \sigma_D$ in an execution trace of the specialized program can be mapped back to the corresponding element $l : (\sigma_S, \sigma_D)$ in the trace of the original program.

6.1.2 Slicing

Slicing [46, 98] computes the set of program points that affect (or are affected by) a given program point called the *slicing criterion*. (*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 is typically performed using an IR called a *system dependence graph* (SDG) [35, 46]. 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 a statement, a condition, 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 [35]. 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. Given an SDG representation of the program and a slicing criterion, an interprocedural-slicing algorithm includes in the slice the SDG nodes that reach (or can be reached from) the slicing criterion by following data- and control-dependence edges. A context-sensitive interprocedural-slicing algorithm uses context-free language (CFL) reachability [69] to reduce the number of nodes in the slice [46, 71].

6.2 Overview

At a very high level, WiPEr is similar to the off-line source-code partial evaluator described in §6.1.1: WiPEr's algorithm works in two phases, BTA and specialization. However, because WiPEr is dealing with machine code, WiPEr's algorithm differs significantly from that of a source-code partial evaluator. In this section, we present an example to illustrate WiPEr's algorithm, while highlighting the following: (i) the issues that arise in machine code, (ii) why techniques used in a source-code partial evaluator are unsatisfactory to resolve the issues, and (iii) how WiPEr resolves the issues.

Fig. 6.5 shows an IA-32 program sum that takes, a, v, and n as inputs, and computes $a + b[n]$. (Array b is statically allocated and has 5 elements. Lines 6–12 in Fig. 6.5 initialize

```
                        ;Initialization loop
                        6: L2:cmp [esp+20],0
                        7: jl L1                        ;Computation of a+b[n]
1: push eax; a          8: mov ebx,[esp+28]             13: L1:mov eax,[esp+32]
2: push ebx; v          9: mov edx,[esp+20]             14: mov ecx,[esp+24]
3: push ecx; n          10: mov [esp+edx*4],ebx         15: mov ebx,[esp+ecx*4]
4: push 4               11: sub [esp+20],1              16: add eax,ebx
5: sub esp,20; b        12: jmp L2                      17: add esp,36
```

Figure 6.5: Input sum program, which computes $a + b[n]$.

the elements of b to v.) n is assumed to be between 0 and 4. The inputs a, v, and n are available in the EAX, EBX, and ECX registers, respectively, at the beginning of the program. The output is available in the EAX register at the end of the program. Let us use WiPEr to partially evaluate sum with respect to the input value $v = 1$.

No source-code variables. Recall from §6.1.1 that the goal of BTA is to compute the division of all program variables as either static or dynamic. The abstraction of a source-code "variable" is absent at the machine-code level. (We assume that the input binary lacks symbol-table and debugging information.) Consequently, a division of source-code variables cannot be obtained at the machine-code level.

However, there are tools [15] that recover "variable-like" abstractions [14] from machine code, and use them to construct IRs such as an SDG, a control flow graph (CFG), etc. WiPEr performs BTA via forward slicing over the SDG recovered from the binary [28]. □

The inputs to WiPEr's BTA phase are: (i) the SDG of the binary, and (ii) the instructions that initialize the dynamic inputs (the slicing criterion). The output is an annotated program, whose instructions are annotated with binding times: static (S), or dynamic (D). In our example, the dynamic inputs a and n are available in the EAX and ECX registers at instructions 1 and 3, respectively. Consequently, instructions 1 and 3 constitute the slicing criterion. All instructions that are in the forward slice depend on a and n, and hence are classified dynamic; the remaining instructions are classified static.

Another feature of ISAs that makes them different from typical high-level languages is that most instructions are multi-assignments: they have several inputs, and several outputs. The next two issues discussed below arise in machine-code partial evaluation because instructions have multiple outputs, and multiple inputs, respectively.

The granularity issue in slicing. Consider instruction 1 in the sum program. 1 modifies the stack-pointer register ESP in addition to copying the value in the EAX register to a memory location. Because all of the remaining instructions in sum either directly or

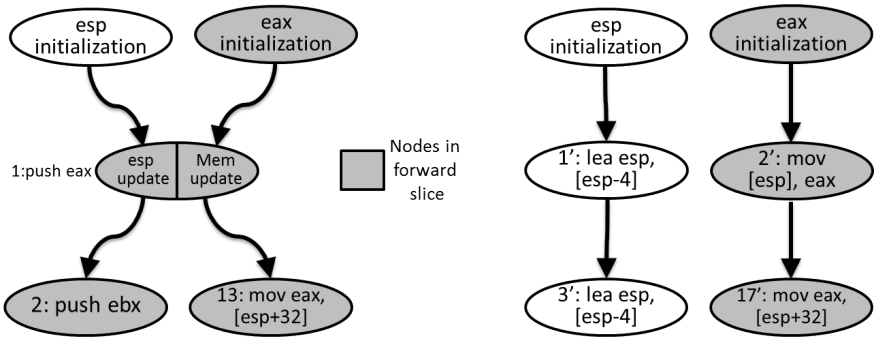


Figure 6.6: SDG snippet to illustrate decoupling.

```
1′:lea esp,[esp-4]
2′:mov [esp],eax ; a
3′:lea esp,[esp-4]
4′:mov [esp],ebx ; v
5′:lea esp,[esp-4]
6′:mov [esp],ecx ; n
7′:lea esp,[esp-4]
8′:mov [esp],4
9′:sub esp,20 ; b

;Initialization loop
10′:L2:cmp [esp+20],0
11′:jl L1
12′:mov ebx,[esp+28]
13′:mov edx,[esp+20]
14′:mov [esp+edx*4],ebx
15′:sub [esp+20],1
16′:jmp L2

;Computation of a+b[n]
17′:L1:mov eax,[esp+32]
18′:mov ecx,[esp+24]
19′:mov ebx,[esp+ecx*4]
20′:add eax,ebx
21′:add esp,36
```

Figure 6.7: Rewritten program sum′.

transitively use ESP, the slice with respect to instruction 1 consists of all the instructions in sum, and thus every instruction in sum would be classified dynamic. This imprecision in slicing is caused by a granularity problem: the push instruction *independently* updates ESP and the memory location. Because the dynamic input is in EAX, the slice requires only the memory update that push performs and not the update to the value of ESP. However, a slice cannot include only a part of an instruction; it has to include the entire instruction. Consequently, the entire push instruction, and the remaining instructions in sum are added to the slice. This issue is illustrated by the SDG snippet on the left in Fig. 6.6.

To resolve this issue, WiPEr rewrites sum by decoupling each instruction that independently updates the stack pointer along with another register or memory location l (e.g., push, pop, leave, etc.). For such instructions, WiPEr uses a machine-code synthesizer [90] (a tool that synthesizes an instruction sequence from a semantic specification) to synthesize an equivalent instruction sequence that updates ESP in one instruction and l in a different instruction. The effect of decoupling on BTA is illustrated by the SDG snippet on the right in Fig. 6.6. □

The rewritten binary sum′ is shown in Fig. 6.7. In sum′, the new slicing criterion includes instructions 2′ and 6′. The new forward slice includes only the instructions highlighted in

Fig. 6.7 in light gray, which are classified dynamic. The remaining instructions are classified static.[3]

The inputs to the specialization phase of WiPEr are: (i) the interprocedural CFG of the binary, (ii) the binding-time annotations, and (iii) an input partial static-store. The output is the residual program. Recall from §2.1.2 that an IA-32 *Store* is a triple that consists of a register map, a flag map, and a memory map. The input partial static-store ρ_0 is

$$\rho_0 \equiv \langle [\boldsymbol{ESP} \mapsto \boldsymbol{1000}][\boldsymbol{EBX} \mapsto \boldsymbol{1}], [\,], [\,] \rangle.$$

In the rest of the chapter, when we use the term "store," we are referring to a partial static-store. The input store has the value ***1*** for v, and ***1000*** for the stack pointer. Let us specialize sum′ with respect to ρ_0.

Non-parameterized operands in instruction ASTs. Recall from §6.1.1 that $\mathbf{PE}_S$ substitutes values for operands while specializing ASTs of dynamic expressions and statements. However, WiPEr cannot use this simple approach for specializing dynamic instructions. For an instruction that has multiple input operands, the instruction AST may not always be parameterized by all operands. For example, the instruction "`adc eax,ebx`" adds the values in registers EAX and EBX, and the value of the carry flag CF. The flag operand is "baked into" the semantics, and is not an explicit syntactic operand. Another example is the instruction "`push eax`," which has the stack-pointer operand (ESP) baked into the semantics.

To handle this issue, WiPEr specializes a representation of the *semantics* of an instruction —instead of its syntax—with respect to a store. WiPEr uses a QFBV formula to represent an instruction's semantics. □

Now let us use the following approach to specialization:

1. Static instructions: Evaluate.

2. Dynamic instructions: Specialize the instruction's QFBV formula with respect to the pre-store, and residuate code. WiPEr uses a machine-code synthesizer to synthesize an instruction-sequence that is equivalent to the specialized QFBV formula. The synthesized instruction-sequence constitutes the residual code.

[3]The division produced by WiPEr's BTA is pointwise, and monovariant for procedures (i.e., an instruction in a procedure is classified as static or dynamic for all calling contexts).

Let us specialize sum′ with respect to ρ_0 using this simple approach. The specializer evaluates instruction 1′ in sum′ because it is static. The post-store is ρ_1.

$$\rho_1 \equiv \langle [\boldsymbol{ESP} \mapsto \boldsymbol{996}][\boldsymbol{EBX} \mapsto \boldsymbol{1}],\ [\,],\ [\,]\rangle. \tag{6.2}$$

The QFBV formula for instruction 2′ is

$$\langle\!\langle 2' \rangle\!\rangle \equiv Mem' = Mem[ESP \mapsto EAX].$$

Because 2′ is dynamic, the specializer specializes $\langle\!\langle 2' \rangle\!\rangle$ with respect to ρ_1 to obtain the specialized formula

$$Mem' = Mem[996 \mapsto EAX],$$

and uses the synthesizer to synthesize the instruction "`mov [996],eax`." After evaluating static instructions 3′ and 4′, the resulting store is

$$\rho_4 \equiv \langle [\boldsymbol{ESP} \mapsto \boldsymbol{992}][\boldsymbol{EBX} \mapsto \boldsymbol{1}],\ [\,],\ [\boldsymbol{992} \mapsto \boldsymbol{1}]\rangle. \tag{6.3}$$

The residual program produced by this simple approach is given below.

```
mov [996],eax   mov ecx,[988]
mov [988],ecx   mov ebx,[964+ecx*4]
mov eax,[996]   add eax,ebx
```

Lifting. The approach described above produces code that can access uninitialized locations. For example, one can see that the instruction "`mov ebx,[964+ecx*4]`" in the residual program dereferences a dynamic pointer to access an element in the static array `b`. However, the static array is not initialized by the residual code. This issue arises because our simple specialization approach did not residuate code for static instructions that produce values that *might* be consumed by a downstream dynamic instruction (a.k.a., a dynamic context).

To resolve this issue, WiPEr conservatively lifts static predecessors of a dynamic instruction. In Fig. 6.7, the boxed instructions are the lifted instructions. After lifted instructions are identified, one can use the following approach to specialization:

Approach $\mathbf{PE}_1$

- Static instructions: Evaluate.
- Lifted instructions: Specialize the instruction's QFBV formula with respect to the pre-store, and residuate code. Then update the store by evaluating the instruction.

```
1₁: mov esp, 996
2₁: mov [esp],eax
5₁: mov esp,988
6₁: mov [esp],ecx
9₁: mov esp, 964
```

```
13¹₁: mov [980], 1
13²₁: mov [976], 1
13³₁: mov [972], 1
13⁴₁: mov [968], 1
13⁵₁: mov [964], 1
```

```
16₁: mov eax,[esp+32]
17₁: mov ecx,[esp+24]
18₁: mov ebx,[esp+ecx*4]
19₁: add eax,ebx
```

Figure 6.8: Residual program sum_1.

- Dynamic instructions: Emit. Because all static locations that might be used by a dynamic instruction d have been initialized by upstream residual lifted instructions, there is no static data that needs to be infused into d via specialization, and the unmodified d can be emitted. □

Let us specialize sum' with respect to ρ_0 using $\mathbf{PE}_1$. Because $1'$ is lifted, the specializer residuates code for it. The QFBV formula for $1'$ is $\langle\langle 1' \rangle\rangle \equiv ESP' = ESP - 4$. The specializer specializes $\langle\langle 1' \rangle\rangle$ with respect to ρ_0 to obtain the specialized formula $ESP' = 996$, and uses the synthesizer to synthesize instruction 1_1 in sum_1 (shown in Fig. 6.8). The specializer also evaluates $1'$ to obtain ρ_1 (Eqn. (6.2)). Because $2'$ is dynamic, the specializer emits it as 2_1. Because $3'$ is static, the specializer evaluates the instruction without residuating code. The new residual program sum_1 is shown in Fig. 6.8. In sum_1, one can see that all the static values required by emitted dynamic instructions (2_1, 6_1, etc.) have been set up by residual lifted instructions (1_1, 5_1, etc.).

Residual specialization-time addresses. In sum_1, one can see that the specialization-time values of the stack pointer have been residuated as constants. Although sum_1 is a correct residual program for sum' partially evaluated with respect to ρ_0, the specialization-time stack layout is hard-wired into the residual code. (The stack begins at address 1000, and grows toward lower addresses.) As a result, the residual code might crash if a different address is used as the base of the runtime stack.

One way to prevent the specializer from emitting stack-pointer values is to make the stack pointer dynamic by including instruction $1'$ in the slicing criterion during BTA. However, that would make all instructions in sum' dynamic.

To resolve this issue, WiPEr uses a *symbolic* store during specialization, instead of a *concrete* store. In the initial symbolic store, specialization-time addresses (e.g., the initial value of ESP) are represented using symbolic constants. Static non-address values are represented using integer and Boolean constants. WiPEr uses $\mathbf{PE}_2$ for specialization.

```
0₂: mov esi,esp
1₂: lea esp,[esi-4]
2₂: mov [esp],eax
5₂: lea esp,[esi-12]
6₂: mov [esp],ecx
9¹₂: lea esp,[esi-16]
9²₂: sub esp,20
```

```
13¹₂: mov [esi-20],1
13²₂: mov [esi-24],1
13³₂: mov [esi-28],1
13⁴₂: mov [esi-32],1
13⁵₂: mov [esi-36],1
```

```
16₂: mov eax,[esp+32]
17₂: mov ecx,[esp+24]
18₂: mov ebx,[esp+ecx*4]
19₂: add eax,ebx
```

Figure 6.9: Residual program sum_2.

Approach $\mathbf{PE}_2$

- Static instructions: Symbolically evaluate. Although specialization-time addresses are symbolic, WiPEr can still access and update static values in memory.
- Lifted instructions: Specialize the instruction's QFBV formula with respect to the symbolic pre-store, and residuate code. Then update the symbolic store by symbolically executing the instruction.
- Dynamic instructions: Emit. □

Let us illustrate $\mathbf{PE}_2$ on sum'. The new initial store is

$$\rho_0^{sym} \equiv \langle [ESP \mapsto c][EBX \mapsto 1],\ [\,],\ [\,] \rangle.$$

Note that the values in the store are not in boldface to indicate that they are logical terms, and not semantic values. The value of the stack pointer ESP is the symbolic constant c to indicate that we are not restricting ESP to a concrete value known at specialization-time.

Before starting specialization, WiPEr residuates instruction 0_2 in sum_2 (shown in Fig. 6.9) to save the initial value of the stack pointer ESP in a dedicated location. (In this example, WiPEr uses the ESI register because ESI is not used in sum'.) This location will be used whenever downstream residual instructions need the initial value of ESP. To residuate code for the lifted instruction $1'$, WiPEr specializes $\langle\langle 1' \rangle\rangle$ with respect to ρ_0^{sym}. The specialized formula is $ESP' = c - 4$. Because at run-time the value of c will be available in ESI, WiPEr replaces c with ESI to produce $ESP' = ESI - 4$. WiPEr uses the synthesizer to synthesize instruction 1_2 for the specialized formula. WiPEr also symbolically evaluates $1'$ to produce ρ_1^{sym}.

$$\rho_1^{sym} \equiv \langle [ESP \mapsto c - 4][EBX \mapsto 1],\ [\,],\ [\,] \rangle.$$

$2'$ is dynamic, and WiPEr emits it as 2_2. After symbolically evaluating static instructions $3'$ and $4'$, we obtain the store

$$\rho_4^{sym} \equiv \langle [ESP \mapsto c - 8][EBX \mapsto 1],\ [\,],\ [c - 8 \mapsto 1] \rangle.$$

The final residual program sum_2 produced by WiPEr is shown in Fig. 6.9. One can see that WiPEr arranges for the initial stack-pointer value to be retrieved from ESI (1_2, 5_2, 9_2, 13_2^1, etc.) instead of residuating the stack-pointer value at residual lifted instructions (cf. 1_1, 5_1, 9_1, 13_1^1, etc. in sum_1); consequently, sum_2 will not crash when the stack base is initialized to different addresses. In contrast, static non-address quantities are residuated at lifted instructions (e.g., the value 1 in instructions 13_2^1 through 13_2^5).

Why do $\mathbf{PE}_1$ and $\mathbf{PE}_2$ work?

To formalize the effect of $\mathbf{PE}_1$ and $\mathbf{PE}_2$, we introduce the notion of an *environment*, which makes explicit the starting address of the stack and heap blocks. An environment maps a symbolic constant denoting the starting address of the stack or a heap block to a concrete address (or "location"). A machine-code state σ consists of an environment η, and a store ρ, which maps locations to values. For example, ρ_4 in Eqn. (6.3) is actually part of a state σ_4

$$\sigma_4 \equiv \langle \eta_4, \rho_4 \rangle \equiv \langle \mathit{Stack} \mapsto \boldsymbol{1000}, \langle [\boldsymbol{ESP} \mapsto \boldsymbol{992}][\boldsymbol{EBX} \mapsto \boldsymbol{1}],\ [\,],\ [\boldsymbol{992} \mapsto \boldsymbol{1}] \rangle \rangle.$$

η_4 maps the start of the stack, denoted by the symbolic constant *Stack*, to the address ***1000***.

Suppose that η and ρ can be partitioned into η_S and η_D, and ρ_S and ρ_D, respectively. The effect of $\mathbf{PE}_1$ on each execution trace of the binary can be expressed by the following reparenthesization:

$$l : (\eta_S, \rho_S, \eta_D, \rho_D) \longrightarrow (l, \eta_S, \rho_S) : (\eta_D, \rho_D) \tag{6.4}$$

Operationally, the partial evaluator creates a new residual instruction for every unique (l, η_S, ρ_S) triple it observes at specialization time. Eqn. (6.4) shows how each element $(l, \eta_S, \rho_S) : (\eta_D, \rho_D)$ in an execution trace of a binary specialized using $\mathbf{PE}_1$ can be mapped back to the corresponding element $l : (\eta_S, \rho_S, \eta_D, \rho_D)$ in the trace of the original binary. Consequently, the specialization-time memory layout (η_S) is hard-wired into the residual code. If the run-time memory layout is different, the residual code produced by $\mathbf{PE}_1$ might crash.

To describe the symbolic techniques used in $\mathbf{PE}_2$, we introduce a static symbolic store

Input: Binary B
Output: Rewritten binary B′
1: **for** each i ∈ B **do**
2: **if** `UpdatesSPAndLoc`(i) **then**
3: I ← `Synth`(⟪ i ⟫)
4: B′ ← `Replace`(B, i, I)
5: **end if**
6: **end for**
7: **return** B′

Algorithm 6.10: Instruction Decoupling (Pre-processing step)

ρ_S^{sym}, which maps symbolic expressions to symbolic expressions, where each such symbolic expression is parameterized on the symbolic constants found in η_S. (Note that the domain of ρ_S^{sym} also contains registers and flags.) ρ_S^{sym} mirrors the static concrete store ρ_S, i.e., $[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$, where the left-hand side denotes the simplification of each symbolic expression in ρ_S^{sym} with respect to the values obtained from η_S. The principal effect of $\mathbf{PE}_2$ on each execution trace of the binary can be expressed by the following reparenthesization:

$$l : (\eta_S, \rho_S, \eta_D, \rho_D) \longrightarrow (l, \rho_S^{sym}) : (\eta_S, \eta_D, \rho_D) \tag{6.5}$$

Eqn. (6.5) shows how each element $(l, \rho_S^{sym}) : (\eta_S, \eta_D, \rho_D)$ in an execution trace of a binary specialized using $\mathbf{PE}_2$ can be mapped back to the corresponding element $l : (\eta_S, \rho_S, \eta_D, \rho_D)$ in the trace of the original binary. In effect, the residual code is independent of η_S. (See also §6.3.4.)

6.3 Algorithm

In this section, we describe the algorithms used by WiPEr. First, we present the algorithm for a pre-processing step used prior to partial evaluation. Second, we present WiPEr's intraprocedural partial-evaluation algorithm. Third, we present extensions to handle multiple procedures. Finally, we present correctness guarantees and threats to the validity of our algorithms.

6.3.1 Pre-processing: Decoupling Instructions

Prior to performing BTA, WiPEr rewrites the input binary by decoupling each instruction *i* that updates the stack-pointer register ESP along with another location *l*. WiPEr uses the machine-code synthesizer McSynth [90] for decoupling. McSynth uses a divide-and-conquer strategy that splits *i*'s QFBV formula φ into two independent sub-formulas φ_1 and φ_2, such

```
Input: SDG, I_D
Output: BTAMap
 1: slice ← ForwardSlice(SDG, I_D)
 2: for each I ∈ slice do
 3:   BTAMap ← BTAMap[I ↦ Dynamic]
 4:   for each static predecessor S of I do
 5:     BTAMap ← BTAMap[S ↦ Lifted]
 6:   end for
 7: end for
 8: for each I ∉ Dom(BTAMap) do
 9:   if I is a call to malloc or an actual argument of a call to malloc then
10:     BTAMap ← BTAMap[I ↦ Lifted]
11:   else
12:     BTAMap ← BTAMap[I ↦ Static]
13:   end if
14: end for
15: return BTAMap
```

Algorithm 6.11: BTA algorithm in WiPEr

that the updates to ESP and l are split between φ_1 and φ_2. Then, McSynth synthesizes code for φ_1 and φ_2, concatenates the results, and returns the resulting instruction sequence I. ESP and l are updated in different instructions in I. The rewriting of the input binary via instruction decoupling is shown as Alg. 6.10. In the algorithm, `UpdatesSPAndLoc` returns true if an instruction updates the stack pointer along with another register or memory location; Recall from §2.2 that $\langle\langle\cdot\rangle\rangle$ encodes an instruction (or instruction sequence) as a QFBV formula; `Synth` invokes McSynth to synthesize an instruction sequence; `Replace` replaces an instruction in a binary with an instruction sequence.

6.3.2 Intraprocedural Partial Evaluation

In this section, we present WiPEr's intraprocedural partial-evaluation algorithm. First, we present the BTA algorithm; then, we present the specialization algorithm.

BTA

The slicing-based BTA algorithm used in WiPEr is shown as Alg. 6.11. In the algorithm, I_D is the set of instructions that initialize dynamic inputs; `ForwardSlice` computes a forward slice with respect to a slicing criterion; BTAMap is a data structure that maps each instruction to its binding time. If a static instruction is control dependent on a dynamic branch, the specialization algorithm might not terminate [48, Chap. 14]. To avoid the possibility of non-termination, `ForwardSlice` follows both data and control dependences during slicing. Lines 4–6 in Alg. 6.11 show the computation of lifted instructions during BTA. WiPEr lifts all static calls to `malloc` (lines 9–11). This step is performed so that heap blocks that

```
Input: CFG, ρ, BTAMap
Output: ρ′
 1: CFG′ ← NewCFG()
 2: bb ← CFG.FirstBB()
 3: worklist ← ⟨bb, ρ⟩
 4: processed ← ∅
 5: ρ′ ← ρ
 6: while worklist ≠ ∅ do
 7:   ⟨bb, ρ⟩ ← RemoveItem(worklist)
 8:   if ⟨bb, ρ⟩ ∈ processed then
 9:     continue
10:   end if
11:   processed ← processed ∪ {⟨bb, ρ⟩}
12:   for each instruction i ∈ bb do
13:     ⟨i′, ρ⟩ ← specialize(i, ρ, BTAMap)
14:     if i′ ≠ ϵ then
15:       CFG′.Emit(i′)
16:     end if
17:   end for
18:   if CFG.FinalBB(bb) then
19:     ρ′ ← ρ
20:   end if
21:   for each successor s of bb do
22:     worklist ← worklist ∪ {⟨s, ρ⟩}
23:   end for
24: end while
25: return ρ′
```

Algorithm 6.12: WiPEr's specialization loop - IntraPE

are allocated at specialization-time also get allocated at run-time. (See ***PE*$_2$ *specializer*** in §6.3.2.)

Specialization

This section presents the specialization algorithm used in WiPEr. We first present the skeleton of WiPEr's specialization loop. We then present two versions of the core specialization function specialize, which gets called in the specialization loop. While the first version produces correct residual code, the residual code cannot be executed with the stack and heap blocks at different positions at run-time than at specialization-time. The second version does not have this limitation, and is used in WiPEr.

Specialization loop. The skeleton of WiPEr's specialization loop is similar to that of a standard partial evaluator [48, p. 87], and is given as Alg. 6.12. specialize is the core specialization function (line 13 of Alg. 6.12), which specializes an instruction with respect to a pre-store to obtain the residual instruction and a post-store. In the algorithm, ϵ denotes

Input: I, ρ, BTAMap
Output: $\langle I', \rho' \rangle$
1: **if** BTAMap[I] = Dynamic **then**
2: **return** $\langle I, \rho \rangle$
3: **else if** BTAMap[I] = Lifted **then**
4: **return** $\langle \mathsf{reduce}_1(I, \rho), \mathsf{eval}_1(I, \rho) \rangle$
5: **else**
6: **return** $\langle \epsilon, \mathsf{eval}_1(I, \rho) \rangle$
7: **end if**

Algorithm 6.13: $\mathsf{specialize}_1$

the empty sequence of instructions; `NewCFG` creates a new CFG; `FirstBB` returns the first basic-block in a CFG; `FinalBB` returns true if the argument is the final basic-block in its parent CFG; `RemoveItem` removes an item from the worklist; `Emit` appends an instruction to a CFG. (We assume that `Emit` creates and connects nodes in the CFG.)

Recall that the register EIP is the IA-32 program counter. Alg. 6.12 follows CFG edges, rather than the value of EIP, because we have no way to give consistent EIP values to new instructions created via decoupling. Alg. 6.12 does not even track EIP in stores, and consequently, it can propagate the same store to the two successors of a branch instruction (lines 21–23).

$\mathbf{PE}_1$ Specializer. $\mathsf{specialize}_1$ is a specialization function that implements $\mathbf{PE}_1$. $\mathsf{specialize}_1$ has two constituent functions, eval_1 and reduce_1. eval_1 evaluates instructions, and reduce_1 residuates code. The algorithm for $\mathsf{specialize}_1$ is given as Alg. 6.13.

eval_1. eval_1 can be implemented by writing an interpreter for IA-32 instructions using the concrete operational-semantics of IA-32. eval_1 evaluates an instruction with respect to a pre-store, and returns the post-store.

$$\mathsf{eval}_1 : \texttt{Instruction} \to \mathit{Store} \to \mathit{Store} \qquad \mathsf{eval}_1(I, \rho) = \mathcal{I}\,[\![I]\!]\,\rho$$

reduce_1. reduce_1 can be defined as follows:

$$\mathsf{reduce}_1 : \texttt{Instruction} \to \mathit{Store} \to \texttt{Instruction-sequence}$$
$$\mathsf{reduce}_1(I,\ \rho) = \texttt{Synth}(\texttt{Subst}_0(\langle\!\langle I \rangle\!\rangle, \rho))$$

reduce_1 converts an instruction into a QFBV formula, specializes the formula with respect to the pre-store (using $\texttt{Subst}_0$), and uses a machine-code synthesizer to synthesize an instruction sequence that is equivalent to the specialized formula. The synthesized

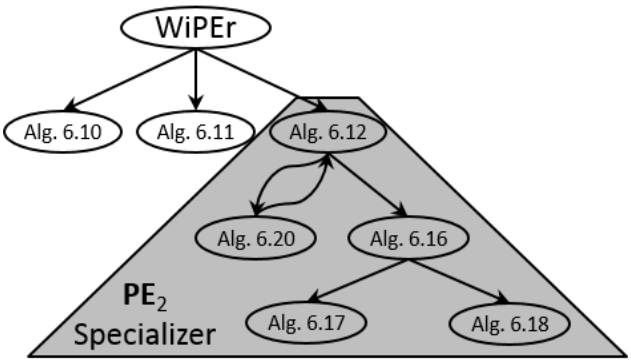


Figure 6.14: Call graph depicting the organization of WiPEr.

instruction-sequence constitutes the residual code. Recall from §2.2 that vocabulary-0 terms and formulas in a QFBV formula represent pre-store locations and predicates, respectively. $\texttt{Subst}_0$ replaces vocabulary-0 terms and formulas in I's QFBV formula with static values from ρ. Because lifted instructions are originally static, they access only static locations. Consequently, $\texttt{Subst}_0$ replaces *all* vocabulary-0 terms and formulas in I's QFBV formula with static values from the store.

For example, suppose that I is the instruction "`add [esp],eax`." I's QFBV formula φ is given below. (For brevity, we only show one of the flags updated by `add`.)

$$\varphi \equiv Mem' = Mem[ESP \mapsto Mem(ESP) + EAX] \wedge SF' = (Mem(ESP) + EAX < 0)$$

Suppose that ρ is $\rho \equiv \langle [\boldsymbol{ESP} \mapsto \boldsymbol{1000}][\boldsymbol{EAX} \mapsto \boldsymbol{2}], [\,], [\boldsymbol{1000} \mapsto \boldsymbol{1}] \rangle$. $\texttt{Subst}_0$ produces the following formula:

$$Mem' = Mem[1000 \mapsto 3] \wedge SF' = False,$$

and $\mathsf{reduce}_1(\mathrm{I}, \rho)$ produces the residual instruction-sequence "`mov [1000], 3; add [1000], 0`." (The second instruction in the sequence is necessary to set the SF flag.)

$\mathbf{PE}_2$ specializer. As discussed in §6.2, Alg. 6.13 specializes the binary with respect to static addresses as well as static values (e.g., sum_1 in Fig. 6.8). In this section, we present the specialization algorithm that WiPEr uses to residuate code that is independent of the specialization-time memory layout. In particular, we illustrate how the algorithm residuates code that allows the stack and heap blocks to be at different positions at run-time than at specialization-time.

WiPEr's call-graph is represented in Fig. 6.14. WiPEr's specializer is highlighted in Fig. 6.14 in light gray. WiPEr uses $\mathsf{specialize}_2$ (Alg. 6.16) as the specialization function in

```
1: mov [esp],16 ;L
2: call malloc ;L
3: mov [eax],1 ;S
4: mov [eax+4],2 ;L
5: add ebx,[eax+4] ;D
```

```
1_R: mov [esp],16
2^1_R: call malloc
2^2_R: mov edi,eax
4_R: mov [edi+4],2
5_R: add ebx,[eax+4]
```

Figure 6.15: Code snippet, and residual program to illustrate specialize$_2$ on heap blocks.

Alg. 6.12. specialize$_2$ implements **PE**$_2$. The remaining components of the specializer (Algs. 6.17, 6.18, and 6.20) will be described later in upcoming paragraphs and sections.

specialize$_2$ uses symbolic addresses and a symbolic store to access and update memory at specialization time. To represent symbolic addresses, we use symbolic constants in L[IA-32] (m, n, etc.). A different constant is used to represent the starting address of the stack and each heap block allocated at specialization-time. specialize$_2$ uses a symbolic store $Store^{sym}$ instead of $Store$.

$$\rho^{sym} \in Store^{sym} = RegMap^{sym} \times FlagMap^{sym} \times MemMap^{sym}$$

$$RegMap^{sym} : Int32Id \rightarrow Term \qquad FlagMap^{sym} : BoolId \rightarrow Formula \qquad MemMap^{sym} : Term \rightarrow Term$$

The residual code generated by specialize$_2$ will use specific run-time addresses to access and update memory using an extra level of indirection. To achieve this effect, specialize$_2$ uses a memory-layout map $\mu \in MemLayout$. A memory-layout map μ maps the symbolic starting-address m of the stack or a heap block to a location l (register or global address) that holds the run-time counterpart of m.

$$\mu \in MemLayout : Int32Id \rightarrow (Register \cup INT)$$

We will illustrate specialize$_2$, and its constituent functions eval$_2$ and reduce$_2$, using the code snippet shown in Fig. 6.15, which allocates and uses a heap block. (We describe how specialize$_2$ handles the stack and stack locations at the end of this sub-section.) The snippet has a static call to `malloc` at instruction 2, and static values are written to the allocated heap-block at instructions 3 and 4. The binding-time annotations for the instructions are shown as comments on the left-hand column of Fig. 6.15. Because WiPEr lifts all static calls to `malloc`, instructions 1 and 2 are lifted.

The algorithms for specialize$_2$, eval$_2$, and reduce$_2$ are given as Algs. 6.16, 6.17, and 6.18, respectively. In Alg. 6.16, we overload ϵ to denote "no symbolic constant" (lines 4 and 13) and "no location" (line 5), respectively. In our example, because instruction 2 allocates memory (checked by `Alloc`), specialize$_2$ adds $[m \mapsto EDI]$ to μ, where m is a fresh symbolic constant (obtained using `NewConst`) that is used to represent the specialization-

Input: I, ρ, BTAMap, μ
Output: $\langle I', \rho', \mu' \rangle$
1: **if** BTAMap[I] = Dynamic **then**
2: **return** $\langle I, \rho, \mu \rangle$
3: **else if** BTAMap[I] = Lifted **then**
4: $m \leftarrow \epsilon$
5: $l \leftarrow \epsilon$
6: **if** `Alloc(I)` **then**
7: $m \leftarrow$ `NewConst()`
8: $l \leftarrow$ `NewLoc()`
9: $\mu' \leftarrow \mu[m \mapsto l]$
10: **end if**
11: **return** $\langle \text{reduce}_2(I, \rho, \mu, l), \text{eval}_2(I, \rho, m), \mu' \rangle$
12: **else**
13: **return** $\langle \epsilon, \text{eval}_2(I, \rho, \epsilon), \mu \rangle$
14: **end if**

Algorithm 6.16: specialize_2

Input: I, ρ, m
Output: ρ'
1: $\rho' \leftarrow \mathcal{J}^{sym}\ [\![I]\!]\ (\rho[\mathit{NextBlock} \mapsto m])$
2: **return** ρ'

Algorithm 6.17: eval_2

Input: I, ρ, μ, l
Output: I'
1: $I' \leftarrow$ `Synth(Subst(Subst`$_0(\langle\!\langle\ I\ \rangle\!\rangle, \rho), \mu))$
2: $I'' \leftarrow \epsilon$
3: **if** `Alloc`(I) **then**
4: $I'' \leftarrow$ `EmitMove`(l, EAX)
5: **end if**
6: **return** $I'; I''$

Algorithm 6.18: reduce_2

time starting address of the heap block, and *EDI* is a location that is not used in the residual binary (obtained using `NewLoc`). This location can also be a global memory location that is otherwise unused in the residual binary. specialize_2 passes m to eval_2, which stores m in ρ under the key *NextBlock*. eval_2 symbolically executes the instruction using $\mathcal{J}^{sym}$, which reinterprets the semantics of an instruction I to symbolically execute I. If I allocates memory, $\mathcal{J}^{sym}$ uses the symbolic constant bound to *NextBlock* in ρ as the next address in the free list. Consequently, in ρ' returned by eval_2 for the call to `malloc`, EAX holds the symbolic constant m—i.e., m is the starting address of the heap block allocated at instruction 2.

specialize_2 then passes I, ρ, μ, and *EDI* to reduce_2. Because instruction 2 allocates memory, reduce_2 residuates two instructions. The first instruction (2^1_R) is the call to `malloc`; the

second instruction (2^2_R) saves the EAX register in EDI (emitted using `EmitMove`). After the residual code executes instruction 2^2_R, the run-time starting address of the heap block will be available in EDI.

specialize$_2$ symbolically executes the static instruction 3, producing the post-store $\langle [EAX \mapsto m], [], [m \mapsto 1] \rangle$. While residuating code for lifted instruction 4, reduce$_2$ uses $\texttt{Subst}_0$ to replace all vocabulary-0 terms and formulas in 4's QFBV formula with terms and formulas, respectively, from the symbolic store. This step produces the formula

$$Mem' = Mem[m + 4 \mapsto 2]. \tag{6.6}$$

reduce$_2$ then replaces each symbolic constant m in the resulting formula with $[\![\mu(m)]\!]_L$ using `Subst`. For our example, for Eqn. (6.6), `Subst` produces $Mem' = Mem[EDI + 4 \mapsto 2]$. Finally, reduce$_2$ uses the synthesizer to residuate instruction 4_R for the specialized formula. One can see that when the residual code executes, instructions 4_R and 5_R use the correct starting address of the heap block.

Handling the stack. Suppose that n is the symbolic constant used to denote the specialization time base of the stack, and l is a dedicated location that will hold the address of the base of the stack at run-time. Because the stack is typically allocated at the beginning of the program, WiPEr adds $[n \mapsto l]$ to μ before specialization begins, and emits "`mov l,esp`" as the first instruction in the residual program. Additionally, in the initial symbolic store, *ESP* is bound to n. In our running example from §6.2, WiPEr adds $[n \mapsto ESI]$ to μ. Consequently, reduce$_2$ uses *ESI* instead of n in the QFBV formulas of all downstream lifted instructions.

In summary, to residuate code that is independent of the specialization-time memory layout, specialize$_2$ uses symbolic addresses, and residuates code that obtains the corresponding run-time addresses from designated locations, using one level of indirection. In effect, the binary is partially evaluated with respect to static values, but not specialization-time addresses.

PE$_2$-Safety. For **PE$_2$** to produce correct residual code for a binary B, B should be ***PE$_2$-safe***. A binary B, along with a specific division of instructions in B as static and dynamic, are **PE$_2$**-safe if they satisfy the following properties:

P1 B does not contain a static branch condition that depends on the starting address of the stack or heap blocks (e.g., `cmp esp,1000`).

P2 Every static instruction that accesses (updates) memory in block M only accesses (updates) memory at a static offset from the base of M, and within the bounds of M.

```
main:                                                    foo:
    :                          :                            push ebp
A1: mov [esp],eax; S       A2: mov [esp],eax; D             mov ebp, esp
C1: call foo               C2: call foo                 R: mov eax,[ebp-8]
R1: mov [ebp-4],eax        R2: mov [ebp-8],eax              pop ebp
    :                          :                            ret
```

Figure 6.19: Code snippet to illustrate issues related to interprocedural BTA.

P1 implies that the partial evaluator does not encounter a symbolic branch condition during specialization of B. P2 implies that every term in every store ρ^{sym} that arises during specialization of B can be simplified to have the form c, or $m + c$, where c is an integer constant, and m is a symbolic constant.

If a binary B violates P1 or P2, B might have different semantics for different layouts of the stack and heap blocks. For example, consider the code snippet

```
1:  mov [esp+1000],10 ;S
2:  mov [esp*2],20 ;S
3:  mov eax,[esp+1000] ;L
```

This snippet violates P2: after symbolically executing instruction 2, we obtain the store ρ_2^{sym}

$$\rho_2^{sym} \equiv \langle [ESP \mapsto n],\ [\,],\ [n + 1000 \mapsto 10][n * 2 \mapsto 20] \rangle$$

In ρ_2^{sym}, the term $n * 2$ cannot be rewritten in the form $m + c$. Note that the value in the EAX register after (concretely) executing instruction 3 depends on the initial value of ESP. If we execute the snippet with the initial concrete store $\rho_0 \equiv \langle [\boldsymbol{ESP} \mapsto \boldsymbol{1000}],\ [\,],\ [\,] \rangle$, the value in EAX after executing instruction 3 will be ***20***. If we execute the code snippet with a different value for ESP in ρ_0, the value in EAX will be ***10***. Consequently, the snippet loads different values into EAX for different stack layouts.

Binaries of memory-safe programs produced by a standard compiler are usually $\mathbf{PE}_2$-safe. Apart from P1 and P2, we will also require that allocated memory chunks do not overlap in the address space used at run-time. Allocated memory chunks also include the chunk in which the residual code resides. (See §6.3.4.)

6.3.3 Interprocedural Partial Evaluation

In this section, we present the extensions to the algorithm to handle binaries with multiple procedures.

BTA

For interprocedural partial-evaluation, WiPEr uses a monovariant division, i.e., the classification of each instruction as static/dynamic holds for *all* calling contexts. This approach to

BTA introduces the two issues discussed below, which will be illustrated using the code snippet shown in Fig. 6.19. Procedure `main` contains two calls to `foo`. The actual parameter of the first call is static, and that of the second call is dynamic. `foo` merely returns the formal parameter.

One possible direction for future work is to investigate polyvariant BTA via *specialization slicing* [12]. For every call to a procedure P with a different combination of static and dynamic actual parameters, specialization slicing can be used to produce a different binding-time annotation for P's instructions. This technique could increase the number of static instructions during specialization, leading to residual code that is more specialized than the code currently residuated by WiPEr.

Parameter mismatches. A forward 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 [19, 46].

For the moment, assume that the forward slice used by BTA is context-sensitive. In the code snippet shown in Fig. 6.19, suppose that the slice includes instructions `A2`, `C2`, `R2`, and the entire body of `foo`. Note that there is a mismatch between the actual parameter `A1`, and `foo`'s formal parameter. (The latter is in the slice, but the former is not.) WiPEr lifts instructions A1 and C1 because they are interprocedural predecessors of `foo`'s instructions; this step ensures that there will be no parameter mismatches in the residual code.

Return mismatches. If the forward slice used during BTA is context-sensitive, there is also a return mismatch between the return value of `foo`, and the use of the return value at instruction `R1`. The former is in the slice, but the latter is not, which violates congruence. (At instruction `R1`, the specializer expects to find a return value for `foo` in the static store, but there is no value.) To prevent return mismatches, WiPEr uses context-insensitive slicing, which causes instruction `R1` to be included in the slice.

In summary, for interprocedural BTA, WiPEr (i) lifts interprocedural static-predecessors of dynamic instructions, and (ii) uses context-insensitive forward slicing. In addition, to ensure that partial evaluation always terminates, WiPEr conservatively classifies all instructions in a recursive procedure as dynamic. (This point is discussed further below.)

Specialization

Input: CFG, ρ, BTAMap, specializedCFGs
Output: ρ'
1: **if** ⟨CFG, ρ⟩ ∈ specializedCFGs **then**
2: **return** ⟨specializedCFGs[⟨CFG, ρ⟩], specializedCFGs⟩
3: **end if**
4: **if** `Recursive`(CFG) **then**
5: specializedCFGs[⟨CFG, ρ⟩] = ρ // See explanation in the text
6: **end if**
7: ⟨ρ', specializedCFGs⟩ ← IntraPE(CFG, ρ, BTAMap, specializedCFGs)
8: specializedCFGs[⟨CFG, ρ⟩] = ρ'
9: **return** ⟨ρ', specializedCFGs⟩

Algorithm 6.20: specializeFn

To perform specialization for interprocedural partial-evaluation, the following lines are added after line [16] of Alg. 6.12. `IsCall` returns true if the argument is a call instruction; `Callee` returns the callee CFG for a call instruction.

if `IsCall`(i) **then**
⟨ρ, specializedCFGs⟩ ← specializeFn(`Callee`(i), ρ, BTAMap, specializedCFGs)
endif

In addition, Alg. 6.12 takes an additional argument, specializedCFGs, which is a map whose entries are of the form ⟨CFG, ρ⟩ ↦ ρ', where ρ is a partial static pre-store and ρ' is a partial static post-store. Alg. 6.12 also returns specializedCFGs as an additional return value. Procedure specializeFn is given as Alg. 6.20. Alg. 6.20 implements function caching—if CFG has already been specialized with respect to ρ, Alg. 6.20 returns ρ' (lines 1–3).

To ensure that partial evaluation always terminates, WiPEr does not attempt to specialize recursive functions. Recall that all instructions in a recursive procedure are classified as dynamic (and the binding-time classification computed by BTA is congruent with respect to that classification). Thus, for a recursive procedure P, the correct values for all variables that are classified static at the return from P are found in the input partial static pre-store ρ. Consequently, Alg. 6.20 can use ρ as the partial static post-store in the entry added to specializedCFGs before IntraPE is called (lines 4–6). This trick ensures that Alg. 6.20 cannot get into an infinite loop while specializing a recursive function.

If a function has not already been specialized, Alg. 6.20 calls Alg. 6.12 to specialize the CFG with respect to ρ, and adds the partial static post-store to the map (lines 7–8).

6.3.4 Correctness

In this section, we present two theorems concerning: (i) termination of WiPEr, and (ii) correctness of the residual code produced by WiPEr. The first theorem applies to binaries

with instructions from the entire IA-32 instruction set. Because it would be difficult to prove the correctness theorem for the entire IA-32 ISA, we prove correctness only for binaries containing instructions from a small instruction subset. (See App. A.)

Theorem 6.21. *Suppose that B is a* ***PE*$_2$*-safe*** *binary, and* ρ_S^{sym} *is a partial static symbolic-store. Then* WiPEr(B, ρ_S^{sym}) *terminates.*

Proof. WiPEr treats recursive functions conservatively, so WiPEr cannot enter into an infinite loop while attempting to specialize a recursive function. WiPEr unrolls static loops finitely many times depending upon the static loop bound. Moreover, the forward slice performed during BTA follows both data and control dependences, which rules out the possibility of there being a static instruction that is control dependent on a dynamic branch. Consequently, WiPEr cannot create infinitely many specialized versions of the same basic block. Thus WiPEr(B, ρ_S^{sym}) is guaranteed to terminate. □

In a standard semantics that formalizes the behavior of an IA-32 processor, an evaluation function would take an IA-32 *Store* as an input, and produce an IA-32 *Store* as an output. In §6.2, we introduced the notion of a state σ consisting of an environment η and an IA-32 store ρ. In this section, we use a non-standard semantics that uses an environment along with an IA-32 store. The environment is really a ghost variable that makes explicit the starting address of the stack and heap blocks, while the concrete part of the state is the store.

Based on the congruent division established by BTA, a state can be partitioned into η_S, ρ_S, η_D, and ρ_D. Thus the type for the evaluation function $[\![\cdot]\!]$ (the function that evaluates an instruction or a sequence of instructions with respect to a state, and returns the post-state) can be written as

$$[\![\cdot]\!] : \text{Instruction} \times \mathit{Env}_S \times \mathit{Store}_S \times \mathit{Env}_D \times \mathit{Store}_D \to \mathit{Env}_S \times \mathit{Store}_S \times \mathit{Env}_D \times \mathit{Store}_D.$$

If we execute an instruction in a partially evaluated binary, ρ_S is never accessed or updated. Thus we will overload $[\![\cdot]\!]$ to also denote the meaning function for instructions in a partially evaluated binary:

$$[\![\cdot]\!] : \text{Instruction} \times \mathit{Env}_S \times \mathit{Env}_D \times \mathit{Store}_D \to \mathit{Env}_S \times \mathit{Env}_D \times \mathit{Store}_D$$

It will always be clear from context which meaning of $[\![\cdot]\!]$ is intended.

We assume that the memory map for a program that has been partially evaluated is equipped with a special area in which the starting addresses of the stack and heap blocks

allocated by lifted instructions are stored. Thus, strictly speaking, the memory maps for a binary B and some residual binary B' of B will be different. However, the only differences are in the special area, and we denote equality of two stores ρ and $\overline{\rho}$ *modulo the special area* by $\rho \approx \overline{\rho}$.

For an input binary B that is $\mathbf{PE}_2$-safe, and an IA-32 state σ that can be partitioned into η_S, ρ_S, η_D, and ρ_D based on a division of inputs to B, WiPEr produces a specialized binary that, when executed, produces an answer that matches the answer that would be produced by B (modulo the special area). The precise statement of the property is given in the following theorem:

Theorem 6.22. *Suppose that B is $\mathbf{PE}_2$-safe. Also suppose that η_S is an environment such that there are no overlaps among (a) any of the chunks in Dom(η_S), (b) the chunk in which the residual code produced by WiPEr is loaded, and (c) any of the chunks allocated during the execution of the residual code. For all $\sigma = (\eta_S, \rho_S, \eta_D, \rho_D)$ such that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D)$ terminates, suppose that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta'_S, \rho'_S, \eta'_D, \rho'_D)$.*
If ρ_S^{sym} is a symbolic store such that $[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$, then $[\![WiPEr(B, \rho_S^{sym})]\!](\eta_S, \eta_D, \rho_D) = (\eta'_S, \eta'_D, \overline{\rho}'_D)$, where $\overline{\rho}'_D \approx \rho'_D$.

Proof. The proof for Thm. 6.22 is given in App. A. □

Thm. 6.22 does not include ρ_S while comparing states because the residual code produced by WiPEr(B, ρ_S^{sym}) does not contain static instructions, and consequently never accesses or updates the static store ρ_S.

6.3.5 Threats to Validity

There are two threats to the validity of our algorithms.

1. If the input binary is not $\mathbf{PE}_2$-safe, the residual code might violate Thm. 6.22.

2. The accuracy of BTA depends on the accuracy of the SDG for the binary. If the methods used to construct the SDG are overly conservative, BTA can classify instructions as "dynamic" that do not depend on dynamic inputs. If the methods used to construct the SDG are under-approximative, BTA can classify instructions as "static" that actually do depend on dynamic inputs. In our experiments, we observed the former behavior. The SDG that WiPEr uses is built using "variable-like abstractions" recovered by machine-code variable-recovery analyses [14]. Each recovered variable-like abstraction could be imprecise—it is often an agglomeration of some source-code variables. Consequently, the SDG recovered from the program binary could be more

```
foo:                                    main:
  push ebp                                push ebp
  mov ebp,esp                             mov ebp,esp
  and esp,0xFFFFFF00                      and esp,0xFFFFFF00
  sub esp,170                             sub esp,150
        :                                       :
```

Figure 6.23: Code snippet to illustrate variable alignment.

imprecise than an SDG that is obtained from the program's source code. As a result, WiPEr's BTA classifies more instructions dynamic than an analogous BTA that starts from source code.

6.4 Implementation

WiPEr uses CodeSurfer/x86 [15] to obtain the SDG, CFG, and call graph for the binary. WiPEr's BTA uses CodeSurfer/x86's forward-slicing operation. WiPEr's specializer uses CodeSurfer/x86's rewriting API to create the sections, CFGs, and instructions in the residual binary. WiPEr uses Transformer Specification Language (TSL) [53] to build its concrete and symbolic interpreters. The concrete operational semantics of the integer subset of IA-32 is written in TSL, and the semantics is interpreted for concrete evaluation, and reinterpreted for symbolic evaluation. WiPEr also uses TSL [54] to convert an instruction into a QFBV formula. WiPEr uses McSynth [91] to synthesize an IA-32 instruction sequence from a QFBV formula. McSynth sometimes requires a set of scratch registers to hold results of intermediate calculations in the residual code. WiPEr supplies dead registers to McSynth to be used as scratch registers. If the number of dead registers at a point is not sufficient, WiPEr supplies global addresses that are unused in the residual program as scratch locations to McSynth.

In the examples presented in this chapter, we have treated memory as if each memory location holds a 32-bit integer. However, our implementation supports the actual IA-32 memory model, in which each memory location holds an 8-bit integer.

The compiler might add instructions at the beginning of each procedure so that stack variables are aligned on 4-byte, 8-byte, or 16-byte boundaries. For example, consider the code snippet shown in Fig. 6.23. In procedures `foo` and `main`, the compiler aligns the stack variables on a 16-byte boundary. To accommodate such alignment, we relax P2 of $\mathbf{PE}_2$-safety properties to allow terms of the form $m + p_0 + p_1 + \ldots + c$, where m is a symbolic constant representing the specialization-time starting address of the stack or a heap block M, c is an integer constant, and p_0, p_1, etc. are symbolic constants that represent the alignment adjustments performed by the procedures Proc0, Proc1, . . . on the current call stack.

Table 6.24: Characteristics of applications for optimization via specialization.

Application	Domain	LOC	Description	Static Input
Power	Mathematical library functions	19	Computes x^n	$n = 100$
Dotproduct	Linear algebra operations	29	Computes the dot product of two vectors with n dimensions	$n = 100$, and co-efficients of the first vector
Interpreter	Language interpreters	71	Interpreter for the minimalist language "Brainf*ck"	Input program
Filter	Image processing	107	Applies a convolution filter of size $m \times m$ on an image of size $n \times n$	$m = 3$, $n = 3$, and elements of the filter
uuencode	Text utilities	129	Base-64 encodes a string of size n	$n = 256$, and the string
SHA1	Cryptographic operations	140	Computes the sha1 digest of a message of size n bits	$n = 1024$, and contents of the first 512 bits
uudecode	Text utilities	167	Decodes a base-64-encoded string of size n	$n = 256$, and the encoded string

If the binary is statically linked, WiPEr does not need any additional input from the user for partial evaluation. If the binary is not statically linked, the user needs to provide a model for each library function as additional inputs to WiPEr. A model consists of the following information: (i) an input-output dependence relation, and (ii) a procedure to compute return values of library-function calls that are classified as static. WiPEr specializes library-function calls in a bimodal manner: if any argument to a library-function call is dynamic, all of the static arguments are lifted along with the call; if all arguments are static, WiPEr uses the user-supplied model to compute a return value for the call (i.e., a fully static call is specialized away.)

We are generally unable to use WiPEr's fully automated end-to-end partial-evaluation algorithm to extract components from a large binary. Binaries of large applications do not have clearly demarcated inputs. They often have only one input—a buffer that stores the entire command line. The binary interprets the contents of the buffer to assign values to variables. If the buffer is classified dynamic, Alg. 6.11 classifies all instructions in the binary dynamic.

For component extraction, we run WiPEr in an alternative experimental mode, called Component Extraction (CE) mode. In CE-mode, WiPEr performs a restricted BTA. (The specialization algorithm is the same in CE and non-CE modes.) In CE-mode, the user specifies a subset I_S of instructions in the original binary to be treated as static instructions. The remaining instructions in the binary are treated as dynamic instructions. WiPEr identifies lifted instructions, and then specializes the binary using WiPEr's original specialization algorithm.

CE-mode is related to *generalized partial computation* [38], in which a program is specialized with respect to constraints. As will become clear below, in CE-mode we are specializing a program P with respect to a condition, such as $\varphi \equiv (EAX = 1)$, which must hold at a given point p in the middle of the program. The goal is to produce a residual program that is (i) a specialization of P, in which (ii) φ holds each time the points residuated from p are reached.

For simple examples of the kind discussed below, CE-mode successfully extracts components, even though congruence is violated. It remains for future work to devise a theory of program specialization that covers CE-mode.

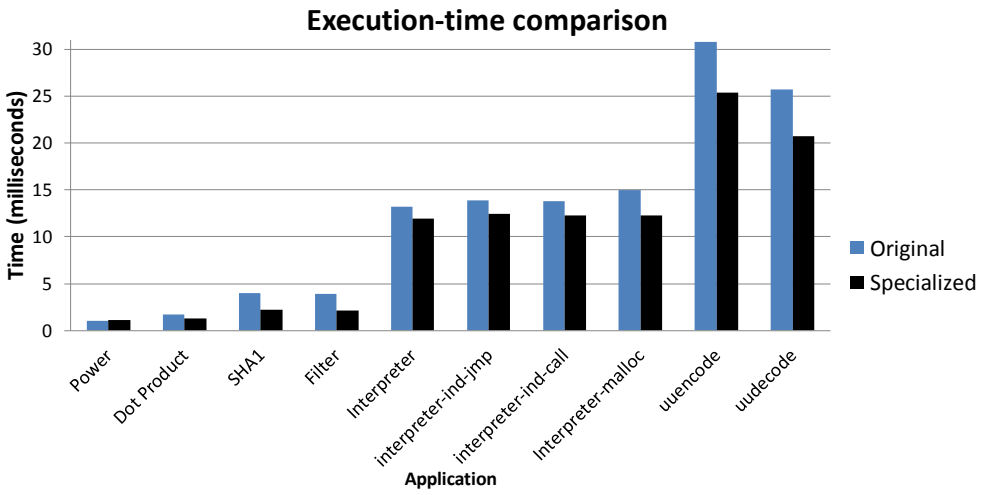


Figure 6.25: Comparison of the execution times of the original and specialized binaries.

6.5 Experiments

The goal of our experiments was to test and evaluate the two primary use cases of WiPEr: (i) optimization via specialization, and (ii) component extraction. Our experiments were run on a system with a 64-core, 2.3 GHz AMD Opteron 6376 processor; however, WiPEr's algorithm is single-threaded. The system has 264 GB of memory and runs RHEL 6.6.

6.5.1 Optimization via Specialization

For this use case, we wanted to answer the following question:

- What is the speedup produced by partial evaluation for different applications?

For the first set of experiments, we used a suite of applications from application domains that were suitable for partial evaluation. Table 6.24 presents the characteristics of the applications. The first two applications listed in the table have been used in prior work [61, 80]. There are four versions of the interpreter application in our test suite: (i) a standard version, (ii) a version that uses indirect jumps (via switch statements), (ii) a version that uses indirect calls (via function pointers), and (iv) a version that uses heap-allocated storage. uuencode and uudecode are FreeBSD utilities [37]. We used CE-mode for partially evaluating the FreeBSD utilities. The results are shown in Fig. 6.25. For these applications, the average speedup produced by partial evaluation, computed via the geometric mean, is 1.3.

For Power, partial evaluation results in a slight slowdown. Power has a tight loop, and we believe that the aggressive loop unrolling caused by partial evaluation disrupts

Table 6.26: Characteristics of applications for component extraction.

Application	KLOC	Description	Component
b64	0.544	Base64 encoder-decoder	decoder
lzfx	0.914	LZF compression-decompression utility	compress
bzip2	7	Compression-decompression utility	compress

Table 6.27: Comparison of sizes and number of procedures in original binary and extracted component.

Application	Size of binary (KB)	No. of procs. in binary	Size of component (KB)	No. of procs. in component
b64	5.4	16	5.9	4
lzfx	7.6	24	6	8
bzip2	90.2	117	64.6	78

instruction caching.

At the other end of the spectrum, we measured a speedup in Filter of 1.9. Filter has four nested for-loops. The inner two loops iterate over the filter elements. Partial evaluation unrolls all four loops, and—because the filter elements are static—replaces almost all instructions in the bodies of the inner two loops with just a few residual instructions.

6.5.2 Component Extraction

For this use case, we wanted to answer the following questions:

- How can we extract an executable component from a binary via partial evaluation?
- How does the size of the extracted component compare to the size of the original binary?

For this set of experiments, we used three applications that bundle several components into a bloated executable. Table 6.26 presents the characteristics of the applications. The first and second applications were obtained from Google Code [1] and SourceForge [2], respectively.

CE-mode of WiPEr was used for this set of experiments. The sizes of the original binary and the extracted components are shown in Table 6.27.

`b64`. The decoder extracted from b64 is bigger than the b64 binary because our current implementation does not optimize the layout of basic blocks in the residual code to reduce the number of jump instructions. For example, b64 has 30 `jmp` instructions, and the extracted decoder has 164 `jmp` instructions. The issues in partially evaluating `b64` are similar to those of `bzip2`, which is discussed below.

lzfx. lzfx is a text-compression utility. In this case study, we present how WiPEr extracts the compression component of lzfx. The decision about whether to call fx_create (which compresses) or fx_read (which decompresses) is made in the following code snippet from the binary:

```
_text:080499D4   test  eax,eax ;S
_text:080499D6   jnz  loc_80499E2 ;S
_text:080499D8   mov  dword [esp+44],0 ;S
                       :
_text:08049A3A   mov  eax,dword [esp+44] ;L
_text:08049A3E   mov  dword [esp+12],eax ;D
                       :
_text:08049A5E   call  fx_create ;D
                       :
loc_80499E2:
                       :
_text:08049A84   call fx_read; D
```

If the value in the eax register is 0 at instruction 80499D4, lzfx calls fx_create. The user specifies that instructions 80499D4 through 8049A3A should be treated as static instructions. WiPEr lifts instruction 8049A3A because it supplies a static value to the dynamic instruction 8049A3E. WiPEr performs specialization and evaluates the static and lifted instructions with respect to the partial store

$$\rho \equiv \langle [EAX \mapsto 0][ESP \mapsto n], [\,], [\,] \rangle,$$

and residuates the instruction mov eax,0 for the lifted instruction 8049A3A, and emits the remaining dynamic instructions. Because eax is 0 at instruction 80499D4, the fall-through branch is taken at 80499D6, which forces WiPEr to enter fx_create and to bypass fx_read. Consequently, WiPEr residuates an executable that only performs compression.

With lzfx, we also went further by embedding the extracted compression component in a different application. We created a small header file with the signature of fx_create, and linked the new application with the compression component extracted by WiPEr.

bzip2. bzip2 is a large application that takes several auxiliary command-line inputs in addition to one that specifies whether to perform compression or decompression. Ultimately, the decision about whether to call compress or uncompress is made in the following code snippet of the binary:

```
_text:0805B539   mov eax,[esp+40] ;S
_text:0805B53E   cmp  eax,1 ;S
```

```
_text:0805B541    jnz   loc_805B5DB ;S
                          :
_text:0805B558    call  compress ;D
                          :
loc_805B5DB:
                          :
_text:0805B5E4    call  uncompress ;D
```

The memory location whose address is $ESP + 40$ holds a value computed from the compression/decompression flag on the command line. If the value is 1, `bzip2` calls `compress`; otherwise, it calls `uncompress`.

The user specifies that instructions `805B539` through `805B541` should be treated as static instructions. WiPEr performs specialization and evaluates the aforementioned instructions with respect to the partial store

$$\rho \equiv \langle [ESP \mapsto n], [\,], [n + 40 \mapsto 1] \rangle.$$

This partial store forces WiPEr to enter `compress` and bypass `uncompress`, and consequently WiPEr residuates an executable that only performs compression.

6.6 Related Work

Partial evaluation. Partial evaluation has been used as a general framework for the specialization of programs belonging to different languages, and different domains. In addition to several functional languages [48, 57], partial evaluation has been used to specialize a flow-chart language [48], C [9, 26], FORTRAN [50], and Java [78, 80]. Partial evaluators can either be *on-line* (performed in a single phase) [43], *off-line* (performed in two phases), or hybrid [80]. WiPEr's high-level architecture resembles that of an off-line partial evaluator for an imperative language. However, the need to handle several machine-code-specific issues makes WiPEr's algorithm significantly different from those of source-code partial evaluators.

Run-time specialization techniques. Run-time specialization (or dynamic compilation) generates optimized code during program execution by partially evaluating user-annotated regions of the program with respect to invariant data computed at run time [13, 25, 32, 52].

Like WiPEr, these tools perform specialization on machine code. However, the partial-evaluator-like component in such tools is part of the compiler. BTA is performed with respect to source-code variables, and carried out on an IR that is constructed from source code. Run-time-specialization tools have to address an issue that does not arise in WiPEr,

namely, the need to create code templates that can be reused at run-time. Some tools accomplish this task by jury-rigging an existing compiler's code generator to inhibit optimizations, such as code motion, by using the `volatile` type qualifier of C.

In contrast, WiPEr faced other issues, such as the need to decouple multiple updates performed by a single instruction, and to create code that allows the stack and heap to be at different positions at run-time than at specialization-time.

Specialization of Java bytecode. WiPEr is not the first partial evaluator that works on low-level code. Lancet [73] performs partial evaluation of Java bytecode snippets that need to be compiled by a just-in-time (JIT) compiler. JScp [51] performs supercompilation on Java bytecode. Like WiPEr, the specializers in Lancet and JScp make use of a static partial store. For Lancet and JScp, the static partial store is a Java virtual-machine (JVM) store, while WiPEr's is an IA-32 store. However, while those specializers work on low-level code, WiPEr works at an "even lower level." The following issues that arise in IA-32 and not in bytecode justify the novel design choices made in WiPEr:

1. Because the JVM is a stack-based machine, specialization of bytecode instructions often involves loading constant values instead of locals/fields. In contrast, the operands of IA-32 instructions vary widely, and the specialization of an instruction with respect to a partial store need not be a variant of the instruction. Moreover, IA-32 has around 43,000 unique opcodes, and WiPEr's specializer must be able to specialize *any* instruction with respect to *any* partial store. WiPEr addressed this issue by using machine-code synthesis.

2. Address computation is not explicit in bytecode (variables are referenced using offsets). Consequently, bytecode specialization does not face the problem of residuating specialization-time addresses. However in machine code, instructions compute addresses, and a naïve specializer, such as $\mathbf{PE}_1$, is unsatisfactory. For this reason, WiPEr uses symbolic techniques for specialization.

Superoptimization and link-time optimization. Superoptimization aims at finding an optimal instruction-sequence for a target instruction-sequence [16, 55, 76]. Peephole superoptimization [16] uses "peepholes" to harvest target instruction-sequences, and replace them with equivalent instruction-sequences that have a lower cost. Link-time optimizers carry out whole-program optimizations at link time [59]. While these optimization techniques optimize a binary for all possible inputs, WiPEr optimizes a binary for a specific input.

7 An Improved Algorithm for Slicing Machine Code

This chapter presents the second instantiation of the semantics-based binary rewriting framework (Framework 1.1) described in Chapter 1: an improved slicer for IA-32.

One of the most useful primitives in program analysis is *slicing* [46, 98]. 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 [12]). Binary analysis and rewriting has received an increasing amount of attention from the academic community in the last decade (e.g., see references in [88, §7], [14, §1], [23, §1], [31, §7]), which has led to the development and wider use of binary analysis and rewriting tools. Improvements in machine-code slicing could significantly increase the precision and/or performance of several existing tools, such as partial evaluators [89], taint trackers [22], and fault localizers [100]. 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 [15, 23] recover an *instruction-level system dependence graph (SDG)*[2] from a binary, and use an existing source-code slicing algorithm [46, 71] 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 ISAs such as IA-32 [47] and ARM [11] 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 microcode,[3] 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

[1] In the literature, program point p and the variable set are called the *slicing criterion* [98]. In this chapter, when we refer to a program point p as the "slicing criterion," we mean p and all the variables used at p.

[2] The SDG is an intermediate representation used for slicing; see §7.1.

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

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 Chapter 6, we sub-optimally addressed the granularity issue via an ad-hoc instruction-decoupling strategy. This chapter proposes a more principled approach to addressing the granularity issue.

In this chapter, 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 [89]. More precise slicing could also be used to reduce the overhead of taint trackers [22] 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 §7.5.1.) 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 [15]. McSlice performs slicing at the microcode level instead of the instruction level. McSlice first converts the instruction-level SDG into a microcode-level SDG or μ-SDG. (McSlice uses QFBV formulas to explicitly represent the microcode at each node.) McSlice then uses an existing interprocedural context-sensitive slicing algorithm [46, 71] 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 McSynth 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

This chapter'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 (§7.2.1).
- 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 (§7.3.1).
- We show how machine-code synthesis can be used to reconstitute a machine-code program from a microcode slice (§7.3.2).
- 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 (§7.5.1).
- 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 §7.1.1.)

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.

7.1 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.

Slicing is typically performed using an IR of the binary called a *system dependence graph* (SDG) [35, 46]. To build an SDG for a program, one needs to know the set of variables that

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

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 CodeSurfer/x86 [15] (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) [14] 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 [14, §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. 7.1 shows the VSA state before each instruction in a small IA-32 code snippet. In Fig. 7.1, 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. 7.1.

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) [53]. 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. 7.2.) 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

```
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^
               (res&255) >>l 1^(res&255^
               (res&255) >>l 1) >>l 2^
               (res&255^(res&255) >>l 1^
               (res&255^(res&255) >>l 1)
                >>l 2) >>l 4)&1) == 0];
      in State(regs1, flags6, memory),
    ...
   )
  )
}
```

Figure 7.2: 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 7.3: BIL code for `add eax,ebx` [24]. BIL is the UAL used in BAP.

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 [35]. For example, in the system-dependence subgraph for the code snippet in Fig. 7.1, 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`.

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. 7.10 for an example SDG with interprocedural edges.)

CodeSurfer/x86 uses an existing interprocedural-slicing algorithm [46, 71] to perform machine-code slicing on the recovered SDG.

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

7.1.1 Other platforms for machine-code slicing.

Apart from CodeSurfer/x86, there are other machine-code analysis platforms, such as Vine [88], REIL [29], and BAP [23]. 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. 7.3.) 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.

7.2 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 7.5: Assembly listing for diff with the imprecise backward slice computed by CodeSurfer/x86.

7.2.1 Granularity Issue in Machine-Code Slicing

Consider the C program diff shown in Fig. 7.4. 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. 7.4). The backward slice with respect to `main`'s return value gets us the desired result. The backward slice is highlighted in gray in Fig. 7.4. (This source-code slice is computed using CodeSurfer/C [10].)

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. 7.5. The corresponding slicing criterion is the boxed instruction in Fig. 7.5. (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. 7.5.) The backward slice with respect to the slicing criterion includes the lines highlighted in gray in Fig. 7.5. (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. 7.5 whose QFBV formula is

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

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

```
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 7.6: 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
36: ret
```

Figure 7.7: Assembly listing for square with the imprecise forward slice computed by CodeSurfer/x86.

before ascending back to `main` from the `square` function, the most recent instruction added to the slice is instruction 10 in Fig. 7.5 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. 7.5. 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. 7.5. 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. 7.6. 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. 7.6). The forward slice of the source code with respect to `a`, highlighted in gray in Fig. 7.6, gets us the desired result. The machine-code forward slice with respect to the analogous slicing criterion (boxed instruction in Fig. 7.7)

includes the lines highlighted in gray in Fig. 7.7. 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. 7.7. 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. 7.21.) 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.)

7.2.2 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 [46, 71] to compute the slice over the μ-SDG. The final slice includes only the microcode updates that are relevant to the slicing criterion.

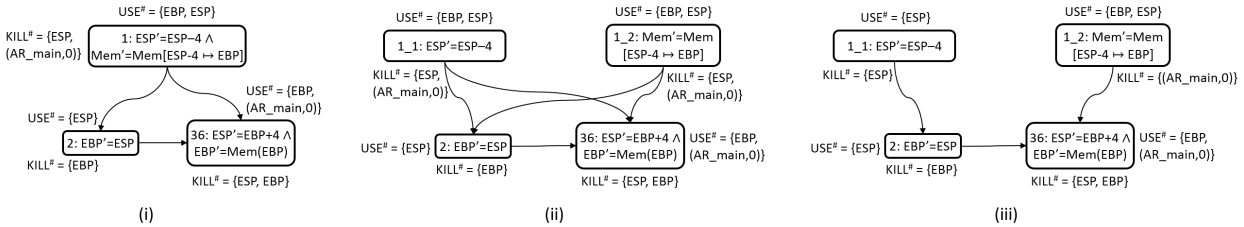


Figure 7.8: SDG snippet illustrating the construction of μ-SDG.

3. McSlice uses an existing machine-code synthesizer [90] to reconstitute a machine-code program from the microcode slice.

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, 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.

This section illustrates the improved slicing algorithm on our two examples from §7.2.1. 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. 7.8(i) shows the SDG snippet[4] for the first few instructions in function `main` in the diff 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. 7.8(ii)). Initially, nodes `1_1` and `1_2` inherit the

[4]In Fig. 7.8 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 number in each node's label. To reduce clutter, we do not show microcode nodes corresponding to flag updates in μ-SDGs.

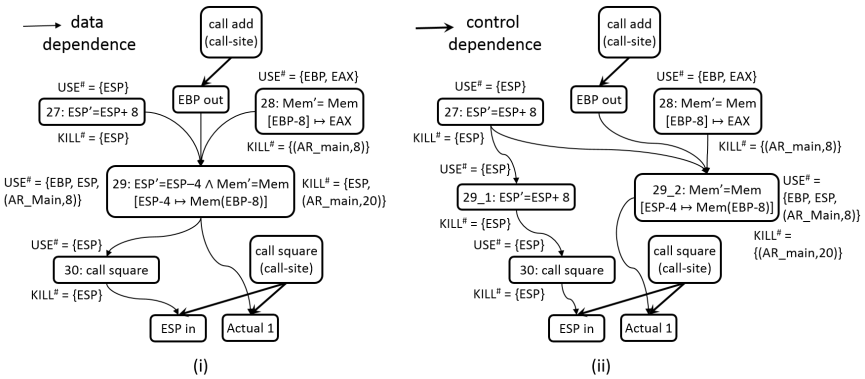


Figure 7.9: SDG snippet illustrating the construction of μ-SDG.

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 §7.3.1.) The μ-SDG snippet, along with the updated USE# and KILL# sets, is shown in Fig. 7.8(iii).

We illustrate McSlice's μ-SDG construction algorithm on another SDG snippet. Fig. 7.9 (i) shows the SDG snippet for the instructions that were used to illustrate the granularity issue for backward slicing in §7.2.1. 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. 7.9(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. 7.10. (To reduce clutter in Fig. 7.10 and Fig. 7.12, some intraprocedural control-dependence edges have been omitted. The omitted edges do not cause additional nodes to 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 §7.2.1 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. 7.10. 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

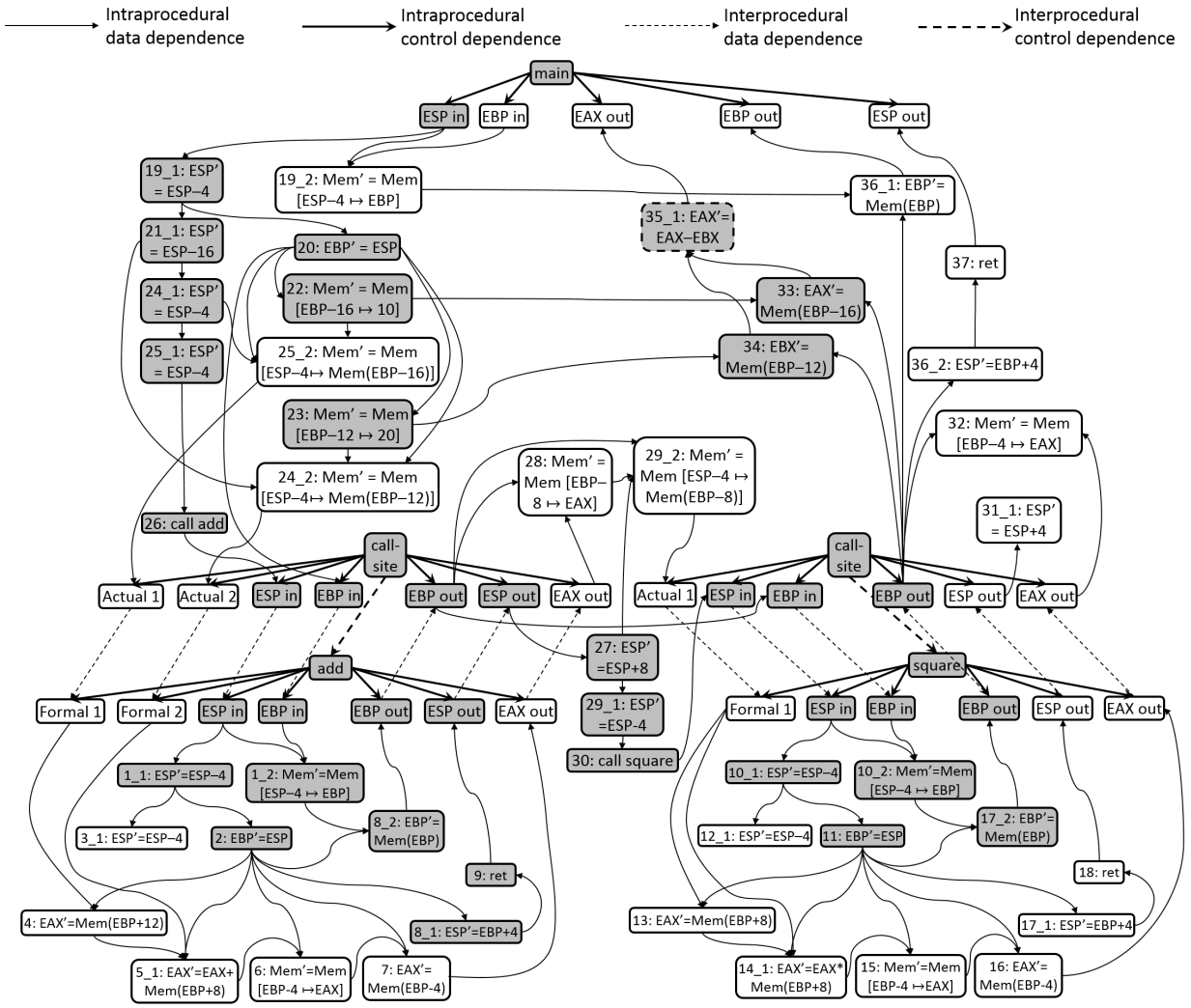


Figure 7.10: Microcode slice over the μ-SDG for the diff program. The slicing criterion is node 35, which is indicated by the dashed box.

computed by McSlice is more precise than the backward slice computed by CodeSurfer/x86 (cf. Fig. 7.5).

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. 7.10 is shown in Fig. 7.11. (To obtain *executable* code from a backward microcode slice, McSlice performs a few additional steps—see §7.3.2.)

```
add:
push ebp
mov ebp,esp
leave
ret
```

```
square:
push ebp
mov ebp,esp
leave
ret
```

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

Figure 7.11: Code generated by McSlice from the microcode backward slice for diff. Highlighted instructions are created by machine-code synthesis.

McSlice computes forward slices in a similar manner: McSlice converts the 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 §7.2.1. The μ-SDG created by McSlice for the program is given as Fig. 7.12. McSlice computes the forward slice over the μ-SDG with node 22 as the slicing criterion. (Recall from §7.2.1 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. 7.12. One can see that the slice computed by McSlice is more precise than the forward slice computed by CodeSurfer/x86 (cf. Fig. 7.7).

7.3 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 (§7.3.1). Then, we describe how McSlice reconstitutes an executable machine-code program from a microcode slice (§7.3.2).

7.3.1 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

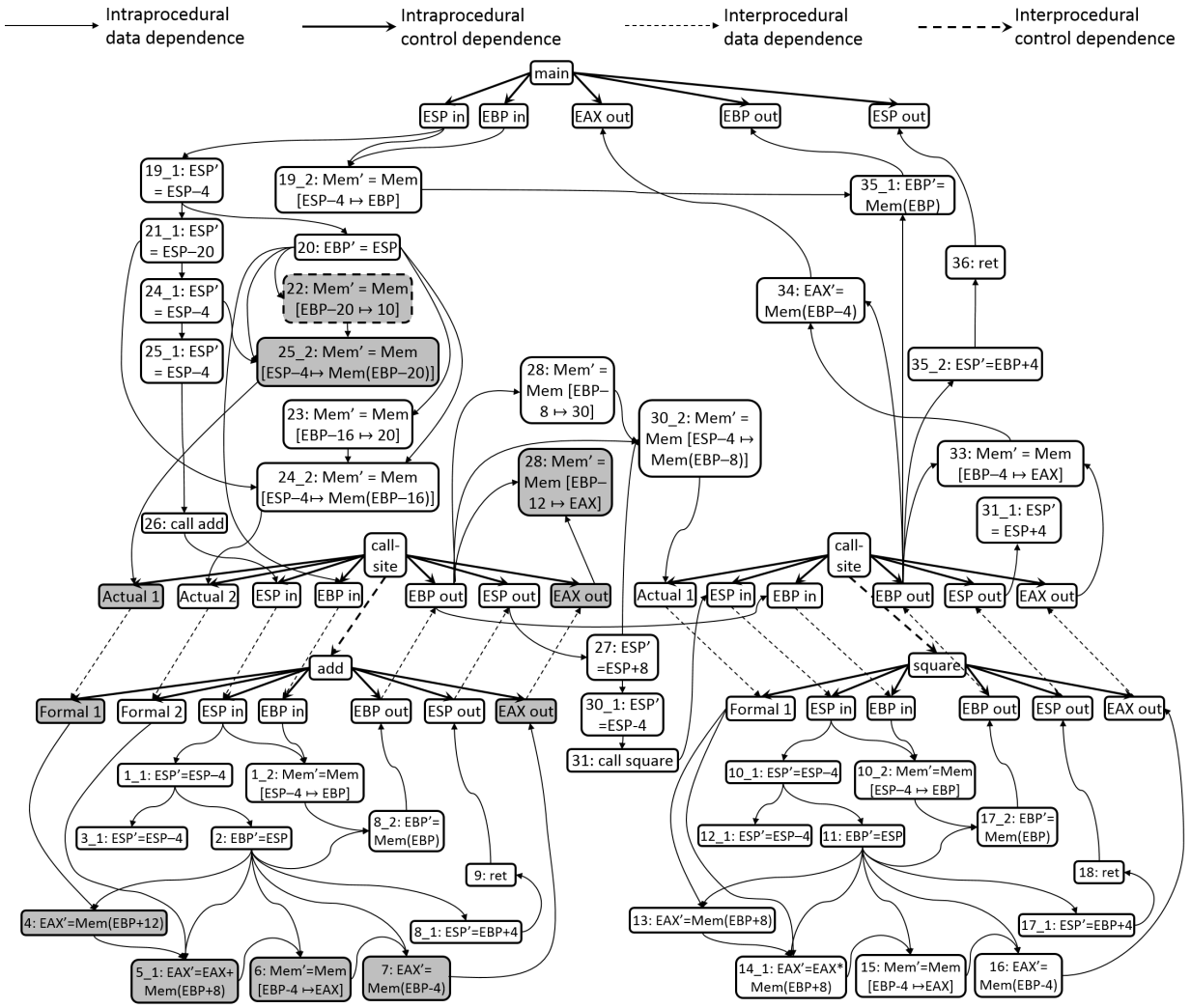


Figure 7.12: Microcode slice over the μ-SDG for the square program. The slicing criterion is node 12, which is indicated by the dashed box.

contains a multi-assignment instruction, and the PDG[5] 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

[5]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.

Output: PDG, CFG, m
Input: Updated PDG, Updated CFG
1: conjuncts $\leftarrow$ `GetConjuncts`(m.microcode)
2: **for** each conjunct $c \in$ conjuncts **do**
3: $n \leftarrow$ `CreateNode`( )
4: n.microcode $\leftarrow c$
5: n.use $\leftarrow$ `Project`(n.microcode, m.use)
6: n.kill $\leftarrow$ `Project`(n.microcode, m.kill)
7: **for** each edge $\leftarrow \langle p, m, \text{data}\rangle \in$ PDG.edges **do**
8: **if** p.kill $\cap$ n.use $\neq \emptyset$ **then**
9: PDG.edges $\leftarrow$ 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 $\leftarrow$ 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 $\leftarrow$ 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 $\leftarrow$ 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 $\leftarrow$ 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 $\leftarrow$ CFG.edges $\cup \{\langle n, s\rangle\}$
28: **end for**
29: **end for**
30: PDG.edges $\leftarrow$ PDG.edges $- \langle *, m, *\rangle - \langle m, *, *\rangle$
31: PDG.nodes $\leftarrow$ PDG.nodes $- \{m\}$
32: CFG.edges $\leftarrow$ CFG.edges $- \langle *, m\rangle - \langle m, *\rangle$
33: CFG.nodes $\leftarrow$ CFG.nodes $- \{m\}$
34: **return** $\langle$PDG, CFG$\rangle$

Algorithm 7.13: Algorithm SplitNode

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. 7.13. In Alg. 7.13, 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

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 ∈ 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 ∈ PDG **do**
10: **for** each edge ⟨p, n, data⟩ ∈ PDG **do**
11: **if** p ∉ ReachingDefs(n) **then**
12: PDG.edges ← PDG.edges − ⟨p, n, data⟩
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: SDG ← `RecomputeSummaryEdges`(SDG)
18: **return** SDG

Algorithm 7.14: Algorithm for μ-SDG construction used in McSlice

microcode has the form shown in Eqn. (2.1). Each conjunct in Eqn. (2.1) is an individual microcode assignment to a register, flag, or memory location. In Alg. 7.13, `GetConjuncts` returns the set of conjuncts 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. 7.8), 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. 7.13, `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-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. 7.14. Alg. 7.14 takes as input the SDG and CFGs constructed by CodeSurfer/x86, and returns a μ-SDG. For each PDG in the SDG, Alg. 7.14 first removes the summary edges from the PDG (via

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

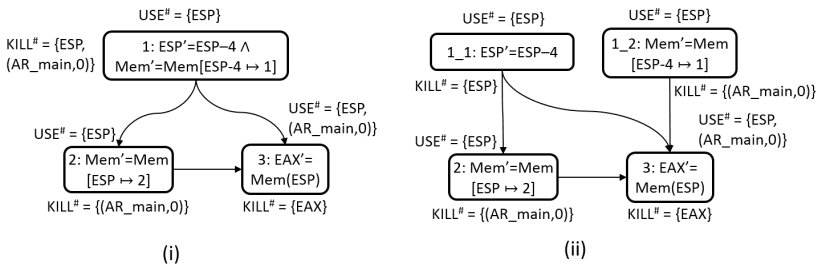


Figure 7.15: Code snippet, and its corresponding SDG and μ-SDG snippets to illustrate removal of data-dependence edges via reaching-definitions analysis.

`RemoveSummaryEdges`) because the existing summary edges in the SDG reflect imprecise transitive dependences across procedure calls (Line 2). Alg. 7.14 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 in Fig. 7.15, and its corresponding SDG given in Fig. 7.15(i). After splitNode splits node `1`, and recomputes data dependences, the μ-SDG obtained is given in Fig. 7.15(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. 7.14 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, $\ldots$, o_m associated with it, `RunReachingDefs` uses the transformer

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

where $l_1, l_2, \ldots l_m$ are the a-locs defined by the actual-out nodes $o_1, o_2, \ldots o_m$, 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. 7.14 removes the edge e from the PDG (Lines 9–15). (In Alg. 7.14, 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. 7.14 uses an existing algorithm (Fig. 5 in [71]) 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

```
int g1, g2;
void foo(int a, int b) {
  g1 = a;
  g2 = b;
}
```

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

Figure 7.16: Example program to illustrate the parameter-mismatch problem in slicing.

[46]) to compute a context-sensitive microcode slice over the computed μ-SDG.

7.3.2 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.

Parameter mismatches [19, 46]. 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 [19, 46]. For example, consider the program shown in Fig. 7.16. The slicing criterion is boxed in Fig. 7.16, 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 [19]. 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. 7.16.

```
add:
push ebp
mov ebp,esp
leave
ret
```

```
square:
push ebp
mov ebp,esp
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
push [ebp-8] *
call square
mov eax,[ebp-16]
mov ebx,[ebp-12]
sub eax,ebx
```

Figure 7.17: Machine code generated from the microcode slice shown in Fig. 7.10 by a naïve method.

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* [12] 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. 7.10, we would obtain the machine-code program shown in Fig. 7.17. One can see that the code in Fig. 7.17 performs computations on uninitialized values. For example, the instruction marked by * in Fig. 7.17 pushes onto the stack the 32-bit value in the memory location EBP $-$ 8, which has not been initialized by upstream instructions.

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 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. 7.18. The input to Alg. 7.18 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. 7.18, `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,

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`$(s', m.\text{instruction})$
7: $s \leftarrow s -$ newNodes
8: **else**
9: $I \leftarrow$ McSynth$(n.\text{microcode})$
10: $s' \leftarrow$ `AddToProgram`(s', I)
11: $s \leftarrow s - \{n\}$
12: **end if**
13: **end for**
14: **return** s'

Algorithm 7.18: Algorithm used by McSlice for generating machine code from a microcode slice

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 [90]. 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. 7.18, McSynth invokes the synthesizer; we assume that the procedure `AddToProgram` takes care of generating code in the correct order, creating new procedures, etc.) Alg. 7.18 returns the final executable machine-code program generated from the microcode slice.

For the microcode slice shown in Fig. 7.10, Alg. 7.18 produces the machine-code program shown in Fig. 7.11.

7.4 Implementation

McSlice uses CodeSurfer/x86 [15] to obtain the SDG for a binary, and the $\text{USE}^{\#}$/$\text{KILL}^{\#}$ sets at each instruction.

McSlice uses Transformer Specification Language (TSL) [53] 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

Table 7.19: Characteristics of applications in our test suite.

Appln.	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`

modes, etc.). The semantics written in TSL is reinterpreted to produce the QFBV formulas for individual instructions [54]. McSlice's slicing algorithm will split any instruction that (i) belongs to one of the 164 opcode variants, (ii) performs a multi-assignment, and (iii) 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 7.19) included instructions belonging to 35 out of the 164 opcode variants.

McSlice uses the machine-code synthesizer McSynth [90] 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 [53, §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.

7.5 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?

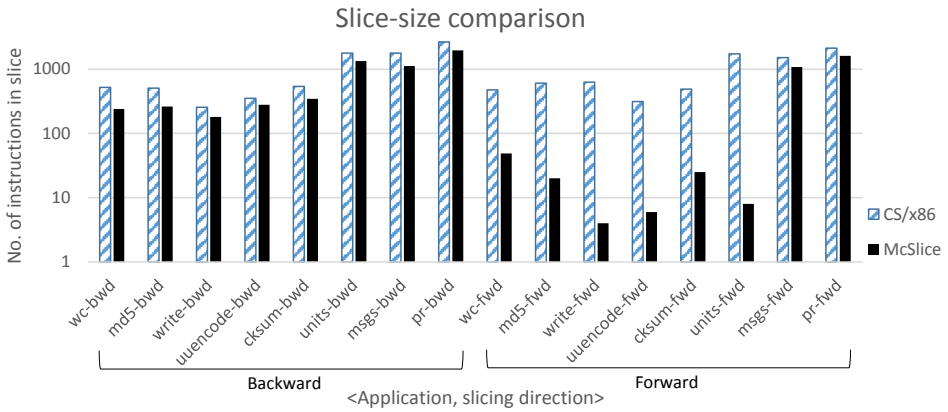


Figure 7.20: Comparison of sizes of slices computed by CodeSurfer/x86 and McSlice (log-scale).

- 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?

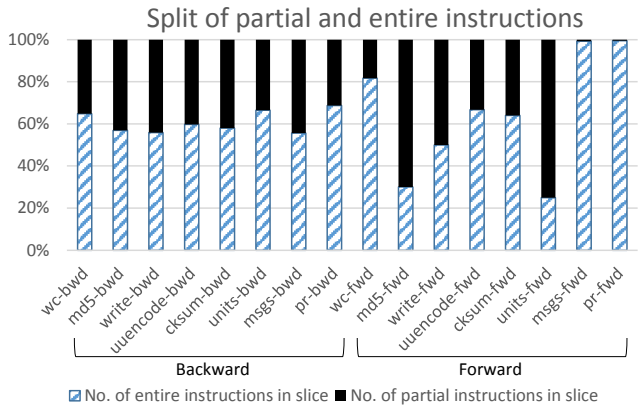
All experiments were run on a system with a quad-core, 3GHz Intel Xeon processor running Windows 7, and 32 GB RAM; however, McSlice's algorithm is single-threaded.

Our test suite consisted of IA-32 binaries of FreeBSD utilities [37]. Table 7.19 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 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. 7.20. 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



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

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. 7.7) 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. 7.21 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. 7.21, 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 7.22: 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

7.5.1 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. 7.18 to reconstitute a machine-code program for the extracted component, and generated an executable slice (§7.3.2). Table 7.22 presents the characteristics of the applications used for component extraction. `wc-lite` is a scaled down version of the `wc` utility.[6] 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 7.22 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 §7.3.2.

7.6 Related Work

Slicing. The literature on program slicing is extensive [20, 58, 96]. Slicing has been—and continues to be—applied to many software-engineering problems [45]. For instance, recently there has been work on language-independent program slicing [21], which repeatedly creates potential slices through statement deletion, and tests the slices against the original program for semantics preservation. Specialization slicing [12] 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 §7.3.2.

[6]The source code for this micro-benchmark is given as Fig. 2 in [70].

Applications of more precise machine-code slicing. WiPEr is a machine-code partial evaluator [89] 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 [60, 79, 94] use dynamic analysis to check if tainted inputs from *taint sources* could affect *taint sinks*. Certain taint trackers such as Minemu [22] 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 [100] 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.

8 Conclusion and Future Directions

This dissertation (i) introduced the problem of machine-code synthesis, (ii) outlined a general framework for semantics-based binary rewriting via machine-code synthesis, (iii) described a suite of approaches—ranging from pruning heuristics to machine learning—that can be used to speed up machine-code synthesis, and (iv) presented two instantiations of the binary-rewriting framework—a machine-code partial evaluator and a machine-code slicer—that can be used to extract an executable component from a binary for purposes of software reuse. This dissertation makes the following contributions that go beyond the realms of binary analysis and rewriting:

1. We have proposed a divide-and-conquer approach to enumerative program synthesis that facilitates breaking a larger synthesis task into a sequence of independent smaller sub-tasks.

2. We have proposed lightweight pruning strategies that use abstractions/overapproximations of enumerated candidates to rapidly prune away useless candidates during synthesis.

3. Our model-assisted synthesis algorithm is the first program-synthesis technique to use models learned from *both* specifications *and* implementations to assist the synthesis search.

4. We have shown, via program synthesis, how one can reconstitute a program that is at a higher level from an intermediate representation (IR) that is at a lower level. (We have demonstrated this synthesis-based approach to program reconstitution in the context of partial evaluation and slicing.) This approach to program reconstitution becomes particularly useful when translation from low-level IR fragments to high-level code constructs is not straightforward, and the search space of high-level code constructs is enormous.

5. We have proposed a symbolic approach to program specialization. This approach to specialization becomes particularly useful when the specializer wants to treat certain inputs as static inputs, but somehow suppress their residuation.

In the remainder of this chapter, we present concluding remarks and future directions for the algorithms and applications, respectively, described in this dissertation.

Algorithms for Machine-Code Synthesis

In Chapter 3, we presented a technique to synthesize a straight-line machine-code instruction sequence from a semantic specification of the desired behavior, given as a QFBV formula. We presented McSynth, a tool that implements our technique. McSynth is parameterized by the ISA of the target instruction-sequence, and is easily adaptable to work on other semantic representations, such as a Universal Assembly Language (UAL) [23]. McSynth is the first machine-code synthesizer that works on the integer subset of a full ISA. A key challenge that McSynth has to address is the enormous size of the synthesis search-space: ISAs like IA-32 have around 43,000 unique instruction schemas. To counter the enormous search space, McSynth uses (i) a divide-and-conquer strategy to split the input formula into several independent smaller sub-formulas, (ii) footprint-based search-space pruning heuristics to prune away candidates during synthesis, and (iii) a novel instantiation of the CEGIS framework as the core synthesis loop. Experiments with the IA-32 instruction set showed that McSynth is 3 to 5 orders of magnitude faster than a baseline enumerative synthesizer.

McSynth brought down synthesis time from days to minutes, but it was still not fast enough: McSynth times out for several larger QFBV formulas; even for smaller formulas, McSynth takes several minutes to find an implementation. This delay might not be tolerable for a binary-rewriting client (e.g., a machine-code partial evaluator) that has to invoke McSynth several times to rewrite an entire binary. In Chapter 4, we made several improvements to the synthesis algorithm used in McSynth, and developed McSynth++ , which is an improved version of McSynth. In addition to a novel bits-lost-based pruning heuristic, the improvements incorporate a number of ideas known from the literature, which we adapted in novel ways for the purpose of speeding up machine-code synthesis. Experiments for IA-32 showed that the improvements enable synthesis of code for 12 out of 14 formulas on which McSynth timed out, speeding up the synthesis time by at least 1981X, and for the remaining formulas, speeds up synthesis by 3X.

In Chapter 5, we presented McSynth-ML, the first low-level-code synthesizer that uses machine learning to assist synthesis. Instead of the linear search used by McSynth and McSynth++, McSynth-ML uses a novel model-assisted best-first search as the core search strategy. The cost heuristic for the best-first search comes from two models trained over a huge corpus of specifications (QFBV formulas) and implementations (instruction se-

quences). One model is an n-gram-based language model, which steers the search towards common/useful instruction sequences. The other is a k-NN-regression model that correlates features of implementations with features of specifications, and steers the search towards instruction sequences that are highly likely to implement the input formula. Experiments for IA-32 showed that the McSynth-ML enables synthesis of code for 6 out of 50 formulas on which McSynth++ timed out, speeding up the synthesis time by at least 549X, and for the remaining formulas, speeds up synthesis by 4.55X.

Future Directions

One possible direction for future work would be to adapt the algorithms discussed in this dissertation to synthesize non-straight-line, but non-looping programs. One approach to loop-free code is to use the *ite* terms in the QFBV formula to create a loop-free CFG skeleton, and then synthesize an appropriate instruction sequence for each basic block.

A second direction would be to investigate how program verifiers can be used in conjunction with machine learning to assist program synthesizers. Research in program-verification technology over the last two decades has resulted in a large collection of tools and techniques for program verification. Using existing program-verification tools, one could build a large corpus of properties and programs that satisfy/violate the properties. Subsequently, the corpus can be used to build models that relate features of properties to features of programs that satisfy/violate the properties. One can then use such models to assist the search performed by an enumerative program synthesizer.

Applications

Our principal motivation for developing algorithms for machine-code synthesis was to develop a general framework for semantics-based binary rewriting, and subsequently instantiate the framework to create binary-rewriting tools. In this dissertation, we presented two instantiations of the framework: a novel machine-code partial evaluator, and an improved machine-code slicer. We outlined a number of potential uses of the two tools, and in particular, we showed how these tools can be used to either modify or extract an executable component from a binary.

In Chapter 6, we presented an algorithm for machine-code partial evaluation, and WiPEr, the first machine-code partial evaluator. WiPEr performs off-line partial evaluation of binaries. WiPEr's algorithm follows the classical two-phase approach of BTA followed by

specialization. WiPEr's specializer specializes an explicit representation of the semantics of an instruction, and emits residual code via machine-code synthesis. Moreover, to create code that allows the stack and heap to be at different positions at run-time than at specialization-time, the specializer represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization. WiPEr can be used to specialize binaries with respect to commonly used inputs to produce faster binaries (e.g., a file-write routine optimized for a certain file descriptor), as well as to extract an executable component from a bloated binary (e.g., extract `compress` from the `gzip` binary).

Future directions in machine-code partial evaluation. One possible direction for future work is to investigate polyvariant BTA via specialization slicing [12]. For every call to a procedure P with a different combination of static and dynamic actual parameters, specialization slicing can be used to produce a different binding-time annotation for P's instructions. This technique could increase the number of static instructions during specialization, leading to residual code that is more specialized than the code currently residuated by WiPEr. A second direction is to use liveness information to reduce the number of specialized copies of a basic block produced by WiPEr. □

In Chapter 7, we introduced the granularity issue caused by using source-code slicing algorithms on an instruction-level SDG, and showed how the issue can lead to very imprecise machine-code slices. We presented a tool called McSlice that slices machine code more accurately. To counter the granularity issue, McSlice 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, McSlice uses machine-code synthesis. Experiments on IA-32 binaries of FreeBSD utilities showed that, in comparison to slices computed by a state-of-the-art tool (CodeSurfer/x86), McSlice reduces the size of backward slices by 33%, and forward slices by 70%. We also showed how the backward executable-slicing functionality of McSlice can be used to extract executable components from binaries (e.g., extract line count from the `wc` binary).

Future directions in machine-code slicing. The monovariant executable-slicing algorithm used in McSlice 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 [12] 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. □

Future Directions

Approximate computing is a promising approach that trades quality of result for energy efficiency [34, 62, 75]. Research in approximate computing has led to the development of microarchitectures and high-level languages that support approximation. However, there are no tools that can port existing binaries that run on deterministic hardware to approximate hardware. A key property that such a tool has to enforce is that values computed by approximate instructions should never affect deterministic instructions. One possible direction for future work is to borrow the ideas explored by this dissertation on machine-code slicing and partial evaluation, and use them in conjunction with machine-code synthesis to develop a tool that can translate deterministic binaries into approximate ones.

A Appendix

Theorem 6.22. *Suppose that B is* ***PE$_2$-safe****. Also suppose that η_S is an environment such that there are no overlaps among (a) any of the chunks in Dom(η_S), (b) the chunk in which the residual code produced by WiPEr is loaded, and (c) any of the chunks allocated during the execution of the residual code. For all $\sigma = (\eta_S, \rho_S, \eta_D, \rho_D)$ such that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D)$ terminates, suppose that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta'_S, \rho'_S, \eta'_D, \rho'_D)$.*
If ρ_S^{sym} is a symbolic store such that $[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$, then $[\![WiPEr(B, \rho_S^{sym}))]\!](\eta_S, \eta_D, \rho_D) = (\eta'_S, \eta'_D, \overline{\rho}'_D)$, where $\overline{\rho}'_D \approx \rho'_D$. □

Proof. We prove Thm. 6.22 for binaries whose instructions come from a small instruction set, TinyIA32. TinyIA32 consists of a small subset of the IA-32 instruction set, augmented with (i) an extra register, and (ii) two new instructions. We decided to exclude instructions related to branches and function calls from TinyIA32 because if we were to prove Thm. 6.22 for the entire IA-32 ISA, the part of the proof concerning branches and function calls would be similar to that of an analogous proof for a source-code partial evaluator. We prove Thm. 6.22 using induction on the length of the trace of a run of the binary, and it is straightforward to extend our trace-based proof to an ISA that includes branches and function calls.

TinyIA32 has three registers, no flags, and seven instructions. The abstract syntax of TinyIA32 is defined in Fig. A.1. In an indirect operand, the register holds the address of the memory location. For instructions with two operands, the operand on the left is the destination operand. For the `stackalloc` instruction, the single operand is the destination operand. For the `heapalloc` instruction, the single operand is both a source and destination operand.

The register `rw1` is a *work register* that is used for holding the results of intermediate calculations in residual code. A work register is used to simulate the fact that the real WiPEr uses dead registers as scratch registers for such calculations. In the rest of this section, we do not show the work register in TinyIA32 stores. Also, if a register is not used in a TinyIA32 binary, it has the default value 0.

The first `mov` instruction loads a 32-bit constant into a register. The second `mov` instruction performs a register-to-register move. The final two `mov` instructions move a 32-bit value from a memory location to a register and from a register to a memory location, respectively. The `add` instruction adds the 32-bit values in the two register operands, leaving the result in the destination-operand register. The `stackalloc`(R) instruction allocates a stack of some fixed, but arbitrarily large, size, whose starting address is available as an output in R. The `heapalloc`(R) instruction allocates a heap block whose size is provided as an input in R, and whose starting address is returned in R as the output. Because TinyIA32 has no flags,

$$\begin{gathered}
\text{INT} ::= \{..., \texttt{-1}, \texttt{0}, \texttt{1}, ...\} \quad \text{Reg} ::= \texttt{r1} \mid \texttt{r2} \mid \texttt{rw1} \\
\text{R} ::= \text{Direct}(\text{Reg}) \quad \text{C} ::= \text{Imm}(\text{INT}) \quad \text{M} ::= \text{Indirect}(\text{R}) \\
\text{I} ::= \texttt{mov}(\text{R}, \text{C}) \mid \texttt{mov}(\text{R1}, \text{R2}) \mid \texttt{mov}(\text{R}, \text{M}) \mid \texttt{mov}(\text{M}, \text{R}) \mid \\
\texttt{add}(\text{R1}, \text{R2}) \mid \texttt{stackalloc}(\text{R}) \mid \texttt{heapalloc}(\text{R})
\end{gathered}$$

Figure A.1: Abstract syntax of TinyIA32.

a TinyIA32 store consists of only a register map and a memory map.

We prove Thm. 6.22 by induction on the length n of a partial trace T of a run of binary B.

Base Case $n = 0$

Thm. 6.22 trivially holds on a partial trace with no instructions.

Induction Hypothesis

Let us assume that Thm. 6.22 holds on a partial trace $T \equiv I_1, I_2, \ldots, I_n$ with n instructions. Suppose that the corresponding residual trace produced by WiPEr for T is $T' \equiv J_1, J_2, \ldots, J_n$, where J_i is the residual instruction sequence produced by the Specialize phase of WiPEr for I_i. (Note that J_i can be an empty sequence of instructions or can consist of several instructions.) Suppose that the initial state is $(\eta_S, \rho_S, \eta_D, \rho_D)$. (Recall from §6.2 that η maps a symbolic constant denoting the starting address of the stack or a heap block to a concrete address.) As the induction hypothesis, we assume the following:

$$\begin{aligned}
&[\![I_1, I_2, \ldots, I_n]\!](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) \\
\wedge\ &[\![J_1, J_2, \ldots, J_n]\!](\eta_S, \eta_D, \rho_D) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n) \\
\wedge\ &\overline{\rho}_D^n \approx \rho_D^n
\end{aligned} \tag{A.1}$$

To account for the actions of WiPEr on static and lifted instructions, we need to introduce yet one more ghost variable, *ghostStaticStore* (*gSS*). At each point k in the trace T', this variable holds the static symbolic store that was used by WiPEr to create the next residual instruction-sequence in T', i.e., $J_{k+1} = \text{Specialize}(I_{k+1}, gSS^k)$.

We strengthen the induction hypothesis with the following conjunct:

$$[\![gSS^n]\!]\eta_S^n = \rho_S^n, \tag{A.2}$$

where the left-hand side denotes the simplification of each symbolic expression in *gSS* with respect to the values obtained from η_S^n. Because gSS^0 is ρ_S^{sym}, by the hypothesis

$[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$ in Thm. 6.22, we have $[\![gSS^0]\!](\eta_S) = \rho_S$, and thus Eqn. (A.2) holds for the base case.

Induction Step

We now prove Thm. 6.22 on a partial trace I_1, I_2, ..., I_{n+1} with $n+1$ instructions. That is, the goal is to show

$$\begin{aligned} &[\![I_1, I_2, \ldots, I_{n+1}]\!](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta_S^{n+1}, \rho_S^{n+1}, \eta_D^{n+1}, \rho_D^{n+1}) \\ \wedge\ &[\![J_1, J_2, \ldots, J_{n+1}]\!](\eta_S, \eta_D, \rho_D) = (\eta_S^{n+1}, \eta_D^{n+1}, \overline{\rho}_D^{n+1}) \qquad \text{(A.3)} \\ \wedge\ &\overline{\rho}_D^{n+1} \approx \rho_D^{n+1} \\ \wedge\ &[\![gSS^{n+1}]\!]\eta_S^{n+1} = \rho_S^{n+1}. \qquad \text{(A.4)} \end{aligned}$$

Some additional intuition about the induction hypothesis can be obtained from the following more precise restatement of reparenthesization (6.5):

$$l : (\eta_S, \rho_S, \eta_D, \rho_D) \longrightarrow (l, gSS) : (\eta_S, \eta_D, \rho_D),$$

where $\rho_S = [\![gSS]\!]\eta_S$.

There are three subcases, for I_{n+1} being static, lifted, or dynamic.

Static instructions. Because static-allocation sites are lifted—see line [10] of Alg. 6.11—a static instruction cannot have the form `stackalloc`(R) or `heapalloc`(R). The argument for each of the static-instruction cases for I_{n+1} has the following form: Suppose that instruction I_{n+1} is static. Then

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^{n+1}, \eta_D^n, \rho_D^n), \qquad \text{(A.5)}$$

where some update is made to ρ_S^n to create ρ_S^{n+1}.

$$\mathsf{Specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \epsilon,$$

where ϵ denotes the empty sequence of instructions. Because J_{n+1} is empty, we have

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n). \qquad \text{(A.6)}$$

Eqns. (A.5) and (A.6), together with Eqn. (A.1), show that Eqn. (A.3) holds.

The congruence property of BTA ensures that WiPEr can completely evaluate I_{n+1} at specialization time, and gSS^n is updated by WiPEr to produce gSS^{n+1} in the same way as ρ_S^n is updated by $[\![I_{n+1}]\!]$ to produce ρ_S^{n+1}. Moreover, because η_S^n is not updated by $[\![I_{n+1}]\!]$, by

Eqn. (A.2) we have $[\![gSS^{n+1}]\!]\eta_S^n = \rho_S^{n+1}$, and Eqn. (A.4) holds. Consequently, the induction hypothesis holds on a trace with $n+1$ instructions, where the $(n+1)^{st}$ instruction is static.

Lifted and dynamic instructions. We now make an observation that is relied on in many of the cases—namely, because static-allocation sites are lifted, if instruction I_i in T is an allocation instruction, then J_i in T' also contains an allocation instruction, i.e., the allocations in the two traces are in lock-step. For purposes of the proof, we can assume that B and $\mathsf{WiPEr}(B, \rho_S^{sym})$ are run in environments that will perform storage allocation in the same way. Consequently, if I_i allocates the stack or a heap block M at address m, an instruction in J_i also allocates M at m. In the respective semantics, such allocations will be recorded by updating η to $\eta[m \mapsto m]$, where m is a symbolic constant that denotes the starting address of M. (Lifted allocation instructions update η_S and η_D; dynamic allocation instructions update η_D.)

Dynamic Instructions: The argument for each of the dynamic-instruction cases for I_{n+1}, except `stackalloc` and `heapalloc`, has the following form: Suppose that instruction I_{n+1} is dynamic. Then

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^{n+1}), \tag{A.7}$$

where some update is made to ρ_D^n to create ρ_D^{n+1}.

$$\mathsf{Specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = I_{n+1}$$

Because J_{n+1} is the same as I_{n+1}, we have

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^{n+1}), \tag{A.8}$$

where, $\overline{\rho}_D^{n+1} \approx \rho_D^{n+1}$. Eqns. (A.7) and (A.8), together with Eqn. (A.1), show that Eqn. (A.3) holds.

The static components of the state, η_S^n, gSS^n, and ρ_S^n, are left unchanged, so Eqn. (A.4) holds.

The cases for `stackalloc` and `heapalloc` are similar, except that those instructions update η_D^n in addition to ρ_D^n. Moreover, because of our assumption about the storage allocator, the updates to η_D^n by $[\![I_{n+1}]\!]$ and $[\![J_{n+1}]\!]$ are identical, and the updates to ρ_D^n by $[\![I_{n+1}]\!]$ and $[\![J_{n+1}]\!]$ are also identical. Consequently, the induction hypothesis holds on a trace with $n+1$ instructions, where the $(n+1)^{st}$ instruction is dynamic.

Lifted instructions: The semantics we use for evaluating static and dynamic instructions is fairly standard—-static instructions access and update ρ_S, and dynamic instructions

access and update ρ_D (in addition to updating η_D). However, the evaluation of lifted instructions is non-standard: lifted instructions were originally classified static, so they must be evaluated just like static instructions; however, they also take "snapshots" of the current static state and dump them into η_D and ρ_D for use by downstream dynamic instructions.

Lifted instructions access ρ_S, but produce values that might be consumed by dynamic successors, so they must update ρ_D. In addition, lifted instructions can also have static successors, so they must also update ρ_S. Thus, in the semantics used for the original program, a lifted instruction performs the same update on both ρ_S and ρ_D. In the semantics used for the residual program, it updates only ρ_D because ρ_S is not present in states. However, for the purposes of this proof, an appropriate update is performed on gSS.

Similarly, if a lifted instruction allocates memory, both static and dynamic instructions downstream might access or update locations in the allocated memory. Thus in both semantics, if a lifted instruction allocates memory, it performs identical updates on η_S and η_D.

The non-overlap hypothesis of Thm. 6.22, together with property P2 of $\mathbf{PE}_2$-safety, imply that each specialization-time address can be represented canonically by a term of the form $m + c$, where m is the symbolic constant that denotes the starting address m of the stack or a heap block M, and c is a constant offset.

The actual symbolic constants used are essentially arbitrary (as long as each new symbolic constant is fresh). However, for the purposes of the proof, we need to relate the symbolic constants used by the original program to those in the residual program. Thus, for the purposes of the proof, we assume that when WiPEr partially evaluates a lifted instruction I_i in T that allocates memory, the symbolic constant m that should be used in gSS to denote the starting address of the allocated stack or heap block is supplied by an oracle. This oracle ensures that the *same* symbolic constant m gets added to the ghost variables η_S and η_D when I_i and the allocation instruction in J_i are evaluated. (If the oracle did not ensure this concordance, we cannot establish the equality in Eqn. (A.2).) WiPEr binds m to a dedicated global address `M-addr`, and uses `M-addr` in the residual code to save the concrete starting address m of the allocated memory. (`M-addr` is located in the special area in the memory map of the residual program discussed in §6.3.4. Recall from the discussion of the $\mathbf{PE}_2$ specializer in §6.3.2 that WiPEr uses a memory-layout map μ to bind a symbolic constant to a dedicated location.)

We handle the lifted-instruction cases according to the form of the instruction. In the cases below, we abbreviate an update to the register map as $\rho[R1 \mapsto v]$ rather than ***let*** $\langle rm, mm\rangle = \rho$ ***in*** $\langle rm[R1 \mapsto v], mm\rangle$, and similarly use $\rho[a \mapsto v]$, to denote an update to the

memory map at address a. Which kind of map-update operation is intended should be clear from context.

1. $I_{n+1} =$ `mov` **(Direct(R1), Imm(v)):**

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[R1 \mapsto v], \eta_D^n, \rho_D^n[R1 \mapsto v]) \tag{A.9}$$

$$\mathsf{Specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov r1,v}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto v]) \tag{A.10}$$

Eqns. (A.9), (A.10), and (A.1) show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto v] \tag{A.11}$$

$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto v] \textit{ (From (A.9))} \tag{A.12}$$

Eqns. (A.11), (A.12), and (A.2) show that Eqn. (A.4) holds.

2. $I_{n+1} =$ `mov` **(Direct(R1), Direct(R2)):** There are two cases that can arise.

(a) Suppose that $\rho_S^n(R2) = u$ and $gSS^n(R2) = v$.

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[R1 \mapsto u], \eta_D^n, \rho_D^n[R1 \mapsto u]) \tag{A.13}$$

$$\mathsf{specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov r1,v}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto v]) \tag{A.14}$$

$$u = \rho_S^n(R2) = ([\![gSS^n]\!]\eta_S^n)(R2) \textit{ (By Eqn. (A.2))}$$

$$= ([\![gSS^n(R2)]\!]\eta_S^n) = [\![v]\!]\eta_S^n = v \tag{A.15}$$

Eqns. (A.13)–(A.15), and (A.1) show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto v] \tag{A.16}$$

$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u] \textit{ (From (A.13)} \tag{A.17}$$

Eqns. (A.15)–(A.17), and (A.2) show that Eqn. (A.4) holds.

(b) Suppose that $\rho_S^n(R2) = u$ and $gSS^n(R2) = m + c$.

$$\text{specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov rw1,M-addr; mov r1,[rw1]; mov rw1,c;}$$
$$\texttt{add r1,rw1;mov rw1,0}$$
$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto m+c]) \tag{A.18}$$
$$u = \rho_S^n(R2) = ([\![gSS^n]\!]\eta_S^n)(R2)\ (By\ \texttt{Eqn.}\ (A.2))$$
$$= ([\![gSS^n(R2)]\!]\eta_S^n) = [\![m + c]\!]\eta_S^n = m + c \tag{A.19}$$

Eqns. (A.13), (A.18), (A.19), and (A.1) show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto m + c] \tag{A.20}$$
$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u]\ (\textit{From (A.13)}) \tag{A.21}$$

Eqns. (A.19)–(A.21), and (A.2) show that Eqn. (A.4) holds. Note that J_{n+1} has the `mov rw1,0` instruction at the end of the residual code to restore the default value 0 in work register `rw1`.

3. I_{n+1} = `mov` (Direct(R1), Indirect(R2)): Because *R2* must hold an address, there are two cases that can arise.

(a) Suppose that $\rho_S^n(R2) = a$ and $\rho_S^n(a) = u$, and $gSS^n(R2) = m + c$ and $gSS^n(m + c) = v$.

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[R1 \mapsto u], \eta_D^n, \rho_D^n[R1 \mapsto u]) \tag{A.22}$$
$$\text{specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov r1,v}$$
$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto v]) \tag{A.23}$$
$$u = \rho_S^n(a) = \rho_S^n(\rho_S^n(R2))$$
$$= \rho_S^n(([\![gSS^n]\!]\eta_S^n)(R2))\ (By\ \texttt{Eqn.}\ (A.2))$$
$$= \rho_S^n([\![gSS^n(R2)]\!]\eta_S^n) = \rho_S^n([\![m + c]\!]\eta_S^n)$$
$$= ([\![gSS^n]\!]\eta_S^n)([\![m + c]\!]\eta_S^n)\ (By\ \texttt{Eqn.}\ (A.2))$$
$$= ([\![gSS^n(m + c)]\!]\eta_S^n) = [\![v]\!]\eta_S^n = v \tag{A.24}$$

Eqns. (A.22)–(A.24), and (A.1) show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto v] \tag{A.25}$$
$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u]\ (\textit{From (A.22}) \tag{A.26}$$

Eqns. (A.24)–(A.26), and (A.2) show that Eqn. (A.4) holds.

(b) Suppose that $\rho_S^n(R2) = a$ and $\rho_S^n(a) = u$, and $gSS^n(R2) = m_1 + c_1$ and $gSS^n(m_1 + c_1) = m + c$.

$$\begin{aligned}
&\mathsf{specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov rw1,M-addr; mov r1,[rw1]; mov rw1,c;}\\
&\quad\texttt{add r1,rw1; mov rw1,0}\\
&[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto m+c]) \qquad \text{(A.27)}\\
&u = \rho_S^n(a) = \rho_S^n(\rho_S^n(R2))\\
&\quad = \rho_S^n(([\![gSS^n]\!]\eta_S^n)(R2))\ (By\ \text{Eqn. (A.2)})\\
&\quad = \rho_S^n([\![gSS^n(R2)]\!]\eta_S^n) = \rho_S^n([\![m_1 + c_1]\!]\eta_S^n)\\
&\quad = ([\![gSS^n]\!]\eta_S^n)([\![m_1 + c_1]\!]\eta_S^n)\ (By\ \text{Eqn. (A.2)})\\
&\quad = ([\![gSS^n(m_1 + c_1)]\!]\eta_S^n) = [\![m + c]\!]\eta_S^n = m + c \qquad \text{(A.28)}
\end{aligned}$$

Eqns. (A.22), (A.27), (A.28), and (A.1) show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto m + c] \qquad \text{(A.29)}$$

$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u]\ (\textit{From (A.22)}) \qquad \text{(A.30)}$$

Eqns. (A.28)–(A.30), and (A.2) show that Eqn. (A.4) holds.

4. $I_{n+1} =$ `mov` **(Indirect(R1), Direct(R2)):** Because *R1* must hold an address, there are two cases that can arise.

(a) Suppose that $\rho_S^n(R1) = a$ and $\rho_S^n(R2) = u$, and $gSS^n(R1) = m + c$ and $gSS^n(R2) = v$.

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[a \mapsto u], \eta_D^n, \rho_D^n[a \mapsto u]) \qquad \text{(A.31)}$$

$$\begin{aligned}
&\mathsf{Specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov rw1,M-addr; mov r1,[rw1]; mov rw1,c;}\\
&\quad\texttt{add r1,rw1; mov [r1],v; mov rw1,0}\\
&[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[m+c \mapsto v]) \qquad \text{(A.32)}
\end{aligned}$$

Similar to the proof of Eqn. (A.15), we can prove that $u = v$. Similar to the proof of Eqn. (A.19), we can prove that $a = m + c$. These equalities, together with Eqns. (A.31),

(A.32), and (A.1), show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^{n}[m + c \mapsto v] \tag{A.33}$$

$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[a \mapsto u] \textit{ (From (A.31))} \tag{A.34}$$

The equalities $u = v$ and $a = m + c$, along with Eqns. (A.33), (A.34), and Eqn. (A.2), show that Eqn. (A.4) holds.

(b) Suppose that $\rho_S^n(R1) = a$ and $\rho_S^n(R2) = u$, and $gSS^n(R1) = m_1 + c_1$ and $gSS^n(R2) = m_2 + c_2$.

$$\begin{aligned}&\textsf{Specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov rw1,}M_1\texttt{-addr; mov r1,[rw1]; mov rw1,}c_1\texttt{;}\\ &\quad\texttt{add r1,rw1; mov rw1,}M_2\texttt{-addr; mov r2,[rw1]; mov rw1,}c_2\texttt{; add r2,rw1;}\\ &\quad\texttt{mov [r1],r2; mov rw1,0}\\ &[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[m_1 + c_1 \mapsto m_2 + c_2])\end{aligned} \tag{A.35}$$

Similar to the proof of Eqn. (A.19), we can prove that $a = m_1 + c_1$ and $u = m_2 + c_2$. These equalities, together with Eqns. (A.31), (A.35), and (A.1), show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^{n}[m_1 + c_1 \mapsto m_2 + c_2] \tag{A.36}$$

$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[a \mapsto u] \textit{ (By Eqn. (A.31))} \tag{A.37}$$

The equalities $a = m_1 + c_1$ and $u = m_2 + c_2$, along with Eqns. (A.36), (A.37), and Eqn. (A.2), show that Eqn. (A.4) holds.

5. $I_{n+1} =$ `add` **(Direct(R1), Direct(R2))** There are two cases that can arise.

(a) Suppose that $\rho_S^n(R1) + \rho_S^n(R2) = u$, and $gSS^n(R1) + gSS^n(R2) = v$.

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[R1 \mapsto u], \eta_D^n, \rho_D^n[R1 \mapsto u]) \tag{A.38}$$

$$\text{specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov r1,v}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto v]) \tag{A.39}$$

$$\begin{aligned} u &= \rho_S^n(R1) + \rho_S^n(R2) = \\ &= ([\![gSS^n]\!]\eta_S^n)(R1) + ([\![gSS^n]\!]\eta_S^n)(R2) \,(By\ \texttt{Eqn.}\ (A.2)) \\ &= ([\![gSS^n(R1)]\!]\eta_S^n) + ([\![gSS^n(R2)]\!]\eta_S^n) \\ &= ([\![gSS^n(R1) + gSS^n(R2)]\!]\eta_S^n) = [\![v]\!]\eta_S^n = v \end{aligned} \tag{A.40}$$

Eqns. (A.38)–(A.40), and (A.1) show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto v] \tag{A.41}$$

$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u] \,(\textit{From (A.38)}) \tag{A.42}$$

Eqns. (A.40)–(A.42), and (A.2) show that Eqn. (A.4) holds.

(b) Suppose that $\rho_S^n(R1) + \rho_S^n(R2) = u$, and $gSS^n(R1) + gSS^n(R2) = m + c$.

$$\begin{aligned} &\text{specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{mov rw1,M-addr; mov r1,[rw1]; mov rw1,c;} \\ &\quad \texttt{add r1,rw1; mov rw1,0} \end{aligned}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto m + c]) \tag{A.43}$$

$$\begin{aligned} u &= \rho_S^n(R1) + \rho_S^n(R2) = \\ &= ([\![gSS^n]\!]\eta_S^n)(R1) + ([\![gSS^n]\!]\eta_S^n)(R2) \,(By\ \texttt{Eqn.}\ (A.2)) \\ &= ([\![gSS^n(R1)]\!]\eta_S^n) + ([\![gSS^n(R2)]\!]\eta_S^n) \\ &= ([\![gSS^n(R1) + gSS^n(R2)]\!]\eta_S^n) \\ &= [\![m + c]\!]\eta_S^n = m + c \end{aligned} \tag{A.44}$$

Eqns. (A.38), (A.43), (A.44), and (A.1) show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto m + c] \tag{A.45}$$

$$\eta_S^{n+1} = \eta_S^n \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u] \,(\textit{From (A.38)}) \tag{A.46}$$

Eqns. (A.44)–(A.46), and (A.2) show that Eqn. (A.4) holds.

6. $I_{n+1} =$ `stackalloc` **(Direct(R1))**

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n[m \mapsto \mathfrak{m}], \rho_S^n[R1 \mapsto \mathfrak{m}], \eta_D^n[m \mapsto \mathfrak{m}], \rho_D^n[R1 \mapsto \mathfrak{m}]) \quad (A.47)$$

$$\text{Specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{stackalloc r1; mov rw1,M-addr;}$$
$$\texttt{mov [rw1],r1; mov rw1,0}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n[m \mapsto \mathfrak{m}], \eta_D^n[m \mapsto \mathfrak{m}], \overline{\rho}_D^n[R1 \mapsto \mathfrak{m}][\texttt{M-addr} \mapsto \mathfrak{m}]) \quad (A.48)$$

$\overline{\rho}_D^{n+1} \approx \rho_D^{n+1}$ because `M-addr` is in the special area of the memory map of the residual binary. This observation, along with Eqns. (A.47), (A.48), and (A.1), show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto m] \quad (A.49)$$

$$\eta_S^{n+1} = \eta_S^n[m \mapsto \mathfrak{m}] \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto \mathfrak{m}]$$
$$\textit{(By Eqn. (A.47))} \quad (A.50)$$

Note that the oracle supplies the correct m while updating gSS, and WiPEr uses m to obtain the correct `M-addr` to use in the residual code. Eqns. (A.49), (A.50), and (A.2) show that Eqn. (A.4) holds.

7. $I_{n+1} =$ `heapalloc` **(Direct(R1))**

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n[m \mapsto \mathfrak{m}], \rho_S^n[R1 \mapsto \mathfrak{m}], \eta_D^n[m \mapsto \mathfrak{m}], \rho_D^n[R1 \mapsto \mathfrak{m}]) \quad (A.51)$$

$$\text{Specialize}(I_{n+1}, gSS^n) \equiv J_{n+1} = \texttt{heapalloc r1; mov rw1,M-addr;}$$
$$\texttt{mov [rw1],r1; mov rw1,0}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n[m \mapsto \mathfrak{m}], \eta_D^n[m \mapsto \mathfrak{m}], \overline{\rho}_D^n[R1 \mapsto \mathfrak{m}][\texttt{M-addr} \mapsto \mathfrak{m}]) \quad (A.52)$$

$\overline{\rho}_D^{n+1} \approx \rho_D^{n+1}$ because `M-addr` is in the special area of the memory map of the residual binary. This observation, along with Eqns. (A.51), (A.52), and (A.1), show that Eqn. (A.3) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto m] \quad (A.53)$$

$$\eta_S^{n+1} = \eta_S^n[m \mapsto \mathfrak{m}] \textit{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto \mathfrak{m}]$$
$$\textit{(By Eqn. (A.51))} \quad (A.54)$$

Note that the oracle supplies the correct m while updating gSS, and WiPEr uses m to obtain the correct `M-addr` to use in the residual code. Eqns. (A.53), (A.54), and (A.2) show that Eqn. (A.4) holds. □

Bibliography

[1] `http://www.code.google.com`.

[2] `http://www.sourceforge.net`.

[3] `https://github.com/mitlm/mitlm`.

[4] `http://scikit-learn.org`.

[5] *Compilers: Principles, Techniques, and Tools*, chapter 8: Code Generation. Addison-Wesley, 2007.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[7] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Prog. Lang. Syst.*, 35(4), 1989.

[8] L. O. Andersen. Binding-time analysis and the taming of C pointers. In *PEPM*, 1993.

[9] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Univ. of Copenhagen, Copenhagen, Denmark, 1994.

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

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

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

[13] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *PLDI*, 1996.

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

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

[16] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[17] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI*, 2008.

[18] A. Bernat, K. Roundy, and B. Miller. Efficient, sensitivity resistant binary instrumentation. In *ISSTA*, 2011.

[19] D. Binkley. Precise executable interprocedural slices. *LOPLAS*, 2:31–45, 1993.

[20] D. Binkley and K. Gallagher. Program slicing. In *Advances in Computers, Vol. 43*. Academic Press, San Diego, CA, 1996.

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

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

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

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

[25] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL*, 1996.

[26] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. *Partial Evaluation*, 1110, 1996.

[27] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *CCS*, 2011.

[28] M. Das, T. Reps, and P. van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *PEPM*, 1995.

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

[30] B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. http://yices.csl.sri.com/.

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

[32] D. Engler, W. Hsieh, and F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL*, 1996.

[33] U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Workshop on New Security Paradigms*, 1999.

[34] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

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

[36] C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code-generator generator. *LOPLAS*, 1(3), 1992.

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

[38] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theor. Comp. Sci.*, 90(1), 1991.

[39] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[40] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *PLDI*, 1991.

[41] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[42] T. Gvero and V. Kunack. Synthesizing java expressions from free-form queries. In *OOPSLA*, 2015.

[43] J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark*, 1998.

[44] J. Henning. SPEC CPU2006 Benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

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

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

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

[48] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., 1993.

[49] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[50] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. *SIGPLAN Not.*, 30(4).

[51] A. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. *Perspectives of Systems Informatics*, 5947:185–192, 2010.

[52] P. Lee and M. Leone. Optimizing ML with run-time code generation. 1996.

[53] J. Lim and T. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Prog. Lang. Syst.*, 35(4), 2013.

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

[55] H. Massalin. Superoptimizer: A look at the smallest program. In *ASPLOS*, 1987.

[56] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *PLDI*, 1991.

[57] T. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *PEPM*, 1995.

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

[59] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. Alto: A link-time optimizer for the compaq alpha. *Softw. Pract. Exper.*, 31(1), 2001.

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

[61] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization: implementation and experimental study. In *Computer Languages*, 1998.

[62] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. FlexJava: language support for safe and modular approximate programming. In *FSE*, 2015.

[63] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[64] P. Phothilimthana, A. Thakur, R. Bodik, and D. Ghurjati. Scaling up superoptimization. In *ASPLOS*, 2016.

[65] P. Phothilimthana, A. Thakur, R. Bodik, and D. Ghurjati. GreenThumb: Superoptimizer construction framework. UCB/EECS-2016-8, University of California–Berkeley Tech Report, Feb. 2016. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-8.pdf`.

[66] N. Ramsey and M. Fernández. Specifying representations of machine instructions. *ACM Trans. Prog. Lang. Syst.*, 19(3), 1997.

[67] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, 2014.

[68] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from“big code". In *POPL*, 2015.

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

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

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

[72] A. Romano. *Methods for Binary Symbolic Execution*. PhD thesis, Stanford University, 2014.

[73] T. Rompf, A. Sujeeth, K. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision JIT compilers. In *PLDI*, 2014.

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

[75] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[76] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[77] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *ASPLOS*, 2013.

[78] U. Schultz, J. Lawall, and C. Consel. Automatic program specialization for Java. *TOPLAS*, 25(4), 2003.

[79] 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.

[80] A. Shali and W. Cook. Hybrid partial evaluation. In *OOPSLA*, 2011.

[81] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *S&P*, 2009.

[82] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *OOPSLA*, 2013.

[83] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *ATC*, 2012.

[84] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[85] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[86] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, 2007.

[87] A. Solar-Lezama, C. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[88] 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.

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

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

[91] V. Srinivasan and T. Reps. An improved algorithm for slicing machince code. In *OOPSLA*, 2016.

[92] V. Srinivasan, T. Sharma, and T. Reps. Speeding-up machine-code synthesis. In *OOPSLA*, 2016.

[93] V. Srinivasan, A. Vartanian, and T. Reps. Model-assisted machine-code synthesis. TR-1843, University of Wisconsin–Madison Tech Report, Feb. 2017.

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

[95] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In *Computer Aided Verif.*, 2010.

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

[97] A. Udupa, A. Raghavan, J. Deshmukh, S. Mador-Haim, M. Martin, and R. Alur. TRANSIT: Specifying protocols with concolic snippets. In *PLDI*, 2013.

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

[99] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *S&P*, 2015.

[100] 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.