Synthesis of Machine Code from Semantics *

Venkatesh Srinivasan
University of Wisconsin
venk@cs.wisc.edu

Thomas Reps
University of Wisconsin and Grammatech, Inc.
reps@cs.wisc.edu

In this paper, we present a technique to synthesize machine-code instructions from a semantic specification, given as a Quantifier-Free Bit-Vector (QFBV) logic formula. Our technique uses an instantiation of the Counter-Example Guided Inductive Synthesis (CEGIS) framework, in combination with search-space pruning heuristics to synthesize instruction-sequences. To counter the exponential cost inherent in enumerative synthesis, our technique uses a divide-and-conquer strategy to break the input QFBV formula into independent sub-formulas, and synthesize instructions for the sub-formulas. Synthesizers created by our technique could be used to create semantics-based binary rewriting tools such as optimizers, partial evaluators, program obfuscators/de-obfuscators, etc. Our experiments for Intel's IA-32 instruction set show that, in comparison to our baseline algorithm, our search-space pruning heuristics reduce the synthesis time by a factor of 473, and our divide-and-conquer strategy reduces the synthesis time by a further 3 to 5 orders of magnitude.

1. Introduction

The analysis of binaries has gotten an increasing amount of attention from the academic community in the last decade (e.g., see references in [Song et al. 2008, §7], [Balakrishnan and Reps 2010, §1], [Brumley et al. 2011, §1]). 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. Examples of semantics-based rewriting include offline optimization, partial evaluation [Jones et al. 1993], and binary translation [Bansal and Aiken 2008]. 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. 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 [Gulwani et al. 2011], or (ii) superoptimize instruction-sequences [Bansal and Aiken 2006]. A peephole-superoptimizer has the following type:

$$Superoptimize : InstrSequence \rightarrow InstrSequence$$

A machine-code synthesizer has the following type:

$$Synthesize : QFBVFormula \rightarrow InstrSequence$$

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 §7.) Moreover, search-space pruning techniques used by superoptimizers cannot be used by a machine-code synthesizer.

In this paper, we present a technique to synthesize straight-line machine-code instruction-sequences from a QFBV formula. The synthesized instruction-sequence implements the input QFBV formula (i.e., is equivalent to the QFBV formula). Our technique 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) [Brumley et al. 2011].

A machine-code synthesizer allows us to create multiple binary-rewriting tools that use the following recipe:

1. Convert instructions in the binary to QFBV formulas.
2. Use analysis results to transform QFBV formulas.
3. Use the synthesizer to produce an instruction-sequence that implements each transformed formula.

One tool that could be created using the above framework is an offline binary optimizer to improve unoptimized bi-

* Supported, in part, by NSF under grant CCF-0904371; by ONR under contract N00014-11-C-0447; by AFRL under contracts FA8650-10-C-7088 and FA8750-14-2-0270; and by DARPA under cooperative agreement HR0011-12-2-0012.Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors,and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc.,which has licensed elements of the technology reported in this publication.

[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.

naries. Analyses like Value-Set Analysis (VSA) [Balakrishnan and Reps 2010] and Def-Use Analysis (DUA) [Lim and Reps 2013] could be used in Step 2 to optimize QFBV formulas using information about constants, live registers and flags, etc. Another example is a machine-code partial evaluator. The partial evaluator can use the synthesizer to produce residual instructions for QFBV formulas specialized with respect to a partial static state. A machine-code synthesizer can also be used to generate obfuscated instruction-sequences for testing malware detectors [Christodorescu and Jha 2004], and to embed security policies in binaries [Erlingsson and Schneider 1999].

We present a tool, called McSynth, which synthesizes Intel IA-32 instructions from a QFBV formula. The core synthesis loop of our tool uses an instantiation of the Counter-Example Guided Inductive Synthesis (CEGIS) framework [Solar-Lezama 2008]. McSynth enumerates instruction-sequences, and uses CEGIS to find an instruction-sequence that implements the QFBV formula. To combat the exponential cost of explicit enumeration, McSynth uses two strategies based on the following observations about QFBV formulas for machine code. First, if an instruction-sequence uses (kills) a location (a register, flag, or memory location) that is not used (killed) by a QFBV formula φ, that instruction-sequence will not implement φ efficiently. Based on this observation, McSynth uses heuristics to prune away useless candidates from the synthesis search space. Second, a QFBV formula for an instruction-sequence (e.g., a basic block) typically has many inputs and many outputs (i.e., registers, flags, and memory locations.) Based on this observation, McSynth uses a *divide-and-conquer* strategy to break an input QFBV formula into sub-formulas, and synthesizes instructions for the sub-formulas.

The contributions of our work include the following:

- We present a technique for the synthesis of machine-code instructions from a QFBV formula. Our technique is parameterized by the ISA, and can be easily adapted to other semantic representations. Our technique is the first of its kind to be applied to a full ISA.
- The core synthesis loop of our technique is a new instantiation of the CEGIS framework (§4.1).
- We have developed heuristics based on the footprint of machine-code QFBV formulas to prune away useless candidates, and reduce the synthesis search-space (§4.2).
- 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 (§4.3). 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. In comparison to the x86 peephole superoptimizer [Bansal and Aiken 2006] (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, and an IA-32 slicer as clients of McSynth. (See §8.)

2. Background

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

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 (*i*, *j*, etc.). L[IA-32] has only one function symbol "Mem", which denotes memory. The syntax of L[IA-32] is defined as follows:

$$T \in \mathit{Term},\ \varphi \in \mathit{Formula},\ \mathit{FE} \in \mathit{FuncExpr}$$

$$c \in \mathit{Int32} = \{..., \text{-1}, 0, 1, ...\} \quad b \in \mathit{Bool} = \{\mathit{True}, \mathit{False}\}$$

$$I_{\mathit{Int32}} \in \mathit{Int32Id} = \{\mathit{EAX}, \mathit{ESP}, \mathit{EBP}, \ldots, i, j, ...\}$$

$$I_{\mathit{Bool}} \in \mathit{BoolId} = \{\mathit{CF}, \mathit{SF}, ...\} \quad F \in \mathit{FuncId} = \{\mathit{Mem}\}$$

$$\mathit{op} \in \mathit{BinOp} = \{+, -, ...\} \quad \mathit{bop} \in \mathit{BoolOp} = \{\wedge, \vee, ...\}$$

$$\mathit{rop} \in \mathit{RelOp} = \{=, \neq, <, >, ...\}$$

$$T ::= c \mid I_{\mathit{Int32}} \mid T_1\ \mathit{op}\ T_2 \mid \mathit{ite}(\varphi, T_1, T_2) \mid \mathit{FE}(T_1)$$

$$\varphi ::= b \mid I_{\mathit{Bool}} \mid T_1\ \mathit{rop}\ T_2 \mid \neg\varphi_1 \mid \varphi_1\ \mathit{bop}\ \varphi_2$$

$$\mathit{FE} ::= F \mid \mathit{FE}_1[T_1 \mapsto T_2]$$

The term of the form $\mathit{ite}(\varphi, T_1, T_2)$ represents an if-then-else expression. A *FuncExpr* of the form $\mathit{FE}[T_1 \mapsto T_2]$ denotes a *function-update* expression.

The function $\langle\langle\cdot\rangle\rangle$ converts an IA-32 instruction-sequence into a QFBV formula. The methodology for this conversion can be found elsewhere [Lim et al. 2011]. To write formulas that express state transitions, all *Int32Id*s, *BoolId*s, and *FuncId*s can be qualified by primes (e.g., *Mem*′). The QFBV formula for an instruction-sequence is a restricted 2-vocabulary formula of the form $\bigwedge_m (I'_m = T_m) \wedge \bigwedge_n (I'_n = \varphi_n) \wedge F' = \mathit{FE}$, where I'_m and I'_n range over the constant symbols for registers and flags, respectively. The primed vocabulary is the post-state vocabulary, and the unprimed vocabulary is the pre-state vocabulary. The QFBV formula for the IA-32 instruction "`push ebp`" is given below. This instruction pushes the

32-bit value in the frame-pointer register `ebp` onto the stack.

$$\langle\langle \texttt{push ebp} \rangle\rangle \equiv ESP' = ESP - 4 \wedge \quad (1)$$
$$Mem' = Mem[ESP - 4 \mapsto EBP]$$

In this section, and in the rest of the paper, we will show only the relevant portions of QFBV formulas. QFBV formulas actually contain identity conjuncts (of the form $I' = I$ or $F' = F$) for constants or functions that are *unmodified*. Because we do not want the synthesizer output to be restricted to an instruction-sequence that uses a specific number of bytes, we drop the conjunct of the form $EIP' = T$. (*EIP* is the program counter for IA-32.) The QFBV formula for the `push` instruction actually looks like

$$ESP' = ESP - 4 \wedge EAX' = EAX \wedge \ldots$$
$$CF' = CF \wedge \ \ldots \ \wedge Mem' = Mem[ESP - 4 \mapsto EBP]$$

and omits the conjunct $EIP' = EIP + 1$.

2.2 Semantics

Intuitively, a QFBV formula represents updates made by an instruction to the machine state. 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 implementation 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. An IA-32 machine-state is a triple of the form:

$$\langle RegMap, FlagMap, MemMap \rangle$$

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. (1) is

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

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

3. Overview

Given a QFBV formula φ, McSynth synthesizes an instruction-sequence for φ in the following way:

1. McSynth 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. McSynth attempts to find an instantiation of a candidate templatized instruction-sequence that is logically equivalent to φ using CEGIS. If an instantiation is found, McSynth returns it. Otherwise, the next templatized sequence is considered.
3. McSynth uses heuristics based on the footprints of QFBV formulas to prune away useless candidates during enumeration.

To counter the exponential cost of brute-force enumeration, McSynth uses a divide-and-conquer strategy; McSynth breaks φ into independent sub-formulas and synthesizes instructions for the sub-formulas. This section presents an example to illustrate our approach. First, we illustrate McSynth's CEGIS loop along with McSynth's footprint-based search-space pruning, and then we illustrate McSynth's divide-and-conquer strategy.

3.1 CEGIS + Footprint-Based Pruning

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

$$\varphi \equiv ESP' = ESP - 4 \ \wedge \ EBP' = ESP - 4 \wedge \quad (2)$$
$$Mem' = Mem[ESP - 4 \mapsto EBP].$$

McSynth starts enumerating templatized one-instruction sequences. Let us assume that the first candidate is $C_1 \equiv$ "`mov eax, <Imm32>`". C_1 is a template to move a 32-bit constant value into the `eax` register. McSynth converts C_1 into a QFBV formula ψ_1. (McSynth uses free constants for template operands.)

$$\psi_1 \equiv \langle\langle C_1 \rangle\rangle \equiv EAX' = i$$

Before processing ψ_1 via CEGIS, McSynth checks if ψ_1 can be pruned away. If an instruction-sequence uses (modifies) a location that is not used (modified) by φ, intuitively, the instruction-sequence can never implement φ in an efficient way. McSynth computes the *abstract semantic USE-footprint* ($\text{SFP}^{\#}_{\text{USE}}$), and the *abstract semantic KILL-footprint* ($\text{SFP}^{\#}_{\text{KILL}}$) for φ and ψ_1. $\text{SFP}^{\#}_{\text{USE}}$ ($\text{SFP}^{\#}_{\text{KILL}}$) is an over-approximation of the locations (registers, flags, or memory) that *might* be used (modified) by a QFBV formula. Concretely, $\text{SFP}^{\#}_{\text{USE}}$ ($\text{SFP}^{\#}_{\text{KILL}}$) for a QFBV formula is a set of constant symbols and/or function symbols from the vocabulary of the QFBV formula. Symbols in $\text{SFP}^{\#}_{\text{KILL}}$ are primed. $\text{SFP}^{\#}_{\text{USE}}$ and $\text{SFP}^{\#}_{\text{KILL}}$ for φ and ψ_1 are given below.

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

$\text{SFP}^{\#}_{\text{USE}}(\psi_1)$ is $\emptyset$ because ψ_1 does not use any registers, flags, or memory locations. (Identity conjuncts like $EBX' = EBX$ do not contribute to $\text{SFP}^{\#}_{\text{USE}}$ and $\text{SFP}^{\#}_{\text{KILL}}$.) $\text{SFP}^{\#}_{\text{KILL}}(\psi_1)$ is $\{EAX'\}$ because ψ_1 might change the value of the `eax` register. $\text{SFP}^{\#}_{\text{KILL}}(\varphi)$ contains Mem' because φ might modify *some* memory location. Because $\text{SFP}^{\#}_{\text{KILL}}(\psi_1) \not\supseteq \text{SFP}^{\#}_{\text{KILL}}(\varphi)$, C_1 might modify a location that is unmodified by φ, and thus, it cannot be equivalent to φ. Consequently, McSynth discards C_1. Moreover, regard-

less of the instruction-sequence that is appended to C_1, the resulting instruction-sequence will 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 McSynth has only *useful-prefixes* as its prefix.

Suppose that McSynth chooses $C_2 \equiv$ "`mov ebp, esp`" as the next candidate. C_2 copies a 32-bit value from the stack-pointer register `esp` to the frame-pointer register `ebp`. The QFBV formula ψ_2 for C_2, and the SFP$^{\#}$ sets for ψ_2 are

$$\psi_2 \equiv \langle\langle C_2 \rangle\rangle \equiv EBP' = ESP$$
$$\text{SFP}^{\#}{}_{\text{USE}}(\psi_2) = \{ESP\} \qquad \text{SFP}^{\#}{}_{\text{KILL}}(\psi_2) = \{EBP'\}.$$

Because $\text{SFP}^{\#}{}_{\text{KILL}}(\psi_2) \subseteq \text{SFP}^{\#}{}_{\text{KILL}}(\varphi)$, and $\text{SFP}^{\#}{}_{\text{USE}}(\psi_2) \subseteq \text{SFP}^{\#}{}_{\text{USE}}(\varphi)$, McSynth proceeds to process ψ_2 via CEGIS.

Given a templatized candidate C, and a finite set of tests $\mathscr{T}$ (where, a *test* is a $\langle$pre-state, post-state$\rangle$ pair), McSynth performs the following steps in its core CEGIS loop:

1. McSynth attempts to find values for the template operands in C, such that the instantiated sequence C_{conc} and φ produce identical post states for each test in test set $\mathscr{T}$. If such an instance cannot be found, McSynth discontinues further processing of C via CEGIS, but retains C as a useful-prefix.
2. If McSynth finds an instance C_{conc} that works for the finite set of tests $\mathscr{T}$, McSynth uses an SMT solver to determine whether $\langle\langle C_{conc} \rangle\rangle$ is equivalent to φ. If the check succeeds, McSynth returns C_{conc}.
3. If the check fails, McSynth adds the counter-example produced by the SMT solver to $\mathscr{T}$, and repeats Step 1.

Suppose that $\mathscr{T}$ has only one test, $\langle \sigma_1, \sigma_1' \rangle$.

$$\sigma_1 \equiv \langle [\text{ESP} \mapsto 100][\text{EBP} \mapsto 200], [\,], [\,] \rangle$$
$$\sigma_1' \equiv \langle [\text{ESP} \mapsto 96][\text{EBP} \mapsto 96], [\,], [96 \mapsto 200] \rangle$$

One can see that $\langle \sigma_1, \sigma_1' \rangle \models \varphi$. McSynth evaluates ψ_2 with respect to $\langle \sigma_1, \sigma_1' \rangle$ (i.e., checks satisfiability), and finds that $\langle \sigma_1, \sigma_1' \rangle \not\models \psi_2$. Hence, McSynth discontinues further processing of ψ_2 via CEGIS, but retains C_2 as a useful-prefix. McSynth uses C_2 as a prefix when enumerating future candidates.

Suppose that McSynth has exhausted all one-instruction candidates, and considers $C_3 \equiv$ "`push ebp; mov ebp, <Imm32>`" as the next candidate. C_3 is a template to save the frame-pointer register `ebp` on the stack, and move a 32-bit constant value into `ebp`. The QFBV formula ψ_3 for C_3 is

$$\psi_3 \equiv \langle\langle C_3 \rangle\rangle \equiv ESP' = ESP - 4 \wedge EBP' = i \wedge Mem' = Mem[ESP - 4 \mapsto EBP].$$

By simplifying ψ_3 with respect to $\langle \sigma_1, \sigma_1' \rangle$, X produces the simplified formula $\psi_3^{\langle \sigma_1, \sigma_1' \rangle}$ shown below.

$$\psi_3^{\langle \sigma_1, \sigma_1' \rangle} \equiv 96 = 96 \wedge 96 = i \wedge Mem' = Mem[96 \mapsto 200] \wedge Mem(96) = 0 \wedge Mem'(96) = 200$$

(To see how McSynth generates the constraints $Mem(96) = 0$ and $Mem'(96) = 200$, see §4.1.2.) McSynth checks the satisfiability of $\psi_3^{\langle \sigma_1, \sigma_1' \rangle}$ using an SMT solver. The solver says that $\psi_3^{\langle \sigma_1, \sigma_1' \rangle}$ is satisfiable, and produces the satisfying assignment $[i \mapsto 96]$. Substituting the assignment in C_3, McSynth obtains the *concrete* instruction-sequence $C_3^{conc} \equiv$ "`push ebp; mov ebp, 96`", and its corresponding QFBV formula, ψ_3^{conc}. A concrete instruction-sequence is a sequence of instructions that do not have any template operands (or holes).

$$\psi_3^{conc} \equiv ESP' = ESP - 4 \wedge EBP' = 96 \wedge Mem' = Mem[ESP - 4 \mapsto EBP]$$

φ and ψ_3^{conc} produce identical post-states for the test case $\langle \sigma_1, \sigma_1' \rangle$. Now that McSynth has found a candidate that is equivalent to φ with respect to one test case, McSynth checks if the candidate is equivalent to φ for all possible test cases. McSynth checks the equivalence of φ and ψ_3^{conc} using an SMT solver. The solver says that the two formulas are not equivalent, and produces a counter-example $\langle \sigma_2, \sigma_2' \rangle$. McSynth adds $\langle \sigma_2, \sigma_2' \rangle$ to $\mathscr{T}$.

$$\sigma_2 \equiv \langle [\text{ESP} \mapsto 104][\text{EBP} \mapsto 200], [\,], [\,] \rangle$$
$$\sigma_2' \equiv \langle [\text{ESP} \mapsto 100][\text{EBP} \mapsto 100], [\,], [100 \mapsto 200] \rangle$$

Eventually, McSynth enumerates the candidate $C_4 \equiv$ "`push ebp; mov ebp, esp`", and obtains the corresponding QFBV formula ψ_4. McSynth simplifies ψ_4 with respect to $\langle \sigma_1, \sigma_1' \rangle$ and $\langle \sigma_2, \sigma_2' \rangle$ to produce the simplified formulas $\psi_4^{\langle \sigma_1, \sigma_1' \rangle}$ and $\psi_4^{\langle \sigma_2, \sigma_2' \rangle}$, respectively. McSynth checks the satisfiability of $\psi_4^{\langle \sigma_1, \sigma_1' \rangle} \wedge \psi_4^{\langle \sigma_2, \sigma_2' \rangle}$ using an SMT solver. The solver says that the formula is satisfiable. McSynth then checks whether φ and ψ_4 are equivalent, and subsequently returns C_4.

3.2 Divide-and-Conquer

For the running example, the synthesis terminates in a few minutes. However, for bigger QFBV formulas, the exponential cost of enumeration causes the synthesis algorithm to run for hours or days. To overcome this problem, McSynth uses a divide-and-conquer strategy. Before synthesizing instructions for the full φ, McSynth attempts to break φ into a sequence of independent sub-formulas. If φ can be split into sub-formulas, McSynth synthesizes instructions for the sub-formulas, appends the synthesized instructions, and returns the result. One possible way to split φ is as $\langle \varphi_1, \varphi_2, \varphi_3 \rangle$, where

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

However, φ_2 and φ_3 both use *ESP*, which is killed by φ_1. (Note that to compare the used and killed locations, the primes are dropped from primed symbols.) If McSynth were to synthesize instructions for φ_1, φ_2, and φ_3, and append them in that order, the result will not be equivalent to φ. We call such a split *illegal*. Another possible way to split φ is

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

In this split, no sub-formula kills a primed location whose unprimed namesake is used by a successor sub-formula. This condition characterizes a *legal* split. McSynth synthesizes the following instructions for the sub-formulas:

$$C_1 \equiv \texttt{mov [esp - 4], ebp}$$
$$C_2 \equiv \texttt{lea ebp, [esp - 4]}$$
$$C_3 \equiv \texttt{lea esp, [esp - 4]}$$

The divide-and-conquer strategy reduces the synthesis time for φ from a few minutes to a few seconds. For the running example, the reduction in synthesis time is small, but for larger QFBV formulas, this strategy brings down the synthesis time by *several orders of magnitude*.

3.3 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, 4294967296`". 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.

4. 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.

4.1 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.

Algorithm 1 Strawman algorithm to synthesize instructions from a QFBV formula

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
```

4.1.1 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. 1 show this strawman algorithm.

McSynth uses an SMT solver to check the satisfiability of a QFBV formula. (Validity queries are expressed as 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 counter-example to validity.

One optimization is to use the counter-examples 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 $\mathcal{T}$. (Note that $\langle \sigma , \sigma' \rangle \models \varphi$, for all $\langle \sigma , \sigma' \rangle \in \mathcal{T}$.) McSynth evaluates ψ with respect to each test $\langle \sigma , \sigma' \rangle$ in $\mathcal{T}$ to check if $\langle \sigma , \sigma' \rangle \models \psi$ (i.e., φ and ψ produce identical post-states for each test in $\mathcal{T}$). 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. 1. In Alg. 1, `TestsPass` evaluates ψ with respect to each test in $\mathcal{T}$.

4.1.2 CEGIS

The search space of Alg. 1 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 [Solar-Lezama et al. 2006]. CEGIS is a popular syn-

thesis 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 counter-example 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 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 $\psi^{\mathscr{T}} \equiv$ **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 subsection, 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.[2]

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

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

$$\psi_1^{\langle \sigma, \sigma' \rangle} \equiv \psi_1 \wedge \chi(\sigma, 0) \wedge \chi(\sigma', 1) \tag{3}$$

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.

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

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.

$$\psi_1^{\langle \sigma, \sigma' \rangle} \equiv \psi_1 \wedge \chi_{RegFlag}(\sigma, 0) \wedge \chi_{RegFlag}(\sigma', 1) \tag{4}$$

However, Eqns. (2) and (3) are not equisatisfiable. This approach results in false positives. Because Eqn. (4) 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), 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:

$$2 = Mem(100) \wedge Mem' = Mem[100 \mapsto 1] \tag{5}$$

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

The only memory location that is accessed or updated in Eqn. (5) is 100. For this location, SimplifyWithTest generates the following constraints from $\langle \sigma, \sigma' \rangle$.

$$Mem(100) = 2 \wedge Mem'(100) = 2 \quad (6)$$

SimplifyWithTest conjoins Eqn. (5) and Eqn. (6) to obtain

$$\psi_1^{\langle \sigma,\sigma' \rangle} \equiv Mem' = Mem[100 \mapsto 1] \wedge Mem(100) = 2 \wedge Mem'(100) = 2. \quad (7)$$

The formulas in Eqn. (3) and Eqn. (7) 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 [<Imm32>], ebx`". C_2 is a template to copy a 32-bit 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 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:

$$2 = Mem(100) \wedge Mem' = Mem[i \mapsto 1] \quad (8)$$

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

$$Mem(100) = 2 \wedge i \neq 100 \Rightarrow Mem(i) = 0 \quad (9)$$

To constrain the post-state value at location i, McSynth generates the following constraint from the memory map $[100 \mapsto 2][200 \mapsto 1]$ in σ':

$$Mem'(100) = 2 \wedge Mem'(200) = 1 \wedge (i \neq 100 \wedge i \neq 200) \Rightarrow Mem'(i) = 0 \quad (10)$$

SimplifyWithTest conjoins Eqns. (8)–(10), 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. 2. 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

Algorithm 2 Algorithm SimplifyWithTest

Input: ψ, $\langle \sigma, \sigma' \rangle$
Output: $\psi^{\langle \sigma,\sigma' \rangle}$

```
1: ψ^⟨σ,σ'⟩ ← Simplify(ψ ∧ χ_RegFlag(σ, 0) ∧ χ_RegFlag(σ', 1))
2: concLocs ← ConcLocs(ψ^⟨σ,σ'⟩)
3: concMemConstr ← true
4: for each a in concLocs do
5:     val ← Lookup(σ, a)
6:     val' ← Lookup(σ', a)
7:     concMemConstr ← concMemConstr ∧ Mem(a) = val ∧ Mem'(a) = val'
8: end for
9: symLocs ← SymLocs(ψ^⟨σ,σ'⟩)
10: if symLocs = ∅ then
11:     return Simplify(ψ^⟨σ,σ'⟩ ∧ concMemConstr)
12: end if
13: return Simplify(ψ^⟨σ,σ'⟩ ∧ concMemConstr ∧ SymMem
    Constr (symLocs, σ, Mem) ∧ SymMemConstr(symLocs,
    σ', Mem'))
```

Algorithm 3 Algorithm CEGIS

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:     ψ^T ← true
3:     for each test-case ⟨σ, σ'⟩ ∈ T do
4:         ψ^T ← ψ^T ∧ SimplifyWithTest(ψ, ⟨σ, σ'⟩)
5:     end for
6:     model = SAT(ψ^T)
7:     if model = ⊥ then
8:         return FAIL
9:     end if
10:    ψ_conc ← Substitute(ψ, model)
11:    model ← SAT(¬(φ ⇔ ψ_conc))
12:    if model = ⊥ then
13:        return Substitute(C, model)
14:    end if
15:    T ← T ∪ model
16: end while
```

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.

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 counter-example returned by the solver to $\mathscr{T}$, and searches for another concrete instruction-sequence that passes the tests. Alg. 3 show McSynth's CEGIS loop. In Alg. 3, the overloaded func-

Algorithm 4 Algorithm Synthesize

Input: φ
Output: C_{conc} or FAIL

```
1:  T ← {⟨σ0, σ0'⟩}
2:  prefixes ← {ϵ}
3:  while prefixes ≠ ∅ do
4:    for each prefix p ∈ prefixes do
5:      prefixes ← prefixes − {p}
6:      for each templatized instruction i in the ISA do
7:        C ← Append(p, i)
8:        ψ ← ⟪C⟫
9:        if SFP#USE(ψ) ⊈ SFP#USE(φ) ∨ SFP#KILL(ψ) ⊈
          SFP#KILL(φ) then
10:         continue
11:       end if
12:       prefixes ← prefixes ∪ {C}
13:       ret = CEGIS(φ, C, ψ, T)
14:       if ret ≠ FAIL then
15:         return ret
16:       end if
17:     end for
18:   end for
19: end while
20: return FAIL
```

tion `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. 4. In the algorithm, ϵ denotes an instruction-sequence with no instructions, and `Append` appends an instruction to an instruction-sequence.

4.2 Pruning the Synthesis Search-Space

ISAs such as Intel's IA-32 have around 43,000 unique templatized instructions. For IA-32, Alg. 4 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 Alg. 4 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.

4.2.1 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 that *are* used by the QFBV formula for *some* input. The *semantic-KILL-footprint* (SFP_{KILL}) is the set of concrete locations (represented as constant symbols) that *are* modified by the QFBV formula for *some* input. For the formula in Eqn. (2), SFP_{USE} and SFP_{KILL} are shown below (with a minor abuse of notation).

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

$0'$, $1'$, $2'$, ... 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 1.

$$\text{SFP}^{\#}_{\text{USE}}(c) = \begin{cases} \{c\} & \text{if } c \in RF \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{SFP}^{\#}_{\text{USE}}(Mem(t)) = \{Mem\} \cup \text{SFP}^{\#}_{\text{USE}}(t), \text{where } t \in T$$
$$\text{SFP}^{\#}_{\text{USE}}(c' = c) = \emptyset, \text{ where } c' \in RF', c \in RF \quad (11)$$
$$\text{SFP}^{\#}_{\text{USE}}(Mem' = Mem) = \emptyset \quad (12)$$

$$\text{SFP}^{\#}_{\text{KILL}}(c') = \begin{cases} \{c'\} & \text{if } c' \in RF' \\ \emptyset & \text{otherwise} \end{cases}$$
$$\text{SFP}^{\#}_{\text{KILL}}(Mem') = \{Mem'\}$$
$$\text{SFP}^{\#}_{\text{KILL}}(c' = c) = \emptyset, \text{ where } c' \in RF', c \in RF \quad (13)$$
$$\text{SFP}^{\#}_{\text{KILL}}(Mem' = Mem) = \emptyset \quad (14)$$

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. (11)–(14) 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:

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

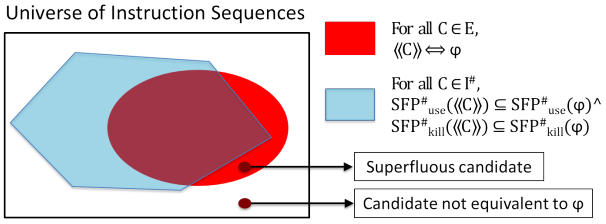

Figure 1: Depiction of the set $I^{\#}$.

The set $I^{\#}$ is depicted in Fig. 1 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. (2) is "`mov eax, ebx`".
2. A candidate C that satisfies the following properties:

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

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

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. (2), "`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`.

4.2.2 Useless-Prefix

Because a location modified (used) by a QFBV formula cannot be "un-modified" ("un-used"), 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 1. *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. 4. Search-space pruning is carried out in Line 9 of Alg. 4.

4.3 Divide-and-Conquer

The candidate enumeration in Alg. 4 has exponential cost. Synthesizing an instruction-sequence that consists of a single instruction takes less than a second; synthesizing a two-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 uses a recursive procedure (DivideAndConquer) 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 DivideAndConquer is shown in Alg. 5. DivideAndConquer has an unusual structure because the base case (Line 13) appears after the recursive calls (Lines 3 and 7). 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 $\text{Synthesize}_{<max>}$ be a version of Alg. 4 that

Algorithm 5 Algorithm DivideAndConquer

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

```
1:  splits ← EnumerateSplits(φ)
2:  for each split ⟨φ1, φ2⟩ ∈ splits do
3:      ret1 ← DivideAndConquer(φ1, max)
4:      if ret1 = FAIL then
5:          continue
6:      end if
7:      ret2 ← DivideAndConquer(φ2, max)
8:      if ret2 ≠ FAIL then
9:          ret ← Concat(ret1, ret2)
10:         return ret
11:     end if
12: end for
13: return Synthesize<max>(φ)
```

is parameterized by the maximum length of candidates to consider during enumeration (*max*). $\text{Synthesize}_{<max>}$ returns FAIL if Alg. 4 cannot find an instruction-sequence with length $\leq$ *max* that implements φ. DivideAndConquer uses EnumerateSplits to enumerate all legal splits of φ.

Definition 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*

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

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*

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

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:

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

For each legal split $\langle\varphi_1, \varphi_2\rangle$ of φ, DivideAndConquer 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, DivideAndConquer 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).

Pseudo-code for EnumerateSplits is shown in Alg. 6. We illustrate the algorithm with the following QFBV formula:

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

Algorithm 6 Algorithm EnumerateSplits

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: **if** `DropPrimes`(SFP$^{\#}_{\text{KILL}}(\varphi_1)$) $\cap$ SFP$^{\#}_{\text{USE}}(\varphi_2) \neq \emptyset$ **then**
14: **continue**
15: **end if**
16: splits $\leftarrow$ splits $\cup \langle\varphi_1, \varphi_2\rangle$
17: **end for**
18: **end for**
19: **return** splits

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 QFBV formula, a set of killed registers and flags, a sequence of memory locations, and returns a QFBV 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

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

$$\varphi_2 \equiv EAX' = EBX \wedge Mem' = Mem[ESP - 12 \mapsto EDI]$$

The legality of a split is checked in Line 13. The split $\langle\varphi_1, \varphi_2\rangle$ shown above constitutes an illegal split, and is discarded. In addition to divide-and-conquer, our implementation of Alg. 5 uses memoization to avoid processing a subformula more than once; the result for a sub-formula is either its synthesized code-sequence or FAIL. Consequently, Alg. 5 really uses a form of dynamic programming. Practical values for Synthesize's parameter *max* are 1 or 2. For these values, DivideAndConquer will either return FAIL or the synthesized instruction-sequence in a few minutes or hours (cf. Fig. 3). If DivideAndConquer returns FAIL, McSynth uses

Algorithm 7 Algorithm McSynth

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

Alg. 4 to synthesize instructions for φ. The full synthesis algorithm used by McSynth is given in Alg. 7.

Lemma 1. *Alg. 4 is sound. (The instruction-sequence returned by Alg. 4 is logically equivalent to the input QFBV formula φ.)*

Proof. By lines 11-14 of Alg. 3, 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 §4.1.2 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 φ_1 is equivalent to $\langle\langle I_1\rangle\rangle$, and φ_2 is equivalent to $\langle\langle I_2\rangle\rangle$, then φ is equivalent to $\langle\langle I_1; I_2\rangle\rangle$.*

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 2. ***Soundness.*** *Alg. 7 is sound.*

Proof. Follows from Lemmas 1 and 2. □

Theorem 3. ***Completeness.*** *Modulo SMT timeouts, if there exists a non-superfluous instruction-sequence I that is equivalent to φ, then Alg. 7 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 (§3.3), 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. 7 might not terminate.

4.4 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.)

Scratch′ is added to $\text{SFP}^{\#}{}_{\text{KILL}}(\varphi)$ in Line 9 of Alg. 4. Consequently, instruction-sequences that use registers in Scratch to hold temporary computations are not pruned away. (Note that instruction-sequences that have upwards-exposed uses of registers in Scratch are still pruned away.) The only other change required is that just before line 13 of Alg. 4, all conjuncts that update registers in Scratch′ are dropped from φ and ψ. (There is one additional minor technical point: to make the Input/Output specification of Alg. 3 correct, all conjuncts of the form $S' = T$, for $S' \in$ Scratch′, are dropped in the two occurrences of $\langle\langle \cdot \rangle\rangle$.)

Quality of synthesized code. Certain clients might want the synthesized code to possess a certain "quality" (small size, short runtime, less energy consumption, etc.). For example, a superoptimizer would like the synthesized code to have a shorter 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. 8. In Alg. 8, 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. 4, 5, and 7 to implement a biased synthesizer:

Algorithm 8 Algorithm Biased X

Input: φ, f, *timeout*
Output: C_{conc} or FAIL

```
 1: seen ← ∅
 2: min ← MaxFn(f)
 3: minSeq ← ϵ
 4: while ! TimeoutExpired(timeout) do
 5:    ret = X(φ, seen)
 6:    if ret = FAIL then
 7:       return FAIL
 8:    end if
 9:    val ← f(ret)
10:    if val < min then
11:       min ← val
12:       minSeq ← ret
13:    end if
14:    seen ← seen ∪ ret
15: end while
16: return minSeq
```

- Algs. 4, 5, and 7 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 14 in Alg. 4, and after line 9 in Alg. 5, respectively:

```
if ret ∈ seen then
   continue
end if
```

5. Implementation

McSynth uses Transformer Specification Language (TSL) [Lim and Reps 2013] 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 [Lim et al. 2011]. McSynth uses ISAL [Lim and Reps 2013, §2.1] to generate the templatized instruction pool for synthesis. McSynth uses Yices [Dutertre and de Moura 2006] 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.

X deviates slightly from the idealized collection of templatized instructions discussed in §3.3. 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 [Ramsey and Fernández 1997] 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

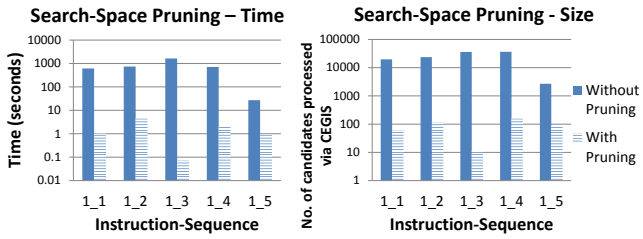

Figure 2: Comparison of synthesis time and search-space size with and without footprint-based search-space pruning.

the corpus lead to the template "`mov eax, <imm>`".) The random generator produces a corpus with a wide variety in the instructions ([Lim and Reps 2013], Fig. 19).

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

6. Experiments

We tested McSynth on QFBV formulas obtained from instruction-sequences from the SPECINT 2006 benchmark suite [Henning 2006]. 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 (§4.2)?
- What is the reduction in synthesis time caused by McSynth's divide-and-conquer strategy (§4.3)?

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. 2. We have presented such results only for QFBV formulas obtained from instruction-sequences of length 1 because synthesis of 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. 4. 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.

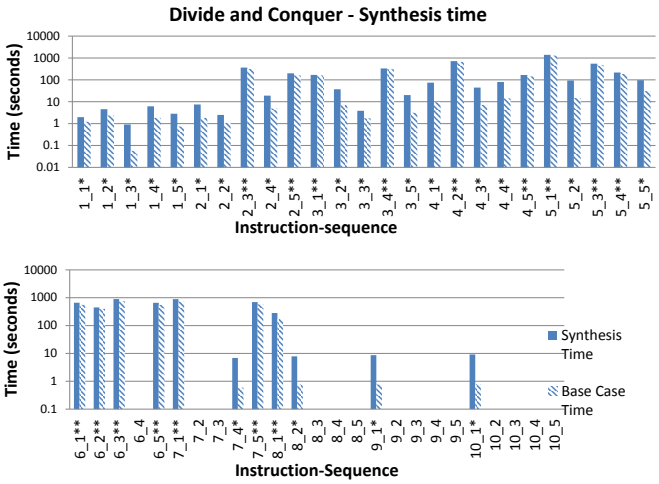

Figure 3: Synthesis times using the divide-and-conquer strategy.

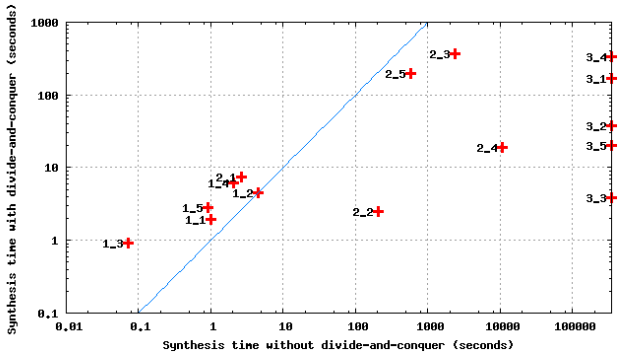

Figure 4: Synthesis times with and without divide-and-conquer.

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. "Synthesis Time" is the total CPU time spent by Alg. 7. "Base Case Time" is the time spent in the base case (Line 12 of Alg. 5). The QFBV formulas for which FAIL was returned in Lines 1 and 5 of Alg. 7 do not have synthesis times reported in Fig. 3. The QFBV formulas for which Alg. 7 returned a result in Line 3 (i.e., *max* = 1 was sufficient for synthesis) are marked by *, and those for which Alg. 7 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. 4, the total CPU time spent by Alg. 4 is compared with the total CPU time spent by Alg. 7. 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. 4 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's divide-and-conquer strategy failed for most of the instruction-sequences of lengths 8 through 10 primarily because of the following reasons:

- The QFBV formula had a term or a sub-formula that can be implemented only by three or more instructions.
- All terms and sub-formulas can be implemented by two instructions or less, but the terms and sub-formulas access or update several independent memory locations. However, because $\text{SFP}^{\#}_{\text{USE}}$ and $\text{SFP}^{\#}_{\text{KILL}}$ do not distinguish between memory locations, splits that are actually legal are conservatively disregarded by Line 13 of Alg. 6.

We believe that a more accurate test for legality of splits will reduce the number of failures in Lines 1 and 5 of Alg. 7, and hope to develop such a test in future work.

For the 36 QFBV formulas for which McSynth synthesized code, Table 1 compares the length of the synthesized instruction-sequence to the length of the corresponding original instruction-sequence. Table 1 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).

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

Table 1: Comparison of the lengths of synthesized and original instruction-sequences.

7. 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* [Solar-Lezama et al. 2005, 2006, 2007, 2008]. 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 [Gulwani et al. 2011]. 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 [Udupa et al. 2013].

Superoptimization. Superoptimization aims at finding an optimal instruction-sequence for a target instruction-sequence [Massalin 1987; Joshi et al. 2002; Bansal and Aiken 2006, 2008; Schkufza et al. 2013]. Peephole superoptimization [Bansal and Aiken 2006] uses "peepholes" to harvest target instruction-sequences, and replace them with equivalent instruction-sequences that have a lower cost. Superoptimization can be viewed as a constrained machine-code synthesis problem, where cost and correctness are constraints to the synthesizer. 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, a superoptimizer can be constructed as follows:

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

However, a synthesizer cannot be constructed from a superoptimizer.

Techniques used by superoptimizers to prune the search space (e.g., testing a candidate and the target instruction-sequence by executing tests on bare metal, canonicalizing instruction-sequences before synthesis, etc.) cannot be used by McSynth because McSynth does not have a specification of the goal as an instruction-sequence. For this reason, we developed new approaches to prune the synthesis search-space.

Applications of machine-code synthesis. Partial Evaluation [Jones et al. 1993] is a program-specialization technique that optimizes a program with respect to certain static inputs. A machine-code synthesizer could play an important role in a machine-code partial evaluator. When the partial evaluator specializes the QFBV formula of a basic block with respect to a partial static state, the synthesizer can be used to synthesize instructions for the specialized QFBV formula.

Semantics-based malware detectors use instruction semantics to detect malicious behavior in binaries [Christodorescu and Jha 2004; Christodorescu et al. 2005]. A machine-code synthesizer can be used to obfuscate instruction-sequences in malware binaries to either (i) suppress the malware signature to allow it to escape detection, or (ii) generate tests for a malware detector to improve detection algorithms.

By introducing suitable biases into a machine-code synthesizer, it may also be possible to use it to de-obfuscate instruction-sequences in malware binaries.

8. Conclusions and Future Work

In this paper, we described an algorithm to synthesize straight-line machine-code instruction-sequences from QFBV formulas. We presented McSynth, a tool that synthesizes IA-32 instructions from a QFBV formula. Our experiments show that, in comparison to our baseline algorithm, McSynth's footprint-based search-space pruning reduces the synthesis time by a factor of 473, and McSynth's strategy of divide-and-conquer plus memoization reduces the synthesis time by a further 3 to 5 orders of magnitude.

We have built an IA-32 partial evaluator using McSynth, and have used the partial evaluator to partially evaluate application binaries (interpreters, image filters, etc.) with respect to static inputs. We have also used the partial evaluator to extract the compression component of the `bzip2` binary.

In addition, we have used McSynth to improve the accuracy of machine-code slicing. Instructions that perform multiple updates to the state (e.g., `push`, `leave`, etc.) reduce the accuracy of machine-code slicing. We used McSynth to "untangle" such instructions by synthesizing equivalent instruction-sequences.

One possible direction for future work is to use McSynth to obfuscate/de-obfuscate instruction-sequences in malware. A second direction would be to adapt the algorithms in McSynth 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 third direction is to create a more accurate test of legality of splits by devising a finer-grained handling of *Mem* in $\text{SFP}^{\#}{}_{\text{USE}}$ and $\text{SFP}^{\#}{}_{\text{KILL}}$.

References

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

S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI*, 2008.

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

M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA*, 2004.

M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *S&P*, 2005.

B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. http://yices.csl.sri.com/.

U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Workshop on New Security Paradigms*, pages 87–95, 1999.

S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

J. Henning. SPEC CPU2006 Benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., 1993.

R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

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

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

H. Massalin. Superoptimizer: A look at the smallest program. In *ASPLOS*, 1987.

N. Ramsey and M. Fernández. Specifying representations of machine instructions. *TOPLAS*, 19(3), 1997.

E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Univ. of Calif., Berkeley, CA, 2008.

A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, 2007.

A. Solar-Lezama, C. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

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 *International Conference on Information Systems Security*, 2008.

A. Udupa, A. Raghavan, J. Deshmukh, S. Mador-Haim, M. Martin, and R. Alur. TRANSIT: Specifying protocols with concolic snippets. In *PLDI*, 2013.