Synthesis of Machine Code from Semantics *

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

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

Abstract
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 QFQBV 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.

Categories and Subject Descriptors D.1.2, I.2.2 [Automatic Programming]: Program Synthesis

General Terms Algorithms

Keywords Machine-code synthesis, CEGIS, divide-and-conquer, SMT, IA-32 instruction set

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 [23, §7], [1, §1], [4, §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 [11], and binary translation [3]. 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 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 [9], or (ii) superoptimize instruction-sequences [2]. A peephole-superoptimizer has the following type:

Superoptimize : InstrSequence → InstrSequence

A machine-code synthesizer has the following type:

Synthesize : QFBVFormula → InstrSequence

Because an instruction-sequence can be converted to a QFQBV 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 QFQBV formula. The synthesized instruction-sequence implements the input QFQBV formula (i.e., is equivalent to the QFQBV formula). Our technique is parameterized by the ISA of the target instruction-sequence, and is inherent in enumerative synthesis, our technique uses a divide-and-conquer strategy to break the input QFQBV formula into independent sub-formulas. The synthesized instruction-sequence implements each transformed formula.

One tool that could be created using the above framework is an offline binary optimizer to improve unoptimized binaries. Analyses like Value-Set Analysis (USA) [1] and Def-Use Analysis (DUA) [13] could be used in Step 2 to optimize QFQBV 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 QFQBV formulas specialized with respect to a partial state. A machine-code synthesizer can also be used to generate obfuscated instruction-sequences for testing malware detectors [5], and to embed security policies in binaries [8].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI’15, June 13–17, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/mmmmm.mmmmm

* Supported, in part, by NSF under grant CCF-0904371; by AFRL under contracts FAF650-10-C-7088 and FAF750-14-2-0270; and by DARPA under cooperative agreement HR0011-12-2-0012; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding under cooperative agreement HR0011-12-2-0012; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

** 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.
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 [18]. 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 \( \varphi \), that instruction-sequence will not implement \( \varphi \) 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 [2] (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 \( \mathsf{L} \) over finite vocabularies of constant symbols and function symbols. We will be dealing with a specific instantiation of \( \mathsf{L} \), denoted by \( \mathsf{L}^{\text{IA-32}} \). (\( \mathsf{L} \) can also be instantiated for other ISAs.) In \( \mathsf{L}^{\text{IA-32}} \), some constants represent IA-32’s registers (\( \text{EAX}, \text{ESP}, \text{EBP}, \text{etc.} \)), some represent flags (\( \text{CF}, \text{SF}, \text{etc.} \)), and some are free constants (\( i, j, \text{etc.} \)). \( \mathsf{L}^{\text{IA-32}} \) has only one function symbol “\( \text{Mem} \)”, which denotes memory. The syntax of \( \mathsf{L}^{\text{IA-32}} \) is defined in Fig. 1. The term of the form

\[ T \in \text{Term}, \varphi \in \text{Formula}, \text{FE} \in \text{FuncExp} \]

\[ c \in \text{Int}_{\text{32}} = \{ ..., -1, 0, 1, ... \} \]

\[ b \in \text{Bool} = \{ \text{True}, \text{False} \} \]

\[ \text{Int}_{\text{32}} \in \text{Int}_{\text{32Id}} = \{ \text{EAX}, \text{ESP}, \text{EBP}, \ldots, i, j, \ldots \} \]

\[ \text{Bool} \in \text{BoolId} = \{ \text{CF}, \text{SF}, \ldots \} \]

\[ F \in \text{FuncId} = \{ \text{Mem} \} \]

\[ \text{op} \in \text{BinOp} = \{ +, -, \ldots \} \]

\[ \text{rop} \in \text{RelOp} = \{ =, \neq, <, >, \ldots \} \]

\[ T ::= c \mid \text{Int}_{\text{32Id}} \mid T_1 \text{op} T_2 \mid \text{ite} (\varphi, T_1, T_2) \mid F(T_1) \]

\[ \varphi ::= b \mid \text{BoolId} \mid T_1 \text{rop} T_2 \mid \neg \varphi_1 \mid \varphi_1 \text{and} \varphi_2 \mid F = \text{FE} \]

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

Figure 1: Syntax of \( \mathsf{L}^{\text{IA-32}} \).

\[ \text{ite}(\varphi, T_1, T_2) \text{ represents an if-then-else expression. A FuncExp of the form } FE[T_1 \mapsto T_2] \text{ denotes a function-update expression.} \]

The function \( \langle \_ \rangle \) converts an IA-32 instruction-sequence into a QFBV formula. The methodology for this conversion can be found elsewhere [14]. To write formulas that express state transitions, all \( \text{Int}_{\text{32Id}}, \text{BoolId}s, \text{ and FuncId}s \) can be qualified by primes (e.g., \( \text{Mem}' \)). The QFBV formula for an instruction-sequence is a restricted 2-vocabulary formula of the form

\[ \bigwedge_n (F_n = T_n) \land \bigwedge_n (\text{Mem}_n = \varphi_n) \land F' = \text{FE}, \]  \hspace{1cm} (1)

where \( F_n, \text{Mem}_n \) 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 “\( \text{push ebp} \)” is given below. This instruction pushes the 32-bit value in the frame-pointer register \( \text{ebp} \) onto the stack.

\[ \langle \text{push ebp} \rangle \equiv \text{ESP}' = \text{ESP} - 4 \land \text{Mem}' = \text{Mem}[\text{ESP} - 4 \mapsto \text{EBP}] \]  \hspace{1cm} (2)

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 \( F' = F \) 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 \( F' = F \). (\( EIP \) is the program counter for IA-32.) The QFBV formula for the \( \text{push} \) instruction actually looks like the formula in Eqn. (3), and omits the conjunct \( EIP' = EIP + 1 \).

\[ \text{ESP}' = \text{ESP} - 4 \land \text{EAX}' = \text{EAX} \land \ldots \]  \hspace{1cm} (3)

\[ CF' = CF \land \ldots \land \text{Mem}' = \text{Mem}[\text{ESP} - 4 \mapsto \text{EBP}] \]

2.2 Semantics

Intuitively, a QFBV formula represents updates made by an instruction to the machine state. QFBV formulas in \( \mathsf{L}^{\text{IA-32}} \) are interpreted as follows: elements of \( \text{Int}_{\text{32}}, \text{Bool}, \text{BinOp}, \text{RelOp}, \) and \( \text{BoolOp} \) 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) \( \text{Mem} \) 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 \( \mathsf{L}^{\text{IA-32}} \) is a set of machine-state pairs (pre-state, post-state) that satisfy the formula. An IA-32 machine-state is a triple of the form:

\[ \langle \text{RegMap}, \text{FlagMap}, \text{MemMap} \rangle \]

\( \text{RegMap}, \text{FlagMap}, \) and \( \text{MemMap} \) map each register, flag, and memory location in the state, respectively, to a value. A (pre-state,
post-state) pair that satisfies Eqn. (2) is
\[
\sigma \equiv ([ESP \mapsto 100][EBP \mapsto 200], [], [])
\]

\[
\sigma' \equiv ([ESP \mapsto 96][EBP \mapsto 200], [], [96 \mapsto 200]).
\]

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 \( \varphi \), McSynth synthesizes an instruction-sequence for \( \varphi \) 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 \( \varphi \) 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 \( \varphi \) 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 \( \varphi \) for this idiom is

\[
\varphi \equiv \text{ESP}' = \text{ESP} - 4 \land \text{EBP}' = \text{ESP} - 4 \land \text{Mem}' = \text{Mem}[\text{ESP} - 4 \mapsto \text{EBP}].
\]

McSynth starts enumerating templatized one-instruction sequences. Let us assume that the first candidate is \( C_1 \equiv \langle \text{mov eax, <Imm32>}; \text{push ebp}; \text{mov ebp, <Imm32>} \rangle \). \( C_1 \) is a template to move a 32-bit constant value into the eax register. McSynth converts \( C_1 \) into a QFBV formula \( \psi_1 \). (McSynth uses free constants for template operands.)

\[
\psi_1 \equiv \langle \{C_1\} \rangle \equiv \text{EAX}' = i
\]

Before processing \( \psi_1 \) via CEGIS, McSynth checks if \( \psi_1 \) can be pruned away. If an instruction-sequence uses (modifies) a location that is not used (modified) by \( \varphi \), intuitively, the instruction-sequence can never implement \( \varphi \) in an efficient way. McSynth computes the abstract semantic USE-footprint (SFP\text{\_USE}) and the abstract semantic KILL-footprint (SFP\text{\_KILL}) for \( \varphi \) and \( \psi_1 \). SFP\text{\_USE} (SFP\text{\_KILL}) is an over-approximation of the locations (registers, flags, or memory) that might be used (modified) by a QFBV formula. Concretely, SFP\text{\_USE} (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 SFP\text{\_KILL} are primed. SFP\text{\_USE} and SFP\text{\_KILL} for \( \varphi \) and \( \psi_1 \) are given below.

\[
\begin{align*}
\text{SFP\text{\_USE}}(\varphi) & = \{\text{ESP}, \text{EBP}\} \\
\text{SFP\text{\_USE}}(\psi_1) & = \emptyset \\
\text{SFP\text{\_KILL}}(\varphi) & = \{\text{ESP}', \text{EBP}'\} \\
\text{SFP\text{\_KILL}}(\psi_1) & = \{\text{EAX}'\} \\
\text{Mem}' & = \text{Mem}[\text{ESP} - 4 \mapsto \text{EBP}].
\end{align*}
\]

SFP\text{\_USE}(\psi_1) is \( \emptyset \) because \( \psi_1 \) does not use any registers, flags, or memory locations. (Identity conjuncts like \( \text{EBX}' = \text{EBX} \) do not contribute to SFP\text{\_USE} and SFP\text{\_KILL}.) SFP\text{\_KILL}(\psi_1) is \( \{\text{EAX}'\} \) because \( \psi_1 \) might change the value of the eax register. SFP\text{\_KILL}(\varphi) contains \( \text{Mem}' \) because \( \varphi \) might modify some memory location. Because SFP\text{\_KILL}(\psi_1) \not\subseteq \text{SFP\text{\_KILL}}(\varphi), \psi_1 \) might modify a location that is unmodified by \( \varphi \), and thus, it cannot be equivalent to \( \varphi \). Consequently, McSynth discards \( C_1 \). Moreover, regardless 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 \( C_2 \equiv \langle \text{mov ebp, esp'} \rangle \) 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 \( \psi_2 \) for \( C_2 \), and the SFP\text{\_USE} sets for \( \psi_2 \) are

\[
\psi_2 \equiv \langle \{C_2\} \rangle \equiv \text{EBP} = \text{ESP}
\]

\[
\text{SFP\text{\_USE}}(\psi_2) = \{\text{ESP}\} \\
\text{SFP\text{\_KILL}}(\psi_2) = \{\text{EBP} \}
\]

Because SFP\text{\_KILL}(\psi_2) \subseteq SFP\text{\_KILL}(\varphi), and SFP\text{\_USE}(\psi_2) \subseteq SFP\text{\_USE}(\psi_1), McSynth proceeds to process \( \psi_2 \) via CEGIS. Given a templatized candidate \( C \), and a finite set of tests \( \mathcal{T} \) (where, a test is a (pre-state, post-state) 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 \( C_{\text{con}} \) and \( \varphi \) produce identical post states for each test in test set \( \mathcal{T} \). If such an instance cannot be found, McSynth continues further processing of \( C \) via CEGIS, but retains \( C \) as a useful-prefix.

2. If McSynth finds an instance \( C_{\text{con}} \) that works for the finite set of tests \( \mathcal{T} \), McSynth uses an SMT solver to determine whether \( \langle C_{\text{con}} \rangle \) is equivalent to \( \varphi \). If the check succeeds, McSynth returns \( C_{\text{con}} \).

3. If the check fails, McSynth adds the counter-example produced by the SMT solver to \( \mathcal{T} \), and repeats Step 1.

Suppose that \( \mathcal{T} \) has only one test, \( \langle \sigma_1, \sigma_1' \rangle \). McSynth evaluates \( \psi_2 \) with respect to \( \langle \sigma_1, \sigma_1' \rangle \) (i.e., checks satisfiability), and finds that \( \sigma_1 \neq \sigma_1' \). Hence, McSynth continues further processing of \( \psi_2 \) 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 \( C_3 \equiv \langle \text{push ebp}; \text{mov ebp, <Imm32>} \rangle \) as the next candidate. \( C_3 \) is a template to save the frame-pointer register esp on the stack, and move a 32-bit constant value into esp. The QFBV formula \( \psi_3 \) for \( C_3 \) is

\[
\psi_3 \equiv \langle \{C_3\} \rangle \equiv \text{ESP}' = \text{ESP} - 4 \land \text{EBP}' = i \land \text{Mem}' = \text{Mem}[\text{ESP} - 4 \mapsto \text{EBP}].
\]

By simplifying \( \psi_3 \) with respect to \( \langle \sigma_1, \sigma_1' \rangle \), McSynth produces the simplified formula \( \psi_3^{(\sigma_1, \sigma_1')} \) shown below.

\[
\psi_3^{(\sigma_1, \sigma_1')} \equiv 96 = 96 \land 96 = i \land \text{Mem}' = \text{Mem}[96 \mapsto 200] \land \text{Mem}(96) = 0 \land \text{Mem}(96) = 200
\]

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

$$\psi^\text{conc} = ESP = ESP - 4 \land EBP = 96 \land Mem = Mem[ESP - 4 \mapsto EBP]$$

$\varphi$ and $\psi^\text{conc}$ produce identical post-states for the test case $\langle \sigma_1, \sigma'_1 \rangle$.

Now that MC$\text{SYNTH}$ has found a candidate that is equivalent to $\varphi$ with respect to one test case, MC$\text{SYNTH}$ checks if the candidate is equivalent to $\varphi$ for all possible test cases. MC$\text{SYNTH}$ checks the equivalence of $\varphi$ and $\psi^\text{conc}$ using an SMT solver. The solver says that the two formulas are not equivalent, and produces a counterexample $\langle \sigma_2, \sigma'_2 \rangle$. MC$\text{SYNTH}$ adds $\langle \sigma_2, \sigma'_2 \rangle$ to $T$.

$$\sigma_2 \equiv \langle ESP \mapsto 104 \rangle[EBP \mapsto 200], [], []$$

$$\sigma'_2 \equiv \langle ESP \mapsto 100 \rangle[EBP \mapsto 100], [], [100 \mapsto 200]$$

Eventually, MC$\text{SYNTH}$ enumerates the candidate $C_4 \equiv \langle \text{push esp; mov esp, esp}', \text{push esp, esp} \rangle$, and obtains the corresponding QFBV formula $\psi_4$. MC$\text{SYNTH}$ simplifies $\psi_4$ 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. MC$\text{SYNTH}$ checks the satisfiability of $\psi_4^{\langle \sigma_1, \sigma'_1 \rangle} \land \psi_4^{\langle \sigma_2, \sigma'_2 \rangle}$ using an SMT solver. The solver says that the formula is satisfiable. MC$\text{SYNTH}$ then checks whether $\varphi$ and $\psi_4$ 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, MC$\text{SYNTH}$ uses a divide-and-conquer strategy. Before synthesizing instructions for the full $\varphi$, MC$\text{SYNTH}$ attempts to break $\varphi$ into a sequence of independent sub-formulas. If $\varphi$ can be split into sub-formulas, MC$\text{SYNTH}$ synthesizes instructions for the sub-formulas, appends the synthesized instructions, and returns the result. One possible way to split $\varphi$ is as $\langle \varphi_1, \varphi_2, \varphi_3 \rangle$, where

$$\varphi_1 \equiv ESP' = ESP - 4 \quad \varphi_2 \equiv EBP' = ESP - 4$$

$$\varphi_3 \equiv Mem' = Mem[ESP - 4 \mapsto EBP]$$

However, $\varphi_2$ and $\varphi_3$ both use $ESP$, which is killed by $\varphi_1$. (Note that to compare the used and killed locations, the primes are dropped from primed symbols.) If MC$\text{SYNTH}$ were to synthesize instructions for $\varphi_1$, $\varphi_2$, and $\varphi_3$, and append them in that order, the result will not be equivalent to $\varphi$. We call such a split illegal. Another possible way to split $\varphi$ is as $\langle \varphi_1, \varphi_2, \varphi_3 \rangle$, where

$$\varphi_1 \equiv Mem' = Mem[ESP - 4 \mapsto EBP]$$

$$\varphi_2 \equiv EBP' = ESP - 4 \quad \varphi_3 \equiv ESP' = ESP - 4$$

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. MC$\text{SYNTH}$ synthesizes the following instructions for the sub-formulas:

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

$$C_3 \equiv \text{lea esp, } [\text{esp - 4}]$$

The divide-and-conquer strategy reduces the synthesis time for $\varphi$ 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 MC$\text{SYNTH}$. First, we present the algorithm for MC$\text{SYNTH}$’s synthesis loop. Second, we present the heuristics used by MC$\text{SYNTH}$ to prune the synthesis search-space. Third, we describe MC$\text{SYNTH}$’s divide-and-conquer strategy, and present the full algorithm used by MC$\text{SYNTH}$.

#### 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 MC$\text{SYNTH}$.

##### 4.1.1 Base Algorithm

Given an input QFBV formula $\varphi$, a naïve first cut is to enumerate every concrete instruction-sequence in the ISA, convert the instruction-sequence into a QFBV formula $\psi$, 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.

MC$\text{SYNTH}$ 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 $\text{SAT}$. 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. MC$\text{SYNTH}$ maintains a finite set of test cases $T$. (Note that $(\sigma, \sigma') \models \varphi$, for all $(\sigma, \sigma') \in T$.) MC$\text{SYNTH}$ evaluates $\psi$ with respect to each test $(\sigma, \sigma')$ in $T$ to check if $(\sigma, \sigma') \models \psi$ (i.e., $\varphi$ and $\psi$ produce identical post-states for each test in $T$). If all the tests pass, $\psi$ is checked for equivalence with $\varphi$ (Line 7); otherwise, it is discarded. The strawman algorithm, along with this optimization, is shown in Alg. 1. In Alg. 1, $\text{TestsPass}$ evaluates $\psi$ with respect to each test in $T$.  

---

**Algorithm 1 Strawman algorithm to synthesize instructions from a QFBV formula**

<table>
<thead>
<tr>
<th>Input: $\varphi$</th>
<th>Output: $C_{\text{conc}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $T \leftarrow \emptyset$</td>
<td>2: for each concrete instruction-sequence $C_{\text{conc}}$ in the ISA do</td>
</tr>
<tr>
<td>3: $\psi \leftarrow \langle C_{\text{conc}} \rangle$</td>
<td>4: if not $\text{TestsPass}(\psi, T)$ then</td>
</tr>
<tr>
<td>5: continue</td>
<td>6: end if</td>
</tr>
<tr>
<td>7: model $\leftarrow \text{SAT}(\neg(\varphi \Leftrightarrow \psi))$</td>
<td>8: if model $\models \bot$ then</td>
</tr>
<tr>
<td>9: return $C_{\text{conc}}$</td>
<td>10: else</td>
</tr>
<tr>
<td>11: $T \leftarrow T \cup \text{model}$</td>
<td>12: end if</td>
</tr>
<tr>
<td>13: end for</td>
<td></td>
</tr>
</tbody>
</table>
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 opcodes. 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 [20]. 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 $\varphi$, (ii) a finite set of tests $T$ 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 $T$, 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 $\varphi$, MCSYNTH bootstraps its test suite $T$ with the test $(\sigma_0, \sigma_0')$. $\sigma_0$ is a machine-code state in which all locations are mapped to 0. MCSYNTH computes $\sigma_0'$ by substituting $\sigma_0$ in $\varphi$. The inputs to CEGIS’S loop are $\varphi$, the test suite $T$, a templatized sequence $C$, and its QFBV formula $\psi \equiv \langle \langle C \rangle \rangle$.

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

Because states have memory arrays, simplifying $\psi$ with respect to $T$ 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 $\varphi$ with respect to $T$, 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 $\varphi$ is

$$\varphi \equiv \text{EAX} = \text{Mem}(\text{ESP}) \land \text{Mem}' = \text{Mem}(\text{EBP} \Rightarrow \text{EBX}).$$

Let us also assume that $T$ has only one test case.\(^2\)

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

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

Consider our first candidate $C_1 \equiv \langle \text{mov eax, esp}; \text{mov [esp], ebx} \rangle$. $C_1$ copies a 32-bit value from the location pointed to by the stack-pointer register $\text{esp}$ to the register $\text{eax}$, and a 32-bit value from the register $\text{ebx}$ to the location pointed to by the frame-pointer register $\text{ebp}$. The QFBV formula $\psi_1$ for $C_1$ is

$$\psi_1 \equiv \langle \langle C_1 \rangle \rangle \equiv \text{EAX} = \text{Mem}(\text{ESP}) \land \text{Mem}' = \text{Mem}(\text{ESP} \Rightarrow \text{EBX}).$$

Our goal is to simplify $\psi_1$ with respect to $(\sigma, \sigma')$ to obtain the simplified formula $\psi_1'$.\(^3\)

Approach 1. Suppose that we have a function $\chi$ that converts a state into a QFBV formula. One way to obtain $\psi_1'(\sigma, \sigma')$ is to convert $\sigma$ and $\sigma'$ into QFBV formulas (using the function $\chi$), and conjoin the resulting formulas with $\psi_1$.

$$\psi_1'(\sigma, \sigma') \equiv \psi_1 \land \chi(\sigma, 0) \land \chi(\sigma', 1)$$

Note that $\chi$ 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 $\chi$ are in the specified vocabulary. We can define $\chi$ as follows:

$$\chi(\sigma, 0) \equiv \text{Mem}(0) = \text{EBP} \Rightarrow 0 \land \text{EBX} = 1.$$  

$$\chi(\sigma, 1) \equiv \text{Mem}(100) = 2 \land \text{Mem}(104) = 0 \land \ldots$$

In most ISAs, addressable memory is usually 2\(^{32}\) or 2\(^{64}\) bytes long. One way to prevent $\chi_{\text{Mem}}$ 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 $\chi_{\text{RegFlag}}$ in place of $\chi$.

$$\psi_1'(\sigma, \sigma') \equiv \psi_1 \land \chi_{\text{RegFlag}}(\sigma, 0) \land \chi_{\text{RegFlag}}(\sigma', 1)$$

However, Eqns. (5) and (6) are not equisatisfiable. This approach results in false positives. Because Eqn. (6) is satisfiable, this approach would conclude that $\psi_1$ is equivalent to $\varphi$ with respect to $(\sigma, \sigma')$, even though it is not.

Approach 3. To obtain a simplified formula that is equisatisfiable with the one in Eqn. (5), 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 $\psi_1$ with respect to $(\sigma, \sigma')$. First, SimplifyWithTest conjoins $\psi_1$ with $\text{RegFlag}(\sigma, 0)$ and $\text{RegFlag}(\sigma', 1)$ to obtain the following formula:

$$2 = \text{Mem}(100) \land \text{Mem}' = \text{Mem}(100 \Rightarrow 1)$$

The only memory location that is accessed or updated in Eqn. (7) is 100. For this location, SimplifyWithTest generates the following constraints from $(\sigma, \sigma')$.

$$\text{Mem}(100) = 2 \land \text{Mem}'(100) = 2$$

SimplifyWithTest conjoins Eqn. (7) and Eqn. (8) to obtain

$$\psi_1'(\sigma, \sigma') \equiv \text{Mem}' = \text{Mem}(100 \Rightarrow 1) \land \text{Mem}(100) = 2 \land \text{Mem}'(100) = 2.$$
Algorithm 2 Algorithm SimplifyWithTest

Input: $\psi(\sigma, \sigma')$
Output: $\psi'(\sigma, \sigma')$

1. $\psi'(\sigma, \sigma') \leftarrow \text{Simplify}(\psi \land \text{RegFlag}(\sigma, 0) \land \text{RegFlag}(\sigma', 1))$
2. concLocs $\leftarrow$ ConcLocs($\psi'(\sigma, \sigma')$)
3. concMemConstr $\leftarrow$ true
4. for each a in concLocs do
5. \quad val $\leftarrow$ Lookup($\sigma$, a)
6. \quad val$'$ $\leftarrow$ Lookup($\sigma'$, a)
7. \quad concMemConstr $\leftarrow$ concMemConstr $\land$ Mem(a) $=$ val $\land$ Mem$'$ (a) $=$ val$'$
8. end for
9. symLocs $\leftarrow$ SymLocs($\psi'(\sigma, \sigma')$)
10. if symLocs $=$ $\emptyset$ then
11. \quad return Simplify($\psi'(\sigma, \sigma')$) $\land$ concMemConstr
12. end if
13. return Simplify($\psi'(\sigma, \sigma')$) $\land$ concMemConstr $\land$ SymMemConstr $(\text{symLocs}, \sigma, \text{Mem})$ $\land$ SymMemConstr $(\text{symLocs}, \sigma', \text{Mem'})$

Consider another candidate $C_2 \equiv \text{"mov eax, [esp]; mov [<Imm32>], ebx"}$. $C_2$ 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 $\psi_2$ for $C_2$ is $\psi_2 \equiv \langle C_2 \rangle \equiv \text{EAX} = \text{Mem(ESP)} \land \text{Mem'}(i \mapsto \text{EBX})$.

After conjointing $\psi_2$ with $\text{RegFlag}(\sigma, 0)$ and $\text{RegFlag}(\sigma', 1)$, SimplifyWithTest produces the following formula:

$2 = \text{Mem}(100) \land \text{Mem'}(i \mapsto 1) \land \text{Mem}(i) = 0$ (10)

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 $i$, $\text{McSynth}$ generates the following constraint from the memory map $[100 \mapsto 2]$ in $\sigma$:

$\text{Mem}(100) = 2 \land i \neq 100 \Rightarrow \text{Mem}(i) = 0$ (11)

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

$\text{Mem'}(100) = 2 \land \text{Mem'}(200) = 1 \land (i \neq 100 \land i \neq 200) \Rightarrow \text{Mem}'(i) = 0$ (12)

SimplifyWithTest conjoints Eqns. (10)–(12), and returns the resulting formula $\psi'(\sigma, \sigma')$. $\text{McSynth}$ checks the satisfiability of $\psi'(\sigma, \sigma')$. The SMT solver says that $\psi'(\sigma, \sigma')$ is satisfiable, and produces the satisfying assignment $[i \mapsto 200]$. Indeed, $\psi$ and $\psi_2$ 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 that ConcLocs and SymLocs collect concrete and symbolic memory locations, respectively, from all nested terms and sub-formulas (e.g., $\text{Mem'}(i \mapsto \text{Mem}(0))$) and not just from those at the top level.

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

Algorithm 3 Algorithm CEGIS

Input: $\varphi, C, \psi = \langle C \rangle, \mathcal{F}$
Output: $\text{Cegis}_{\text{con}}$ of $C$ such that $\langle C_{\text{con}} \rangle \Leftrightarrow \varphi$, or $\text{FAIL}$

1. while true do
2. \quad $\psi \leftarrow \text{true}$
3. \quad for each test-case $\langle \sigma, \sigma' \rangle \in \mathcal{F}$ do
4. \quad \quad $\psi \leftarrow \psi \land \text{SimplifyWithTest}(\psi, \langle \sigma, \sigma' \rangle)$
5. \quad end for
6. \quad model$_1 = \text{SAT}(\psi)$
7. \quad if model$_1 = \bot$ then
8. \quad \quad return FAIL
9. \quad end if
10. \quad $\psi_{\text{con}} \leftarrow \text{Substitute}(\psi, \text{model}_1)$
11. \quad model$_2 \leftarrow \text{SAT}(\varphi \Rightarrow \psi_{\text{con}})$
12. \quad if model$_2 = \bot$ then
13. \quad \quad return Substitute($C, \text{model}_1$)
14. \quad end if
15. \quad $\mathcal{F} \leftarrow \mathcal{F} \cup \text{model}_2$
16. end while

The CEGIS loop. Once $\text{McSynth}$ obtains $C_{\text{con}}$ (and its corresponding QFBV formula $\psi_{\text{con}}$) that is equivalent to $\varphi$ with respect to $\mathcal{F}$, $\text{McSynth}$ checks if $\psi_{\text{con}}$ is equivalent to $\varphi$ using an SMT solver. If they are equivalent, $\text{McSynth}$ returns $C_{\text{con}}$. Otherwise, $\text{McSynth}$ adds the counter-example returned by the solver to $\mathcal{F}$, and searches for another concrete instruction-sequence that passes the tests. Alg. 3 show $\text{McSynth}$’s CEGIS loop. In Alg. 3, 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. 4. In the algorithm, $\epsilon$ 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 (SFPUSE) is the set of concrete locations (represented as constant symbols) that are used by the QFBV formula for some input. The semantic-KILL-footprint (SFPKILL) is the set of concrete locations that are modified by the QFBV formula for some input. For the formula in Eqn. (4), SFPUSE and SFPKILL are shown below (with a minor abuse of notation).

\[
\begin{align*}
\text{SFP}_{\text{USE}}(\varphi) &= \{\text{ESP}, \text{EBP}\} \\
\text{SFP}_{\text{KILL}}(\varphi) &= \{\text{ESP}', \text{EBP}', 0', 1', 2', \ldots \}
\end{align*}
\]

0', 1', 2', \ldots are in SFPKILL because \(\varphi\) 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 (SFP\textsuperscript{a}USE) is an over-approximation of SFP\textsubscript{USE}, and the abstract semantic-KILL-footprint (SFP\textsubscript{a}KILL) is an over-approximation of SFP\textsubscript{KILL}. We identify SFP\textsuperscript{a}USE and SFP\textsubscript{a}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.**

\[
\begin{align*}
\text{SFP}_{\text{a} \text{USE}}(c) &= \begin{cases} \{c\} & \text{if } c \in RF \\
\emptyset & \text{otherwise} \end{cases} \\
\text{SFP}_{\text{a} \text{USE}}(\text{Mem}(t)) &= \{\text{Mem}\} \cup \text{SFP}_{\text{a} \text{USE}}(t), \text{ where } t \in T \\
\text{SFP}_{\text{a} \text{USE}}(c') &= \emptyset, \text{ where } c' \in RF', c \in RF \\
\text{SFP}_{\text{a} \text{USE}}(\text{Mem}') &= \emptyset
\end{align*}
\]

For all other cases, SFP\textsuperscript{a}USE (SFP\textsubscript{a}KILL) is the union of SFP\textsubscript{a}USE (SFP\textsubscript{a}KILL) of the constituents.

Eqns. (13)–(16) represent the computation of SFP\textsuperscript{a}USE and SFP\textsuperscript{a}KILL for the identify conjuncts in a QFBV formula (representing the unmodified portions of the state). For an input QFBV formula \(\varphi\), consider the set of instruction-sequences \(I^\#\) that has the following property:

\[
\forall C \in I^\#, \text{ SFP}_{\text{a} \text{USE}}(\langle\langle C\rangle\rangle) \subseteq \text{SFP}_{\text{a} \text{USE}}(\varphi) \cap \text{SFP}_{\text{a} \text{KILL}}(\langle\langle C\rangle\rangle) 
\]

The set \(I^\#\) is depicted in Fig. 2 as a hexagon. If \texttt{McSYNTH} restricts the synthesis search-space to \(I^\#\), \texttt{McSYNTH} will miss two types of candidates.

1. A candidate that is not equivalent to \(\varphi\). An example of such a candidate for the QFBV formula in Eqn. (4) is "\texttt{mov eax, ebx}".

2. A candidate \(C\) that satisfies the following properties:
   (a) \(\langle\langle C\rangle\rangle \leftrightarrow \varphi\)
   (b) \(\text{SFP}_{\text{a} \text{USE}}(\langle\langle C\rangle\rangle) \not\subseteq \text{SFP}_{\text{a} \text{USE}}(\varphi) \lor \text{SFP}_{\text{a} \text{KILL}}(\langle\langle C\rangle\rangle) \not\subseteq \text{SFP}_{\text{a} \text{KILL}}(\varphi)\)

We call such a candidate superfluous. Although \(\langle\langle C\rangle\rangle\) semantically uses and modifies the same locations as \(\varphi\) (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 \(\varphi\), and might not implement \(\varphi\) efficiently. Therefore, \texttt{McSYNTH} prunes away superfluous candidates. For the QFBV formula in Eqn. (4), "\texttt{push ebp; lea esp, [esp+eax]}; \texttt{lea ebp, [ebp-eax]}" is an example of a superfluous candidate; the final value of \texttt{ebp} depends on the value of \texttt{esp}, but does not depend on the value of \texttt{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 \texttt{McSYNTH} finds 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.

Benmarks 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.
the pre-state and post-state vocabularies of \( \varphi \)
constant and function symbols: \( \langle, \rangle \)
Input:
Output: \( \text{splits} \)
\begin{algorithm}
\textbf{Algorithm 6} Algorithm EnumerateSplits
\begin{enumerate}
\item \text{splits} \leftarrow \emptyset
\item \text{killedRegsFlags} \leftarrow \text{KilledRegs}(\varphi) \cup \text{KilledFlags}(\varphi)
\item \text{killedMem} \leftarrow \text{KilledMem}(\varphi)
\item \text{regFlagSplits} \leftarrow \text{SplitSet}(\text{killedRegsFlags})
\item \text{memSplits} \leftarrow \text{SplitSequence}(\text{killedMem})
\item for each \( (s1, s2) \in \text{regFlagSplits} \) do
\item \text{for each} \( (\text{prefix}, \text{suffix}) \in \text{memSplits} \) do
\item \text{if} \( (s1 = \emptyset \text{ and } \text{prefix} = \langle \rangle ) \lor (s2 = \emptyset \text{ and } \text{suffix} = \langle \rangle ) \) then
\item \text{continue}
\item \text{end if}
\item \( \varphi_1 \leftarrow \text{TruncateFormula}(\varphi, s1, \text{prefix}) \)
\item \( \varphi_2 \leftarrow \text{TruncateFormula}(\varphi, s2, \text{suffix}) \)
\item \text{if} \( \text{DropPrimes}(\text{SFP}_{\text{KILL}}(\varphi_1)) \cap \text{SFP}_{\text{USE}}(\varphi_2) \neq \emptyset \) then
\item \text{continue}
\item \text{end if}
\item \text{splits} \leftarrow \text{splits} \cup \langle \varphi_1, \varphi_2 \rangle
\item \text{end for}
\item \text{end for}
\item \text{return} \text{splits}
\end{enumerate}
\end{algorithm}

McSYNTH uses a recursive procedure (\text{DivideAndConquer})
that splits \( \varphi \) into two sub-formulas \( \varphi_1 \) and \( \varphi_2 \), and
synthesizes instructions for \( \varphi_1 \) and \( \varphi_2 \). For pragmatic reasons, an implementation
of the splitting step would typically construct \( \varphi_1 \) and \( \varphi_2 \) from sub-formulas of \( \varphi \).
The pseudo-code for \text{DivideAndConquer} is shown in Alg. 5. \text{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) \text{EnumerateSplits} returns an empty set of splits in Line 1, or (ii) for each split, at least one recursive call returns \text{FAIL}. Let \text{Synthesize}_{\text{max}} be a version of Alg. 4
that is parameterized by the maximum length of candidates to consider during enumeration (\text{max}). \text{Synthesize}_{\text{max}} returns \text{FAIL} if Alg. 4 cannot find an instruction-sequence with length \( \leq \text{max} \) that implements \( \varphi \). \text{DivideAndConquer} uses \text{EnumerateSplits} to enumerate all legal splits of \( \varphi \).

\begin{algorithm}
\textbf{Algorithm 7} Algorithm McSYNTH
\begin{enumerate}
\item \text{return} \text{DivideAndConquer}(\varphi, 1)
\item \text{if} \text{return} \neq \text{FAIL} \text{then}
\item \text{return} \text{return}
\item \text{end if}
\item \text{return} \text{DivideAndConquer}(\varphi, 2)
\item \text{if} \text{return} \neq \text{FAIL} \text{then}
\item \text{return} \text{return}
\item \text{end if}
\item \text{return} \text{Synthesize}(\varphi)
\end{enumerate}
\end{algorithm}

Pseudo-code for \text{EnumerateSplits} is shown in Alg. 6. We illustrate the algorithm with the following QFBV formula:
\begin{align*}
\varphi & \equiv \text{ESP}' = \text{ESP} - 12 \land \text{EBP}' = \text{ESP} - 4 \land \\
\text{EAX'} = \text{EBX} \land \text{Mem}' = \text{Mem}[\text{ESP} - 12 \rightarrow \text{EDI}] \\
& \quad [\text{ESP} - 8 \rightarrow \text{ESI}][\text{ESP} - 4 \rightarrow \text{EBP}]
\end{align*}
For \( \varphi \), \text{KilledMem} (Line 2) returns \{\text{ESP}', \text{EBP}', \text{EAX}'\}, and \text{KilledMem} (Line 3) returns the sequence \{(\text{ESP} - 4, \text{ESP} - 8, \text{ESP} - 12)\}. Note that the sequence preserves the temporal order of memory updates. \text{SplitSequence} returns the set of disjoint subset pairs for the argument set. For example, \{(\text{EBP}', \text{ESP}, \text{EAX}')\} and \{(\{\text{ESP}', \text{EBP}', \text{EAX}'\}\} \text{regFlagSplits} (Line 4). \text{SplitSequence} returns the set of non-overlapping \( (\text{prefix}, \text{suffix}) \) pairs that partition the argument sequence. For example, \{(\text{ESP} - 4, \text{ESP} - 8), (\text{ESP} - 12)\} \text{in memSplits} (Line 5), but \{(\text{ESP} - 4, \text{ESP} - 12), (\text{ESP} - 8)\} is not. \text{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 \( (s1, s2) \in \{\text{ESP}', \text{EBP}', \text{EAX}'\}\} \text{in Line 6 and (prefix, suffix) = ((ESP - 4, ESP - 8), (ESP - 12)\} \text{in Line 7,} \varphi_1 \text{and} \varphi_2 \text{in Lines 11 and 12, respectively, are}
\begin{align*}
\varphi_1 & \equiv \text{ESP}' = \text{ESP} - 12 \land \text{EBP}' = \text{ESP} - 4 \land \\
\text{Mem}' & = \text{Mem}[\text{ESP} - 8 \rightarrow \text{ESI}][\text{ESP} - 4 \rightarrow \text{EBP}] \\
\varphi_2 & \equiv \text{EAX'} = \text{EBX} \land \text{Mem}' = \text{Mem}[\text{ESP} - 12 \rightarrow \text{EDI}]
\end{align*}
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 sub-formula more than once; the result for a sub-formula is either its synthesized code-sequence or \text{FAIL}. Consequently, Alg. 5 really uses a form of dynamic programming. Practical values for \text{Synthesize’s parameter max} are 1 or 2. For these values, \text{DivideAndConquer} will either return \text{FAIL}, or the synthesized instruction-sequence in a few minutes or hours (cf. Fig. 4). If \text{DivideAndConquer} returns \text{FAIL}, McSYNTH uses Alg. 4 to synthesize instructions for \( \varphi \). The full synthesis algorithm used by McSYNTH is given in Alg. 7.

\begin{lemma}
Alg. 4 is sound. (The instruction-sequence returned by Alg. 4 is logically equivalent to the input QFBV formula \( \varphi \).)
\end{lemma}
\begin{proof}
By lines 11-14 of Alg. 3, the returned instruction-sequence is logically equivalent to \( \varphi \).
\end{proof}

Suppose that \text{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 \( \chi \) from §4.1.2 to mean the operator that converts a symbolic state into a QFBV formula. \( \langle I \rangle \) can be defined as follows: \( \langle I \rangle \equiv \chi(\text{sym.exec}(I, i, j)) \). 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 \((\varphi_1, \varphi_2)\) of \(\varphi\), if \(\varphi_1 \iff \langle\langle I_1\rangle\rangle\), and \(\varphi_2 \iff \langle\langle I_2\rangle\rangle\), then \(\varphi \iff \langle\langle I_1; I_2\rangle\rangle\).

Proof.
\[
\langle\langle I_1; I_2\rangle\rangle \iff \chi(sym\_exec(I_1; I_2, 0, 1))
\]
\[
iff \chi(sym\_exec(I_2, 2, 1) \circ sym\_exec(I_1, 0, 2)),
\]
\[
iff \exists voc_2 \cdot \chi(sym\_exec(I_2, 2, 1))
\]
\[
\land \chi(sym\_exec(I_1, 0, 2) \iff (\text{because vocabulary 2 acts as an intermediate vocabulary})
\]
\[
iff \exists voc_2 \cdot change\_voc(\varphi_1, 1, 2)
\]
\[
\land change\_voc(\varphi_2, 0, 2) \iff (\text{because } \varphi_1 \text{ is equivalent to } \langle\langle I_1\rangle\rangle, \text{ and } \varphi_2 \text{ is equivalent to } \langle\langle I_2\rangle\rangle)
\]
\[
iff \varphi \iff (\text{because } \langle\langle \varphi_1, \varphi_2\rangle\rangle \text{ is a legal split of } \varphi)
\]

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 \(\varphi\), then Alg. 7 will find \(I\) and terminate.

Proof. M\(^{\text{CSYNTH}}\) enumerates templatized instruction-sequences of increasing length. Because the templatized instruction-sequences span the full set of IA-32 instruction-sequences (§3.3), M\(^{\text{CSYNTH}}\) searches through all non-superfluous instruction-sequences in IA-32 to find an instruction-sequence \(I\) that is equivalent to \(\varphi\).

Note that if such an instruction-sequence does not exist (if all instruction-sequences that implement \(\varphi\) 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. M\(^{\text{CSYNTH}}\) 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 SF\(^\text{P}_{\text{KILL}}\)(\(\varphi\)) just before Line 9 of Alg. 4. 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 13 of Alg. 4, all conjuncts that update registers in Scratch’ need to be dropped from \(\varphi\) and \(\psi\). (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 \(S' \in \text{Scratch}'\), should be dropped in the two occurrences of \(\langle\langle\rangle\rangle\).

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. 8. In Alg. 8, the parameter \(f\) represents the quality-evaluation function, the parameter \(\text{timeout}\) represents the timeout value for the biased synthesizer, the function \(\text{MaxFn}\) returns the maximum value for a quality-evaluation function, and the call to the function \(\text{TimeoutExpired}\) returns true if \(\text{timeout}\) has expired. Additionally, the following changes have to be made to Algs. 4, 5, and 7 to implement a biased synthesizer:

- Algs. 4, 5, and 7 should take an additional parameter \(\text{seen}\), which is the set of instruction-sequences that have already been synthesized by M\(^{\text{CSYNTH}}\).
- The following lines of code should be inserted after line 14 in Alg. 4, and after line 9 in Alg. 5, respectively:

\[
\text{if } ret \in \text{seen} \text{ then continue}
\]

5. Implementation

M\(^{\text{CSYNTH}}\) uses Transformer Specification Language (TSL) [13] 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 [14]. M\(^{\text{CSYNTH}}\) uses ISAL [13, §2.1] to generate the templatized instruction pool for synthesis. M\(^{\text{CSYNTH}}\) uses Yices [7] 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.

M\(^{\text{CSYNTH}}\) 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 [16] 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 ([13], 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 MC SYNTH on QFBV formulas obtained from instruction-sequences from the SPECINT 2006 benchmark suite [10]. Our experiments were designed to answer the following questions:

- What is the time taken by MC SYNTH to synthesize instruction-sequences of varying length?
- What is the reduction in (i) synthesis time, and (ii) search-space size caused by MC SYNTH’s footprint-based search-space pruning heuristic (§4.2)?
- What is the reduction in synthesis time caused by MC SYNTH’s divide-and-conquer strategy (§4.3)?

All experiments were run on a system with a quad-core, 3GHz Intel Xeon processor; however, MC SYNTH’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 mn, 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 MC SYNTH’s footprint-based search-space pruning. The results are shown in Fig. 3. 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.

Divide-and-Conquer. The second set of experiments measured the synthesis times for formulas created from instruction-sequences of lengths 1 through 10 using MC SYNTH’s divide-and-conquer strategy (as well as footprint-based pruning). The results are shown in Fig. 4. “Synthesis Time” is the total CPU time spent by Alg. 7. “Base Case Time” is the time spent in the base case (Line 13 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. 4. 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. 5, 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. 5 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.
Superoptimization. Superoptimization aims at finding an optimal instruction-sequence for a target instruction-sequence [2, 3, 12, 15, 17]. Peephole superoptimization [2] 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 $⟨⟨·⟩⟩$ converts an instruction-sequence into a QFBV formula. Suppose that $\text{SynthOptimize}$ is a client of the synthesizer that is biased to synthesize short instruction-sequences, a superoptimizer can be constructed as follows:

$$\text{Superoptimize}(\text{InstrSeq}) = \text{SynthOptimize}(⟨⟨\text{InstrSeq}⟩⟩)$$

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 $\text{McSYNTH}$ because $\text{McSYNTH}$ 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 [11] 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 [5, 6]. 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 $\text{McSYNTH}$, a tool that synthesizes IA-32 instructions from a QFBV formula. Our experiments show that, in comparison to our baseline algorithm, $\text{McSYNTH}$’s footprint-based search-space pruning reduces the synthesis time by a factor of 473, and $\text{McSYNTH}$’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 $\text{McSYNTH}$, 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 $\text{gzip2}$ binary.

In addition, we have used $\text{McSYNTH}$ to improve the accuracy of machine-code slicing. Instructions that perform multiple updates to the state (e.g., $\text{push}$, $\text{leave}$, etc.) reduce the accuracy of machine-code slicing. We used $\text{McSYNTH}$ to “untangle” such instructions by synthesizing equivalent instruction-sequences.

One possible direction for future work is to use $\text{McSYNTH}$ to obfuscate/de-obfuscate instruction-sequences in malware. A second direction would be to adapt the algorithms in $\text{McSYNTH}$ to synthesize non-straight-line, but non-looping programs. One approach to loop-free code is to use the $\text{ite}$ 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 $\text{Mem}$ in $\text{SFP}$$_\text{USE}$ and $\text{SFP}$$_\text{KILL}$.
Acknowledgments

We thank Junghee Lim and Tushar Sharma for answering several TSL-related questions during the implementation of McSYNTH. We thank Brian Alliet for implementing McSYNTH’s IA-32 AST parser. We thank the anonymous reviewers for their invaluable feedback.

References