Model-Assisted Machine-Code Synthesis

VENKATESH SRINIVASAN*, University of Wisconsin–Madison, USA
ARA VARTANIAN, University of Wisconsin–Madison, USA
THOMAS REPS, University of Wisconsin–Madison, USA and GrammaTech, Inc., USA

Binary rewriters are tools that are used to modify the functionality of binaries lacking source code. Binary rewriters can be used to rewrite binaries for a variety of purposes including optimization, hardening, and extraction of executable components. To rewrite a binary based on semantic criteria, an essential primitive to have is a machine-code synthesizer—a tool that synthesizes an instruction sequence from a specification of the desired behavior, often given as a formula in quantifier-free bit-vector logic (QFBV). However, state-of-the-art machine-code synthesizers such as McSynth++ employ naïve search strategies for synthesis: McSynth++ merely enumerates candidates of increasing length without performing any form of prioritization. This inefficient search strategy is compounded by the huge number of unique instruction schemas in instruction sets (e.g., around 43,000 in Intel's IA-32) and the exponential cost inherent in enumeration. The effect is slow synthesis: even for relatively small specifications, McSynth++ might take several minutes or a few hours to find an implementation.

In this paper, we describe how we use machine learning to make the search in McSynth++ smarter and potentially faster. We converted the linear search in McSynth++ into a best-first search over the space of instruction sequences. The cost heuristic for the best-first search comes from two models—used together—built from a corpus of ⟨QFBV-formula, instruction-sequence⟩ pairs: (i) a language model that favors useful instruction sequences, and (ii) a regression model that correlates features of instruction sequences with features of QFBV formulas, and favors instruction sequences that are more likely to implement the input formula. Our experiments for IA-32 showed that our model-assisted synthesizer enables synthesis of code for 6 out of 50 formulas on which McSynth++ times out, speeding up the synthesis time by at least 526×, and for the remaining formulas, speeds up synthesis by 4.55×.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies** → *Supervised learning by regression*;

Additional Key Words and Phrases: machine-code synthesis, machine learning, best-first search, n-gram language model, regression model, IA-32 instruction set

ACM Reference Format:
Venkatesh Srinivasan, Ara Vartanian, and Thomas Reps. 2017. Model-Assisted Machine-Code Synthesis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 61 (October 2017), 26 pages. https://doi.org/10.1145/3133885

1 INTRODUCTION

Binary analysis and rewriting has received 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

*This author is now at Google, Inc.

Authors' addresses: Venkatesh Srinivasan, University of Wisconsin–Madison, USA, venk@cs.wisc.edu; Ara Vartanian, University of Wisconsin–Madison, USA, aravart@cs.wisc.edu; Thomas Reps, University of Wisconsin–Madison, USA , GrammaTech, Inc. USA, reps@cs.wisc.edu.

2475-1421/2017/10-ART61
https://doi.org/10.1145/3133885

2010, §1], [Brumley et al. 2011, §1], [ElWazeer et al. 2013, §7]), which has led to the development and wider use of binary analysis and rewriting tools. Binary rewriting becomes particularly important if one wishes to modify the functionality of a binary that lacks source code and/or the compiler toolchain used to build the binary. Binary rewriting can be done for purposes of software reuse (e.g., slicing [Srinivasan and Reps 2016], partial evaluation [Srinivasan and Reps 2015a], binary translation [Bansal and Aiken 2008]), optimization (e.g., superoptimization [Bansal and Aiken 2006]), and software security (e.g., binary hardening [Abadi et al. 2005; Erlingsson and Schneider 1999; Slowinska et al. 2012]).

Recently [Srinivasan and Reps 2015b], it has been observed that a machine-code synthesizer[1] can be used to create a general framework for semantics-based binary rewriting (see §2.2), and one can instantiate the framework with different analyses to create different rewriters, such as partial evaluators [Srinivasan and Reps 2015a] and slicers [Srinivasan and Reps 2016]. A machine-code synthesizer is a tool that synthesizes a straight-line instruction sequence that implements a semantic specification of the desired behavior, which is often given as a formula in QFBV. A key challenge in machine-code synthesis is the enormous size of the synthesis search-space: for example, Intel's IA-32 instruction-set architecture (ISA) has around 43,000 unique instruction schemas; this huge instruction pool, along with the exponential cost inherent in enumerative synthesis, results in an enormous search space for a synthesizer. For example, with this search space, a naïve enumerative synthesizer will take a few days just to synthesize an instruction sequence of length 3 (Fig. 5 in [Srinivasan and Reps 2015b]).

To cope with the enormous synthesis search-space, a state-of-the-art machine-code synthesizer McSynth++ [Srinivasan et al. 2016] uses a *master-slave* architecture. The master splits the input formula into a sequence of independent smaller sub-formulas, and delegates the sub-formulas to slave synthesizers. The slaves perform the actual enumerative synthesis. McSynth++ brought down the synthesis time from days to several minutes, but it is still not fast enough: McSynth++ times out for larger QFBV formulas; even for smaller formulas, McSynth++ sometimes takes a few hours to find an implementation (see Fig. 6). Consequently, if a binary-rewriter client supplies a formula as input to McSynth++, the client has to wait several minutes or hours before McSynth++ finds an implementation. This delay might not be tolerable for a client that has to invoke the synthesizer multiple times to rewrite an entire binary, e.g., a machine-code partial evaluator [Srinivasan and Reps 2015a].

A key limitation that McSynth++ suffers from is the core search strategy used by the slave synthesizers: each slave performs a *linear search* over the space of instruction sequences, i.e., it first exhausts one-instruction sequences, then moves to two-instruction sequences, and so on. The slave subsequently tests every enumerated candidate for equivalence with the input formula.[2] One can see that this search strategy is clearly inefficient: not all enumerated candidates are equally likely to implement the input formula.

This paper describes how we used *machine learning* to make the search in McSynth++'s slaves smarter and almost always faster. Our technique is based on the following insight:

> If a slave synthesizer can *prioritize* candidates, it can potentially find an implementation faster, while retaining its completeness guarantees. In essence, the goal is to turn an enumerative synthesizer into a *best-first-search synthesizer* that is informed by a trained model.

To implement this idea, we converted the linear search in McSynth++'s slaves into a best-first search, where the cost heuristic comes from models learned from a huge corpus of specifications

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

[2] McSynth++ is equipped with two pruners (see §2.2) that prune away useless candidates prior to equivalence testing.

and implementations: 4.4 million ⟨QFBV-formula, instruction-sequence⟩ pairs. The model-assisted best-first search prioritizes (i) instruction sequences that are commonly used to implement idioms in programs, and (ii) instruction sequences that contain instructions that are highly likely to implement the input formula (e.g., if the formula is $\varphi \equiv EAX' = EAX + 10$, the search prioritizes instruction sequences that contain the `add` instruction). We have equipped McSynth++'s slaves with this model-assisted best-first search to build an improved machine-code synthesizer called McSynth-ML.

Given a huge corpus of ⟨QFBV-formula, instruction-sequence⟩ pairs, McSynth-ML first learns (i) a *language model* for instruction sequences in the instruction set, and (ii) a regression model that correlates features of a QFBV formula with features of its equivalent instruction sequence. In particular, we use an *n-gram* model for the former, and *k-nearest-neighbor (k-NN) regression* for the latter. McSynth-ML then performs the actual synthesis. Given an input formula φ,

- McSynth-ML uses features of φ along with k-NN regression to restrict the slave's instruction pool to contain only instructions that are highly likely to implement φ.
- McSynth-ML then begins a best-first search over the truncated instruction-sequence space[3] to find an implementation for φ. The cost heuristic for the search comes from both the n-gram model and the k-NN-regression model: the former allows the search to prioritize useful instruction sequences that are commonly used to implement idioms in binaries; the latter allows the search to prioritize instruction sequences that contain instructions most likely to implement φ, and steer the search away from instruction sequences with instructions that are irrelevant to φ.

The effect of the model-assisted best-first search is potentially faster synthesis in slave synthesizers. The transitive effect is potentially faster client binary-rewriters. McSynth-ML should also allow existing clients to work on larger QFBV formulas for the purpose of obtaining output binaries of better quality. Moreover, McSynth-ML should also facilitate building of new clients that were impractical to build with McSynth++. (See §8 for possible future directions.)

Contributions. This paper's contributions include the following:

- Our technique is the first of its kind to employ machine learning for the synthesis of low-level code.
- While existing approaches for low-level-code synthesis employ enumerative, symbolic, and stochastic search strategies [Phothilimthana et al. 2016b; Schkufza et al. 2013; Srinivasan and Reps 2015b; Srinivasan et al. 2016], we employ a novel best-first search assisted by models learned from a corpus of specifications and implementations.
- Our technique makes novel usage of language models to steer the search towards useful instruction sequences; prior approaches have used language models only for purposes of finding most likely completions of partial programs [Gvero and Kuncak 2015; Raychev et al. 2014].
- Ours is the first synthesis technique that employs a model (specifically, k-NN regression) that uses features of *both* specifications *and* implementations; prior model-assisted synthesis tools use features learned from only a corpus of programs [Gvero and Kuncak 2015; Raychev et al. 2015, 2014]. We use the model to steer the search towards instruction sequences that are highly likely to implement the input specification.

Our techniques have been implemented in McSynth-ML, a model-assisted synthesizer for IA-32. We evaluated McSynth-ML on a test suite consisting of 50 formulas. Our experiments show that McSynth-ML synthesizes code for all 6 of the formulas on which McSynth++ times out, speeding up

[3]To preserve its completeness guarantees, if synthesis with the truncated instruction pool fails to find an implementation, McSynth-ML attempts synthesis with the untruncated instruction pool. See §4.2.

$$T \in \mathit{Term},\ \varphi \in \mathit{Formula},\ \mathit{FE} \in \mathit{FuncExpr}$$
$$c \in \mathit{Int32} = \{\ldots,\ \text{-}1,\ 0,\ 1,\ \ldots\} \quad b \in \mathit{Bool} = \{\mathit{True},\ \mathit{False}\}$$
$$I_{\mathit{Int32}} \in \mathit{Int32Id} = \{\mathit{EAX},\ \mathit{ESP},\ \mathit{EBP},\ \ldots\}$$
$$I_{\mathit{Bool}} \in \mathit{BoolId} = \{\mathit{CF},\ \mathit{SF},\ \ldots\} \quad F \in \mathit{FuncId} = \{\mathit{Mem}\}$$
$$\mathit{op} \in \mathit{BinOp} = \{+,\ -,\ \ldots\} \quad \mathit{bop} \in \mathit{BoolOp} = \{\wedge,\ \vee,\ \ldots\} \quad \mathit{rop} \in \mathit{RelOp} = \{=,\ \neq,\ <,\ >,\ \ldots\}$$
$$T ::= c \mid I_{\mathit{Int32}} \mid T_1\ \mathit{op}\ T_2 \mid \mathit{ite}(\varphi, T_1, T_2) \mid F(T_1)$$
$$\varphi ::= b \mid I_{\mathit{Bool}} \mid T_1\ \mathit{rop}\ T_2 \mid \neg\varphi_1 \mid \varphi_1\ \mathit{bop}\ \varphi_2 \mid F = \mathit{FE}$$
$$\mathit{FE} ::= F \mid \mathit{FE}_1[T_1 \mapsto T_2]$$

Fig. 1. Syntax of L[IA-32].

their synthesis by at least 526×. For the remaining 44 formulas, McSynth-ML speeds up synthesis by 4.55×.

While the paper itself addresses the problem of speeding up machine-code synthesis, the basic idea described in the paper is not restricted to machine code, and could be used to speed up other enumerative program synthesizers, as well. In particular, if one has available, or can create, a corpus of ⟨specification, implementation⟩ pairs (e.g., via logs from a program verifier, or via symbolic execution—see §3.1), machine-learning techniques can be used to build a model M that correlates features of implementations with features of specifications. Then an enumerative program synthesizer S can (i) use best-first search as its core search strategy, where M is used to steer the best-first search toward most likely implementations, and (ii) use model M to truncate the pool of program elements that S uses for enumeration.

2 BACKGROUND

In this section, we briefly describe the logic in which input formulas are expressed (§2.1). The logic allows a client to specify some desired state change in a specific hardware platform—in our case, Intel IA-32 (the 32-bit subset of x86). We also motivate the problem of machine-code synthesis, give an overview of a state-of-the-art machine-code synthesizer McSynth++, and briefly describe how binary-rewriter clients use a machine-code synthesizer to rewrite binaries (§2.2).

2.1 QFBV Formulas for Expressing Specifications

Input specifications to McSynth-ML can be expressed formally by QFBV formulas. Because our specifications express the desired IA-32 state transformation, we will use a variant of QFBV that is specific to the domain of IA-32 instructions.[4]

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]. (Note that L can also be instantiated for other ISAs such as ARM, MIPS, PowerPC, etc.) In L[IA-32], some constant symbols represent IA-32's registers (*EAX*, *ESP*, *EBP*, etc.), and some represent flags (*CF*, *SF*, etc.). L[IA-32] has only one function symbol "*Mem*," which denotes IA-32's memory. The syntax of L[IA-32] is defined in Fig. 1. A 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.

[4]This logic is the same variant of QFBV that is used as an input language in IA-32 analysis and translation tools that use Satisfiability Modulo Theories (SMT) solvers. Thus, such formulas are naturally available in the symbolic components of existing tools.

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

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

where I'_m and $\mathcal{J}'_n$ range over the constant symbols for registers and flags, respectively. The primed vocabulary is the post-state vocabulary, and the unprimed vocabulary is the pre-state vocabulary. For example, the QFBV formula for the specification "push the 32-bit value in the frame-pointer register EBP onto the stack" is given below. (Note that the IA-32 stack pointer is register ESP.)

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

In this section, and in the rest of the paper, we show only the portions of QFBV formulas that express how the state is *modified*. QFBV formulas actually contain identity conjuncts of the form $I' = I$, $\mathcal{J}' = \mathcal{J}$, and $Mem' = Mem$ for constants and functions that are *unmodified*. Because we do not want the synthesizer output to be restricted to an instruction sequence that is located at a specific address, specifications do not contain conjuncts of the form $EIP' = T$. (EIP is the program counter for IA-32.)

Expressing semantics of instruction sequences. In addition to input specifications, McSynth++ also uses L[IA-32] formulas to express the semantics of the candidate instruction-sequences it considers. The function $\langle\!\langle \cdot \rangle\!\rangle$ encodes a given IA-32 instruction-sequence as a QFBV formula. While others have created such encodings by hand (e.g., [Saïdi 2008]), we use a method that takes a specification of the concrete operational semantics of IA-32 instructions and creates a QFBV encoder automatically. The method reinterprets each semantic operator as a QFBV formula-constructor or term-constructor (see [Lim et al. 2011]).

Certain IA-32 string instructions contain an implicit microcode loop, e.g., instructions with the `rep` prefix, which perform an *a priori* unbounded amount of work determined by the value in the ECX register at the start of the instruction. In other applications that use the infrastructure on which McSynth-ML is built, this implicit microcode loop is converted into an explicit loop whose body is an instruction that performs the actions performed by the body of the microcode loop. (More details about this conversion is available elsewhere [Lim et al. 2011, §6].) However, the semantics cannot be expressed as a single QFBV formula. Because of this expressibility limitation, neither McSynth++ nor McSynth-ML tries to synthesize instructions that use the `rep` prefix.

2.2 Overview of McSynth++

McSynth++ [Srinivasan et al. 2016] is an improved version of a prior machine-code synthesizer called McSynth [Srinivasan and Reps 2015b]. In this section, we motivate the problem of machine-code synthesis, and give an overview of McSynth++. (Note that the overview also includes components inherited from McSynth.) We also summarize McSynth++'s algorithm while highlighting its key limitations.

The motivation for machine-code synthesis arises in the context of more principled methods for rewriting binaries (i.e., logic-based or semantics-based methods). For example, suppose that one has an unoptimized binary that lacks source code, and one would like to optimize the binary. An automated and semantics-based way of rewriting the binary would first run analyses like value-set analysis [Balakrishnan and Reps 2010], def-use analysis [Lim and Reps 2013], etc. to gather information about constants, and live registers and flags at various instructions in the binary. Then one would

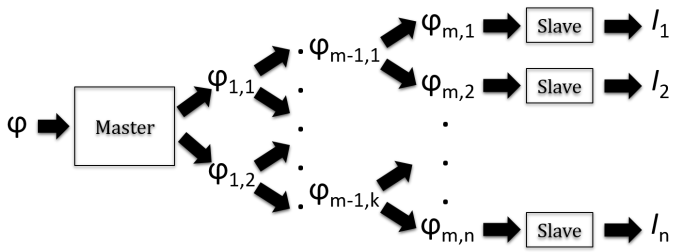

Fig. 2. Master-slave architecture of McSynth++.

(1) convert instruction sequences in the binary into some semantic representation so that one knows what the instruction sequence does,
(2) use the analysis results to transform the semantic representation, such that the transformed semantic representation acts as a specification for an optimized instruction sequence, and
(3) find an instruction sequence that implements the transformed semantic representation.

Recall from §2.1 that a QFBV formula clearly describes what an instruction sequence does. Consequently, one can use QFBV formulas as the semantic representation in the aforementioned recipe for semantics-based binary rewriting.

Note that in the above recipe, a programmer or a binary analyst does not come up with a QFBV specification; the specification comes from transforming a semantic representation of instructions that are already in the binary.

One can use symbolic execution to perform step (1), and one can simplify QFBV formulas with respect to analysis results to perform step (2). To perform step (3) one needs a tool that is capable of searching for an instruction sequence that implements a QFBV formula. This step is where a machine-code synthesizer comes in. By repeating steps (1), (2), and (3) on different straight-line instruction sequences in a binary, one can produce an optimized binary. Note that if one has a machine-code synthesizer for performing step (3), one can use different analyses and transformation mechanisms in step (2) to build different semantics-based binary rewriters (e.g., partial evaluators [Srinivasan and Reps 2015a], slicers [Srinivasan and Reps 2016], binary translators [Bansal and Aiken 2008], etc.).

McSynth++ is a state-of-the-art machine-code synthesizer. McSynth++ synthesizes a straight-line machine-code instruction sequence from a semantic specification of the desired behavior, given as a QFBV formula. The synthesized instruction-sequence implements the input formula (i.e., is equivalent to the formula). McSynth++ is parameterized by the ISA of the target instruction-sequence.

McSynth++ uses enumerative strategies for synthesis. However, an ISA like IA-32 has around 43,000 unique instruction schemas, which, when combined with the exponential cost inherent in enumeration, results in an enormous search space for synthesis. McSynth++ attempts to cope with the enormous search space using a *master-slave* architecture. The design of McSynth++ is depicted in Fig. 2. Given a QFBV formula φ, McSynth++ synthesizes an instruction sequence for φ in the following way:

(1) The master uses a combination of *divide-and-conquer* [Srinivasan and Reps 2015b, §4.3] and *flattening* [Srinivasan et al. 2016, §4.1.2] strategies to split φ into a sequence of independent sub-formulas, and hands over each sub-formula to a slave synthesizer.
(2) The slave enumerates *templatized* instruction-sequences, in which template operands (or holes) replace one or more constant values. Each slave uses an instantiation of the

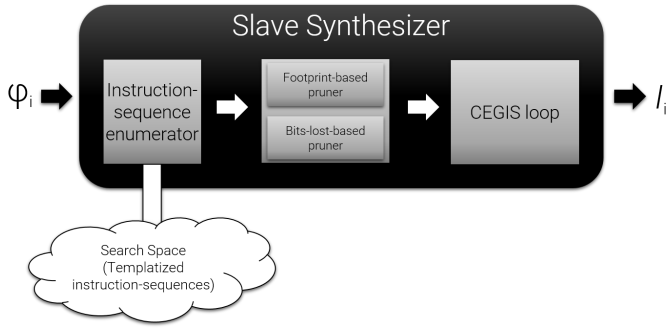

Fig. 3. Design of McSynth++'s slave.

counterexample-guided inductive synthesis (CEGIS) framework [Srinivasan and Reps 2015b, §4.1] along with two pruners—a *footprint-based* pruner [Srinivasan and Reps 2015b, §4.2] and a *bits-lost-based* pruner [Srinivasan et al. 2016, §4.2.2]—to synthesize code for a sub-formula. (The design of the slave is depicted as Fig. 3.)

(3) If a slave times out, the master uses an alternative split. (For example in Fig. 2, if synthesis of code for $\varphi_{m,\,1}$ times out, the master tries out an alternative split of $\varphi_{m\text{-}1,\,1}$.) If all candidate splits for a sub-formula time out, the master hands over the entire sub-formula to a slave. (For example in Fig. 2, if all candidate splits of $\varphi_{m\text{-}1,\,1}$ time out, the master supplies $\varphi_{m\text{-}1,\,1}$ as an input to a slave.)

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

(5) Additionally for pragmatic purposes, McSynth++ uses a *"move-to-front"* heuristic [Srinivasan et al. 2016, §4.3], which moves instructions[5] that occur in synthesized code to the front of the instruction pool for use in the next synthesis task. This heuristic prioritizes useful instructions: useful instructions "bubble up" the instruction pool over the course of several synthesis tasks.

In the remainder of this section, we present an example to illustrate McSynth++'s algorithm, while highlighting its key limitations. (The limitations of McSynth++ we present in this section lie in McSynth++'s slave synthesizers.) Along the way, we provide details for only the components of McSynth++ that are necessary to understand the design of McSynth-ML. A detailed description of the design of McSynth++ is available elsewhere [Srinivasan and Reps 2015a; Srinivasan et al. 2016].

Consider the following QFBV formula φ:

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

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

McSynth++'s master splits φ into the sequence of independent sub-formulas $\langle \varphi_1, \varphi_2 \rangle$ shown below. The master splits φ in such a way that if one were to synthesize instruction sequences for φ_1 and φ_2 independently and concatenate the sequences in the same order, the result will be equivalent

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

to φ. Such a split is said to be *legal* (Defn. 2 in [Srinivasan and Reps 2015b]).

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

McSynth++'s master then gives φ_1 and φ_2 to slave synthesizers.

We now illustrate the limitations in McSynth++'s slave synthesizer using φ_2 as an example. Note that one implementation of φ_2 is "`add esp, 10; lea esp, [esp+8]`." (For this example, we pretend that the `add` instruction sets only the zero flag ZF. The `lea` instruction given above increments ESP by 8 without affecting flags.) McSynth++'s slave enumerates templatized instruction-sequences of increasing length, and uses a CEGIS loop along with two pruners to synthesize code for a sub-formula. However, the slave suffers from the following limitations:

Retaining instructions that are irrelevant to φ in the instruction pool. The slave does not attempt to prune its instruction pool based on the QFBV operators present in the input formula φ. The existing pruners in McSynth++ prune from the instruction pool instructions whose *operands* are inconsistent with those of φ. (For example, the existing pruners will prune the instruction "`add ebp, ⟨Imm32⟩`" because the instruction uses and modifies the EBP register, which is untouched by φ_2.) However, instructions whose *opcode variants* (defined in §3.1.1) are highly unlikely to implement the input formula are left unpruned. (For example, instructions belonging to the IMUL opcode variant could never be used to implement φ_2; yet existing pruners do not prune instructions such as "`imul esp, ⟨Imm32⟩`" from the instruction pool.) Consequently, candidate instruction-sequences containing such instructions are wastefully enumerated and subsequently discarded by the slave.

Linear search. The basic search strategy used by McSynth++'s slave is *linear search*: the slave first exhausts all one-instruction sequences, followed by two-instruction sequences, and so on. During the search, it might be better to prioritize (i) commonly used instruction-sequence prefixes (e.g., instruction sequences that implement common idioms), and (ii) prefixes with instructions that are highly likely to implement φ. However, McSynth++'s slave does not attempt to perform any prioritization. For example, when a McSynth++ slave searches for an implementation for φ_2, let us assume that the slave encounters the prefix $P_1 \equiv$ "`ror esp, ⟨Imm32⟩`" (rotate right instruction) before the prefix $P_2 \equiv$ "`add esp, ⟨Imm32⟩`" during its linear search. The slave would try to expand P_1 before P_2, and would take a long time before it eventually enumerates the templatized candidate $C \equiv$ "`add esp, ⟨Imm32⟩; lea esp, [esp+⟨Imm32⟩]`." The slave then uses the CEGIS loop to find the instantiation $C_{conc} \equiv$ "`add esp, 10; lea esp, [esp+8]`" of C that implements φ_2. However, compared to P_1, code like P_2 is much more frequently found in binaries. Furthermore, P_2 contains an `add` instruction, which is much more likely to implement φ_2 than the `ror` instruction in P_1. The slave could have found the implementation C_{conc} much faster had it expanded P_2 first.

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

In effect, McSynth++ takes around 10 minutes to synthesize an implementation for our example φ. Given the enormous size of the search space that McSynth++ has to deal with, this number is not high by itself. (Note that a naïve enumerative synthesizer would have taken several hours to find an implementation for φ.) However, in the context of a binary-rewriting client that makes several calls to the synthesizer (e.g., the machine-code partial evaluator WiPEr [Srinivasan and Reps 2015a]), such synthesis times would cause the client to take hours or days to rewrite an entire binary.

3 OVERVIEW

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

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

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

Step 1 is more of an optimization: instead of first enumerating a candidate C and then postpone processing C because of C's low value of the cost heuristic, step 1 eagerly truncates the instruction pool so that C never gets enumerated.[6] Because of the smarter model-assisted search, McSynth-ML's slave is typically much faster than McSynth++'s slave.[7]

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

3.1 Training Phase

To train models that assist machine-code synthesis, one needs a huge corpus of specifications (QFBV formulas) and their corresponding implementations (instruction sequences). However, creating such a huge corpus via synthesis can take a very long time. (Recall from §2.2 that finding an implementation with McSynth++ can take several minutes. Finding implementations for millions of specifications can take several days.) An important observation is that, while finding an instruction sequence for a QFBV formula involves search/synthesis and is slow, converting an instruction sequence into a QFBV formula can be done very quickly via *symbolic execution*. There are many readily available tools that can perform this conversion [Lim et al. 2011]. We used this observation to create a corpus of millions of equivalent ⟨QFBV-formula, instruction-sequence⟩ pairs as follows: (i) we harvested several straight-line instruction sequences from binaries, (ii) converted each instruction sequence I into a QFBV formula φ via symbolic execution, and (iii) added $\langle\varphi, \mathrm{I}\rangle$ to the corpus. It took almost 12 hours to create a corpus of 4.4 million pairs by this method. (This number represents the total time spent to harvest and canonicalize instruction sequences from binaries, and perform symbolic execution.) §6 provides more details about creating the training corpus.

3.1.1 k-NN regression. Our intended use case for k-NN regression during synthesis is to predict how likely it is for an IA-32 instruction to occur in an implementation of the input formula. k-NN regression is a non-parametric regression technique that stores all available training data and

[6] To preserve the McSynth-ML's completeness guarantees (Thm. 2), if synthesis with the truncated instruction pool fails to find an implementation, McSynth-ML subsequently attempts synthesis with the untruncated instruction pool. (See §4.2.)
[7] Note that McSynth-ML's slave is not guaranteed to be faster because its heuristics are sensitive to training data.

predicts a numerical target based on a *metric*[8] over its input space. In the next few paragraphs, we describe a specific instantiation of k-NN regression that we use in our context.

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

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

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

In our case, the presence or absence of QFBV operators in the specification φ constitute the features of the input space: an entry in the input feature-vector is 1 if φ contains the QFBV operator corresponding to that entry; it is 0 otherwise. There are 70 QFBV operators in the logic described in §2.1, and so the length of each input vector is 70 (i.e., $m = 70$).

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

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

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

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

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

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

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

The assumption is that the probability of observing the i^{th} word X_i in the context history of the preceding $i-1$ words can be approximated by the probability of observing it in the shortened

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

context history of the preceding $n-1$ words. The conditional probabilities in the above equation can be calculated from frequency counts as follows:

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

Typically, n-gram probabilities are not directly derived from frequency counts, and incorporate some *smoothing* mechanism to account for unseen words or n-grams.[9]

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

3.2 Synthesis Phase

We now illustrate the workings of McSynth-ML using the same example that was used in §2.2.

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

McSynth-ML uses the same master as McSynth++, and so it splits φ into the same sub-formulas φ_1 and φ_2 from §2.2.

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

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

We now illustrate how McSynth-ML's model-assisted slave synthesizes code for φ_2. McSynth-ML first obtains a feature vector for φ_2 based on the QFBV operators that appear in φ_2. Some QFBV operators that would have a 1 in their corresponding entries in the feature vector for φ_2 are **+** (QFBV_PLUS), **∧** (QFBV_AND), and **=** (QFBV_EQUALS). McSynth-ML then gives the feature vector as input to the k-NN model, and obtains as output the label vector, which contains a k-NN-regression probability for every opcode variant in IA-32. Given a query point q (input feature-vector), k-NN predicts the probability distribution over the opcode variants as follows: k-NN finds the k training-set items nearest to q using Euclidean distance as the metric, averages their label values, and returns the resulting label vector as the output. Intuitively, the probability value for an opcode variant represents the likelihood of an implementation of φ_2 to contain an instruction belonging to that opcode variant. McSynth-ML discards opcode variants whose probabilities are below a certain threshold (say, 0.1 for this example). For our running example, this step reduces the size of the instruction pool from 240 to 20 instructions. (Note that the pruners inherited from McSynth++ had already reduced the size of the instruction pool from around 43,000 to 240.) Let us use $\mathscr{P}$ to denote this remnant instruction pool.

Once the instruction pool has been truncated, McSynth-ML starts best-first search. The search maintains a fringe of instruction-sequence prefixes of varying lengths. The search starts with the empty prefix (ϵ). At any given point during the search, the prefix p that has the highest score according to the cost heuristic c gets expanded, i.e., McSynth-ML appends every instruction in $\mathscr{P}$ to p, thus expanding the contour of the fringe. The cost heuristic c for a prefix p is computed via a combination of (i) p's n-gram probability according to Eqn. (1), and (ii) the combined k-NN-regression probability for the opcode variants of individual instructions in p. (c is defined in Line 25 in Alg. 3.) Because ϵ is the only prefix in the initial fringe, McSynth-ML expands it by appending

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

Algorithm 1 Algorithm TrainModels

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

every instruction in $\mathscr{P}$ to it. McSynth-ML then checks if any of the newly created candidates implements φ_2 via CEGIS. (Note that the pruners inherited from McSynth++ attempt to prune away a newly created candidate before testing it via CEGIS.) None of the one-instruction sequences in the fringe implement φ_2, and so McSynth-ML looks for the next prefix to expand. Suppose that the one-instruction prefix "`mov esp, ⟨Imm32⟩`" has the highest score according to the cost heuristic. McSynth-ML expands it, and tests if any of the newly created two-instruction candidates implements φ_2; none of them do. Note that now the fringe of the search contains both one-instruction and two-instruction prefixes. Suppose that the one-instruction prefix "`add esp, ⟨Imm32⟩`" now has the highest score according to the cost heuristic. McSynth-ML expands it, and tests the resulting candidates via CEGIS. The instantiation "`add esp, 10; lea esp, [esp+8]`" of the candidate C ≡ "`add esp, ⟨Imm32⟩; lea esp, [esp + ⟨Imm32⟩]`" implements φ_2, and so the slave returns that concrete instruction-sequence as the implementation for φ_2.

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

4 ALGORITHM

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

4.1 Training Phase

During the training phase, one needs to train the k-NN-regression model and the n-gram language model. Recall from §3 that the corpus for the training phase consists of millions of equivalent ⟨QFBV-formula, instruction-sequence⟩ pairs produced via symbolic execution. The algorithm for training the models is given as Alg. 1. The algorithm takes as input the corpus, the hyperparameter k for k-NN regression, and the length n of n-grams up to which the model should maintain n-gram counts. (For example, if $n = 3$, the model maintains counts for unigrams, bigrams, and trigrams.) The output is a pair of models: kNNModel and nGramModel.

In Alg. 1, `InitKNNModel` and `InitNGramModel` initialize parameters of the k-NN model and the n-gram model, respectively (Lines 1 and 2). `GetQFBVOperators`(φ) returns an ordered list of ⟨opr, isPresent⟩ pairs, where isPresent is 1 if operator opr is present in φ; it is 0 otherwise (Line 4). `GetQFBVOperators` traverses the AST of φ, and collects QFBV operators. `GetQFBVOperators` orders its output by QFBV operator. `GetOpcodeVariants`(I) returns

Algorithm 2 Algorithm TruncateInstrPool

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

an ordered list of ⟨opc, isPresent⟩ pairs, where isPresent is 1 if instruction sequence I contains an instruction belonging to the opcode variant opc; it is 0 otherwise (Line 5). Recall from §3.1.1 that the opcode variant is a conjunction of opcode category and operand sizes. `GetOpcodeVariants` inspects the ASTs of instructions in I to produce its output, which is ordered by opcode variant. `AddToModel` adds an ⟨input-vector, label-vector⟩ pair to the input space of the k-NN-regression model (Line 6). `UpdateCounts` teases apart the various n-grams of length up to n from the instruction sequence I, and updates their counts in the model (Line 7). Recall from §3.1.2 that in the context of McSynth-ML, an n-gram is an instruction subsequence of length n. Alg. 1 finally returns the two models (Line 9).

4.2 Synthesis Phase

Recall from §3 that McSynth-ML uses the same master as McSynth++, and so in this sub-section, we describe only the algorithm used by McSynth-ML's slave. In this sub-section, we use φ to refer to a sub-formula given as input to a slave synthesizer.

Before presenting the algorithm used by the slave, we present a procedure called TruncateInstrPool that uses features of φ along with k-NN regression to retain in the instruction pool only those instructions that are highly likely to implement φ. The algorithm for TruncateInstrPool is given as Alg. 2. The algorithm takes as inputs a formula φ, the instruction pool, and the k-NN model built during the training phase. The role of the remaining input (t_p) will be explained in the next paragraph. TruncateInstrPool returns the remnant instruction pool, along with the label vector for φ, as the output.

Alg. 2 first obtains the QFBV operators present in φ via `GetQFBVOperators` (Line 1). It supplies that set of operators as the test vector to the k-NN-regression model, and queries for the opcode-variant probabilities via `GetLabelVector` (Line 2). `GetLabelVector` produces as output an ordered list of ⟨opc, prob⟩ pairs (ordered by opc), where opc is an opcode variant, and prob is its corresponding k-NN-regression probability. Recall from §3.2 that prob represents the likelihood of an implementation of φ to contain an instruction belonging to the opcode variant opc. Starting with an empty instruction pool instrPool′ (Line 3), TruncateInstrPool repeatedly adds to instrPool′ instructions from instrPool belonging to opcode variants whose k-NN-regression probabilities are greater than a threshold value t_p (Lines 4–10). TruncateInstrPool finally returns the remnant instruction pool instrPool′, along with the label vector for φ (Line 11). instrPool′ contains only the instructions that, according to k-NN regression, are most likely to implement the input φ.

Algorithm 3 Algorithm McSynth-MLSlave

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

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

Now we are ready to present the algorithm used by McSynth-ML's slave. Recall from §3 that the slave performs the actual enumerative synthesis in McSynth-ML. The algorithm used by the slave is given as Alg. 3. Alg. 3 takes the following inputs: (i) the input formula φ, (ii) a k-NN-regression model, (iii) the threshold probability t_p used in TruncateInstrPool, (iv) an n-gram-based language model, and for pragmatic purposes (v) the maximum length *max* of instruction-sequence prefixes to enumerate during synthesis. The slave also takes a timeout value as an additional input (not shown in Alg. 3). The output of Alg. 3 is either a concrete instruction-sequence C_{conc} that implements φ, or FAIL if the slave could not find an implementation before the timeout expires.

The slave first reads the list of instructions from a file (via `ReadInstrPool`) (Line 1), and uses the footprint-based pruner and the bits-lost-based pruner inherited from McSynth++ to prune away useless instructions from the instruction pool via `PruneFootprint` and `PruneBitsLost`, respectively (Lines 2–6). The slave then calls TruncateInstrPool, and obtains a smaller pool of instructions (Line 7), and subsequently begins its best-first search. The fringe of the search is

implemented as a priority queue, with instruction-sequence prefix as key and the cost heuristic c (defined in Line 25) as priority. Recall from §3.2 that the cost-heuristic value c for a prefix p is a combination of (i) the n-gram probability of p according to the language model (obtained via nGramModel.`GetProb`), and (ii) the combined k-NN-regression probability of the opcode variant of each instruction i in p (obtained via labelVector.`GetProb`(`OpcodeVariant`(i))). In Line 25, λ denotes the weighting parameter for the scores obtained from the two models. The slave initially inserts the empty prefix into the queue (Line 9). At any given point in the search, the slave picks the prefix with the highest priority, and expands it by appending to it every instruction in the instruction pool (Lines 11–13). The slave then checks if a candidate instruction-sequence obtained via expansion is useless, and thus can be pruned away using the footprint-based and bits-lost-based pruners (Lines 14–17). Candidates that escape pruning are tested for equivalence with φ via CEGIS (Line 18). If CEGIS finds an instantiation C_{conc} of a candidate C that implements φ, the slave returns that instantiation as the implementation (Lines 19–21). Otherwise, the slave adds the (useful) candidate as a prefix in the priority queue (Line 26) provided that the length of the candidate is not greater than *max* (Lines 22–24).

To preserve the completeness guarantees of McSynth-ML (see Thm. 2), if the slave fails to find an implementation for an input φ with the truncated instruction pool, then the slave attempts to find an implementation without truncating the instruction pool. In Alg. 3, if an implementation could not be found when $t_p > 0$, Alg. 3 recursively calls itself with $t_p = 0$ (Line 32). Consequently in that recursive call, TruncateInstrPool will default to merely returning the input instruction pool. If an implementation could not be found even in this second attempt, the slave returns FAIL (Line 30).

4.3 Correctness

In this sub-section, we present the soundness and completeness guarantees of McSynth-ML. Because McSynth-ML uses the same master as McSynth++, and correctness properties of McSynth++ have been proven elsewhere [Srinivasan et al. 2016], we only discuss correctness of McSynth-MLSlave (Alg. 3) in this section.

Theorem 1. ***Soundness.*** *Alg. 3 is sound. (The formula $\langle\langle I \rangle\rangle$ for instruction sequence I returned by Alg. 3 is logically equivalent to the input QFBV formula φ.)*

Proof. The CEGIS loop of the slave (Line 18 in Alg. 3) returns an instruction sequence I only if $\langle\langle I \rangle\rangle$ is equivalent to φ. □

McSynth-ML's slave has the same completeness guarantees as that of McSynth++ (Thm. 2 in [Srinivasan et al. 2016]).

Theorem 2. ***Completeness.*** *Modulo SMT timeouts and candidates that are pruned away because of imprecision in $BITS^{\#}_{required}(\varphi)$ ([Srinivasan et al. 2016, §4.2.2]), if there exists an instruction sequence I that (i) is equivalent to φ, and (ii) does not superfluously use/modify locations that are otherwise unused/unmodified by φ, then Alg. 3 will find I and terminate.*

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

and 15–17 in Alg. 3), and McSynth-ML's slave has the same completeness guarantees as that of McSynth++. □

4.4 Threats to Validity

There are three threats to the validity of our algorithms.

(1) The parameters of our models and algorithms need to be tuned for McSynth-ML to synthesize code effectively. If the parameters are significantly off, one cannot guarantee low synthesis times using McSynth-ML. (§5 describes how we did the tuning for our experiments.)
(2) The models used by McSynth-ML should be trained with sufficient training data; failure to do so might result in higher synthesis times.
(3) If one wishes to synthesize code that possesses a certain "quality" using McSynth-ML, the implementations in the training data should also possess that "quality." For example, if one wishes to find optimal implementations using McSynth-ML, one should generate training examples using a superoptimizer [Bansal and Aiken 2006, 2008; Joshi et al. 2002; Massalin 1987; Phothilimthana et al. 2016a,b; Schkufza et al. 2013]. For our experiments, the instruction sequences in our training data come from code sequences generated by a standard compiler (`gcc -O2`), and so the implementations produced by McSynth-ML resemble those produced by a compiler.

5 IMPLEMENTATION

Because McSynth-ML is an extension of McSynth++, McSynth-ML has the same underlying components as McSynth++: Transformer Specification Language (TSL) [Lim and Reps 2013] to convert instruction sequences into QFBV formulas; ISAL [Lim and Reps 2013, §2.1] to generate the templatized instruction pool for synthesis; and Yices [Dutertre and de Moura 2006] as its SMT solver. Just like McSynth++, in McSynth-ML, memory is addressed at the level of individual bytes; McSynth-ML is also capable of accepting scratch registers for synthesis [Srinivasan and Reps 2015b, §4.4]. To implement k-NN regression, we used the scikit-learn toolkit [Pedregosa et al. 2011]. We used the MIT Language Modeling Toolkit [Hsu and Glass 2008] for our n-gram model.

6 EXPERIMENTS

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

(1) In comparison with McSynth++, what is the speedup in synthesis time caused by McSynth-ML's best-first search assisted only by the n-gram model ($\lambda = 0.0$)? What is the speedup when McSynth-ML's best-first search is assisted only by k-NN regression ($\lambda = 1.0$)? What is the speedup when the search is assisted by both models ($\lambda = 0.25$)?
(2) In comparison with McSynth++, on how many formulas does McSynth-ML timeout?
(3) In comparison with McSynth++, what is the reduction in the size of the instruction pool caused by McSynth-ML's k-NN-based instruction-pool truncation?
(4) To help understand the breakdown of (potential) benefits of (i) instruction-pool truncation, and (ii) model-assisted search, how well does McSynth++ perform when its slaves are equipped with only k-NN-based instruction-pool truncation and model-assisted best-first search, respectively?

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.

Test suite ($\mathscr{T}$): To answer the questions, we used a test suite that is similar to the one used to test McSynth++ in prior work [Srinivasan et al. 2016], consisting of QFBV formulas obtained from a representative set of "important" instruction sequences that occur in real programs. In the remainder of this section, we use $\mathscr{T}$ to denote the test suite. We used 10 binaries from the SPECINT 2006 suite (astar, gcc, gobmk, h264ref, hmmer, libquantum, omnetpp, perl, sjeng, xalan). The total number of non-control-modifying instructions in all binaries is around 1.8 million. For each of the aforementioned SPECINT 2006 binaries, we harvested the most frequently occurring instruction sequence of lengths 6 through 10, respectively (50 instruction sequences in total), avoiding overlaps—i.e., for binary B, the fragments $\{I_6, I_7, \ldots, I_{10}\}$ were chosen to be disjoint. We converted each instruction sequence into a QFBV formula to produce a test suite of 50 formulas. This test suite is more challenging for the synthesizer than the test suite that was used to test McSynth++ in prior work [Srinivasan et al. 2016]: this test suite consists of larger formulas obtained from longer instruction sequences.

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

Test suite for estimating the weighting parameter λ and truncation-threshold parameter t_p ($\mathscr{T}'$): To create the test suite that was used to estimate λ and t_p, we used the same method as the one outlined above to create test suite $\mathscr{T}$, but we harvested the second most frequently occurring instruction sequence from each binary, again avoiding overlaps. Note that $\mathscr{T}$ and $\mathscr{T}'$ are disjoint.

Training corpus: If a test formula t in either $\mathscr{T}$ or $\mathscr{T}'$ came from an instruction sequence in binary B, the training data for t came from the 9 binaries *other than* B. For example, if a test formula t came from a six-instruction sequence harvested from binary B_1, the training data for t came from binaries $\{B_2, B_3, \ldots, B_{10}\}$. The goal was to keep the training and test data independent from each other.

To create the training data for a specific test formula t obtained from binary B, we harvested all straight-line instruction sequences of lengths 1 through 4 from binaries other than B, which resulted in a training corpus of around 4.4 million instruction sequences, and 612,000 unique instruction sequences (after canonicalizing immediate operands) on average. (We report the average because the size of the training data varies with the binary B from which t was obtained.)

Parameters for models: For our n-gram model, we chose to use Kneser-Ney smoothing. We used 10-fold cross-validation on the training corpus to pick 3 as the maximum length n up to which the n-gram model has to maintain n-gram counts (see Alg. 1). We set as 2 the hyperparameter k of k-NN regression (see Alg. 1) by the same cross-validation scheme.

Weighting parameter λ: We determined the weighting parameter λ (see Alg. 3) as follows: we used the separate test suite $\mathscr{T}'$ of 50 formulas, and for different values of λ, we measured the average synthesis time obtained via McSynth-ML for $\mathscr{T}'$. We were able to obtain the lowest average synthesis-time for the value $\lambda = 0.25$; so we set $\lambda = 0.25$ in our experiments.

Truncation-threshold parameter t_p: We determined the truncation-threshold parameter t_p (see Alg. 2) as follows: using untruncated pools (i.e., truncation-threshold probability = 0.0), we synthesized code for each formula in test suite $\mathscr{T}'$. We recorded k-NN-regression probabilities of opcode variants during each synthesis run for $\mathscr{T}'$. Suppose that O is the set of opcode variants of instructions occurring in the instruction sequences synthesized from formulas in $\mathscr{T}'$. We chose the truncation-threshold parameter t_p to be the smallest non-zero k-NN-regression probability recorded for any opcode variant in O. The value was around 0.1. □

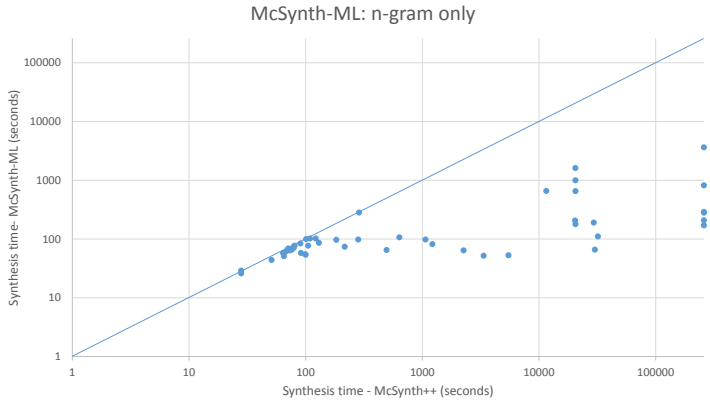

Fig. 4. Effect of only the n-gram model assisting McSynth-ML's best-first search for the corpus of 50 QFBV formulas.

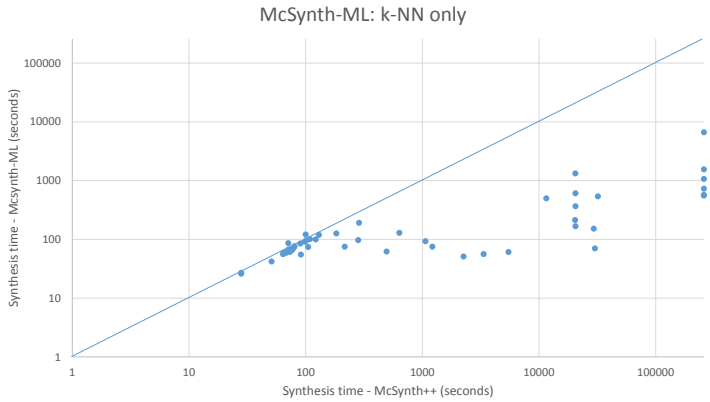

Fig. 5. Effect of only k-NN regression assisting McSynth-ML's best-first search for the corpus of 50 QFBV formulas.

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

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

The results are shown in Figs. 4, 5, and 6, respectively. In the figures, the blue lines represent the diagonals of the scatter plots. If a point lies below and to the right of the diagonal, the baseline performs worse. All axes use logarithmic scales. McSynth++ timed out on 6 formulas. (The timeout value was three days.) McSynth-ML did not timeout on any formula in the test suite. For the formulas that timed out in McSynth++ but did not timeout in McSynth-ML, the average speedup in

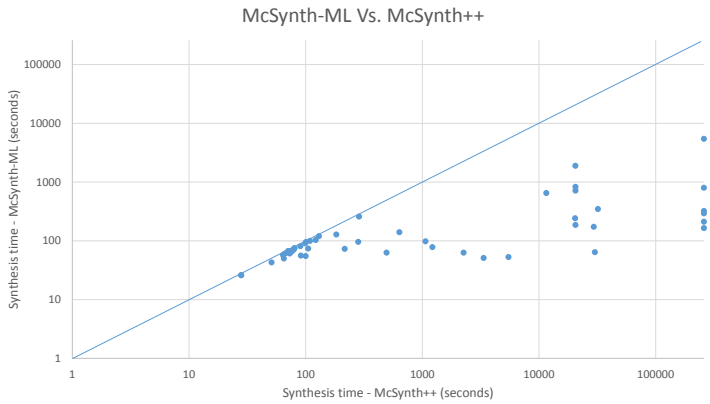

Fig. 6. Effect of both models assisting McSynth-ML's best-first search for the corpus of 50 QFBV formulas.

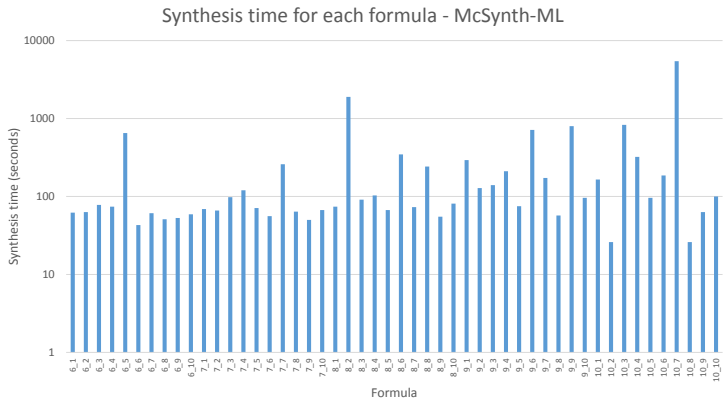

Fig. 7. Per formula synthesis-time numbers. (Formulas are sorted by the length of the instruction sequence that produced the formula.)

synthesis time caused by improvements (i), (ii), and (iii) are over 573×, 222×, and 526×, respectively (computed as a geometric mean). For the formulas that did not timeout, the average speedup in synthesis time caused by improvements (i), (ii), and (iii) are 4.5×, 4.56×, and 4.55×, respectively. If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedups are more pronounced: 12.3×, 13×, and 12.6×, respectively. The n-gram model has a more pronounced effect on the speedup for formulas that timed out in McSynth++, whereas k-NN regression has a more pronounced effect on the speedup for the other formulas.

One can see that the points in Figs. 4–6 form clusters. The points that occupy similar positions in the three scatter plots are actually the same formulas. The following are the characteristics of formulas in various clusters:

- Cluster along the diagonal: The master breaks these formulas into a sequence of sub-formulas such that each sub-formula can be implemented by a one-instruction sequence. Consequently, the slaves of McSynth++ find the implementations quickly, and the improvements in McSynth-ML do not cause much of a difference.

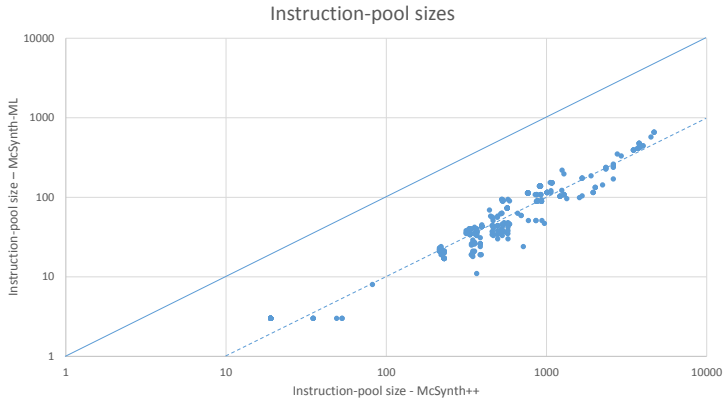

Fig. 8. Reduction in instruction-pool sizes caused by McSynth-ML's k-NN-based truncation.

- Vertical cluster at the right end of the scatter plot: These are the six formulas that timeout in McSynth++. Three of the formulas come from instruction sequences of length 9, and three from sequences of length 10. The master breaks each of these formulas into a sequence of sub-formulas such that there exists at least one sub-formula that can only be implemented by a three-instruction sequence. McSynth++ times out attempting to find such an instruction sequence.
- Other two clusters: The master breaks these formulas into a sequence of sub-formulas such that each sub-formula can be implemented by a one-instruction sequence or a two-instruction sequence. The slaves of McSynth++ do not timeout for these formulas, but spend a non-trivial amount of time trying to find an implementation.

Fig. 7 shows the synthesis-time numbers produced by McSynth-ML on a per-formula basis. Each formula in the test suite is identified by m_n, where m is the length of the instruction sequence that produced the formula, and n identifies a specific formula. The y axis uses a logarithmic scale. One can see that the synthesis time does not increase with m. In general, if the master breaks up a formula φ into a sequence of sub-formulas such that the sub-formulas can only be implemented by longer instruction sequences, synthesis of code for φ takes longer. For example, suppose that φ_1 is obtained from an instruction sequence of length 6 and φ_2 from a sequence of length 10. However, all sub-formulas of φ_2 can be implemented by one-instruction sequences, whereas φ_1 has a sub-formula that can only be implemented by a three-instruction sequence; then the synthesis time for φ_1 will be greater than that for φ_2.

To answer the third research question, we measured the sizes of the instruction pools used by McSynth++ and McSynth-ML, respectively, for each formula in the test suite. The results are shown in Fig. 8. The blue line represents the diagonal of the scatter plot. If a point lies below and to the right of the diagonal, the baseline has a larger instruction pool. The axes use logarithmic scales. Synthesis of code for each formula in the test suite requires several slave invocations, and each slave invocation creates a new instruction pool, which is represented by a single data point in Fig. 8. Consequently in Fig. 8, there are more than 50 data points (369 exactly). The average reduction in instruction-pool size caused by truncation based on k-NN-regression probabilities is 9.83× (computed as a geometric mean). One can notice this order-of-magnitude improvement in Fig. 8: all the data points roughly lie along the dotted line that represents an order-of-magnitude improvement over the baseline.

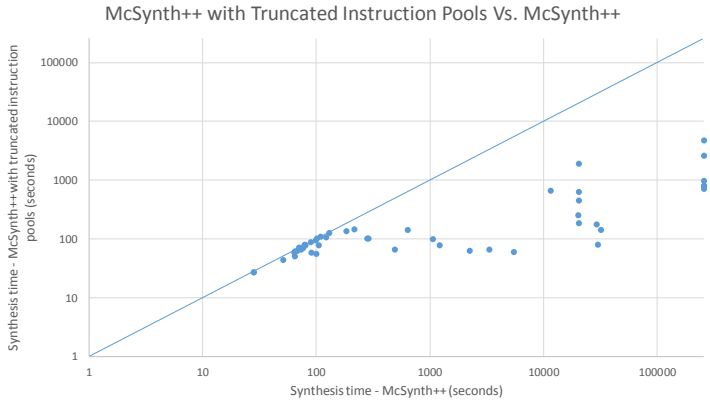

Fig. 9. Effect of k-NN-based instruction-pool truncation on McSynth++ for the corpus of 50 QFBV formulas.

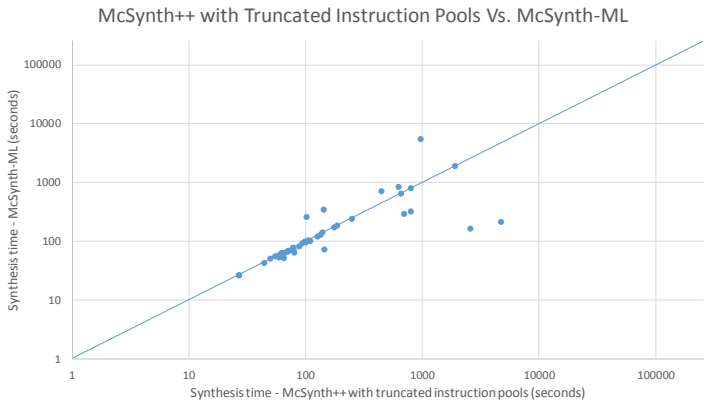

Fig. 10. Comparison of McSynth-ML against McSynth++ with truncated instruction pools.

To answer the fourth research question, we performed two control experiments in which we used in McSynth++'s slaves (i) linear search with instruction pools truncated by k-NN regression, and (ii) model-assisted best-first search with untruncated pools. The goals of these experiments are to demonstrate how well (i) the linear search in McSynth++ can perform when it is presented with the same instruction pools as McSynth-ML, and (ii) the model-assisted search in McSynth-ML can perform when it is presented with the same instruction pools as McSynth++. We compared the numbers against the synthesis-time numbers obtained from McSynth++ and McSynth-ML. The results are shown in Figs. 9–12. The blue lines represent the diagonals of the scatter plot. If a point lies below and to the right of the diagonal, the baseline performs worse. All axes use logarithmic scales.

For the 6 formulas that timed out in McSynth++, the average speedup in synthesis time caused only by truncating instruction pools is over 200× (computed as a geometric mean), versus 526× speedup for McSynth-ML. For 2 out of these 6 formulas, linear search performed better than model-assisted best-first search. For the formulas that did not timeout, the average speedup in synthesis time caused only by truncating instruction pools is 4× (computed as a geometric mean),

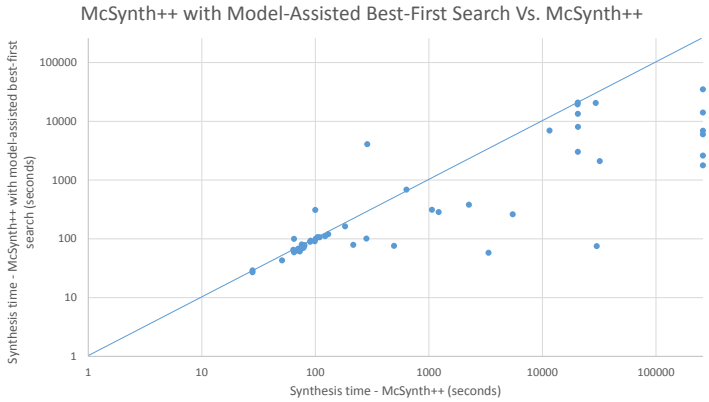

Fig. 11. Effect of model-assisted best-first search on McSynth++ for the corpus of 50 QFBV formulas.

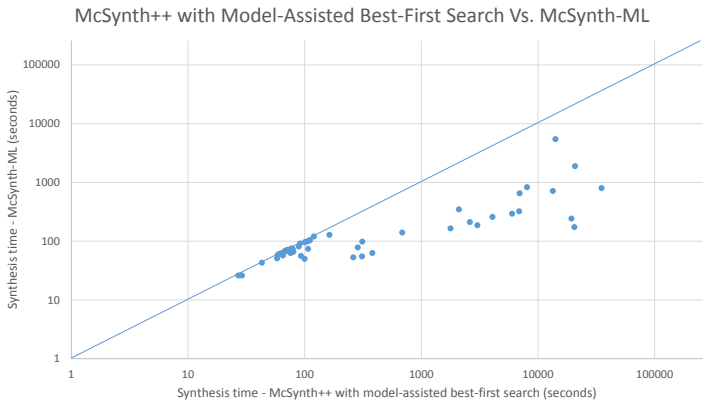

Fig. 12. Comparison of McSynth-ML against McSynth++ equipped with model-assisted best-first search.

versus 4.55× for McSynth-ML. For 5 out of these 44 formulas, linear search performed better than model-assisted best-first search.

For the 6 formulas that timed out in McSynth++, the average speedup in synthesis time caused only by the model-assisted best-first search is over 38× (computed as a geometric mean), versus 526× speedup for McSynth-ML. Model-assisted best-first search performed better than linear search for all of these 6 formulas. (McSynth++ equipped with model-assisted search did not timeout for any of these 6 formulas.) For the formulas that did not timeout in McSynth++, the average speedup in synthesis time caused only by the model-assisted best-first search is 1.78× (computed as a geometric mean), versus 4.55× for McSynth-ML. For 10 out of these 44 formulas, linear search performed better than model-assisted best-first search. One can see that k-NN-based instruction-pool truncation has a more pronounced effect on the speedup than model-assisted best-first search.

A summary of our experimental results is presented in Table 1. In summary, in comparison with McSynth++, McSynth-ML speeds up the synthesis time by over 526× for the 6 formulas that timed out in McSynth++ but did not timeout in McSynth-ML. Moreover, McSynth-ML does not timeout on any formula in our test suite. For the formulas that did not timeout in McSynth++, McSynth++

Table 1. Summary of experimental resuts.

	Search strategy						
Instruction-pool truncation via k-NN		Linear			Model-assisted best-first		
		No. of timeouts	Speedup for formulas that timeout	Speedup for remaining formulas	No. of timeouts	Speedup for formulas that timeout	Speedup for remaining formulas
	No	6	--	--	0	At least 38×	1.78×
	Yes	0	At least 200×	4×	0	At least 526×	4.55×

speeds up the synthesis time by 4.55×. If we consider only formulas whose baseline synthesis-time numbers are 100 seconds or more, the speedup is 12.6×.

We also summarize the speedups produced by McSynth-ML in comparison with those produced by McSynth [Srinivasan and Reps 2015b] and McSynth++.

- In comparison with a baseline enumerative machine-code synthesizer, McSynth's search-space pruning heuristics reduce the synthesis time by a factor of 473, and its divide-and-conquer strategy reduces the synthesis time by a further 3 to 5 orders of magnitude.
- McSynth++ synthesizes code for 12 out of 14 formulas on which McSynth times out, speeding up the synthesis time by at least 1981×, and for the remaining formulas, speeds up synthesis by 3×.
- McSynth-ML synthesizes code for 6 out of 50 formulas on which McSynth++ times out, speeding up the synthesis time by at least 526×, and for the remaining formulas, speeds up synthesis by 4.55×. (The test suite of formulas used to compare McSynth++ to McSynth was different from the test suite used to compare McSynth-ML to McSynth++.)

7 RELATED WORK

Search strategies in superoptimization. Superoptimization aims to find an optimal instruction sequence for a target instruction-sequence. (A detailed comparison of superoptimization and machine-code synthesis is available elsewhere: see the paragraph titled "Superoptimization" in [Srinivasan et al. 2016, §7]. While early attempts at superoptimization used linear search as the core search strategy [Bansal and Aiken 2006, 2008; Joshi et al. 2002; Massalin 1987], modern superoptimizers use other approaches.

The stochastic superoptimizer STOKE [Schkufza et al. 2013] uses stochastic techniques for finding an optimal instruction sequence. STOKE uses Markov Chain Monte Carlo (MCMC) sampling to search through the space of instruction sequences, and encodes its cost heuristic in terms of correctness and performance. To find implementations that are algorithmically different from the input instruction-sequence, STOKE begins its search from random points in the space of instruction sequences (instead of starting the search from the input instruction-sequence). The stochastic search strategy, along with the fast MCMC sampling allows STOKE to quickly synthesize larger programs (10 - 15 x86 instructions).

The GreenThumb superoptimizer framework [Phothilimthana et al. 2016b] employs a cooperative search strategy: complementary enumerative, symbolic, and stochastic search strategies work in conjunction by exchanging the best programs each search instance has discovered so far. The cooperative strategy, along with other improvements, allow instantiations of the GreenThumb framework to perform better than STOKE: the GreenThumb instantiations optimize the benchmarks faster, and obtain better (faster) implementations.

STOKE and GreenThumb employ stochastic strategies for the search, which renders the search incomplete. In comparison, McSynth-ML uses a model-assisted best-first search, which merely prioritizes (reorders) the candidates during search, while maintaining completeness guarantees. Moreover, unlike McSynth-ML, superoptimizers only use information obtained from the input

instruction-sequence to guide the search; they do not use models learned from code-sequences to guide the search.

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

- SLANG is a code-completion tool that uses API calls to fill holes in a partial program [Raychev et al. 2014]. SLANG learns two language models from a corpus of API-call sequences harvested from programs: an n-gram model and a recurrent neural-network (RNN) model. SLANG fills the holes in a test program with the most likely API calls according to the models.
- *anyCode* synthesizes well-formed Java expressions from free-form queries containing a mixture of English and Java [Gvero and Kuncak 2015]. *anyCode* uses a conjunction of the following to synthesize and rank expressions: (i) natural-language processing (NLP) tools to process the input English text, and (ii) language models and a probabilistic context-free-grammar model built from corpus of Java programs.
- JSNICE predicts names and types for identifiers in JavaScript programs [Raychev et al. 2015]. JSNICE learns a probabilistic model (employing conditional random fields) for program properties from a huge corpus of existing programs, and uses the model to predict names and type annotations in a test program.

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

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

8 CONCLUSION

In this paper, we presented McSynth-ML, the first low-level-code synthesizer that uses machine learning to assist synthesis. Instead of the linear search used by McSynth++ (a state-of-the-art machine-code synthesizer), McSynth-ML uses a novel model-assisted best-first search as the core search strategy. The cost heuristic for the search comes from two models trained over a corpus of 4.4 million specifications (QFBV formulas) and implementations (instruction sequences). One model is an n-gram-based language model, which steers the search towards common/useful instruction sequences. The other is a k-NN-regression model that correlates features of implementations with features of specifications, and steers the search towards instruction sequences that are highly likely to implement the input formula. We evaluated McSynth-ML on a test suite consisting of 50 formulas. Our experiments show that McSynth-ML synthesizes code for all 6 of the formulas on which McSynth++ times out, speeding up their synthesis by at least 526×. For the remaining 44 formulas, McSynth-ML speeds up synthesis by 4.55×.

One possible direction for future work is to develop binary-rewriting clients that deal with large QFBV formulas, and subsequently measure the improvements to rewriting times caused by McSynth-ML. Existing binary rewriters that use McSynth++ include the machine-code partial evaluator WiPEr [Srinivasan and Reps 2015a] and the machine-code slicer McSlice [Srinivasan and Reps 2016]. However, both WiPEr and McSlice perform rewriting on a per-instruction basis, and produce formulas that could be implemented mostly by one-instruction sequences (and sometimes by two-instruction sequences). For these small formulas, McSynth++ proves to be sufficient for

purposes of synthesis. Potential candidates for rewriting tools that deal with larger formulas are tools that perform rewriting on a per-basic-block basis: binary translators [Bansal and Aiken 2008], deobfuscators [Yadegari et al. 2015], etc. It remains for future work to implement such a tool, and compare the rewriting times obtained via McSynth++ and McSynth-ML.

ACKNOWLEDGMENTS

This work was supported in part by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; by NSF under grant CCF-1423237; 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.

Thomas Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

REFERENCES

M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. 2005. Control-flow Integrity. In *CCS*.

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

S. Bansal and A. Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *ASPLOS*.

S. Bansal and A. Aiken. 2008. Binary Translation Using Peephole Superoptimizers. In *OSDI*.

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

B. Dutertre and L. de Moura. 2006. Yices: An SMT Solver. (2006). http://yices.csl.sri.com/.

K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. 2013. Scalable Variable and Data Type Detection in a Binary Rewriter. In *PLDI*.

Ú. Erlingsson and F.B. Schneider. 1999. SASI Enforcement of Security Policies: A Retrospective. In *Workshop on New Security Paradigms*.

T. Gvero and V. Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *OOPSLA*.

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

B. Hsu and J. Glass. 2008. Iterative Language Model Estimation: Efficient Data Structure and Algorithms. In *Interspeech*.

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

J. Lim, A. Lal, and T. Reps. 2011. Symbolic Analysis via Semantic Reinterpretation. *Softw. Tools for Tech. Transfer* 13, 1 (2011), 61–87.

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

H. Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *ASPLOS*.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

P. Phothilimthana, A. Thakur, R. Bodik, and D. Ghurjati. 2016a. *GreenThumb: Superoptimizer Construction Framework*. UCB/EECS-2016-8. University of California–Berkeley Tech Report. http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-8.pdf

P. Phothilimthana, A. Thakur, R. Bodik, and D. Ghurjati. 2016b. Scaling up Superoptimization. In *ASPLOS*.

V. Raychev, M. Vechev, and A. Krause. 2015. Predicting Program Properties from"Big Code". In *POPL*.

V. Raychev, M. Vechev, and E. Yahav. 2014. Code Completion with Statistical Language Models. In *PLDI*.

H. Saïdi. 2008. Logical Foundation for Static Analysis: Application to Binary Static Analysis for Security. *ACM SIGAda Ada Letters* 28, 1 (2008), 96–102.

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

A. Slowinska, T. Stancescu, and H. Bos. 2012. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In *ATC*.

D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Int. Conf. on Information Systems Security*.

V. Srinivasan and T. Reps. 2015a. Partial Evaluation of Machine Code. In *OOPSLA*.

V. Srinivasan and T. Reps. 2015b. Synthesis of Machine Code from Semantics. In *PLDI*.

V. Srinivasan and T. Reps. 2016. An Improved Algorithm for Slicing Machince Code. In *OOPSLA*.

V. Srinivasan, T. Sharma, and T. Reps. 2016. Speeding-up Machine-Code Synthesis. In *OOPSLA*.
B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *S&P*.