

Automated Synthesis of Symbolic Instruction Encodings from I/O Samples

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Ankur Taly*
Stanford University
ataly@stanford.edu

Abstract

Symbolic execution is a key component of precise binary program analysis tools. We discuss how to automatically boot-strap the construction of a symbolic execution engine for a processor instruction set such as x86, x64 or ARM. We show how to automatically synthesize symbolic representations of individual processor instructions from input/output examples and express them as bit-vector constraints. We present and compare various synthesis algorithms and instruction sampling strategies. We introduce a new synthesis algorithm based on *smart sampling* which we show is one to two orders of magnitude faster than previous synthesis algorithms in our context. With this new algorithm, we can automatically synthesize bit-vector circuits for over 500 x86 instructions (8/16/32-bits, outputs, EFLAGS) using only 6 synthesis templates and in less than two hours using the Z3 SMT solver on a regular machine. During this work, we also discovered several inconsistencies across x86 processors, errors in the x86 Intel spec, and several bugs in previous manually-written x86 instruction handlers.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Program Synthesis, Symbolic Execution, x86

1. Introduction

Symbolic execution is a key component of precise binary program analysis tools, for test generation [6, 17], program verification [2, 20], malware analysis [1, 23], and other applications. Symbolic execution engines are traditionally written by hand [1, 2, 6, 17, 20, 23]: the effect of executing each individual instruction is described by a symbolic constraint, called *symbolic instruction encoding*, written manually and derived by reading the processor instruction manual. Unfortunately, the semantics of the instruction set of general-purpose processors such as x86, x64 or ARM

is very complex. For instance, x86 includes hundreds of instructions whose semantics is described in more than 2,000 pages divided in 3 volumes. This complexity makes the manual development of symbolic-execution engines tedious (many instructions), error-prone (many corner cases), partial (not all instructions are usually covered) and imprecise (approximations are often used). Moreover, the official reference manual is often under-specified and may itself contain errors.

In this work, we explore a radically different approach to developing symbolic-execution engines: what if most symbolic instruction encodings could be *synthesized automatically*?

To do this, we study how to adapt and extend recent advances on *automatic program synthesis*. Given a functional *specification* and a set of building blocks, called *components*, possibly combined together as described in a solution *template* or program sketch (i.e., a program with holes), automatic program synthesis consists of searching the space of all possible template completions for a fully-defined program (i.e., with no holes left) that satisfies the specification. In our context, we do not have access to a full functional specification of individual processor instructions — such a specification is precisely what we want to infer. But we have access to a cheap and fast *specification oracle*: we can execute instructions on a processor and observe their input/output behaviors.

Program synthesis from I/O samples has been recently investigated in [13]. There, an *I/O oracle-guided* synthesis algorithm is presented for loop-free programs. This algorithm consists of computing a set of I/O samples, then synthesizing a candidate program that satisfies those samples (to check whether such a program exists), then computing a *distinguishing input* that distinguishes this candidate program P from some other non-equivalent candidate program P' (to check whether P is unique), and if such an input exists, then query the I/O oracle with this distinguishing input to eliminate either P or P' as a possible solution. This *counter-example-guided iterative synthesis* process is repeated until one unique solution remains, or no solution exists if the synthesis template is too constrained. The algorithm assumes the existence of a *verification oracle* which can determine whether a solution is “correct”.

In our context, we do not have access to such a verification oracle. For instructions with small I/O signatures, such as 8-bit instructions, exhaustive testing can provide a verification oracle, but exhaustive testing does not scale to 16-bit or 32-bit instructions. Another practical hurdle is that the counter-example-guided iterative synthesis algorithm of [13] can be very expensive when many iterations are needed, as we will show with results of experiments in Section 6.

To improve on this, we propose a new synthesis algorithm based on *smart sampling* which we show is one to two orders of magnitude faster than previous synthesis algorithms in our context. Given a specific template, the main idea is to generate upfront a set of distinguishing inputs which uniquely determine each possible so-

* The work of this author was done mostly while visiting Microsoft.

lution refining the template. This way, our new synthesis algorithm converges faster to the unique solution, without requiring any additional expensive synthesis-refinement steps. Synthesis with smart sampling is more efficient when a template is repeatedly used for many instructions, as is the case in our context.

In this work, we want to *automatically generate concise yet precise* symbolic instruction encodings that can be used for bit-precise symbolic execution of large programs and long execution paths. Conciseness is important for scalability, while precision is key to detect subtle program bugs (for instance due to integer overflows). For these reasons, we adopt *bit-vector constraints* supported by SMT solvers as synthesis components.

For automatic synthesis to be practical, synthesis templates should be constrained enough to define a tractable search space, yet abstract enough to allow a simple representation of many possible solutions. Perhaps surprisingly, we show that the I/O behavior of over 500 x86 instructions (8/16/32-bits, outputs, EFLAGS) can be precisely captured with only 6 synthesis templates. Using these templates and our new smart sampling synthesis algorithm, we can automatically synthesize bit-vector circuits for all those 500+ instructions in less than two hours using the Z3 SMT solver on a regular x86 machine. Moreover, the size of the synthesized circuits is either constant or linear in the number of input/output bits, satisfying our conciseness requirement.

This paper is organized as follows. In Section 2, we precisely define the problem addressed in this work. Then, we review in Section 3 existing automatic synthesis approaches and discuss their applicability to our context. We introduce in Section 4 our new synthesis algorithm based on smart sampling. In Section 5, we present 6 synthesis templates that together abstract the semantics of over 500 x86 instructions, and we discuss properties of those templates. Next, we compare in Section 6 the performance of several synthesis algorithms with those templates. We discuss our overall results for x86, lessons learned and limitations in Section 7. We then discuss other related work in Section 8, and conclude in Section 9.

2. Problem Definition

We consider a *processor* which can execute a set of *instructions*. In this work, we focus on ALU instructions and will not consider floating point and SIMD instructions. We will also ignore specific addressing modes and assume that an instruction has some inputs and outputs, but we will not distinguish where those inputs or outputs are being stored, e.g., in a register or a memory location. However, we do consider the sizes of the input and output arguments of an instruction, which we assume are known (i.e., are not inferred automatically).

Formally, we define an instruction instance I as a *function* that takes a known fixed ordered set \vec{i} of *inputs*, each of a known fixed size, which may be read during the execution of the instruction, and returns a known fixed ordered set \vec{o} of *outputs*, each of a known fixed size, which may be written during the execution of the instruction. We thus assume that the execution of each instruction instance is *deterministic* and always terminates. In what follows, we will treat each output o in \vec{o} separately, abuse notation by writing $o = f(\vec{i})$, and calling such a function an *instruction instance*.

For instance, SHL is an x86 instruction, while SHL8 is an instruction instance that takes two 8-bit values as inputs and returns an 8-bit value as output representing the main result, and SHL8^{CF} is another instruction instance that takes again two 8-bit values as inputs but returns a boolean value representing the value of the CF flag (part of the x86 “EFLAGS”) after executing the instruction.

The problem we consider in this work is to automatically synthesize a (concise and precise) representation of function f for a

given instruction instance. We call such a representation a *symbolic representation*, or *symbolic encoding*, or *instruction handler*.

However, function f is unknown. All we are given to learn about f is a processor \mathcal{P} implementing the instruction instance, which we can *sample* by providing some input values, executing the instruction instance, and then observing the output value. In other words, processor \mathcal{P} is a *black-box input/output oracle* for instruction instance I . This oracle is denoted $\Phi(\mathcal{P}, I)$ in what follows.

Because we want to generate concise and precise symbolic representations, we will represent inputs and outputs by bit-vectors, and functions by logic expressions using the theory of bit-vectors as defined by modern SMT solvers. We will sometimes call such function representations *circuits*. If \vec{i} represents two inputs i_1 and i_2 , each of size s , we will write $i_1[j]$ with $0 \leq j < s$ to denote the j th bit of i_1 .

In summary, the problem considered in this work is:

Given a black-box processor \mathcal{P} and an instruction instance I , how to automatically synthesize a function f that is semantically equivalent to the oracle $\Phi(\mathcal{P}, I)$?

3. Synthesis Procedures

In this section, we review prior synthesis approaches to the function synthesis problem given a black-box I/O oracle $\phi = \Phi(\mathcal{P}, I)$. We present two procedures ExhaustVal and DInputVal and discuss their correctness and scalability.

Both procedures involve three stages: a *sampling* stage, a *synthesis* stage, and a *verification* stage. In the sampling stage, the I/O oracle is queried on an initial set of inputs and a set of I/O samples \mathcal{S} is obtained. In the synthesis stage, a function is synthesized whose behavior respects the samples \mathcal{S} . This is done by using a *template-based approach* (see below) to find a function f that satisfies $\bigwedge_{\vec{i}, o \in \mathcal{S}} f(\vec{i}) = o$. The synthesized function f is then passed to the verification stage to check whether it matches the I/O oracle on other samples outside \mathcal{S} . Samples that fail the verification check are sent back to the synthesis stage, and the procedure is repeated until the verification check succeeds. The two procedures differ in the specific verification checks used.

Before presenting the two procedures in more details, we review the motivation for template-based synthesis.

3.1 Template-Based Synthesis

A direct approach for synthesizing a function f satisfying the samples \mathcal{S} is to check the satisfiability of the formula

$$\exists f : \bigwedge_{\vec{i}, o \in \mathcal{S}} f(\vec{i}) = o$$

Unfortunately, this second-order logic formula can be expensive or even impossible to check. A common approach to get rid of the quantification over functions is to use a *function template*. Informally, a function template T is a function with some free variables \vec{c} called *coefficients*. Instantiating the coefficients with concrete values defines a closed (i.e., fully-defined) function $T(\vec{c})$, called *concretization*. The set $\gamma(T)$ of all possible concretizations of a template T is thus defined as the set $\{C \mid \exists \vec{c} : C = T(\vec{c})\}$. By replacing the function f in the synthesis formula above by a function template T and by existentially quantifying its coefficients, the synthesis problem is reduced to satisfiability checking of a first-order logic formula

$$\text{SYN}_{T, \mathcal{S}}(\vec{c}) := \exists \vec{c} : \bigwedge_{\vec{i}, o \in \mathcal{S}} T(\vec{c}, \vec{i}) = o$$

where $T(\vec{c}, \vec{i})$ denotes application of function $T(\vec{c})$ with inputs \vec{i} .

ExhaustVal(ϕ, T, n_{syn})

1. $\mathcal{I} := n_{syn}$ valid inputs for ϕ picked randomly
2. $\mathcal{I}_{ex} :=$ All valid inputs for ϕ
3. $\mathcal{S} := \text{Sample}(\mathcal{I}, \phi)$
4. $\mathcal{S}_{ex} := \text{Sample}(\mathcal{I}_{ex}, \phi)$
5. $\vec{\alpha} := \text{SAT}(\text{SYN}_{T,S}(\vec{c}))$ // returns \perp if UNSAT
6. **if** ($\vec{\alpha} = \perp$) **return** “Insufficient Template”
7. $C := T(\vec{\alpha})$
8. $\mathcal{S}_{fail} := \text{Verify}(C, \mathcal{S}_{ex})$
9. **if** ($\mathcal{S}_{fail} == \emptyset$) **return** C
10. $\mathcal{S} := \mathcal{S} \cup \mathcal{S}_{fail}$
11. **goto** step 5

Figure 1. Procedure ExhaustVal: Exhaustive Validation

If the template T is expressed as a quantifier-free formula using the theory of bit-vectors and if the set of possible values for the coefficients is finite, checking the satisfiability of the formula $\text{SYN}_{T,S}(\vec{c})$ is decidable.

If the above formula is unsatisfiable, then the template cannot be used to synthesize a function that satisfies the samples \mathcal{S} . We define this property of *template sufficiency* as follows.

DEFINITION 1. [Template Sufficiency] A template T is *sufficient* for abstracting a function C if $C \in \gamma(T)$. ■

We now describe the two procedures ExhaustVal and DInputVal for efficiently solving the template-based synthesis problem. In what follows, $\text{Sample}(\mathcal{I}, \phi)$ for a set of inputs \mathcal{I} and an I/O oracle $\phi = \Phi(\mathcal{P}, I)$ denotes the set $\{(\vec{\alpha}, \phi(\vec{\alpha})) \mid \vec{\alpha} \in \mathcal{I}\}$ of samples (I/O pairs) obtained by executing each input by the oracle. Moreover, $\text{Verify}(C, \mathcal{S})$ for a function C and a set of samples \mathcal{S} denotes the set $\{(\vec{i}, o) \in \mathcal{S} \mid C(\vec{i}) \neq o\}$ of samples which do not agree with C (if any).

3.2 Procedure ExhaustVal

The procedure ExhaustVal is described in Figure 1. It takes as input an I/O oracle ϕ , a template T and a number of synthesis samples n_{syn} .

During the sampling stage (lines 1 – 4), n_{syn} valid inputs are chosen randomly to define the set \mathcal{I} , while the set \mathcal{I}_{ex} contains all possible valid inputs. Next, two sets \mathcal{S} and \mathcal{S}_{ex} of I/O samples are constructed by querying the oracle ϕ on inputs in \mathcal{I} and \mathcal{I}_{ex} respectively.

During the synthesis stage (lines 5 – 8), the synthesis formula $\text{SYN}_{T,S}(\vec{c})$ is checked for satisfiability. If the formula is unsatisfiable then the procedure returns “Insufficient Template”, otherwise the satisfying assignment $\vec{\alpha}$ for all the template coefficients is used to construct the function $C := T(\vec{\alpha})$.

The verification stage (lines 9 – 11) checks whether the synthesized function C agrees with all possible I/O samples \mathcal{S}_{ex} using the procedure Verify. The procedure returns the set \mathcal{S}_{fail} of all samples that fail. If $\mathcal{S}_{fail} = \emptyset$ then the function C is returned, else the set of failed samples \mathcal{S}_{fail} are added to the set of synthesis samples \mathcal{S} and the procedure loops back to the synthesis stage (line 3).

Note that the synthesis stage is more expensive (NP-hard) than the verification stage (linear in the size of each sample), which explains why the procedure ExhaustVal does not consider immediately the \mathcal{S}_{ex} in its synthesis stage. We now state the main properties of the procedure ExhaustVal.

DInputVal($\phi, T, n_{syn}, n_{ver}$)

1. $\mathcal{I} := n_{syn}$ valid inputs for ϕ picked randomly
2. $\mathcal{S} := \text{Sample}(\mathcal{I}, \phi)$
3. $\vec{\alpha} := \text{SAT}(\text{SYN}_{T,S}(\vec{c}))$ // returns \perp if UNSAT
4. **if** ($\vec{\alpha} = \perp$) **return** “Insufficient Template”
5. $C := T(\vec{\alpha})$
6. $\mathcal{I}_{ver} := n_{ver}$ valid inputs for ϕ picked randomly
7. $\mathcal{S}_{ver} := \text{Sample}(\mathcal{I}_{ver}, \phi)$
8. $\mathcal{S}_{fail} := \text{Verify}(C, \mathcal{S}_{ver})$
9. **if** ($\mathcal{S}_{fail} == \emptyset$)
10. $\vec{i} := \text{SAT}(\text{DISTINCT}_{T,C,S}(\vec{i}))$
11. **if** ($\vec{i} = \perp$) **return** C
12. $\mathcal{S} := \mathcal{S} \cup \text{Sample}(\{\vec{i}\}, \phi)$
13. **goto** step 3
14. $\mathcal{S} := \mathcal{S} \cup \mathcal{S}_{fail}$
15. **goto** step 3

Figure 2. Procedure DInputVal: Distinguishing-Input based Validation

THEOREM 1. Given a template T , an oracle ϕ , and any $n_{syn} > 0$, the following holds:

1. If the template T is sufficient for abstracting the oracle ϕ , then the procedure ExhaustVal(ϕ, T, n_{syn}) returns a function semantically equivalent to ϕ ;
2. Else the procedure returns “Insufficient Template”.

In other words, procedure ExhaustVal is both sound and complete, thanks to its exhaustive validation. Unfortunately, exhaustive validation does not scale to large inputs, such as 16-bit and 32-bit instructions. For instance, any 16-bit instruction instance with two 16-bit inputs requires an exhaustive sample set of 2^{32} samples, that is, more than a billion samples to verify.

3.3 Procedure DInputVal

We now present the procedure DInputVal, described in Figure 2 which offers weaker verification guarantees but scales to larger input signatures. Besides an I/O oracle ϕ and a template T , this procedure takes two additional inputs n_{syn} and n_{ver} denoting the number of synthesis and verification samples, respectively.

This procedure *assumes* that the input template T is sufficient for abstracting the I/O oracle ϕ . Its goal is to search through all functions abstracted by the template and to return one that is semantically equivalent to the oracle. If this assumption is right, the procedure returns a correct function. But if the assumption is wrong, it may either detect that the template is not sufficient, or return a wrong function.

Similarly to the procedure ExhaustVal, the sampling stage (lines 1 – 2) computes a set of synthesis samples \mathcal{S} by querying the oracle ϕ on a set \mathcal{I} of n_{syn} inputs chosen randomly. The synthesis stage (lines 3 – 5) checks for a satisfying assignment for the formula $\text{SYN}_{T,S}(\vec{c})$. If the formula is unsatisfiable then the procedure returns “Insufficient Template”, otherwise the satisfying assignment $\vec{\alpha}$ for all the template coefficients is used to construct the function $C := T(\vec{\alpha})$. The procedure differs from ExhaustVal in its verification stage (lines 6 – 15). We explain the main idea first and then present the details.

Intuitively, if we assume that the template T is sufficient for abstracting the oracle ϕ , and if we can show that all the concretization functions for T that satisfy samples \mathcal{S} are all semantically equivalent, then the synthesized function C is semantically equivalent to the oracle. In order to show this, we make use of the *distinguishing input* check, introduced in [13]. A distinguishing input $\text{DISTINCT}_{T,C,S}(\vec{i})$ for a function C , a template T and a set \mathcal{S} of samples is an input \vec{i} that can be used to distinguish C from another function that also concretizes the template and satisfies all the samples in \mathcal{S} . Formally, we define $\text{DISTINCT}_{T,C,S}(\vec{i})$ as

$$\exists \vec{c}: \left(\bigwedge_{\vec{j}, o \in \mathcal{S}} T(\vec{c}, \vec{j}) = o \right) \wedge T(\vec{c}, \vec{i}) \neq C(\vec{i})$$

If an input \vec{i} satisfies the above formula, then there are at least two non-equivalent functions that concretize the template and satisfy the samples in \mathcal{S} ; therefore, by querying the oracle with \vec{i} and adding the resulting I/O sample to \mathcal{S} , we can strictly reduce the number of non-equivalent functions that concretize the template and satisfy all previous samples. Otherwise, if the formula is unsatisfiable, then C is guaranteed to be equivalent to all other functions that concretize the template and satisfy all previous samples.

We now describe the pseudo-code for the validation stage. The first step (lines 6 – 8) is to sample n_{ver} inputs at random and query the oracle on them, thereby building the set of I/O samples \mathcal{S}_{ver} . Next the synthesized function C is verified against the samples \mathcal{S}_{ver} , and the samples that fail are collected in the set \mathcal{S}_{fail} . If $\mathcal{S}_{fail} \neq \emptyset$, then the samples are added (line 14) to the set of synthesis samples \mathcal{S} and the procedure loops back to the synthesis stage (line 3). If $\mathcal{S}_{fail} = \emptyset$, then the formula $\text{DISTINCT}_{T,C,S}(\vec{i})$ is checked for satisfiability. If it is unsatisfiable, then the function C is returned; otherwise, the oracle is queried with the satisfying assignment \vec{i} , the sample $\text{Sample}(\{\vec{i}\}, \phi)$ is added to the set of synthesis samples \mathcal{S} , and the procedure then loops back to the synthesis stage.

The reason for the verification stage with the samples \mathcal{S}_{ver} is to reduce the number of expensive satisfiability checks of formulas $\text{DISTINCT}_{T,C,S}(\vec{i})$. If the template is sufficient for abstracting the oracle, any input that distinguishes the synthesized function C from the oracle is also a distinguishing input. Thus, checking C against a set of randomly chosen samples provides a cheap way of searching for distinguishing inputs. We now state the main property of the procedure DInputVal .

THEOREM 2. *Given a template T , an oracle ϕ , and any $n_{syn}, n_{ver} > 0$, if T is sufficient for abstracting ϕ , then the procedure $\text{DInputVal}(\phi, T, n_{syn}, n_{ver})$ returns a function C semantically equivalent to ϕ .*

A useful corollary is that, if the procedure DInputVal returns “Insufficient Template” for any n_{syn}, n_{ver} , then the template T is indeed insufficient for abstracting ϕ . However, this theorem is weaker than Theorem 1 as it does not guarantee that the procedure returns “Insufficient Template” whenever the template is insufficient.

Just like procedure ExhaustVal , DInputVal provides a satisfactory solution to the instruction handler synthesis problem provided we can find a template that is sufficient for abstracting the oracle $\Phi(\mathcal{P}, I)$. Although the procedure DInputVal scales to instruction instances with large inputs, the running time can still be very long for some templates, as we will see in Section 6. The most expensive step in the procedure is checking the satisfiability of formulas $\text{DISTINCT}_{T,C,S}(\vec{i})$. Since the satisfiability of each such formula depends on a set \mathcal{S} of random samples, the running time of the procedure can vary significantly across various invocations. In the next section, we present a new synthesis procedure that pro-

vides the same correctness guarantee as the procedure DInputVal , but alleviates the above limitations.

4. Smart Sampling

In this section, we present a new template-based synthesis procedure SmartVal that provides the same correctness guarantee as procedure DInputVal but does not require any distinguishing input check. As a result, the procedure has a significantly better and more predictable running time than the procedure DInputVal . As with procedure DInputVal , the procedure SmartVal also assumes that the given template is sufficient for abstracting the given I/O oracle.

Given a specific template, the main idea is to generate upfront a set of distinguishing inputs which uniquely determine each possible solution refining the template. This way, the new synthesis algorithm converges directly to the unique solution, without requiring any other expensive synthesis-iteration and distinguishing-input steps.

Recall from section 3.3 that the distinguishing input check is performed to guarantee that for a template T and a set of synthesis samples \mathcal{S} , the synthesized function C is semantically equivalent to all functions that concretize template T and satisfy all the samples in \mathcal{S} . In order to avoid the distinguishing input check, we want to run a (unique) synthesis step with a set of samples obtained with an input set \mathcal{I} such that all the functions that concretize template T and satisfy all the samples in $\text{Sample}(\mathcal{I}, \phi)$, are all semantically equivalent. We call such an input set \mathcal{I} *smart* for the template T and oracle ϕ . Formally, we have the following.

DEFINITION 2. [Smart Inputs] A set \mathcal{I} of inputs is *smart* for a template T and an oracle ϕ if

$$\forall \vec{c}: \left(\bigwedge_{\vec{j}, o \in \mathcal{S}} T(\vec{c}, \vec{j}) = o \right) \Rightarrow \neg \text{DISTINCT}_{T,T(\vec{c}),\mathcal{S}}(\vec{i})$$

with $\mathcal{S} = \text{Sample}(\mathcal{I}, \phi)$. ■

When a set \mathcal{I} of inputs is smart for a template T and an oracle ϕ , we write $\text{Smart}_{T,\phi}(\mathcal{I})$.

The previous definition depends on a specific oracle ϕ . We can generalize it and define a stronger property on input sets, which we call *universal smartness* for a template T independently of any specific oracle ϕ . A set \mathcal{I} of inputs is universally smart for a template T if any two functions that concretize the template and agree on all the inputs in \mathcal{I} are semantically equivalent. Formally, we have the following.

DEFINITION 3. [Universally Smart Inputs] A set \mathcal{I} of inputs is *universally smart* for a template T , denoted by $\text{USmart}_T(\mathcal{I})$, if

$$\forall \vec{c}, \vec{d}, \vec{i}: \left(\bigwedge_{\vec{j} \in \mathcal{I}} T(\vec{c}, \vec{j}) = T(\vec{d}, \vec{j}) \right) \Rightarrow T(\vec{c}, \vec{i}) = T(\vec{d}, \vec{i}) \quad \blacksquare$$

By definition, a universally smart input set for a template is also smart for the template and any possible oracle ϕ .

LEMMA 3. $\text{USmart}_T(\mathcal{I}) \Rightarrow \forall \phi: \text{Smart}_{T,\phi}(\mathcal{I})$

Thus, for a fixed template, and given a set of universally smart inputs for that template, we are then guaranteed that, irrespective of the oracle, all functions synthesized by sampling those inputs do not require any distinguishing input check.

The procedure SmartVal is described in Figure 3. It takes as input an I/O oracle ϕ , a template T and a set \mathcal{I} of inputs such as $\text{Smart}_{T,\phi}(\mathcal{I})$. The procedure has only one sampling and one synthesis stage. The sampling stage (lines 1) computes a set of samples for the smart set of inputs \mathcal{I} by querying the oracle ϕ . The synthesis stage (lines 2 – 4) checks the formula $\text{SYN}_{T,S}(\vec{c})$ for satisfiability. If the formula is unsatisfiable, then the procedure returns “Insufficient Template”; otherwise, the satisfying assignment \vec{c} defines

$\text{SmartVal}(\phi, T, \mathcal{I})$

1. $S := \text{Sample}(\mathcal{I}, \phi)$
2. $\vec{\alpha} := \text{SAT}(\text{SYN}_{T,S}(\vec{c}))$ // returns \perp if UNSAT
3. **if** $(\vec{\alpha} = \perp)$ **return** “Insufficient Template”
4. **return** $T(\vec{\alpha})$

Figure 3. Procedure SmartVal

the concrete function $T(\vec{\alpha})$, which is then returned. We prove the following.

THEOREM 4. *Given a template T , an I/O oracle ϕ and a smart set \mathcal{I} of inputs for T and ϕ , if T is sufficient for abstracting ϕ , then the procedure $\text{SmartVal}(\phi, T, \mathcal{I})$ returns a function C semantically equivalent to ϕ .*

Like procedure DInputVal , SmartVal is guaranteed to return a function semantically equivalent to the oracle ϕ only if the template T is sufficient for abstracting ϕ ; if this assumption is wrong, it may either detect that the template is not sufficient, or return a wrong function. As will be shown in Section 6, SmartVal can be much faster than DInputVal . But it also requires a set of smart inputs.

We now discuss several approaches to compute smart or universally smart inputs. Trivially, the set of all possible inputs is universally smart for any template, but we want smart input sets to be as small as possible so that the synthesis step is fast.

Brute-force approach. Using Definition 3 for $\text{USmart}_T(\mathcal{I})$, we can compute a set \mathcal{I} by checking the satisfiability of the formula

$$\exists \mathcal{I} : \forall \vec{c}, \vec{d}, \vec{i} : \left(\bigwedge_{\vec{j} \in \mathcal{I}} T(\vec{c}, \vec{j}) = T(\vec{d}, \vec{j}) \right) \Rightarrow T(\vec{c}, \vec{i}) = T(\vec{d}, \vec{i})$$

If such a set \mathcal{I} exists, then it is universally smart for T , by definition. Therefore, a straightforward procedure for synthesizing universally smart inputs is to check the satisfiability of the above formula for any set \mathcal{I} of size 1, then 2, then 3, and so on. This approach can return a universally smart input set of minimum cardinality. Its drawback is that checking satisfiability of a $\exists \forall$ formula can be expensive.

Greedy approach. Here is a straightforward greedy procedure for constructing a universally smart set of inputs:

1. $\mathcal{I} := \emptyset$
2. If $\text{USmart}_T(\mathcal{I})$ holds then return \mathcal{I}
3. Else there exist coefficients \vec{c}, \vec{d} and an input \vec{i} such that $(\bigwedge_{\vec{j} \in \mathcal{I}} T(\vec{c}, \vec{j}) = T(\vec{d}, \vec{j})) \wedge T(\vec{c}, \vec{i}) \neq T(\vec{d}, \vec{i})$
Add \vec{i} to the set \mathcal{I} and go to step (2)

Unlike the brute-force approach, this approach may not return a universally smart input set of minimum cardinality. But checking the validity of $\text{USmart}_T(\mathcal{I})$ can be cheaper because it avoids the existential quantifier on \mathcal{I} .

Manual approach. Universally smart input sets can be inferred from the structure of a template. Therefore, they can also be defined manually while designing the template. A manually defined set \mathcal{I} can be verified for universal smartness using the predicate $\text{USmart}_T(\mathcal{I})$ and an SMT solver. If the verification fails, then the set can be used as the initial set for the greedy approach described above, which would then iteratively enlarge it and eventually return a universally smart input set.

Note that universally smart input sets need to be constructed only once for each template, and can then be re-used for all the instruction instances (oracles) covered by each template. Synthesis with smart sampling is thus especially attractive when a template

is repeatedly used for many instructions instances, as is the case in our context.

5. Synthesis Templates for x86

We discuss in this section how to partially automate the construction of a symbolic execution engine for the x86 processor instruction set using the synthesis techniques described in the previous sections. Specifically, we present 6 synthesis templates that together abstract the semantics of over 500 x86 instruction instances. These templates are expressed using bit-vector constraints for the conciseness and precision requirements discussed in Section 2.

The x86 processor has a complex instruction set architecture (CISC) with 8, 16 and 32 bit instructions. The instructions can be divided into three broad groups: ALU instructions, floating-point instructions and SIMD instructions. In this work, we only focus on ALU instructions since modern SMT solvers supporting the theory of bit-vectors provide the required building-blocks for expressing their semantics (concisely and precisely). Each x86 ALU instruction takes from 0 to 3 inputs and has 0 to 2 outputs, all being stored in either registers or memory locations. The size of the register and memory locations determines the size of the individual inputs and outputs. Typically, most instructions are “overloaded” and can be executed with 8, 16 or 32 bit inputs. Moreover, the execution of each instruction can also set or reset special boolean flags, called EFLAGS. In this work, we consider the 5 most commonly used flags: the carry flag CF, the overflow flag OF, the zero flag ZF, the sign flag SF, and the parity flag PF.

As explained in Section 2, when defining instruction instances, we ignore where the inputs and outputs are stored, and only consider their sizes. For any instruction instance, all its inputs are of size either 8, 16 or 32 bits. The output of each instruction instance is either one of the *main outputs*, whose size is either 8, 16 or 32 bits, or one of the *flag outputs*, which are all 1-bit in size. As an example, the instruction SHL corresponds to $3 \times 6 = 18$ instruction instances: for each size of 8, 16 and 32 bits, there are 6 instances, 1 for the main output and 5 for each of the flag outputs.

In order to define a few tractable synthesis templates that are abstract enough to cover the semantics of many x86 ALU instructions, we first consulted the Intel x86 instruction set reference manual. Based on a preliminary study, we partitioned the ALU instructions into 3 groups, based on similarities in their execution semantics.

1. *Bit-wise group* (BW) contains instructions that perform bit-wise operations, such as AND, OR, and XOR.
2. *Arithmetic group* (ARI) includes instructions that perform arithmetic operations, such as ADD, SUB, and MUL.
3. *Bit-Shift group* (BS) contains instructions that perform shift, rotate and bit-flip operations, such as SHL, ROL, and BTS.

For each instruction group, we define two templates: one for the main output instruction instances, called the *main template*, and the other for the flag output instruction instances, called the *flag template*. Since, for a particular instruction, there are different instruction instances for the different sizes of inputs and outputs, we define synthesis templates that are parametric on the input and output size. From the Intel x86 specification, we learn that the flag outputs of an instruction often depend on the main output. For example, the zero flag is often set when the main output is zero. For this reason, we define each flag template as an extension of the main template for the corresponding instruction group: the flag template for an instruction is defined using a symbolic instruction encoding (a circuit) C_{main} for the main output of the same instruction, and this circuit C_{main} is synthesized first.

We now present the 6 templates. Throughout the description, we use i_1, i_2, i_3 to denote inputs, and o_{main}, o_{flag} to denote the main

and flag outputs, respectively. To simplify the presentation, each template T is specified as a relation over the coefficients, inputs and output. In each case, the output is a (deterministic) function of the other parameters.

5.1 Templates for BW instruction instances

We now describe the main and flag templates for instruction instances in the BW group, which is the simplest of all 3 groups. The main and flag outputs depend only on two inputs i_1, i_2 , which are both of the same size s . The main output o_{main} is also always of size s .

Main template. For all BW instructions, we “guess” that the i th bit of the main output is the result of a certain bit-wise operation performed on the i th bits of the two inputs. Since there are at most 16 different bit-wise operations, i.e., 16 possible functions from 2-bit inputs to 1-bit output, the search space is small. For a given size s , each of the 16 bit-wise operations can be expressed as functions in the theory of bit-vectors, which we denote by $BW_0[s], \dots, BW_{15}[s]$. Thus, we design the main template such that its concretizations are exactly the set $\{BW_0[s], \dots, BW_{15}[s]\}$ of functions. The template formula $T-BW^{main}(c, i_1, i_2, o)$ is formally defined as

$$\bigvee_{0 \leq \alpha \leq 15} (c = \alpha \wedge o_{main} = BW_\alpha[s](i_1, i_2))$$

It has one coefficient c which ranges over $\{0, \dots, 15\}$. When $c = \alpha$, the template behaves as the concrete function $BW_\alpha[s]$. The coefficient c can thus be viewed as a “non-deterministic” choice; once the value of c is fixed, the nondeterminism disappears.

Note that we first “guessed” the above template from a superficial (i.e., non detailed) reading of the Intel x86 spec. Later, the application of the synthesis algorithms discussed in the previous sections, including their sampling stages, confirmed in an automated way (see Sections 6 and 7) that the above template was indeed sufficient to abstract BW instructions.

Flag template. We now define the flag template using the circuit C_{main} previously synthesized for the main output. From the specification manual, we learn that the flag output depends on the truth value of certain predicates over the inputs i_1, i_2 and the main output $o_{main} = C_{main}(i_1, i_2)$. We call each such predicate a *factor*. An example of a factor is $msb(i_1) = 0$ which denotes that the most significant bit of the first input i_1 is 0.

Given all the factors $\vec{F} := F_1, \dots, F_n$, the flag circuit is essentially some function from the truth values of some of the factors to the set $\{0, 1\}$. Thus, we define the flag template such that its concretizations are all the possible functions from the truth values of the factors to the set $\{0, 1\}$. A general definition of such a template for the flag output o_{flag} and coefficients $\vec{c} := c_0, \dots, c_{N-1}$, where $N = 2^n$, is given by the formula $ENUM(\vec{c}, \vec{F}, o_{flag})$, defined as

$$\begin{aligned} & (\neg F_1 \wedge \dots \wedge \neg F_n \wedge o_{flag} = c_0) \\ \vee & (\neg F_1 \wedge \dots \wedge \neg F_{n-1} \wedge F_n \wedge o_{flag} = c_1) \\ \vee & (\neg F_1 \wedge \dots \wedge F_{n-1} \wedge \neg F_n \wedge o_{flag} = c_2) \\ \vee & \dots \\ \vee & (F_1 \wedge \dots \wedge F_n \wedge o_{flag} = c_{N-1}) \end{aligned}$$

We now define the factors $\vec{F}_{BW} := F_1, \dots, F_3$ used in defining the flag template $T-BW^{flag}$ for BW instructions.

$$\begin{aligned} F_1 & := msb(C_{main}(i_1, i_2)) = 1 \\ F_2 & := C_{main}(i_1, i_2) = 0 \\ F_3 & := parity(C_{main}(i_1, i_2)) = 1 \end{aligned}$$

Here msb and $parity$ are the *most-significant-bit* and *parity* operators provided by the theory of bit-vectors. The template formula $T-BW^{flag}(\vec{c}, i_1, i_2)$ is formally defined as $ENUM(\vec{c}, \vec{F}_{BW}, o_{flag})$ and makes use of $2^3 = 8$ coefficients ranging over $\{0, 1\}$.

5.2 Templates for ARI instruction instances

We now describe the main and flag templates for instruction instances in the set ARI. The main and flag outputs for these instructions only depend on the first two inputs i_1, i_2 , which could be of different sizes, denoted by s_1, s_2 respectively. We use s_o for the size of the main output. The flag outputs of ARI instructions not only depend on the main output but also on an internally computed overflow output, which gets discarded at the end of the computation. For instance, the carry flag after executing an ADD instruction is set whenever the overflow output is strictly greater than zero. We use o_{of} to denote the overflow output, whose size is also s_o .

In order to define the flag template, we design our main template such that for each concrete value of the coefficients, we synthesize two functions C_{main} and C_{of} for the main and overflow outputs respectively. This is done by first extending the inputs to size $s_{max} := 2max(s_1, s_2, s_o)$ and then applying them to some arithmetic operation (depending on the coefficients), as discussed below. This generates an output of size s_{max} , whose bits 0 to $s_o - 1$ are considered as the main output while bits s_o to $2s_o - 1$ define the overflow output.

Main template. All ARI instructions perform standard arithmetic operations: addition, subtraction, multiplication and division. However, they differ in whether the inputs are considered signed or unsigned. We design the main template $T-ARI^{main}$ such that its concretizations cover all these possibilities.

The template has 5 coefficients denoted by $\vec{c} := c_0, \dots, c_4$. Coefficients c_0 and c_1 range over $\{1, 2, 3\}$ and determine whether the inputs i_1, i_2 respectively must be sign-extended (case 1), zero-extended (case 2) or replaced with a constant (case 3). The constants used for the third case are the values of the coefficients c_2 and c_3 , which both range over $\{0, \dots, 2^{s_{max}} - 1\}$. The extended value of inputs i_1, i_2 are given by the expressions i_1^{ext}, i_2^{ext} defined as

$$\begin{aligned} i_1^{ext} & := ITE(c_0 = 1, zExt(i_1, s_{max}), \\ & \quad ITE(c_0 = 2, sExt(i_1, s_{max}), c_2)) \\ i_2^{ext} & := ITE(c_1 = 1, zExt(i_2, s_{max}), \\ & \quad ITE(c_1 = 2, sExt(i_2, s_{max}), c_3)) \end{aligned}$$

Here ITE, zExt and sExt are, respectively, the *if-then-else*, *zero-extend* and *sign-extend* operators provided by the theory of bit-vectors.

Coefficient c_4 ranges over $\{1, \dots, 7\}$ and determines the arithmetic operation that must be applied to the two inputs. The operations provided by the theory of bit-vectors are: addition $bvadd$ (case 1), subtraction $bvsub$ (case 2), multiplication $bvmul$ (case 3), unsigned division $bvudiv$ (case 4), unsigned remainder $bvurem$ (case 5), signed division $bvsdiv$ (case 6), and signed remainder $bvsrem$ (case 7). For convenience, we write ARI_1, \dots, ARI_7 to refer to these operations. If $c_4 = k$, then operation ARI_k is applied to the extended inputs, and bits 0 to $s_o - 1$ of the output are considered as the main output of the instance. Thus, in summary, the template $T-ARI^{main}(\vec{c}, i_1, i_2, o_{main})$ is defined as

$$\bigvee_{1 \leq \alpha \leq 7} c_4 = \alpha \wedge o_{main} = ARI_\alpha(i_1^{ext}, i_2^{ext})[0, s_o - 1]$$

Once the coefficients \vec{c} for the main template have been synthesized, we define the function for the overflow output as

$$C_{of}(i_1, i_2) := ARI_\alpha(i_1^{ext}, i_2^{ext})[s_o, 2s_o - 1]$$

Here α is the value of coefficient c_4 in \vec{c} , and i_1^{ext}, i_2^{ext} are obtained by instantiating the concrete values of the coefficients c_0, \dots, c_3 fixed in \vec{c} .

Flag template. As mentioned earlier, the flag template is built upon the circuits C_{main} and C_{of} for the main and overflow output respectively. Like $T-BW^{flag}$, the flag outputs for ARI instructions

also depend on the truth value of certain factors defined over the inputs i_1, i_2 , the main output $o_{main} = C_{main}(i_1, i_2)$ and overflow output $o_{of} = C_{of}(i_1, i_2)$. We therefore make use of the construction $\text{ENUM}(\vec{c}, \vec{F}, o_{flag})$ defined for flag output of BW instructions. The factors $\vec{F}_{\text{ARI}} := F_1, \dots, F_7$ used in defining the flag template $T\text{-ARI}^{flag}$ for ARI instructions are defined as follows:

$$\begin{aligned} F_1 &:= \text{msb}(i_1) = 1 \\ F_2 &:= \text{msb}(i_2) = 1 \\ F_3 &:= \text{msb}(C_{main}(i_1, i_2)) = 1 \\ F_4 &:= \text{parity}(C_{main}(i_1, i_2)) = 1 \\ F_5 &:= C_{main}(i_1, i_2) = 0 \\ F_6 &:= C_{of}(i_1, i_2) = 0 \\ F_7 &:= C_{of}(i_1, i_2) = 2^{s_o} - 1 \end{aligned}$$

The template $T\text{-ARI}^{flag}(\vec{c}, i_1, i_2)$ is formally defined as $\text{ENUM}(\vec{c}, \vec{F}_{\text{ARI}}, o_{flag})$ and makes use of $2^7 = 128$ coefficients ranging over $\{0, 1\}$.

5.3 Templates for BS instruction instances

We now describe the main and flag templates for BS instructions. The main and flag outputs for these instructions depend on all three inputs i_1, i_2, i_3 . While describing BS instructions we will call the inputs i_1, i_2 as the *shift inputs* and i_3 as the *count input*. The size of the shift inputs and the main outputs is the same and is denoted by s . The size of the count input is always 8 bits.

Main template. From the specification manual, we learn that the execution of all BS instruction instances share the following common properties. First, the count input is always used after bit-masking to the lower 5 bits (equivalent to a modulo(%) 32 operation). Second, for a fixed value of the count input, each bit of the main output is either fixed to 0 or 1, or is a specific bit of one of the two shift inputs. We use these two properties to design the main template $T\text{-BS}^{main}$. The main idea is to case split on the count input i_3 bit-masked to the lower 5 bits and then, for each value $\alpha \in \{0, \dots, 31\}$, use coefficients $\vec{c}_\alpha := c_{\alpha,0}, \dots, c_{\alpha,s-1}$, each ranging over $\{0, \dots, 2s+1\}$, to determine the mapping between each of the s bits of the output and the bits of the shift inputs. For a fixed value α of the (bit-masked) count input, the mapping is given by the relation $P_\alpha(\vec{c}_\alpha, i_1, i_2, o_m)$ defined below

$$\bigvee_{0 \leq \beta \leq s-1} \left\{ \begin{array}{l} \vee (0 \leq c_{\alpha,\beta} \leq s-1 \wedge o_m[\beta] = i_1[c_{\alpha,\beta}]) \\ \vee (s \leq c_{\alpha,\beta} \leq 2s-1 \wedge o_m[\beta] = i_2[c_{\alpha,\beta} - s]) \\ \vee (c_{\alpha,\beta} = 2s \wedge o_m[\beta] = 0) \\ \vee (c_{\alpha,\beta} = 2s+1 \wedge o_m[\beta] = 1) \end{array} \right\}$$

Template $T\text{-BS}^{main}(\vec{c}, i_1, i_2, i_3, o_m)$ is formally defined as

$$\bigvee_{0 \leq \alpha \leq 31} i_3 \% 32 = \alpha \wedge P_\alpha(\vec{c}_\alpha, i_1, i_2, o_m)$$

It uses $32s$ coefficients, each ranging over $\{0, \dots, 2s+1\}$.

Flag template. We now define the flag template using the circuit C_{main} for the main output. From the specification manual, we learn that all the BS instructions set the flag output in a similar way: for a fixed count input value, the flag output is either a specific bit of one of the shift inputs, or is set based on the *parity* or *zerness* of the main output, or the xor of the most-significant bits of the first shift input and the main output, or is one of the constants 0 or 1. Thus we define the flag template $T\text{-BS}^{flag}$ such that its concretizations cover all the above cases.

For a fixed value α of the (bit-masked) count input, the template uses a coefficient c_α ranging over $\{0, \dots, 2s+4\}$, to define the mapping between the shift inputs i_1, i_2 and the flag output o_{flag} . This mapping is given by the relation $P_\alpha^{flag}(c_\alpha, i_1, i_2, o_{flag})$ defined as

$$0 \leq c_\alpha \leq s-1 \wedge o_{flag} = i_1[c_\alpha]$$

$$\begin{aligned} \vee s \leq c_\alpha \leq 2s-1 \wedge o_{flag} = i_2[c_\alpha] \\ \vee c_\alpha = 2s \wedge (o_{flag} = 1 \Leftrightarrow C_{main}(i_1, i_2) = 0) \\ \vee c_\alpha = 2s+1 \wedge (o_{flag} = 1 \Leftrightarrow \text{parity}(C_{main}(i_1, i_2)) = 1) \\ \vee c_\alpha = 2s+2 \wedge (o_{flag} = 1 \Leftrightarrow (\text{msb}(i_1) \oplus \text{msb}(C_{main}(i_1, i_2)))) \\ \vee c_\alpha = 2s+3 \wedge o_{flag} = 0 \\ \vee c_\alpha = 2s+4 \wedge o_{flag} = 1 \end{aligned}$$

The template formula $T\text{-BS}^{flag}(\vec{c}, i_1, i_2, i_3, o_{flag})$ is formally defined as

$$\bigvee_{0 \leq \alpha \leq 31} i_3 \% 32 = \alpha \wedge P_\alpha^{flag}(c_\alpha, i_1, i_2, o_{flag})$$

Altogether, the template uses 32 coefficients $\vec{c} := c_0, \dots, c_{31}$, ranging over $\{0, \dots, 2s+4\}$.

5.4 Smart Inputs and Summary

We presented 6 templates (3 for main outputs and 3 for flag outputs) which abstract the semantics of a large number of x86 ALU instructions. In this section, we discuss some key properties of these templates. We start by discussing universally smart input sets, followed by a summary of the *search space* (size of the concretization set) and the *circuit size* (size of the circuits generated) for each template.

Smart inputs. We now present universally smart inputs for the main templates $T\text{-BW}^{main}$ and $T\text{-BS}^{main}$, and smart inputs for the main template $T\text{-ARI}^{main}$ and a large subset of ARI instructions. Those input sets were inferred manually from the structure of the corresponding template. As will be discussed in the next two sections, experiments with the procedure `DInputVal` show that this procedure performs reasonably well on all the flag templates, even for instructions with large (32 bits) input sizes, because of the structural simplicity (DNF formulas with few disjuncts) of the flag templates; therefore, we do not discuss smart inputs for those. In contrast, the procedure `DInputVal` has the worst performance on the $T\text{-BS}^{main}$ template, for which the use of smart inputs is much more important. As an illustration, we explain the methodology used for arriving at a (single!) universally smart input for the template $T\text{-BW}^{main}$. The methodology for the other templates is similar.

As discussed in Section 5.1, the concretizations of the template $T\text{-BW}^{main}$ are all the 16 possible bit-wise operations BW_i , which each take two bits as inputs and return one bit as output. How many inputs are needed to uniquely identify any of those 16 BW_i functions? The answer is 4 provided the 4 inputs cover all four possible combinations of 0 and 1, namely is the set $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$. Therefore, if a single pair of (bit-vector) inputs i_1, i_2 , each of size s , for the template $T\text{-BW}^{main}$ covers these 4 boolean combinations, this single input pair (i_1, i_2) is universally smart for the template. A sufficient condition for finding such an input pair is that there exist bit indices k_1, k_2, k_3, k_4 such that the set of bit pairs $\{(i_1[k_1], i_2[k_1]), \dots, (i_1[k_4], i_2[k_4])\}$ is equal to the set $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$. A pair of inputs that satisfies the above condition is $i_1 = 12, i_2 = 10$. Indeed, the bit-vector representations of these decimal numbers are $i_1 = 12 = 0\dots 01100, i_2 = 10 = 0\dots 01010$, for any input size $s > 4$. Clearly the condition above is satisfied by the bit indices $0, \dots, 3$. Thus, the input set $\mathcal{I}_{\text{BW}} := \{(10, 12)\}$ is universally smart for the template $T\text{-BW}^{main}$.

We now give a set of universally smart inputs for the template $T\text{-BS}^{main}$. For brevity, we only give the input set $\mathcal{I}_{\text{BS}_8}$ for the template instantiated with size 8 bits. Recall from Section 5.3 that BS instructions take 3 inputs and therefore the set $\mathcal{I}_{\text{BS}_8}$ is a set of

Template	Search space	Circuit size
$T\text{-}BW^{main}$	16	$O(1)$
$T\text{-}BW^{flag}$	2^8	$O(1)$
$T\text{-}ARI^{main}$	21×2^{2s}	$O(1)$
$T\text{-}ARI^{flag}$	2^{128}	$O(1)$
$T\text{-}BS^{main}$	$(2s + 2)^{32s}$	$O(s)$
$T\text{-}BS^{flag}$	$(2s + 5)^{32}$	$O(1)$

Figure 4. Templates summary

triples (i_1, i_2, i_3) . It is defined as follows:

$$\begin{aligned} & \{(255, 0, \alpha) \mid 0 \leq \alpha \leq 31\} \\ \cup & \{(1, 1, \alpha) \mid 0 \leq \alpha \leq 31\} \\ \cup & \{(170, 170, \alpha) \mid 0 \leq \alpha \leq 31\} \\ \cup & \{(204, 204, \alpha) \mid 0 \leq \alpha \leq 31\} \\ \cup & \{(240, 240, \alpha) \mid 0 \leq \alpha \leq 31\} \end{aligned}$$

The total number of inputs in the set \mathcal{I}_{BS^s} is $32 \times 5 = 160$. In general for size parameter s , we define a universally smart set of inputs of size $32 \times (\log(s) + 2)$.

For the template $T\text{-}ARI^{main}$, we define a set \mathcal{I}_{ARI} of inputs which is smart for a large subset of ARI instructions. The set \mathcal{I}_{ARI} is defined as

$$\{(17, 5), (200, 59), (170, -59)\}$$

This set is not universally smart for the template $T\text{-}ARI^{main}$ because the set was designed for an earlier version of the template which did *not* use the operations `bvsvdiv` and `bvsvrem`, which since then have been added and are currently handled by the values 6 and 7 of coefficient c_4 . The earlier template was extended so that it could cover the main output of the instruction `IDIV` and a few other instructions. For the current template, we verified by running the smart inputs check (Definition 2) that the set \mathcal{I}_{ARI} is still smart for the current template and all ARI instructions except `IDIV`, `DIV`, `CWD`, `CWDE`, `CDQ`. In practice, as validated by our experiments, the set \mathcal{I}_{ARI} is a good initial set for seeding the `DInputVal` procedure for those remaining instructions, and sufficient for our purposes (see Section 7).

We also performed preliminary experiments to automatically generate sets of universally smart inputs. For instance, the set \mathcal{I}_{ARI} can be augmented to a universally smart input set using the “greedy” approach described in Section 4: in the 8-bit case, this results in a (not necessarily minimal) set of 10 universally smart inputs obtained in 8 secs. In contrast, generating universally smart input sets with the “brute-force” approach (see Section 4) is much more computationally expensive. This topic should be investigated further in future work (see also the discussion in Section 8 on related work in machine learning).

Template summary. In Figure 4, we present a summary of the *Search space* and *Circuit size* for all the templates. The values for each of these properties are expressed as functions of the size s .

For the main template $T\text{-}BW^{main}$, the size of the search space is 16, which is the number of possible bit-wise operations, and the circuits generated are always of constant size as they are just one of the bit-wise operations. The flag template $T\text{-}BW^{flag}$ has a search space of size 2^8 , which is the number of functions from 3 bits (from the 3 factors) to 1 bit. The circuits generated are again of constant size, in particular equal to the size of the formula $\text{ENUM}(\vec{c}, \vec{F}_{BW}, o_{flag})$. The main template $T\text{-}ARI^{main}$ has a search space of size 21×2^{2s} , the 2^{2s} factor comes from the third and fourth coefficients which vary over $\{0, \dots, 2^s - 1\}$. The circuits generated are of constant size. The flag template $T\text{-}ARI^{flag}$ has a search space of 2^8 , which is the number of functions from 7

Instr	n_{syn}	S-Iters	Time (ms)
AND8	10	1	26,808
AND8	10^2	1	26,478
AND8	10^3	1	26,692
MUL8	10	1	32,462
MUL8	10^2	1	33,581
MUL8	10^3	1	40,730
SHL8	10	41	1,171,060
SHL8	10^2	18	542,963
SHL8	10^3	1	181,857

Figure 5. Synthesis using Procedure ExhaustVal

bits (from the 7 factors) to 1 bit, and the circuits generated are again of constant size. The main template $T\text{-}BS^{main}$ has a search space of size $(2s + 2)^{32s}$, since it has $32s$ coefficients ranging over $\{0, \dots, 2s + 1\}$. The circuits generated are of size $O(s)$ as each bit of the output is described individually using a separate circuit. The flag template $T\text{-}BS^{flag}$ has a search space of size $(2s + 5)^{32}$, as there are 32 coefficients ranging over $\{0, \dots, 2s + 4\}$, and the circuits generated are of constant size.

6. Experimental Results

We report results of experiments performed with the synthesis algorithms and templates presented in the previous sections. All experiments were performed on a x86 HP xw4400 PC with a 32-bit 2.4GHz Intel Core2 processor, 2Gb of RAM and running Windows Vista. We used the Z3 [3] SMT solver for implementing all the synthesis algorithms.

We present results of detailed experiments for 3 instructions, each covered by a different synthesis template: AND with template $T\text{-}BW^{main}$, MUL with template $T\text{-}ARI^{main}$, and SHL with template $T\text{-}BS^{main}$. For each instruction, we consider their 8-bit, 16-bit and 32-bit versions to measure the impact of I/O sizes. In all the following experiments, the maximum number of failed verification samples (i.e., $|\mathcal{S}_{fail}|$ in Figure 1) is set to 10: when 10 samples have failed to be verified, the verification stage stops and those samples are fed back to the synthesis stage.

Figure 5 presents results obtained using the synthesis algorithm `ExhaustVal` and for various synthesis sample set sizes n_{syn} as defined in Figure 1. The exhaustive verification part of Procedure `ExhaustVal` does not scale to the 16-bit and 32-bit versions of those instructions (which each take two inputs of that size), so no results are presented for those cases. The number **S-Iters** of synthesis iterations is given in the third column. The overall time (in msec) required to synthesize a verified circuit is given in the last column. The best runtime for an instruction is highlighted in boldface.

For MUL8, the best runtime is obtained with $n_{syn} = 10$ as 10 random synthesis samples are sufficient to identify the correct circuit, so more samples are not necessary. For SHL8, starting with 10 or 100 random synthesis samples requires several expensive synthesis iterations, while $n_{syn} = 10^3$ converges faster to the correct circuit. For AND8, all the runtimes are very close, and the differences are insignificant with respect to the overall runtime.

Figure 6 presents results obtained with the Procedure `DInputVal` of Section 3.3, for various numbers of synthesis samples n_{syn} and verification samples n_{ver} . The number of distinguishing-input checks is given under the column **D-Iters**. The best overall time for any given instruction is again highlighted in boldface.

For AND and MUL, a small set of 10 random synthesis samples is sufficient to synthesize the correct circuit in one iteration (**S-Iters**=1) with a single passing distinguishing-input check (**D-**

Instr	n_{syn}	n_{ver}	S-Iters	D-Iters	Time (ms)	Instr	n_{syn}	n_{ver}	S-Iters	D-Iters	Time (ms)
AND8	10	10^2	1	1	48	MUL8	10	10^2	1	1	189
AND8	10	10^3	1	1	414	MUL8	10	10^3	1	1	637
AND8	10	10^4	1	1	4,007	MUL8	10	10^4	1	1	4,996
AND8	10^2	10^2	1	1	68	MUL8	10^2	10^2	1	1	1,781
AND8	10^2	10^3	1	1	429	MUL8	10^2	10^3	1	1	1,686
AND8	10^2	10^4	1	1	4,023	MUL8	10^2	10^4	1	1	6,659
AND8	10^3	10^2	1	1	273	MUL8	10^3	10^2	1	1	8,805
AND8	10^3	10^3	1	1	637	MUL8	10^3	10^3	1	1	13,365
AND8	10^3	10^4	1	1	4,206	MUL8	10^3	10^4	1	1	14,692
AND16	10	10^2	1	1	55	MUL16	10	10^2	1	1	609
AND16	10	10^3	1	1	484	MUL16	10	10^3	1	1	1,070
AND16	10	10^4	1	1	4,791	MUL16	10	10^4	1	1	6,179
AND16	10^2	10^2	1	1	79	MUL16	10^2	10^2	1	1	2,864
AND16	10^2	10^3	1	1	509	MUL16	10^2	10^3	1	1	3,729
AND16	10^2	10^4	1	1	4,813	MUL16	10^2	10^4	1	1	8,363
AND16	10^3	10^2	1	1	338	MUL16	10^3	10^2	-	-	OOM
AND16	10^3	10^3	1	1	760	MUL16	10^3	10^3	-	-	OOM
AND16	10^3	10^4	1	1	5,040	MUL16	10^3	10^4	-	-	OOM
AND32	10	10^2	1	1	71	MUL32	10	10^2	1	1	1,997
AND32	10	10^3	1	1	619	MUL32	10	10^3	1	1	2,437
AND32	10	10^4	1	1	6,497	MUL32	10	10^4	1	1	9,133
AND32	10^2	10^2	1	1	98	MUL32	10^2	10^2	1	1	5,469
AND32	10^2	10^3	1	1	642	MUL32	10^2	10^3	1	1	5,008
AND32	10^2	10^4	1	1	6,124	MUL32	10^2	10^4	1	1	11,587
AND32	10^3	10^2	1	1	411	MUL32	10^3	10^2	-	-	OOM
AND32	10^3	10^3	1	1	963	MUL32	10^3	10^3	-	-	OOM
AND32	10^3	10^4	1	1	6,451	MUL32	10^3	10^4	-	-	OOM
SHL8	10	10^2	14	4	82,184	SHL16	10	10^2	20	3	938,122
SHL8	10	10^3	16	3	62,569	SHL16	10	10^3	22	1	485,531
SHL8	10	10^4	13	1	60,730	SHL16	10	10^4	20	3	1,019,270
SHL8	10^2	10^2	8	2	41,690	SHL16	10^2	10^2	13	3	795,375
SHL8	10^2	10^3	9	1	38,691	SHL16	10^2	10^3	12	2	678,463
SHL8	10^2	10^4	8	1	58,968	SHL16	10^2	10^4	11	1	457,520
SHL8	10^3	10^2	1	1	21,501	SHL16	10^3	10^2	1	1	250,105
SHL8	10^3	10^3	1	1	23,045	SHL16	10^3	10^3	1	1	301,424
SHL8	10^3	10^4	1	1	49,105	SHL16	10^3	10^4	1	1	291,452
SHL8	10^4	10^2	1	1	78,391	SHL16	10^4	10^2	1	1	967,358
SHL8	10^4	10^3	1	1	84,433	SHL16	10^4	10^3	1	1	1,064,090
SHL8	10^4	10^4	1	1	103,264	SHL16	10^4	10^4	1	1	1,046,099
SHL32	10	10^2	31	4	24,168,853	SHL32	10	10^2	31	3	20,107,259
SHL32	10	10^3	31	3	11,754,805	SHL32	10	10^3	31	1	11,754,805
SHL32	10	10^4	31	1	16,877,223	SHL32	10^2	10^2	21	3	16,877,223
SHL32	10^2	10^2	21	3	17,577,444	SHL32	10^2	10^3	22	3	17,577,444
SHL32	10^2	10^3	20	4	21,620,686	SHL32	10^2	10^4	20	4	21,620,686
SHL32	10^3	10^2	1	1	4,382,472	SHL32	10^3	10^2	1	1	4,382,472
SHL32	10^3	10^3	1	1	4,456,942	SHL32	10^3	10^3	1	1	4,456,942
SHL32	10^3	10^4	1	1	4,707,855	SHL32	10^3	10^4	1	1	4,707,855
SHL32	10^4	10^2	-	-	OOM	SHL32	10^4	10^2	-	-	OOM
SHL32	10^4	10^3	-	-	OOM	SHL32	10^4	10^3	-	-	OOM
SHL32	10^4	10^4	-	-	OOM	SHL32	10^4	10^4	-	-	OOM

Figure 6. Synthesis using Procedure DInputVal

Iters=1), and the fastest runtime is achieved with the fewest verification samples ($n_{ver} = 10^2$). Note that for MUL16 and MUL32 with $n_{syn} = 10^3$, the synthesis algorithm runs out of memory (denoted by “OOM”) and is unable to generate a circuit.

For SHL, the best times are achieved with $n_{syn} = 10^3$ and the smallest number of verification samples ($n_{ver} = 100$) we consider. For smaller numbers of synthesis samples, the DInputVal synthesis algorithm requires several synthesis stages and sometimes feedback from several distinguishing-input checks, which are expensive and increase overall runtime. (As an extreme example not shown in the figure, with $n_{syn} = 0$ and $n_{ver} = 100$, the DInputVal algorithm times out after 12h for SHL32.) With a larger set of synthesis samples ($n_{syn} = 10^4$), the DInputVal algorithm takes more time, or runs out of memory in the SHL32 case.

Figure 7 presents in its last column the runtimes obtained with the smart sampling synthesis algorithm SmartVal of Section 4. These results are compared to the best times obtained with the two other algorithms as reported in Figures 5 and 6. The speed-up obtained by an algorithm compared to the one to the immediate left is indicated by the symbol \div . In our experiments, the SmartVal algorithm is between 11 to 68 times faster than the best time obtained with the DInputVal algorithm, which is itself between 9 to 551 times faster than the ExhaustVal algorithm when the latter is applicable (i.e., in the 8-bit case only).

7. Overall Results, Lessons Learned, Limitations

Using the 6 templates presented in Section 4 and their associated smart inputs, we can automatically synthesize bit-vector circuits for 534 x86 instruction instances (8/16/32-bits, outputs, EFLAGS) in less than two hours on the regular machine described in the previous section (2Gb of RAM, 2.4GHz processor). We used the SmartVal algorithm whenever possible, i.e., whenever a set of universally smart inputs is available, and used the DInputVal

Instr	Exhaust	DInput	Smart Sampl
AND8	26,478	48 (\div 551)	3 (\div 16)
AND16	-	55	4 (\div 14)
AND32	-	71	4 (\div 18)
MUL8	32,462	189 (\div 172)	17 (\div 11)
MUL16	-	609	20 (\div 30)
MUL32	-	1,997	29 (\div 68)
SHL8	181,857	21,501 (\div 9)	867 (\div 25)
SHL16	-	250,105	8,064 (\div 31)
SHL32	-	4,382,472	303,970 (\div 14)

Figure 7. Synthesis using Procedure SmartVal: runtime (in msec) and comparison

algorithm otherwise (e.g., for all EFLAG circuits) still seeded with the smart inputs defined for that instruction family. We also used a verification stage with 1000 random verification samples for each circuit.

Instructions covered include SHL, SHR, SAR, SAL, SHLD, SHRD, ROL, ROR, RCL, RCR, BT, BTR, BTS, BSWAP, AND, OR, XOR, TEST, NOT, NEG, XADD, ADD, SUB, INC, DEC, MUL, IMUL, DIV, IDIV, CWD, CWDE, CDQ, CBW, MOVZX, MOVXS, CMPXCHG.

During the course of this work, we discovered several interesting and sometimes surprising details about the semantics of x86 instructions.

The Intel x86 reference manual often defines the semantics of x86 instructions *partially*, leaving some corner cases “undefined”. In contrast, our automatic synthesis approach gives a precise semantics to all x86 instructions on the processor which is sampled, uncovering sometimes seemingly bizarre behaviors. As an example, the Intel specification says that the carry flag CF is undefined after a ROR8 instruction when the count argument modulo 8 is 0; on

an Intel XEON3.7 processor, the CF flag is actually set to 0 when the count argument is 0, and to 1 when the count argument is 16, 24 or 32. As another example, the Intel spec says that the OF flag of an ADD instruction is set “according to the result” (i.e., the output); however, on an Intel XEON3.7 processor, the OF flag is 1 only when the XOR of the most-significant bit of the *two inputs* is the negation of the most-significant bit of the output.

We also discovered cases where observed behaviors contradicts the x86 reference manual (which is unsurprising given the size and complexity of the spec). For instance, we discovered while debugging our template $T-AR^{main}$ that the overflow OF flag should be set to 0 after executing IMUL8 with 65 and 254 as inputs according to the Intel spec, while the OF flag is actually set to 1 after the execution of this instruction with those inputs on an Intel XEON3.7 processor.

Moreover, we discovered that the semantics of instructions varies across Intel processors. For instance, on an Intel XEON3.7 or Core2 or i7 M620 processors and in accordance with the x86 spec, executing instructions ROL, SHL or SHR does not set the overflow OF flag if the count argument is not 1. However, on an Intel i7-2620M processor (HP EliteBook 2760p, 2.7Ghz, 8Gb RAM, 64-bit) processor, the OF flag is set to 1 even for certain cases when the count argument is greater than 1. Our template $T-BS^{flag}$ is actually unable to capture this behavior, which is why we detected these corner cases.

Finally, and unsurprisingly, we also discovered several errors in previous manually-written x86 instruction handlers used in the whitebox fuzzer SAGE [6].

Our current implementation has several limitations. First, instructions like DIV crash (trigger an error) on certain inputs, for instance when the quotient is larger than the output range. Currently, we use manually-written input preconditions to prevent such cases from occurring during synthesis. Such preconditions should be used as “active checkers” [5] during symbolic execution to check whether those error cases can be triggered during program analysis. In the future, those preconditions could be synthesized automatically by generating a special additional ERROR output. Second, instructions like SHL leave the flags ZF, PF and SF “unchanged” when the count operand is 0, therefore those flags should also be considered as inputs in those cases. This is not currently handled by our template $T-BS^{flag}$.

8. Other Related Work

Synthesizing transfer functions for embedded processors. The closest work to our work is [19] which presents a system for automatically synthesizing transfer functions for embedded processor instructions, which can be used for static analysis of embedded object code. This prior work synthesizes transfer functions in a given abstract domain (like intervals or bit-wise domain) and therefore performs a sound over-approximation of the concrete semantics. In contrast, our goal is to automatically synthesize a bit-precise symbolic representation of the concrete semantics. The approach used in [19] involves building a complete truth table by exhaustively sampling the processor, lifting the table to the abstract domain and then encoding it using BDDs. Due to the exhaustive sampling, this approach does not scale beyond 8-bit instructions. Moreover, the BDD encodings are often too large (several Kbs) to satisfy our conciseness requirement, which is imperative in our context to allow for bit-precise symbolic execution of long program execution traces as is needed for whitebox fuzzing [6]. In follow-up work, [18] develops another technique that assumes a structural constraint on the function being synthesized (analogous to template-based synthesis) and scales to larger instructions. However, the synthesized functions are again for certain abstract domains. Another differ-

ence with our work is that [18] generates abstract transfer functions using a simple custom brute-force solver, whereas we encode our templates as logic formulas in the theory of bit-vectors and carry out the search using an SMT solver.

Connection to machine learning. There is a close connection between the notion of universally smart inputs for a template and the notion of “teaching dimension for a concept class” [7]. Informally, the teaching dimension of a concept class (consisting of classifiers) is the minimum number of samples that a teacher must reveal in order to uniquely identify *any* concept in the class. The paper [7] investigates upper and lower bounds on the teaching dimension and its relation to structural properties of a concept class. Function templates can essentially be thought of as concept classes (concepts being the functions represented as relations over inputs and outputs). These results on teaching dimension shed light on the connection between templates and the set of universally smart inputs, and on the complexity of automatically synthesizing the smallest set of universally smart inputs. Another interesting (and related) connection that we plan to explore further in the future is that between the descriptive complexity [11] of a template and the size of the smallest set of universally smart inputs.

Template-based synthesis. In the last few years, there has been a large amount of work on automated synthesis using deductive techniques. A central theme of all these techniques is to express the synthesis problem as a search problem over a restricted space. The restricted space is defined either using a template [24, 25], or using a sketch [21, 22], or using a set of building blocks [9, 13], or using a restricted language [8, 12]. The synthesis techniques used in this paper are inspired from and build upon this prior work. The unique challenges associated with our specific application domain were the lack of an initial specification and the lack of a final verification oracle.

Black-box analysis of processors/assemblers. Another area of recent related work is work on designing and testing CPU emulators [14–16], especially for x86 processors. The goal of this work is to test whether an emulator faithfully mimics all aspects of the processor, including various addressing modes, privilege levels and clock cycles per instruction. [14] uses the architecture specification as the starting point for determining what instruction operand combinations are valid, and then intelligently modifying and testing each combination with various possible “edge case” values. On the other hand, [15] uses the CPU as the oracle to determine what byte sequences represent valid instructions and how instructions are encoded. The valid byte sequences are then run with various memory and register states. In the same spirit, [10] presents a technique for testing and reverse engineering assemblers for a given architecture, by testing them with permutations of assembly code and then decoding the output. The main relation with our work is the idea of analyzing a black-box system by strategically testing its interface and then inferring internal properties of the system from the outputs.

9. Conclusion

We showed that automatic synthesis of precise and concise symbolic representations of individual processor instructions is possible for a complex processor like x86. The main practical advantage of automatic synthesis is the gain in manual labor: instead of defining manually (probably incomplete and incorrect) detailed instruction handlers for 534 x86 ALU instruction instances, synthesis allowed us to define only 6 abstract instruction handlers (templates), from which 534 correct and precise instruction handlers were generated automatically, from input/output examples and expressed concisely as bit-vector constraints.

In this work, we focused on ALU instructions since those are used in virtually all programs. When symbolically executing a program, each instruction is executed symbolically using the corresponding symbolic instruction handler. Sequences of instructions are handled by combining the symbolic encodings (circuits) of individual instructions. Conditional statements (jumps) are handled using the flag circuits. For instance, a `jz <addr>` jump instruction is simply mapped to a constraint $ZF == 1$ where ZF is the zero-flag circuit synthesized for the instruction immediately prior to the jump instruction. The circuit for regular memory operations such as `ld` (load), `mov` (move), etc. is simply the identity function. Handling all those operations (ALU, conditional jumps and regular memory operations) is already useful for (partial) symbolic execution of many applications, like whitebox fuzzing of file and packet parsers [6].

We believe our templates could be extended to cover x86 SIMD instructions, since they are essentially ALU instructions applied to vectors of registers. In contrast, extending our approach to floating-point instructions seems more challenging since most SMT solvers do not currently handle floating-point arithmetic. Fortunately, for the purpose of whitebox fuzzing, precise symbolic execution of floating-point instructions can often be avoided [4].

It would be interesting to design instruction templates for other processors such as x64 or ARM. Our general synthesis-based approach should be applicable to those processors as well, but details of the templates for those could be significantly different.

Acknowledgments

We thank David Molnar for suggesting the problem addressed in this work, Sumit Gulwani for helpful comments on program synthesis, and Ella Bounimova for interesting discussions on x86 semantics. We also thank the anonymous reviewers for their constructive comments to improve the presentation.

References

- [1] D. Brumley, I. Jager, Th. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *CAV'2011*, July 2011.
- [2] A. Chlipala. Modular Development of Certified Program Verifiers with a Proof Assistant. In *ICFP'2006*, September 2006.
- [3] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS'2008*, April 2008.
- [4] P. Godefroid and J. Kinder. Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis. In *ISSTA'2010*, July 2010.
- [5] P. Godefroid, M.Y. Levin, and D. Molnar. Active Property Checking. In *EMSOFT'2008*, October 2008.
- [6] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'2008*, February 2008.
- [7] S. A. Goldman and M. J. Kearns. On the Complexity of Teaching. *Journal of Computer and System Sciences*, 50:303–314, 1992.
- [8] S. Gulwani. Automating String Processing in Spreadsheets using Input-Output Examples. In *POPL'2011*, January 2011.
- [9] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing Geometry Constructions. In *PLDI'2011*, May 2011.
- [10] W. C. Hsieh, D. R. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *USENIX'2001*, June 2001.
- [11] N. Immerman. *Descriptive complexity*. Springer, 1999.
- [12] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A Simple Inductive Synthesis Methodology and its Applications. In *OOPSLA'2010*, October 2010.
- [13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-Guided Component-Based Program Synthesis. In *ICSE'2010*, May 2010.
- [14] W. Ma, A. Forin, and J. Liu. Rapid Prototyping and Compact Testing of CPU Emulators. In *Proceedings of the 21st IEEE International Symposium on Rapid System Prototyping*, June 2010.
- [15] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing CPU Emulators. In *ISSTA'2009*, July 2009.
- [16] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing System Virtual Machines. In *ISSTA'2010*, July 2010.
- [17] D. Molnar, X. C. Li, and D. Wagner. Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs. In *Proc. of the 18th Usenix Security Symposium*, August 2009.
- [18] J. Regehr and U. Duongsaa. Deriving Abstract Transfer Functions for Analyzing Embedded Software. In *LCTES'2006*, 2006.
- [19] J. Regehr and A. Reid. HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. In *ASPLOS'2004*, 2004.
- [20] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, Th. Braibant, M. O. Myreen, and J. Aglave. The Semantics of x86-CC Multiprocessor Machine Code. In *POPL'2009*, January 2009.
- [21] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by Sketching for Bit-Streaming Programs. In *PLDI'2005*, May 2005.
- [22] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial Sketching for Finite Programs. In *ASPLOS'2006*, 2006.
- [23] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Pooankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS'2008*, December 2008.
- [24] A. Taly, S. Gulwani, and A. Tiwari. Synthesizing Switching Logic Using Constraint Solving. In *VMCAI'2009*, January 2009.
- [25] A. Taly and A. Tiwari. Switching Logic Synthesis for Reachability. In *EMSOFT'2010*, October 2010.