Oracle-Guided Component-Based Program Synthesis

Susmit Jha
UC Berkeley
jha@eecs.berkeley.edu

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

Ashish Tiwari
SRI International
tiwari@csl.sri.com

ABSTRACT

We present a novel approach to automatic synthesis of loop-free programs. The approach is based on a combination of oracle-guided learning from examples, and constraint-based synthesis from components using satisfiability modulo theories (SMT) solvers. Our approach is suitable for many applications, including as an aid to program understanding tasks such as deobfuscating malware. We demonstrate the efficiency and effectiveness of our approach by synthesizing bit-manipulating programs and by deobfuscating programs.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence**]: Program Synthesis; K.3.2 [**Learning**]: Concept Learning

Keywords

Program synthesis, Oracle-based learning, SMT, SAT

1. INTRODUCTION

Automatic synthesis of programs has long been one of the holy grails of software engineering. It has found many practical applications: generating optimal code sequences [20, 11], optimizing performance-critical inner loops, generating general-purpose peephole optimizers [2, 3], automating repetitive programming tasks [15], and filling in low-level details after the higher-level intent has been expressed [24]. Two applications of synthesis are of particular interest in this paper. The first is that of automating the discovery of non-intuitive algorithms (e.g., [8]). The second application, as we show in this paper, is *program understanding*, and more specifically, *program deobfuscation*. The need for deobfuscation techniques has arisen in recent years, especially due to an increase in the amount of malicious, and mostly obfuscated, code (malware) [28]. Currently, human experts use decompilers and manually deobfuscate the resulting code (see, e.g., [22]). Clearly, this is a tedious task that could benefit from automated tool support.

A traditional view of program synthesis is that of synthesis from complete specifications. One approach is to give a specification as a formula in a suitable logic [19, 26, 10, 8]. Another is to write the specification as a simpler, but possibly far less efficient program [20, 11, 24]. While these approaches have the advantage of completeness of specification, such specifications are often unavailable, difficult to write, or expensive to check against using automated verification techniques. In this paper, we propose a novel *oracle-guided* approach to program synthesis, where an *I/O oracle* that maps a given program input to the desired output is used as an alternative to having a complete specification. The key idea of our algorithm is to query the I/O oracle on an input that can distinguish between non-equivalent programs that are consistent with the past interaction with the I/O oracle. The process is repeated until a semantically unique program is obtained. Our experimental results show that only few rounds of interaction are needed.

We apply the oracle-guided approach to automated synthesis of *loop-free programs*, those that compute functions of their input and terminate. Such programs arise in a variety of application contexts, such as low-level bit-manipulating code, scientific computing kernels, parts of control software in graphical languages such as LabVIEW, and even applications in high-level scripting languages such as Javascript and Ruby that are formed by chaining multiple high-level operators. A key characteristic of our method is that it is *component-based*, meaning that we synthesize a program by performing a circuit-style, loop-free composition of components drawn from a given component library. We can also address the challenge of identifying whether the given set of components is insufficient to synthesize the desired program. For this purpose, we additionally require making only one query to a more expensive *validation oracle* that checks whether the program is correct or not.

Our synthesis algorithm is based on a novel *constraint-based* approach that reduces the synthesis problem to that of solving two kinds of constraints: the *I/O-behavioral constraint* whose solution yields a candidate program consistent with the interaction with the I/O oracle, and the *distinguishing constraint* whose solution provides the input that distinguishes between non-equivalent candidate programs. These constraints can be solved using off-the-shelf SMT (Satisfiability Modulo Theory) solvers. Traditional synthesis algorithms perform a expensive combinatorial search over the space of all possible programs. In contrast, our technique leaves the inherent exponential nature of the problem to the underlying SMT solver, whose engineering advances over the years allow them to effectively deal with problem instances that arise in practice, which are usually not hard, and hence end up not requiring exponential reasoning.

Contributions and Organization.

- We propose a novel oracle-guided approach to synthesis, where an I/O oracle obviates the need for complete specifications. Our approach has interesting connections to results from computational learning theory (Section 5).
- We present an instantiation of the oracle-guided approach to synthesis of loop-free programs over a given set of components (see problem definition in Section 3). This is enabled by a novel constraint-based technique that involves

ICSE '10, May 2-8 2010, Cape Town, South Africa

an interaction between SMT solvers and the I/O oracle (Section 4). We also present an interesting optimization that leverages biased sampling (Section 4.6).

- We demonstrate the utility of our synthesis technique to discovery of bit-manipulating programs [29], which are often needed for optimizing performance (Section 6.1). These programs are quite unintuitive and can be difficult for even expert programmers to discover. The upcoming 4th volume of the classic series *the art of computer programming* by Knuth has a chapter on bitwise tricks [13].
- We propose a novel application of program synthesis to program understanding. We demonstrate this in the context of malware deobfuscation by deobfuscating examples drawn from and inspired by the Conficker and MyDoom viruses using our synthesis technique (Section 6.2).

2. MOTIVATING EXAMPLES

We present two examples in this section to introduce the synthesis problem and motivate our approach.

2.1 Bit Manipulation

Consider the following programming problem: Given a bit-vector integer x, of finite but arbitrary length, construct a new bit-vector y that corresponds to x with the rightmost string of contiguous 1s turned off, i.e., reset to 0s. Such programming problems often arise while developing low level embedded code, network applications or in other domains where bit-level manipulation is needed.

Let us contemplate writing a formal specification for this problem. The most natural and easiest specification involves the use of alternating quantifiers, where n is the length of x:

$$\begin{aligned}\exists i, j.\ \{\ 0 \le i, j < n \wedge (\forall k.\ j \le k \le i \implies x[k] = 1)\\ \wedge\ (\forall k.\ 0 \le k < j \implies x[k] = 0)\\ \wedge\ (x[i+1] = 0 \vee i = n-1)\\ \wedge\ (\forall k.\ i < k < n \implies x[k] = y[k])\\ \wedge\ (\forall k.\ 0 \le k \le i \implies y[k] = 0)\ \}\end{aligned}$$

The above specification is not easy to write. Moreover, verifying any candidate implementation against the above specification is challenging due to the presence of quantifiers.

Let us consider writing some sample input-output pairs, or examples, for the problem. For any input x, it is easy to provide the corresponding output y. Some example (x, y) pairs are $(0110, 0000)$, $(0101, 0100)$, $(110110, 110000)$.

Finally, let us contemplate writing a program for the above problem. A straightforward, but inefficient, implementation is a loop that iterates through the bits of x and zeroes out the rightmost contiguous string of 1s. Can we synthesize a shorter and more efficient implementation? It is difficult to answer this, but it is easy to speculate that the elementary operations that may be used inside such an efficient implementation will be the standard bit-vector operators: bit-wise logical operations ($|$, $\&$, $\oplus$, $\sim$), and basic arithmetic operations ($+, -, *, /, \%$).

Given a set of possible elementary operations, and an ability to generate outputs for given inputs, our oracle-guided synthesis tool Brahma will synthesize the following nontrivial and tricky procedure for solving the above problem.

```
1 turnOffRightMostOneBitString (x)
2 { t1 = x-1; t2 = (x | t1); t3 = t2+1;
3   t4 = (t3 & x); return t4; }
```

A programmer will require considerable familiarity with bit-level manipulations to come up with such an implementation. Hence, automated synthesis of such difficult-to-write programs is of great practical significance.

```
1  genStringObs(int input)
2  {
3    a1=1, a2=0; b1=1, b2=0; c1=0; c2=0;
4    if (input == 0)
5       { a1 = 0; a2 = 0; b1=0; b2 = 0; }
6    else if (input == 1)
7         { c1=0; c2 = 1; }
8    else if (input == 2)
9         { a1 =1; a2 = 0; c1=1; c2=1; }
10   else if (input == 3)
11        { b1 = 0; b2 = 0; c1=1; c2=1; }
12   else return NULL;
13   c = 2*c1 + c2;
14   if(c == 1) { return rot13("EPCG GB, 7"); }
15   else
16   if (c == 2) {
17      if (input * (input-1) % 2 == 0)
18         return rot13("EPCG GB", 7);
19      else
20        return rot13("RUYB", 4);
21   }
22   else {
23        if (b1 ⊕ b2) return rot13("ZNVY SEBZ",9)
24        else if ((a1 ⊕ a2) = (b1 ⊕ b2))
25             return rot13("RUYB", 4);
26        else return rot13("QNGN", 4);
27   }
28 }
29 rot13(char *buf, int sz)
30 {
31   char *buf1 = malloc((sz+1) * sizeof(char));
32   char a;
33   while (a =~ *buf)
34   {
35     *buf1 = (~a-1/(~((a | 32))/13*2-11)*13);
36     buf++; buf1++;
37   }
38   return buf1;
39 }
```

Figure 1: Obfuscated program inspired by MyDoom

2.2 Deobfuscation

A major challenge of dealing with malware is simply to understand what the malicious code is doing. We introduce here an example inspired by the obfuscations introduced in variants of MyDoom, an e-mail virus affecting Microsoft Windows that became the fastest-spreading virus when it was first released in 2004 [21].

The example involves the construction of the SMTP header for the e-mail sent by the virus. SMTP requires a prescribed sequence of messages of different types, initially starting with a "hello" message, followed by the "from", "reply to", and other similar control fields, followed finally by the data segment of the e-mail. The fragment we have constructed is a program `genStringObs`, given in Figure 1, which constructs a string corresponding to the message type.

We used two types of obfuscations in this example. The first is a control-flow obfuscation drawn from several obfuscations given by Collberg et al. [5]. The second obfuscation is one that is directly used in MyDoom, involving the modification of each alphanumeric character in the message type string by the `rot13` substitution cipher [14]. The reader can appreciate the difficulty of decoding exactly what this program is doing.

Is there a simpler (deobfuscated) program that performs

```
1 genString(int input)
2 { if(input == 0) return "EHLO";
3   else if (input == 1) return "RCPT TO";
4   else if (input == 2) return "MAIL FROM";
5   else if (input == 3) return "DATA";
6   else return NULL;
7 }
```

Figure 2: Deobfuscated version of `genStringObs`

the same function as `genStringObs`? It is difficult to answer this question. However, looking at the obfuscated code in Figure 1, it is easier to guess that a deobfuscated program will probably use the following types of components:

- Conditional if-then-else (ternary) operators,
- Boolean expressions that occur in `genStringObs`, and
- Operators that return strings of size bounded by size of the largest string in `genStringObs`.

Given these components and the obfuscated program `genStringObs`, our oracle-guided synthesis tool BRAHMA automatically computes the (deobfuscated) program given in Figure 2. The reader can appreciate how much easier this program is to understand.

3. PROBLEM DEFINITION

The goal is to synthesize a loop-free program using a given set of base components and using input-ouput examples. We assume the presence of an I/O oracle that can be queried on any input. The I/O oracle, when given an input, returns the output of the desired program (that we wish to synthesize) on that input. We also assume the presence of a validation oracle that validates the correctness of a candidate program. Finally, we assume that we are given a set of (base) components that should be used as building blocks in the synthesized program. Each component is given in the form of its input-output specification, which is written as a logical formula relating the inputs and the outputs of that component. For ease of presentation, we assume that all components have exactly one output. We also assume that all inputs and outputs have the same *type*. These restrictions are easily removed.

Formally, the synthesis problem in our proposed programming methodology requires the following:

- A validation oracle $\mathcal{V}$ that, given any candidate program (constructed from base components), returns a Boolean answer indicating whether the candidate program is the desired one or not. We discuss how a validation oracle can be implemented for the program classes considered in this paper in Sections 6.1 and 6.2.
- An I/O oracle $\mathcal{I}$ that, given any program input, returns the output of the desired program on that input.
- A set of specifications $\{\langle \vec{I}_i, O_i, \phi_i(\vec{I}_i, O_i)\rangle \mid i = 1, \ldots, N\}$, called a library, where $\langle \vec{I}_i, O_i, \phi_i(\vec{I}_i, O_i)\rangle$ is the specification for the base component f_i, which includes
  - a tuple of input variables $\vec{I}_i$ and an output variable O_i
  - an expression $\phi_i(\vec{I}_i, O_i)$ over variables $\vec{I}_i$ and O_i that specifies the input-output relationship of the i-th component.

  The symbols $\vec{I}_i, O_i$ are assumed to be distinct.

The goal of the synthesis problem is to synthesize a program P that can be validated by the validation oracle $\mathcal{V}$, i.e., $\mathcal{V}(P) = \texttt{true}$. Furthermore, program P should be constructed using only the set of base components in the library, i.e., Program P should take $\vec{I}$ as its inputs and use the set $\{O_1, \ldots, O_N\}$ as temporary variables in the following form:

$$\underline{P(\vec{I}):}$$
$$O_{\pi_1} := f_{\pi_1}(\vec{V}_{\pi_1});\ \ \ldots\ ;\ O_{\pi_N} := f_{\pi_N}(\vec{V}_{\pi_N});$$
$$\texttt{return } O_{\pi_N};$$

where

C1. each variable in $\vec{V}_{\pi_i}$ is either an input variable from $\vec{I}$, or a temporary variable O_{π_j} such that $j < i$, and

C2. $\pi_1, \ldots, \pi_N$ is a permutation of $1, \ldots, N$.

Program P above appears to be a straight-line program, but, in fact, it can be more complex because the base components f_i's can be complex. In particular, base components can be "if-then-else" functions, and using these components, Program P can describe arbitrary loop-free programs.

We note that the program P above is using *all* components from the library. We can assume this without any loss of generality. Even when there is a correct Program P using fewer components, that program can always be extended to a program that uses all components by adding dead code. Dead code can be easily statically identified and removed in a post-processing step.

We also note that program P above is using each base component only once. We can assume this without any loss of generality. If there is a Program P using *multiple* copies of the same base component, we assume that the user provides multiple copies explicitly in the library. Such a restriction of using each base component only once is interesting in two regards. First, it can be used to enforce efficient or minimal programs. Second, it prunes the search space of possible programs making the synthesis problem finite and tractable.

Informally, the synthesis problem is to come up with a program – using only the base components in the given library – that is accepted by the validation oracle.

4. ORACLE-BASED SYNTHESIS

In this section, we provide our solution for the program synthesis problem formally described above. Our solution is based on encoding the space of all possible programs by a formula (Section 4.1). Given a set of input-output pairs, we then constrain this formula further so that it encodes only those programs that work correctly on the given input-output pairs (Section 4.2). By solving this constraint, we generate a candidate solution. If the candidate solution is not the desired program, we provide a way to generate a new input-output pair (Section 4.3). The overall procedure that combines these parts to solve the program synthesis problem is presented in Section 4.4. We present enhancements to our basic procedure in Section 4.6.

4.1 Background: Encoding Programs

We present an encoding of the space of *well-formed candidate programs*, that is, of programs P satisfying constraints C1 and C2, as formulas. This encoding is drawn from recent work [8]. Note, however, that the material in subsequent sections depends only on the *existence* of such an encoding. Our proposed approach can be used with alternative encodings as well.

Intuitively, the encoding we use involves viewing the space of candidate programs as all ways of *connecting* components from the library that satisfy syntactic and semantic well-formedness constraints. Each connection is encoded using an integer-valued *location variable*. Put another way, the value of a location variable determines which component goes on which location (line-number), and from which location (line-number or circuit input) it gets its input arguments.

The main property of the encoding that our approach relies upon is distilled into the following theorem. This theorem states the existence of two formulas (encodings): the first formula $\psi_{\texttt{wfp}}$ represents the set of all *syntactically* well-formed programs; whereas the second formula $\phi_{\texttt{func}}$ represents the set of all *semantic* input-output behaviors of a well-formed program.

THEOREM 1. *There exists a set of integer-valued location variables L, a* well-formedness constraint $\psi_{\texttt{wfp}}(L)$ *over L, a mapping* `Lval2Prog`*, and a* functional constraint $\phi_{\texttt{func}}(L, \vec{I}, O)$ *over $L \cup \{\vec{I}, O\}$ such that the following properties hold:*

- `Lval2Prog` *is a bijective mapping from the set of values L that satisfy the constraint $\psi_{\texttt{wfp}}(L)$ to the set of programs that satisfy constraints C1 and C2.*
- *Let L_0 be a satisfying assignment to the formula $\psi_{\texttt{wfp}}$. If α and β are any candidate input and output values, then the formula $\phi_{\texttt{func}}(L_0, \alpha, \beta)$ is true iff the program* `Lval2Prog`(L_0) *returns the value β on the input α.*

The proof of Theorem 1 follows from the results stated in [8].

We now describe the encoding more formally. Let $\mathbf{P}$ and $\mathbf{R}$ denote the union of all formal inputs (parameters) and formal outputs (return variables) of the components respectively, that is,

$$\mathbf{P} := \bigcup_{i=1}^{N} \vec{I}_i \qquad \mathbf{R} := \bigcup_{i=1}^{N} \{O_i\} = \{O_1, \ldots, O_N\}$$

Any straight-line program constructed using N components can be described by a set of *location variables* L

$$L := \{l_x \mid x \in \mathbf{P} \cup \mathbf{R}\}$$

that contains one new variable l_x for each variable x in $\mathbf{P} \cup \mathbf{R}$ with the following interpretation associated with each of these variables.

- If x is the output variable O_i of the component f_i, then l_x is the line number in the program where the component f_i is used.
- If x is the j^{th} input parameter of the component f_i, then l_x is the line number "from where component f_i gets its j^{th} input".

In the above description, line number refers to either a line of the program, or to some input. For uniformity, each input in $\vec{I}$ is assigned a line number from $0, \ldots, |\vec{I}| - 1$ and the program line numbers then take values from $|\vec{I}|, \ldots, |\vec{I}| + N - 1$. Let $M = |\vec{I}| + N$. The variables L take values in the range $0, \ldots, M-1$ and these new line numbers have the following interpretation.

- For $0 \leq j < |\vec{I}|$, line number j is blank; it takes the value of the j^{th} input of the program.
- For $|\vec{I}| \leq j < M$, line number j contains the $(j - |\vec{I}| + 1)$-th assignment statement of the original program P.

The well-formedness constraint $\psi_{\texttt{wfp}}(L)$, defined below, encodes the interpretation of the location variables l_x along with syntactic well-formedness constraints, such as consistency and acyclicity constraints.

$$\psi_{\texttt{wfp}}(L) \stackrel{\text{def}}{=} \bigwedge_{x \in \mathbf{P}} (0 \leq l_x < M) \wedge \bigwedge_{x \in \mathbf{R}} (|\vec{I}| \leq l_x < M) \wedge \psi_{\texttt{cons}}(L) \wedge \psi_{\texttt{acyc}}(L)$$

$$\psi_{\texttt{cons}} \stackrel{\text{def}}{=} \bigwedge_{x,y \in \mathbf{R}, x \neq y} (l_x \neq l_y)$$

$$\psi_{\texttt{acyc}} \stackrel{\text{def}}{=} \bigwedge_{i=1}^{N} \bigwedge_{x \in \vec{I}_i, y \equiv O_i} l_x < l_y$$

The consistency constraint $\psi_{\texttt{cons}}$ encodes that every line in the program should have at most one component, while the acyclicity constraint $\psi_{\texttt{acyc}}$ encodes that every variable should be initialized *before* it is used.

The function `Lval2Prog` returns the program corresponding to a given valuation L as follows: in the i^{th} line of `Lval2Prog`(L), we have the assignment $O_j := f_j(O_{\sigma(1)}, .., O_{\sigma(t)})$ if $l_{O_j} = i$, $l_{I_j^k} = l_{O_{\sigma(k)}}$ for $k = 1, .., t$, where t is the arity of component f_j, and $(I_j^1, .., I_j^t)$ is the tuple of input variables $\vec{I}_j$ of f_j. The well-formedness constraint describes syntactically correct programs, but it does not describe the semantics of these programs.

The functional constraint $\phi_{\texttt{func}}(L, \vec{I}, O)$ is obtained by taking $\psi_{\texttt{wfp}}(L)$ and adding to it constraints capturing the dataflow semantics and semantics of components.

$$\phi_{\texttt{func}}(L, \vec{I}, O) \stackrel{\text{def}}{=} \exists \mathbf{P}, \mathbf{R}\, \psi_{\texttt{wfp}}(L) \wedge \phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) \wedge \psi_{\texttt{conn}}(L, \vec{I}, O, \mathbf{P}, \mathbf{R})$$

$$\phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) \stackrel{\text{def}}{=} (\bigwedge_{i=1}^{N} \phi_i(\vec{I}_i, O_i))$$

$$\psi_{\texttt{conn}}(L, \vec{I}, O, \mathbf{P}, \mathbf{R}) \stackrel{\text{def}}{=} \bigwedge_{x,y \in \mathbf{P} \cup \mathbf{R} \cup \vec{I} \cup \{O\}} (l_x = l_y \Rightarrow x = y)$$

where $\phi_{\texttt{lib}}$ represents the semantics of the base components (that relates the inputs and outputs of each component), and $\psi_{\texttt{conn}}$ represents the dataflow semantics (that matches the inputs and output of the different components and the inputs and output of the overall program with each other, in accordance with values of location variables).

The formula $\phi_{\texttt{func}}(L, \vec{I}, O)$ represents the class of all syntactically well-formed programs P, constructed using only the N base components, that on input $\vec{I}$ return output O. Hence, we can solve the program synthesis problem by finding appropriate values for the L variables. We need to find values for L such that the input-output behavior of the resulting program matches the input-output behavior specified by the I/O oracle.

A key step in our solution of the program synthesis problem is to *synthesize programs that work for finitely many input-output pairs*. We discuss this next.

4.2 I/O-behavioral Constraint

In this section, we show how to generate a constraint whose solution provides a candidate program whose input-output behavior matches a given *finite* set of input-output examples.

Given a set E of input-output examples $\{(\alpha_j, \beta_j)\}_j$, we use the notation $\texttt{Behave}_E$ to denote the following constraint, which we refer to as *I/O-behavioral constraint*.

$$\texttt{Behave}_E(L) \stackrel{\text{def}}{=} \bigwedge_{(\alpha_j, \beta_j) \in E} \phi_{\texttt{func}}(L, \alpha_j, \beta_j)$$

Let L_0 be a set of values such that $\texttt{Behave}_E(L_0)$ is true. It follows from the definition of the I/O-behavioral constraint that the program encoded by L_0 will give output β_j, whenever it is given an input α_j, for all pairs (α_j, β_j) in E. This property of the I/O-behavioral constraint is stated below.

THEOREM 2 (I/O-BEHAVIORAL CONSTRAINT). *For any satisfying solution L_0 to the I/O-behavioral constraint, the input-output behavior of the program* `Lval2Prog`(L_0) *matches all the input-output examples in the set E.*

The proof of the above theorem is immediate from the definition of an I/O-behavioral contraint and Theorem 1.

We next check if the program, which is synthesized by considering finitely many input-output pairs, is the desired program. We want to avoid the use of the validation oracle, since it is expensive. Here we use what is perhaps the central idea of our approach: *generate a "distinguishing" input that differentiates this program from another candidate program.*

4.3 Distinguishing Constraint

In this section, we show how to generate a constraint whose solution provides an input that distinguishes a given candidate program from another non-equivalent candidate program, both of which have a given set of input-output pairs in their respective input-output behavior.

Let E be a set of input-output pairs. Let P be a candidate program, defined by values L, whose input-output behavior matches the set E. Suppose P is not the desired program. Then, there should be some input $\vec{I}$ such that P gives incorrect output on $\vec{I}$. But, how do we find such an $\vec{I}$?

If P is not the desired program, then let us assume that there is a correct program P'. Clearly, for all input-output pairs (α_j, β_j) in E, the program P' should return β_j when it is given input α_j. But since P is not the desired program, whereas P' is the desired program, P and P' should give different outputs on some new input.

We say $\vec{I}$ is a distinguishing input if there is another program P' whose input-output behavior also matches E, but P and P' give different outputs on the input $\vec{I}$. The constraint $\texttt{Distinct}_{E,P}(\vec{I})$, defined below, represents the set of all distinguishing inputs $\vec{I}$ and we refer to it as *distinguishing constraint.*

$$\texttt{Distinct}_{E,L}(\vec{I}) \stackrel{\text{def}}{=} \exists L', O, O' \ \texttt{Behave}_E(L') \wedge \phi_{\texttt{func}}(L, \vec{I}, O) \wedge \phi_{\texttt{func}}(L', \vec{I}, O') \wedge O \neq O'$$

THEOREM 3 (DISTINGUISHING CONSTRAINT). *If α is a satisfying solution to the distinguishing constraint $\texttt{Distinct}_{E,P}(\vec{I})$, then there exists a program P' such that P and P' have different behaviors on input α, but have the same behavior on all the inputs in the set E.*

The proof of Theorem 3 follows from the definition of the distinguishing constraint, Theorem 2 and Theorem 1. We now have all the ingredients for describing our overall procedure for solving the synthesis problem.

4.4 Oracle-Guided Synthesis

In this section, we describe our oracle-guided iterative synthesis procedure. The description uses the I/O-behavioral constraint and the distinguishing constraint described above.

The procedure works by iteratively synthesizing new programs that work correctly on more and more inputs. It starts with a set containing just one arbitrarily chosen input. In each iteration, the procedure synthesizes a program that works correctly on the current finite set of inputs. If such a program is found, then the procedure attempts to find a distinguishing input. If a distinguishing input is found, then it is added into the set of inputs for subsequent iterations. In all other cases, the procedure terminates. It either returns the correct program, or it notes that the components provided are insufficient for synthesizing the correct program.

For solving the I/O-behavioral constraint and the distinguishing constraint, the procedure makes use of a function `T-SAT`. Given a formula $\phi(A)$, the function $\texttt{T-SAT}(\phi(A))$ searches for values for A that will make the formula ϕ true. If successful, then $\texttt{T-SAT}(\phi(A))$ returns one such specific value for A. Otherwise, it returns $\perp$. The function `T-SAT` is implemented as a call to a Satisfiability Modulo Theory (SMT) solver. SMT solvers check for satisfiability of a first-order formula with respect to underlying background theories [4].

```
IterativeSynthesis():
1  // Input:  Set of base components used in
2  // construction of Behave_E and Distinct_{E,L}
3  // Output:  Candidate Program
4  E := {(α0, I(α0))} // Pick any value α0 for I⃗
5  while (1) {
6    L := T-SAT(Behave_E(L));
7    if (L == ⊥) return "Components insufficient";
8    α := T-SAT(Distinct_{E,L}(I⃗));
9    if (α == ⊥) {
10     P := Lval2Prog(L);
11     if (V(P)) return P;
12     else return "Components insufficient"; }
13   E := E ∪ {α, I(α)}; }
```

Figure 3: Oracle-guided Synthesis Procedure

The pseudo-code for the procedure is given in Figure 3. The procedure maintains a set E of input-output examples constructed by querying the I/O oracle $\mathcal{I}$ on a new input at the start of the while loop (Line 4) and in each iteration of the while loop (Line 13). In each iteration of the while loop, the procedure attemps to synthesize a candidate program P (represented by L) that satisfies the set E of input-output examples (Line 6). If it fails, then it returns failure (Line 7). Otherwise, it checks (in Line 9) whether the candidate program P is the semantically unique program that satisfies the given set of input-output examples. A program is semantically unique if any other program that satisfies the given set of input-output examples produces the same output as the program for any other input. If P is the semantically unique program, then the procedure either returns P (Line 11) or failure (Line 12) depending on whether the validation oracle $\mathcal{V}$ validates P or not. If the candidate program is not semantically unique, then an input α is obtained that is added to E to help narrow down the choice of candidate programs (Line 13).

The following theorem states the correctness of Procedure `IterativeSynthesis`. Note that if the inputs $\vec{I}$ take values from a finite domain, then the number of iterations of the loop in the procedure is bounded by the total number of different inputs; and hence, in such cases the procedure is guaranteed to terminate.

THEOREM 4. *If Procedure* `IterativeSynthesis`*, given in Figure 3, returns a program P, then $\mathcal{V}(P)$ is true. If Procedure* `IterativeSynthesis` *returns "Components insufficient", then there does not exist any program P constructed from the set of base-components such that $\mathcal{V}(P)$ is true. Furthermore, Procedure* `IterativeSynthesis` *is guaranteed to terminate when the inputs $\vec{I}$ take values from a finite domain.*

The proof of the correctness theorem follows immediately from the description of the procedure in Figure 3, combined with the properties stated in Theorem 1, Theorem 2 and Theorem 3. We also illustrate in Figure 4 all three cases in which Procedure `IterativeSynthesis`terminates. The first case corresponds to step 7 and the second and third cases correspond to step 11 and step 12 respectively.

4.5 Illustration on Running Example

We illustrate the oracle-guided synthesis approach on the running example presented in Section 2.1. The problem was,

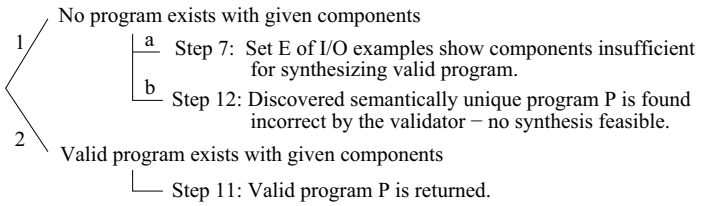

Figure 4: Termination cases of Synthesis Procedure. The validation oracle is needed only to ensure correctness in case 1b.

given a bit-vector integer x, of finite but arbitrary length, to construct a new bit-vector y that corresponds to x with the rightmost string of contiguous 1s turned off.

Our technique starts with a random input 01011 and the I/O oracle $\mathcal{I}$ (the user) is used to obtain the corresponding expected output 01000. This step corresponds to Line 4 of the algorithm presented in Figure 3.

Given the input/output pair $(01011, 01000)$, our technique generates the following candidate program (Line 6): (we give only the expression returned)

$$(x+1)\,\&\,(x-1)$$

Then, it checks whether a semantically different program can be generated in Line 8. In this case, our technique generates the following alternative program and the distinguishing input 00000:

$$(x+1)\,\&\,x$$

The I/O oracle is used to obtain the output 00000 for this input (Line 13). This is added to the set of input/output pairs E. Note that the newly added pair rules out one of the candidate programs, namely, $(x+1)\,\&\,(x-1)$.

In the next iteration, with the updated set E, the technique finds the program

$$-(\neg x)\,\&\,x$$

and the check in Line 8 generates the alternate program

$$(((x\&-x)\,|\,-(x-1))\&x)\oplus x$$

and the input 00101. Hence, we add $(00101, 00100)$ to E. This rules out $(((x\&-x)\,|\,-(x-1))\&x)\oplus x$.

Note that at this stage, the program $(x+1)\&x$ remains a candidate, since it was not ruled out in the earlier iterations. In next four iterations, Brahma generates $(01111, 00000)$, $(00110, 00000)$, $(01100, 00000)$ and $(01010, 01000)$ as input-output examples and adds them to E. The semantically unique program generated from the resulting set E is the desired program:

$$(((x-1)|x)+1)\&x.$$

4.6 Optimization

The basic procedure described above can be improved by using alternate ways to generate the inputs that are used by the procedure for synthesis.

`IterativeSynthesis` uses an SMT solver in two ways:
(a) First, an SMT solver is used to generate a candidate program that works for the current set of inputs.
(b) Second, an SMT solver is used to generate a new distinguishing input on which the currently synthesized program and the desired program potentially differ.

Although SMT solvers are fast and capable of handling very large formulas, using them in every iteration compromises efficiency. It is tempting to speculate that the use of SMT solvers for generating a distinguishing input (case (b) above) can be avoided; for example, by replacing it by a function that finds new inputs by sampling the input space

```
ConstrainedRandomInput:
1  // cnt is a global variable initialized to 0
2  // K is a parameter (#rightmost bits to set)
3  if (cnt < 2^K) {
4     α := sample(Inputs);
5     α :=Set rightmost K bits of α to cnt;
6     cnt := cnt + 1; }
7  else α := T-SAT(Distinct_{E,L}(Ī));
```

Figure 5: Strategy for generating a new input based on sampling from the input space with an application-dependent bias.

in some way. We explore two alternative ways for sampling the input space.

Sampling Uniformly at Random

Let $\mathcal{I}$`nputs` be the set of all possible valuations for the input variables. Let `sample(`$\mathcal{I}$`nputs)` be a function that returns a particular input from the input space $\mathcal{I}$`nputs` by sampling the set $\mathcal{I}$`nputs` uniformly at random. The function `sample(`$\mathcal{I}$`nputs)` can be used to find a new input, in place of the call to the SMT solver, in Line 8 of Procedure `IterativeSynthesis`. We will call this new variant as `Random`.

Sampling With Bias

The second approach we consider is based on biasing the search for inputs towards a certain part of the input space. Not all inputs in the input space are equally important. For example, a program may take an integer input i, but have the same behavior for all $i > 5$ and have interesting behaviors only on values $0 \leq i \leq 5$. For many applications, the user knows a-priori which inputs are more crucial in defining the overall program. The idea behind the sampling with bias strategy is to search for distinguishing inputs by biasing the search to this part of the input space.

In the bitvector benchmarks used in this paper, the input space consists of all (tuples of) bitvectors of a certain bit width. It is well-known that, for a very large class of commonly-used bitvector functions, the rightmost bits influence the output more than the leftmost bits.

Property 1 (See [29], Chapter 2). *A function mapping bitvectors to bitvectors can be implemented with add, subtract, bitwise and, bitwise or, and bitwise not instructions if and only if each bit of the output depends only on bits at and to the right of that bit in each input operand.*

This suggests that we should bias the sampling so that we get more variety on the rightmost bits.

The code `ConstrainedRandomInput` in Figure 5 uses a constrained random strategy for generating a new input. It starts with an input α that is sampled uniformly at random, but then it sets its rightmost K bits to the (rightmost K bits in the) number *cnt*. Since *cnt* is incremented each time, we get a new combination in each time. Specifically, if $K = 2$, then in four calls to the Function `ConstrainedRandomInput`, we will get all four combinations – 00, 01, 10 and 11 – in the rightmost 2 digits of I. The code `ConstrainedRandomInput` finds the first 2^K inputs this way. If more are needed, then it goes back to using the SMT solver. The new variant of `IterativeSynthesis` – obtained by replacing the call to the SMT solver in Line 8 by the code `ConstrainedRandomInput` – will be called `Constrained Random`.

We will compare the performance of `IterativeSynthesis`, `Random`, and `Constrained Random` in Section 6.

5. DISCUSSION

5.1 Choosing Base Components

It is reasonable to ask how base components are chosen in our approach and what happens when the given set of base components is either insufficient or very large.

The choice of base components is made by the user and is guided by the application domain. This allows the user to use his/her knowledge to guide the synthesis and influence success. It is not unreasonable to expect users to provide this information. In several application domains, there is a natural choice for the set of base components. For example, a natural set of base components for synthesizing bitvector algorithms will contain components that perform bitwise and, or, not, xor, negation, increment and decrement operations. In our experiments on synthesizing bitvector programs (Section 6), we started with such a set of base components, referred to as the *standard library*. If the synthesis procedure found that this set of components was insufficient, the standard library was augmented with a set of new components suggested by the user and the synthesis procedure was re-run with this *extended library*.

Nevertheless, choosing a reasonable set of base components is crucial for the feasibility of our synthesis approach. The search space of candidate programs grows exponentially with the number of base components. The strategy of starting with a small set of base components, and then incrementally adding components, can partly avoid the need to deal with very large set of base components. However, it can be successful only if the synthesis engine not only synthesizes correct programs quickly, but also reports infeasibility of the synthesis problem quickly. In our experiments, we show that our technique can detect infeasibility efficiently.

Choosing Base Components for Deobfuscation. When using our program synthesis approach for performing program deobfuscation, the base components are picked from the assignment and conditional statements in the obfuscated code. For example, consider the obfuscated program in Figure 1. Although it is difficult to understand the obfuscated program, it is easier to guess that the set of important base components will include if-then-else components and equality comparators. Similarly, for the other deobfuscation examples (reported in Section 6), the base components used for synthesis contain only operators (such as left-shift and bitwise-xor) that appear explicitly in the obfuscated code (see Figure 7).

5.2 Connections to Learning

Our oracle-guided synthesis framework has close connections to certain fundamental results in computational learning theory. We explore these connections in this section.

Our oracle-based model is similar to the query-based learning model proposed by Angluin [1], but with some important distinctions. In Angluin's model, a learner interacts with an oracle through the use of *membership* and *equivalence* queries in order to learn a *target concept*. In our setting, the *target concept* is the program we seek to synthesize. A membership query is similar to the query we make to an I/O oracle, except that the former returns a binary answer whereas the I/O oracle returns an output value. An equivalence query is similar to a query to the validation oracle, except that, in Angluin's model, if the candidate concept is not equivalent to the target concept, the oracle returns a counterexample as evidence for this non-equivalence. In our context, since the validation oracle is called only at the end, when we are left with a semantically unique program consistent with the set of examples, such a counterexample is not needed. Moreover, Angluin's model treats both kinds of queries as equally expensive. We make a distinction between the cheaper queries to the I/O oracle and the more expensive queries to the validation oracle, which allows us to optimize our implementation. Finally, our algorithm iterates by finding distinguishing inputs, which is not an operation supported by Angluin's model.

Two other results from learning theory also shed light on why our oracle-based approach is effective in practice.

First, note that our focus on loop-free programs that compute functions of finite-precision bit-vector inputs indicates a connection to the work on learning Boolean circuits. In particular, the classic result on learning constant-depth Boolean (AC^0) circuits from a few test inputs [17] provides a partial explanation for the effectiveness of this strategy. The result relies on a theorem stating that AC^0 circuits can be approximated well by low-degree polynomials, which in turn are known to be identifiable by their behavior on few inputs.

The second relevant result relates to the notion of *teaching dimension* introduced by Goldman and Kearns [7]. Informally, the teaching dimension of a concept class is the minimum number of examples a teacher (oracle) must reveal to uniquely identify *any* target concept from that class. As our experiments show, we need very few examples to synthesize our target programs in practice, indicating that these programs form a concept class with a low teaching dimension. Moreover, our algorithm fits closely with a result from the Goldman-Kearns paper [7], showing that the generation of an *optimal teaching sequence* of examples is equivalent to a minimum set cover problem. In the set cover problem for a given target concept, the universe of elements is the set of all incorrect concepts (programs) and each set S_i, corresponding to example x_i, contains concepts that are differentiated from the target concept by this example x_i. We can see that our Procedure `IterativeSynthesis` computes such a distinguishing example in each iteration, and terminates when it has computed a "set cover" that distinguishes the target concept from all other candidate concepts (the "universe"). Given this close connection, it does seem that the classes of functions corresponding to the bit-manipulating and deobfuscation examples we consider have small teaching dimension, and also Procedure `IterativeSynthesis` is effective at generating a sequence of examples close to the optimal teaching sequence.

These connections to learning theory are very intriguing. We leave a formal exploration of these links to future work.

6. EXPERIMENTAL RESULTS

We present experimental evaluation of our technique and compare different approaches namely `IterativeSynthesis`, `Random`, and `Constrained Random` discussed in Section 4.

Setup and Benchmarks.

We have implemented `IterativeSynthesis` in a tool called Brahma. It uses Yices 1.0.21 [25] as the underlying SMT solver. We ran our experiments on 8x Intel(R) Xeon(R) CPU 1.86GHz with 4GB of RAM. Brahma was able to synthesize the desired circuit for each of the benchmark examples. Semi-biased Brahma implements `Constrained Random` with the parameter $K = 2$. Thus, it differs only in first 4 steps from Brahma. As mentioned in Section 4, this is specially targetted towards synthesis of bitvector programs.

We selected a set of 25 benchmark examples to evaluate our technique. 22 benchmarks (P1-P22) are bit-manipulation programs from the book *Hacker's Delight*, commonly referred to as the Bible of bit twiddling hacks [29]. 3 benchmarks were used as examples to illustrate the use of our technique for deobfuscation. These benchmarks reflect obfusca-

P1(x) : Turn-off rightmost 1 bit.

```
1  o1=(x - 1)
2  res=(x & o1)
```

P19(x) : Turn-off the rightmost contiguous string of 1 bits

```
1  o1=(x - 1)
2  o2=(x | o1)
3  o3=(o2 + 1)
4  res=(o3 & x)
```

P21(x) : Next higher unsigned number with same number of 1 bits

```
1  o1=(- x)
2  o2=(x & o1)
3  o3=(x + o2)
4  o4=(x ⊕ o2)
5  o5=(o4 >> 2)
6  o6=(o5 / o2)
7  res=(o6 | o3)
```

P22(x) : Round up to the next highest power of 2

```
1  o1=(x - 1)
2  o2=(o1 >> 1)
3  o3=(o1 | o2)
4  o4=(o3 >> 2)
5  o5=(o3 | o4)
6  o6=(o5 >> 4)
7  o7=(o5 | o6)
8  o8=(o7 >> 8)
9  o9=(o7 | o8)
10 o10=(o9 >> 16)
11 o11=(o9 | o10)
12 res=(o10 + 1)
```

Figure 6: Selected Bit-vector Benchmarks

tion strategies from literature on obfuscation techniques [5] (P23) and Internet worms - Conficker [22] (P24) and MyDoom [21] (P25). Some of these examples are presented in Figure 6 and Figure 7. All the benchmarks used in the experiments are listed in a more detailed version of the paper made available as a technical report [9].

6.1 Bit-Manipulating Programs

Recall from Section 5.1 that the bitvector benchmarks were run using a *standard library* of base components, and if necessary, an *extended library*. In Table 1, we report the runtime when using the standard library (col. 4) and when using the user-augmented extended library (col. 5), in case the standard library was not sufficient. Note that our tool quickly terminates when the given library is insufficient.

For bitvector benchmarks, the user plays the role of the I/O oracle as well as the validation oracle. If the user guarantees that the provided set of base components is sufficient to encode the desired solution, then we do not require the validation oracle. Otherwise, it is theoretically impossible to know whether or not the generated solution is the correct one without a validation oracle. However, in practice, our algorithm detects insufficiency of the base components by discovering inconsistency, and not by a query to the validation oracle. This suggests that in the absence of any validation oracle, we can consider the semantically unique candidate program returned by our algorithm to be the correct program for all practical purposes.

We now compare the three approaches on bit-vector benchmarks using two metrics - the total runtime and the number of iterations. We present the ratio of runtimes of random input generation (col 2 of Table 1) and semi-biased Brahma (col 5 of Table 1) in Figure 8. Semi-biased Brahma is 1.5 times to 12 times faster than random technique. For P22, the random technique did not finish in 1 hour while semi-biased Brahma was able to synthesize it in 186 seconds. Also, the number of iterations required to synthesize a program is also reduced significantly as shown in Table 1. Brahma and semi-biased Brahma is compared in Figure 9. While the number of iterations is more for semi-biased Brahma, it is faster than the Brahma on larger benchmarks. It reduces the runtime for P18 from 140.65 seconds to 25.55 seconds, P21 from 527.91 seconds to 272.28 seconds and P22 from 1108.15 seconds to 187.17 seconds. This validates the optimization proposed in Section 4.

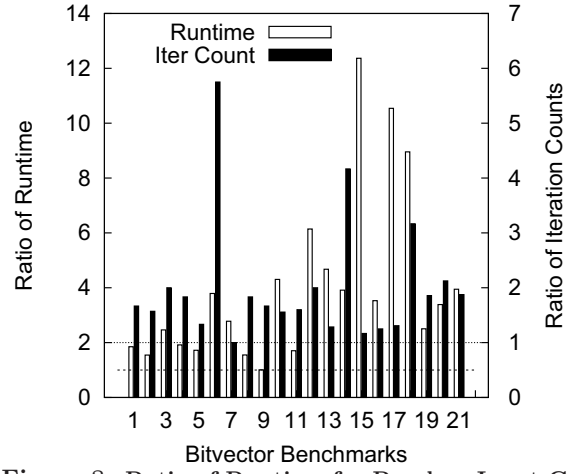

Figure 8: Ratio of Runtime for Random Input Generation to SemiBiased Brahma

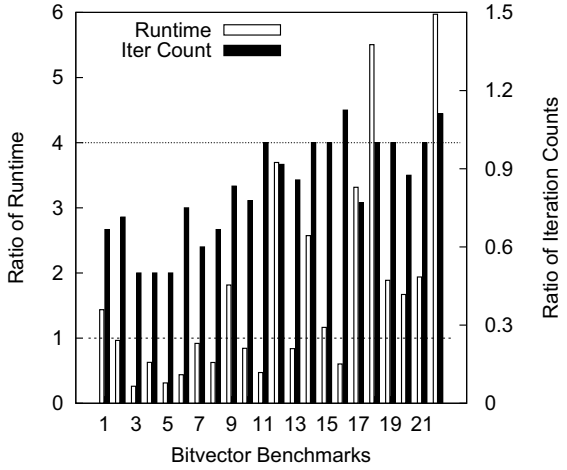

Figure 9: Ration of Runtime for Brahma **to SemiBiased** Brahma

6.2 Deobfuscation

The I/O oracle involves simply evaluating the obfuscated program on the given input. The validation oracle can be a program equivalence checking tool or the user.

An additional challenge that this domain offers is the presence of arbitrary string constants. Our synthesis framework can be easily extended to discovering such constants. For this purpose, we introduce a generic base component f_c that simply outputs some arbitrary constant c. The component f_c takes no input and returns one output O and its functional specification is written as $O = c$. The only change to the framework is that since c is allowed to be arbitrary, we existentially quantify over c in the functional constraint $\phi_{\texttt{func}}$ described in Section 4.1.

For the three examples that we used in experiments, observe that Brahma gives the best performance. The key observation from the experiments is that random input generation does not work well for examples such as P25 where randomly generating integers has a rare chance of 1 in 2^{32} to pick an input which produces any of the first 4 possible outputs. Brahma takes exactly 5 iterations to query the I/O oracle with inputs that generate all the 5 possible outputs.

The experimental results indicate that adding a distinguishing input is better than adding a random input to E because it guarantees that atleast one candidate program is definitely removed from the search space. Thus, it guaran-

P23: Interchange the source and destination addresses.

```
interchangeObs(IPaddress* src, IPadress* dest)
{ *src = *src ⊕ *dest;
  if (*src == *src ⊕ *dest) { *src = *src ⊕ *dest;
     if (*src == *src ⊕ *dest) { *dest = *src ⊕ *dest;
        if (*dest == *src ⊕ *dest) { *src = *dest ⊕ *src;
           return; }
        else { *src = *src ⊕ *dest; *dest = *src ⊕ *dest;
          return;} }
     else *src = *src ⊕ *dest; }
  *dest = *src ⊕ *dest; *src = *src ⊕ *dest; return;}
interchange(IPaddress* src, IPadress* dest)
{ *dest = *src ⊕ *dest; *src = *src ⊕ *dest;
*dest = *src ⊕ *dest;return; }
```

P24: Multiply by 45

```
int multiply45Obs(int y)
{ a=1; b=0; z=1; c=0;
  while(1) { if (a == 0) {
    if (b == 0) { y=z+y; a =¬a;
    b=¬b;c=¬c; if (¬ c) break; }
    else {
    z=z+y; a=¬a; b=¬b; c=¬c;
    if (¬ c) break; } }
    else if(b == 0) {z=y << 2; a=¬a;}
    else { z=y << 3; a=¬a; b=¬b; }}}
return y;}
multiply45(int y)
{ z = y << 2; y = z + y;
  z = y << 3; y = z + y; return y;
}
```

Figure 7: Deobfuscation Benchmarks

Bench	Random Inputs		Semibiased Brahma		
Names	Runtime	Iter	Runtime Standard Lib	Runtime Extended Lib	Iter
1	2	3	4	5	6
P1	1.48	5	0.80*	0.80	3
P2	7.35	11	4.75*	4.75	7
P3	1.60	8	0.65*	0.65	4
P4	1.65	11	0.86*	0.86	6
P5	3.92	8	2.28*	2.28	6
P6	6.22	23	1.64*	1.64	4
P7	1.39	5	0.50*	0.50	5
P8	2.20	11	1.42*	1.42	6
P9	4.95	10	3.85	4.90	6
P10	13.99	14	4.57	3.25	9
P11	24.31	16	2.86	14.27	10
P12	279.49	24	2.64	45.52	12
P13	32.50	9	3.02	6.95	7
P14	14.32	25	3.00	3.66	6
P15	167.84	7	4.50	13.57	6
P16	66.93	10	4.95	18.97	8
P17	217.34	17	5.89	20.62	13
P18	228.78	19	7.98	25.55	6
P19	163.82	13	65.45*	65.45	7
P20	214.14	17	19.30	63.23	8
P21	1074.04	15	13.28	272.28	8
P22	timeout	NA	187.17	185.57	9

Table 1: Random input generation and Semi-biased Brahma on Bitvector Examples. NA denotes not applicable. * denotes that the extended library was same as standard library. Runtimes in sec.

Bench	Brahma		Random		Semibiased Brahma	
Names	Runtime (sec)	Iter	Runtime (sec)	Iter	Runtime (sec)	Iter
P23	1.380	3	24.28	9	12.12	5
P24	5.28	2	11.96	4	2.94	2
P25	0.50	5	timeout	NA	0.86	9

Table 2: Deobfuscation Examples

tees progress. Moreover, it possibly also removes a set of other similar designs from the search space.

7. RELATED WORK

There is a vast body of work on automated program synthesis. In this section, we describe some of the different approaches used for program synthesis.

Component-based Synthesis

Synthesis of straight-line code-fragments constructed from a given set of components has received significant attention.

From Type Signatures: Jungloid mining tool [18] synthesizes code-fragments (over a given set of API methods annotated with their type signatures) given a simple query that describes the desired code using input and output types.

From Functional Specifications: [8] uses SMT solving technology to synthesize a straight-line sequence of instructions from functional description of the desired code sequence. DIPACS [10] uses an AI planner to implement a programmer-defined abstract algorithm using a sequence of library calls. The behavior of the library procedures and the abstract algorithm is specified using high-level *abstractions*, e.g., predicates *sorted* and *permutation*. It uses interaction with the programmer to prune undesirable compositions.

From unoptimal code sequences: Superoptimizers generate an optimal code sequence for a given straight-line sequence of instructions. One approach is to enumerate sequences of increasing length or cost, testing each for equality with the given sequence [20]. Another approach is to constrain the search space to a set of equality-preserving transformations expressed by the system designer [11] and then select the one with the lowest cost. Superoptimization is useful in optimizing performance-critical inner loops. Recent work has used superoptimization [2, 3] to automatically generate general-purpose peephole optimizers by optimizing a small set of instructions in the code.

Program Synthesis

In deductive program synthesis [19, 26], a program is synthesized by constructively proving a theorem which states that forall inputs in a given set, there exists an output, such that a given functional specification predicate holds. Deductive program synthesis assumes that a full functional specification is given. Moreover, it requires advanced deduction technology that is hard to automate.

In inductive program synthesis [27, 12], recursive programs are generated from input-output examples in two steps. In the first step, a set of I/O examples are written as one large conditional expression. In the second step, this initial program is generalized into a recursive program by searching for syntactic regularities in the initial program. In contrast, we do not require a "good" set of I/O examples be given a-priori. Moreover, we do not explicitly generalize – generalization happens implicitly from synthesizing a function using only a given set of components.

Shapiro's Algorithmic Debugging System [23] performs synthesis by repeatedly using the oracle to identify errors in the current (incorrect) program and then *fixing* those errors. We do not fix incorrect programs. We use the incorrect program to identify a distinguishing input and then re-synthesize a new program that works on the new input-output pair as well.

In programming by demonstration [15, 16, 6], the user demonstrates how to perform a task and the system learns an appropriate representation of the task procedure. Unlike our method, these approaches do not make active oracle queries, but rely on the demonstrations the user chooses. Making active queries is important for efficiency and terminating quickly (so that user is not overwhelmed with queries).

In programming by sketching [24], implementations are synthesized from sketches – partially-specified programs with holes. The SKETCH system uses SAT solving within a counterexample-guided loop that constantly interacts with a verifier to check candidate implementations against a complete specification, where the verifier provides counterexamples until a correct solution has been found. In contrast, we do not use counterexamples for synthesis. Further, we require a validation oracle only when the specification can not be realized using the provided components. This verifier is not required to return a counter-example. In practice, we never require a query to the verifier and our technique correctly identifies infeasible scenarios without calling the validation oracle.

8. CONCLUSION

We have presented a novel approach to program synthesis based on oracle-guided learning from examples and SMT solvers. Applications to synthesis of bit-vector programs and deobfuscation have been demonstrated. Experiments indicate that our approach can be efficient and effective for discovering unintuitive code and for program understanding.

Acknowledgments

We are grateful to Rastislav Bodik, George Necula, John Rushby, Natarajan Shankar, Hassen Saidi, Dawn Song, Richard Waldinger and the anonymous reviewers for their insightful comments. The UC Berkeley authors were supported in part by NSF grants CNS-0644436 and CNS-0627734, and by an Alfred P. Sloan Research Fellowship. The fourth author was supported in part by NSF grants CNS-0720721 and CSR-0917398.

9. REFERENCES

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

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

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

[4] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[5] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. Comp. Sci., The Univ. of Auckland, July 1997.

[6] A. Cypher, editor. *Watch what I do: Programming by demonstration*. MIT Press, 1993.

[7] S. A. Goldman and M. J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50:20–31, 1995.

[8] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Component based synthesis applied to bitvector circuits. Technical Report MSR-TR-2010-12, Microsoft Research, Feb 2010.

[9] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. Technical Report UCB/EECS-2010-15, EECS Department, University of California, Berkeley, Feb 2010.

[10] T. A. Johnson and R. Eigenmann. Context-sensitive domain-independent algorithm composition and selection. In *PLDI*, 2006.

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

[12] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Machine Learning Res.*, 7:429–454, 2006.

[13] D. E. Knuth. The art of computer programming. `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`.

[14] J. Kominek. rot13 implementation. `http://www.miranda.org/~jkominek/rot13/`, Accessed Sep. 2009.

[15] T. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[16] H. Lieberman, editor. *Your wish is my command: Giving users the power to instruct their software*. Morgan Kaufmann, 2001.

[17] N. Linial, Y. Mansour, and N. Nisan. Constant depth circuits, fourier transform, and learnability. In *FOCS*, pages 574–579, 1989.

[18] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[19] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.

[20] H. Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, pages 122–126, 1987.

[21] MyDoom Wikipedia Article. `http://en.wikipedia.org/wiki/Mydoom`, URL accessed Sep. 2009.

[22] P. Porras, H. Saidi, and V. Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, March 2009.

[23] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

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

[25] SRI Intl. *Yices: An SMT solver*. `http://yices.csl.sri.com/`.

[26] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *CADE*, 1994.

[27] P. D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1), 1977.

[28] Symantec Corporation. Internet security threat report volume XIV. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`, April 2009.

[29] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman, Boston, MA, USA, 2002.