Control-Flow Deobfuscation Using Trace-Informed Compositional Program Synthesis

BENJAMIN MARIANO* and ZITENG WANG*, University of Texas at Austin, USA
SHANKARA PAILOOR, University of Texas at Austin, USA
CHRISTIAN COLLBERG, University of Arizona, USA
IŞIL DILLIG, University of Texas at Austin, USA

Code deobfuscation, which attempts to simplify code that has been intentionally obfuscated to prevent understanding, is a critical technique for downstream security analysis tasks like malware detection. While there has been significant prior work on code deobfuscation, most techniques either do not handle *control flow obfuscations* that modify control flow or they target specific classes of control flow obfuscations, making them unsuitable for handling new types of obfuscations or combinations of existing ones. In this paper, we study a new deobfuscation technique that is based on program synthesis and that can handle a broad class of control flow obfuscations. Given an obfuscated program P, our approach aims to synthesize a smallest program that is a *control-flow reduction* of P and that is semantically equivalent. Since our method does not assume knowledge about the types of obfuscations that have been applied to the original program, the underlying synthesis problem ends up being very challenging. To address this challenge, we propose a novel *trace-informed compositional synthesis algorithm* that leverages hints present in dynamic traces of the obfuscated program to decompose the synthesis problem into a set of simpler subproblems. In particular, we show how dynamic traces can be useful for inferring a suitable *control-flow skeleton* of the deobfuscated program and performing independent synthesis of each basic block. We have implemented this approach in a tool called Chisel and evaluate it on 546 benchmarks that have been obfuscated using combinations of six different obfuscation techniques. Our evaluation shows that our approach is effective and that it produces code that is almost identical (modulo variable renaming) to the original (non-obfuscated) program in 86% of cases. Our evaluation also shows that Chisel significantly outperforms existing techniques.

CCS Concepts: • **Security and privacy → Malware and its mitigation**; • **Software and its engineering → Search-based software engineering**.

Additional Key Words and Phrases: Program Synthesis, Deobfuscation, Obfuscation

ACM Reference Format:
Benjamin Mariano, Ziteng Wang, Shankara Pailoor, Christian Collberg, and Işil Dillig. 2024. Control-Flow Deobfuscation Using Trace-Informed Compositional Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 349 (October 2024), 31 pages. https://doi.org/10.1145/3689789

*Both authors contributed equally to this research.

Authors' Contact Information: Benjamin Mariano, bmariano@cs.utexas.edu; Ziteng Wang, ziteng@cs.utexas.edu, University of Texas at Austin, USA, Austin; Shankara Pailoor, University of Texas at Austin, USA, Austin, spailoor@cs.utexas.edu; Christian Collberg, University of Arizona, USA, Tucson, collberg@cs.arizona.edu; Işil Dillig, University of Texas at Austin, USA, Austin, isil@cs.utexas.edu.

ACM 2475-1421/2024/10-ART349
https://doi.org/10.1145/3689789

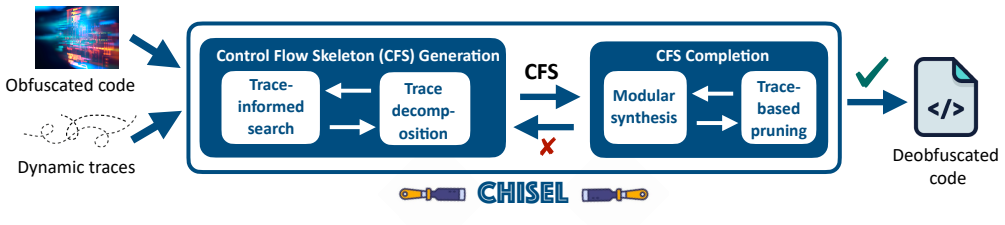


Fig. 1. Illustration of our approach

1 Introduction

Code obfuscation is a frequently-used software protection technique that thwarts reverse engineering by rewriting code into an equivalent but more complex version. Techniques for obfuscating code can take a variety of different forms, including introduction of dead or irrelevant code [16], altering variable and function names [12], unrolling loops [37], replacing program expressions [77], and introducing bogus control flow [13, 19]. While such obfuscation techniques can be used for legitimate reasons like IP protection [8, 23, 48], they are also commonly used by hackers to hide malicious functionality[44, 49, 73] or for avoiding plagiarism detection software [36, 43]. Hence, techniques for *deobfuscating* code play a crucial role for downstream analysis tasks, such as malware identification or software plagiarism detection.

While there has been ample prior work on code deobfuscation techniques for recovering the original code from its obfuscated version[9, 17, 18, 21, 22, 25, 32, 34, 40, 47, 53, 56, 62–65, 68, 70, 71], most techniques in this space are designed to handle *specific* obfuscations. For instance, one technique [63] targets the *opaque predicate* transformation that introduces dead-code guarded by obfuscated conditionals, and another deobfuscation tool [34] targets obfuscations introduced by the OLLVM [33] obfuscator. Such techniques are *point solutions* in that they handle individual obfuscations and can be easily circumvented by combining multiple obfuscation techniques or slightly modifying existing ones.

In this paper, we present a new *general* deobfuscation technique based on program synthesis that can handle a broad class of control-flow obfuscations. Specifically, we consider the class of obfuscations that introduce bogus control flow and refer to this class as *control flow extension (CFE)*. CFE obfuscations cover a wide array of known techniques, including flattening [13], dead code insertion [16], irrelevant code insertion [31], loop unrolling [37], dummy loop insertion [37], block fission [61], and combinations thereof. Our technique does not require knowledge about the specific obfuscations used a priori, can handle programs with multiple obfuscations; it can also adapt to previously unknown obfuscation techniques.

While there have been prior attempts at using program synthesis for code deobfuscation [9, 18, 32, 35, 39, 47, 76], most of these techniques perform deobfuscation at the *expression* level. Expression-level synthesis is useful for handling so-called *dataflow obfuscations* that replace a simple expression like $a + b$ with a much more complex equivalent expression, but such techniques are *not* adequate for obfuscations that modify control flow, as they are much more global in nature. Unfortunately, the global nature of these obfuscations also makes the corresponding synthesis problem more challenging: the code snippets to be reverse engineered are no longer simple expressions but entire code fragments involving conditionals, loops, and sometimes entire functions.

Our proposed method deals with this challenge using a novel *trace-informed compositional synthesis* technique that leverages hints present in dynamic traces of the obfuscated program to dramatically reduce the search space. As illustrated in Figure 1, a key idea underlying our technique

```
1  bool binSearch(int a[], int n, int m) {
2      int l=0; int u = n-1; int mid;
3      while (l <= u) {
4          mid = (l + u) / 2;
5          if (m == a[mid]) { return true; }
6          else if (m < a[mid]) {u = mid - 1;}
7          else { l = mid + 1; }
8      }
9      return false;
10 }
```

Fig. 2. Binary search implementation in C

is to synthesize a so-called *control-flow skeleton (CFS)* of the target program by analyzing dynamic traces. Intuitively, a CFS completely fixes the control flow structure of the program such that the only remaining parts to be synthesized are the contents of the individual basic blocks. Thus, in the first phase (labeled as CFS Generation in Figure 1), our synthesis algorithm obtains the most likely control flow skeleton of the deobfuscated program by performing trace-informed search. In addition to biasing the search towards more likely control flow skeletons, the dynamic traces also facilitate compositional synthesis, as we can infer specifications (in the form of sub-traces) of the unknown parts of the program. Then, in the second CFS completion phase, our algorithm synthesizes each basic block independently in a *modular* fashion and uses the sub-traces associated with that basic block to prune the search space. Overall, dynamic traces play a prominent role in our synthesis technique and help both prune and prioritize the search.

We have evaluated our proposed approach on a benchmark set containing 546 obfuscated programs obtained using 6 control-flow obfuscation techniques from the literature as well as their combinations. We show that Chisel can successfully deobfuscate 86% of these benchmarks within a time limit of 20 minutes. Furthermore, we compare to multiple baselines, including the state-of-the-art control-flow deobfuscator from Yadegari et al. [68], and show that none of them can deobfuscate nearly as many programs as Chisel. Additionally, we run a detailed performance evaluation of Chisel on 100 programs generated by Tigress [14] and find that our technique can scale to quite complex programs, including obfuscated programs with nearly 150,000 lines of code. Finally, we perform a detailed ablation study where we evaluate the importance of each key component of our algorithm and show that all components are critical.

In short, this paper makes the following contributions:

- We propose a new deobfuscation algorithm that targets a large family of obfuscations that we formalize as *control-flow extension*.
- We present a *trace-informed compositional synthesis* algorithm that leverages dynamic program traces to make program synthesis tractable in this context.
- We implement a proof-of-concept source-to-source tool, Chisel, that simplifies obfuscated C programs.
- We present the results of an extensive experimental evaluation consisting of 546 benchmarks obfuscated using a variety of different techniques.

2 Overview

In this section, we give a high-level overview of our deobfuscation method using the program in Figure 2, which gives a standard implementation of binary search in C. Figure 3 shows the resulting program when Tigress [14], a well-known obfuscator, is used to apply two different obfuscations: flattening and dead-code insertion. Flattening [13] replaces standard control flow structures (like

```
1  bool obfFunc(int *var1, int var2, int var3) {
2    int var4; int var5; int var6; ... bool var23;
3    ...
4    var8[11] = &&label11;
5    var9 = 6UL;
6
7    goto *(var8[var9]);
8   label2: var6=(var4+var5)/2; goto label1;
9   label5: return 1;
10  label7: if (var4 <= var5) {goto label2;}
11          else {goto label4;}
12  label1: if (var3==*(var1+var6))
13             {goto label5;}
14          else
15             {goto label10;}
16  label6: var4 = 0; goto label9;
17  label9: var5 = var2-1; goto label7;
18  label10:
19      if ( var21 != var22 ) {
20         if (var3<*(var1+var6))
21             goto label11;
22      } else {
23        while ( (var3<*(var1+var6))<=var5 ) {
24          return 1;
25      }
26    }
27  label0:  var4 = var6 + 1; goto label7;
28  label11: var5 = var6 - 1; goto label7;
29  label4: return 0;
30 }
```

Fig. 3. Binary search obfuscated by Tigress with flattening and dead-code insertion. Note that the presentation of the program has been altered slightly from the output of Tigress for clarity. Opaque predicates introduced by dead-code insertion are highlighted in pink.

`while` loops and `if-then-else` statements) with gotos and labels as can be seen in lines 7-26 of Figure 3. Dead-code insertion introduces one or more sections of code that are never executed and thus have no impact on a program's behavior. Dead-code is often guarded by a so-called *opaque predicate*, which is a complicated expression that always evaluates to a constant value but whose behavior is difficult to determine statically. In this case, there are two opaque predicates: `var21 != var22` always evaluates to `true` and the guard `(var3 < *(var1+var6)) <= var5` always evaluates to `false`.

There are several previously published techniques for returning a flattened function to its original form [21, 34, 40, 64, 65] and for identifying and removing dead code protected by opaque predicates [53, 56, 62, 63]. These techniques, by themselves, do not handle programs such as the one in Figure 3 that compose two or more transformations, particularly given that the choice of order affects the generated code. For example, the program in Figure 3 was produced by first applying Tigress' `Flatten` transformation and then the `AddOpaque` transformation; applying these in the reverse order would have generated different code. This makes rule-based approaches to deobfuscation unappealing, as they are unlikely to be successful for compositions of different obfuscations as well as previously unseen ones.

Traces generated using `a = [1,2,3,4,5], n = 5, m = 4`

	Original		Obfuscated	
Index i	Statement s	Valuation σ	Statement s'	Valuation σ'
1	-	-	goto *(var8[var9])	-
2	l = 0	{l ↦ 0}	var4 = 0	{var4 ↦ 0}
3	-	-	goto label9	-
4	u = n-1	{u ↦ 4}	var5 = var2-1	{var5 ↦ 4}
5	-	-	goto label7	-
6	l <= u	true	var4 <= var5	true
7	-	-	goto label2	-
8	mid = (l+u)/2	{mid ↦ 2}	var6 = (var4+var5)/2	{var6 ↦ 2}
9	-	-	goto label1	-
10	m == a[mid]	false	var3 == *(var1 + var6)	false
11	-	-	goto label10	-
12	-	-	var21 != var22	true
14	m < a[mid]	false	var3 < *(var1+var6)	false
15	-	-	(var3 < *(var1+var6)) <= var5	false
16	l = mid+1	{l ↦ 3}	(var4 = var6+1	{var4 ↦ 3}
17	-	-	goto label7	-
18	l <= u	true	var4 <= var5	true
19	-	-	goto label2	-
20	mid = (l+u)/2	{mid ↦ 3}	var6 = (var4+var5)/2	{var6 ↦ 3}
21	-	-	goto label1	-
22	m == a[mid]	true	var3 == *(var1+var6)	true
23	-	-	goto label5	-
24	return true	-	return 1	-

Fig. 4. Dynamic traces of obfuscated and deobfuscated programs. Loop guards are highlighted in blue.

One solution to this problem is *program synthesis*, which treats the deobfuscation task as a search problem for the smallest program equivalent to the obfuscated code. In fact, synthesis has been highly successful at addressing a similar deobfuscation problem: synthesizing expressions that have been obfuscated using Mixed-Boolean-Arithmetic (MBA) [9, 18]. However, techniques from that domain do not translate directly to control-flow obfuscation, which may introduce a multitude of complex boolean expressions and control-flow operators. To address this shortcoming, we introduce a new synthesis-based algorithm for automatically deobfuscating so-called *control-flow extended* programs. *Control-flow extension (CFE)* describes a family of control-flow obfuscations that preserve the existing control- and data-flow behavior of the program but augment it with additional variables, statements, and control-flow constructs. More formally, a program P' is a *control-flow extension* of P if, for any given trace t' of P' (consisting of a sequence of atomic statements), the corresponding trace t of P is a subsequence of t'. Several obfuscation techniques, including control-flow flattening and dead code insertion shown in Figure 3 as well as their combinations, fall under the umbrella of CFE techniques.

Our algorithm relies on a key observation about control-flow extended programs: *dynamic program traces of the obfuscated program can help intelligently guide search for the deobfuscated program.* As an example, consider the two dynamic program traces of the obfuscated and deobfuscated binary search programs shown in Figure 4. The trace of the original program is shown in the two columns on the left and the obfuscated trace in the two columns on the right (the left-most column gives an index i for each trace element). The traces are generated when we try to find element `m=4` in the array $a = [1, 2, 3, 4, 5]$ (which is of length $n = 5$). It should be noted that the start of both traces have been omitted for simplicity of presentation.

Each trace is a sequence of tuples (s, σ), where s is the statement executed (or the guard of the statement if the statement is an `if-then-else` or `while` loop) and a valuation σ which maps variables to their values. For simplicity, valuations only show updated variable values and a valuation of "-" indicates no state is changed. For guards, instead of listing the valuation, we give the value of the guard when executed (`true` or `false`). Additionally, rows where the statement is marked "-" correspond to no-ops which have been introduced to simplify the presentation.

Our first observation is that the obfuscated trace contains hints as to the control-flow structure of the deobfuscated program. For instance, the length 5 sequence of instructions starting with the guard `var4 <= var5` evaluating to `true` is repeated twice. Additionally, no other guard in this trace evaluates to `true` twice. Thus, if there is a loop in the deobfuscated code, we deduce that `var4 <= var5` is the most likely guard of that loop. Furthermore, if we deduce that `var4 <= var5` is the guard of a loop in the deobfuscated program, it is likely that the statements appearing in between `true` invocations of the guard correspond to traces of the loop body.

Our second observation is that, for each element (s, σ) of the original program trace, there is a "matching" element of the obfuscated trace (s', σ') which is almost identical to the original program (modulo variable renaming). Thus, when deobfuscating control-flow-extended programs, we only need to consider those programs which produce traces that are "reductions" (i.e., subsequences modulo variable renaming) of the obfuscated trace.

We leverage these observations to formulate a two-staged deobfuscation algorithm that works as follows. The first stage utilizes a set of rules based on common trace patterns of standard *unobfuscated programs* to produce the most-likely *control-flow skeletons* (CFS) that fix the control flow structure. Critically, this stage biases the search towards programs that are more likely given the observed traces of the obfuscated program. Once such a control flow skeleton is conjectured, our algorithm also produces a so-called *trace decomposition*, which maps unknowns in the sketch to subtraces of the obfuscated program. This decomposition is critical to the scalability of our approach, as it allows each unknown in the sketch to be synthesized *independently*.

To gain more intuition about these ideas, suppose our algorithm produces the following sketch, where $?^s_i$ indicates an unknown program fragment that could be filled with sequential code:

$$?^s_1 \,;\, \texttt{while(var3 < *(var1 + var6)},\ ?^s_2) \,;\, ?^s_3$$

Our rules would assign low probability to such a partial program, as the guard `var3 < *(var1 + var6)` appears only once in the trace and evaluates to `false`. On the other hand, consider the following alternative CFS:

$$?^s_1 \,;\, \texttt{while(var4 <= var5},\ ?^s_2) \,;\, ?^s_3$$

This sketch would be considered much more promising, as the guard `var4 <= var5` appears multiple times, evaluating to `true` each time. Given this high-probability sketch, we can identify which portions of the trace likely correspond to each hole. In particular, the part of the sketch before the first occurrence of `var4 <= var5` corresponds to $?^s_1$, while each portion of the trace after `var4 <= var5` evaluates to `true` corresponds to $?^s_2$. Thus, a high probability decomposition inferred for this trace is the following [1]:

$$\begin{array}{ll} ?^s_1 \mapsto & \{[(s'_1, \sigma'_1),\ \ldots,\ (s'_5, \sigma'_5)]\} \\ ?_2 \mapsto & \{[(s'_7, \sigma'_7),\ \ldots,\ (s'_{17}, \sigma'_{17})],\ \ [(s'_{19}, \sigma'_{19}),\ \ldots,\ (s'_{24}, \sigma'_{24})]\} \\ ?^s_3 \mapsto & \emptyset \end{array} \tag{1}$$

[1] Note that this is the decomposition inferred for a single trace, and thus some control-paths are not explored (e.g., currently there are no execution paths that exercise $?^s_3$). In practice, we develop a decomposition over multiple traces which explore different paths within the program.

$$
\begin{array}{ll}
\text{Prog } P & \rightarrow S \mid P; S \\
\text{Stmt Groups } S & \rightarrow C \mid B \mid \textit{label} : S \\
\text{Control-flow Stmt } C & \rightarrow \text{ite}(E, P_1, P_2) \mid \text{while}(E, P) \mid \text{break} \mid \text{continue} \mid \textit{goto} \\
\text{Stmt Block } B & \rightarrow E \mid L := E \mid B_1; B_2 \mid \text{return } E \\
\text{Expr } E & \rightarrow \textit{func}(\bar{E}) \mid V \mid \textit{const} \mid E_1[E_2] \mid (*E) \mid \text{alloc}(E) \\
\text{LHS Expr } L & \rightarrow L[E] \mid V \mid (*L) \\
\text{Var } V & \rightarrow v_1 \mid \ldots \mid v_n
\end{array}
$$

Fig. 5. Source language $\mathcal{L}_{obf}$ for obfuscated programs and target language $\mathcal{L}$ for deobfuscated programs.

Given the control flow structure of the program and a trace decomposition mapping each unknown to a set of sub-traces, our algorithm "completes" the sketch by replacing each unknown with straight-line code. Crucially, the trace decomposition allows the algorithm to synthesize each unknown independently: We can use the trace decomposition to figure out the correspondence between each hole in the sketch and its corresponding code fragment in the obfuscated program. This gives us a "specification" for each hole, allowing them to be synthesized independently.

3 Problem Statement

We introduce the control-flow deobfuscation problem in the context of an imperative C-like language $\mathcal{L}$ shown in Figure 5. We describe the syntax of the underlying programming language using a context-free grammar, which includes a set of non-terminals $\mathcal{V}$, terminals Σ, productions R, and a start symbol α. Given a string in $s \in (\Sigma \cup \mathcal{V})^*$, we use $s \Rightarrow s'$ to indicate that s' is obtained from s by replacing some non-terminal $N \in \mathcal{V}$ with a string $w \in (\Sigma \cup \mathcal{V})$ where $N \rightarrow w$ is a production in R. We write $\Rightarrow^*$ to indicate the transitive closure of $\Rightarrow$.

Programs in $\mathcal{L}$ consist of sequences of statements, which include both *control-flow statements* (i.e., if-then-else (ite) and while loops) as well as *non-control-flow statements*, which include expressions and assignments. We differentiate between the language $\mathcal{L}$ used for expressing *deobfuscated* programs and the language $\mathcal{L}_{obf}$ used for expressing *obfuscated* programs: The only difference is that obfuscated programs may contain gotos while deobfuscated programs may not.

3.1 Traces and Program Equivalence

As mentioned in Section 2, control-flow extended programs alter the control-flow behavior of a program while maintaining a relationship between traces of the source and target programs. As is standard, we consider a trace t to be a sequence of tuples of the form (s, σ), where s is an *atomic statement* and σ is a valuation mapping variables to their values (after s is executed). Atomic statements do not have control flow and include (1) assignments and expressions from Figure 5 and (2) boolean guards g that encode control flow predicates. The guards track predicates used in conditionals and loops and whether they evaluate to true or false. For example, consider the program $ite(x < 0, y := 0, y := 1)$ with input x and suppose we execute this program on input $x = 5$. This execution would be encoded using the following trace t:

$$(!(x < 0), [x \mapsto 5]), (y := 1, [x \mapsto 5, y \mapsto 1])$$

The first element in t shows that the predicate of the if statement evaluates to false in this execution, and the second element corresponds to the execution of the false branch.

We use the notation $\sigma \subseteq \sigma'$ to indicate that the valuation σ' is an extension of σ, meaning that (1) σ, σ' agree on the values of all variables in the domain of σ and (2) σ' may contain additional variables not in the domain of σ. Given a program P with input σ, we denote the trace of P on

input σ as $\text{Trace}(P, \sigma)$, and we write $t.\sigma_{in}$ to denote the the input that produced t. We also use the notation $[\![P]\!]_\sigma$ to denote the program output when P is executed on σ. We say two programs $P, \hat{P}$ are *semantically equivalent*, denoted $P \equiv \hat{P}$, if, for all input valuations σ, we have $[\![P]\!]_\sigma = [\![\hat{P}]\!]_\sigma$.

3.2 Control-flow Extensions and Reductions

We formalize the class of obfuscations targeted in this paper as *control-flow extensions*. To define this concept, we first introduce the notions of *guard-free trace* and *trace extension*:

Definition 3.1 (**Guard-free trace**). *Given trace t, $\chi(t)$ yields a trace t' such that (1) t' does not contain any boolean guards, and (2) t' is the longest subsequence of t satisfying the first condition.*

Example 3.1. *Suppose t is the trace $(!(x < 0), [x \mapsto 5]), (y := 1, [x \mapsto 5, y \mapsto 1])$ from above. Then $\chi(t)$ is $(y := 1, [x \mapsto 5, y \mapsto 1])$.*

Control flow obfuscation can modify control flow predicates in subtle ways – for example, the guard of a loop may be negated so that `while(x) {...}` becomes `while(true) { if(!x) break; ...}` or a conditional guard `x && y` may be split into nested guards `x` and `y` separately. To address this, we define *trace extensions* over *guard-free* traces $\chi(t)$, which remove guards from the trace t.

Definition 3.2 (**Trace Extension**). *We say that a trace $\hat{t}$ is an extension of trace t if there exists a subsequence $\hat{t}'$ of $\hat{t}$ such that for every trace element $\chi(t)[i] = (s, \sigma)$, $\chi(\hat{t}')[i] = (s, \hat{\sigma})$ where $\sigma \subseteq \hat{\sigma}$.*

Example 3.2. *Consider the three following traces:*

$$t_1 = [(\texttt{x++}, \{x \mapsto 4\})]$$
$$t_2 = [(\texttt{x--}, \{x \mapsto 2\}), (\texttt{x++}, \{x \mapsto 3\})]$$
$$t_3 = [(\texttt{y--}, \{x \mapsto 3, y \mapsto 1\}), (\texttt{x++}, \{x \mapsto 4, y \mapsto 1\})]$$

Here, t_3 is an extension of t_1 as both have an element `x++` where x is assigned to 4. No other trace is an extension of any other one.

We note that checking that a trace t is an extension of a trace t' can be done efficiently, as one can first project the states of t onto the variables V occurring in t' to derive a trace t_V, and then check that t' is a subtrace of t_V.

Definition 3.3 (**Control-flow Extension**). *Program $\hat{P}$ is a control-flow extension of program P iff P is observationally equivalent to $\hat{P}$ (written $P \equiv \hat{P}$) and, for every input σ, $\text{Trace}(\hat{P}, \sigma)$ is an extension of $\text{Trace}(P, \sigma)$.*

Example 3.3. *Suppose P is* `x := 1; return x;` *Then the following program is a control-flow extension of P:*

```
y := 1; if (y >= 1){x := 1;} return x
```

as any trace of both programs will contain an entry with `x := 1` *and where x maps to* 1.

Definition 3.4 (**Control-flow Reduction**). *We say that a program P is a control-flow reduction of program $\hat{P}$ iff $\hat{P}$ is a control-flow extension of P.*

3.3 Problem Statement

When deobfuscating a control-flow extended program, our goal is to find the "simplest" deobfuscated program. We formalize this notion as a *minimum control-flow reduction*:

Definition 3.5 (**Minimum Control-flow Reduction**). *P is a minimum control-flow reduction of $\hat{P}$ if P is a control-flow reduction of $\hat{P}$ and for any control-flow reduction P' of $\hat{P}$, we have $|P| \leq |P'|$, where $|\cdot|$ returns the AST size of a program.*

We now define the *control-flow deobfuscation problem*:

Definition 3.6 (**Control-flow Deobfuscation Problem**). *Given some (obfuscated) program $\hat{P}$ from $\mathcal{L}_{obf}$, find a minimum control-flow reduction P of $\hat{P}$ in $\mathcal{L}$.*

4 Deobfuscation Algorithm

In this section, we first introduce the concept of a *trace-augmented sketch* (or *t-sketch* for short) which plays a key role in our synthesis algorithm. We then describe the top-level deobfuscation procedure, followed by a discussion of its two core components.

4.1 Trace Augmented Program Sketch

A *trace-augmented sketch*, abbreviated *t*-sketch, consists of two parts: a regular sketch and a trace decomposition Δ. A regular sketch is a program with unknown parts:

Definition 4.1 (**Sketch**). *Let $\mathcal{L} = (\mathcal{V}, \Sigma, R, \alpha)$ be the language from Figure 5. A program sketch $\mathcal{S}$ over $\mathcal{L}$ is a string in $(\Sigma \cup \mathcal{V})^*$ where $\alpha \Rightarrow^* \mathcal{S}$. We refer to any non-terminal $N \in \mathcal{V}$ in $\mathcal{S}$ as a* hole. *We distinguish between two types of holes. A control flow hole is a hole $?^c$ such that $?^c \Rightarrow^* C$, while a statement hole $?^s$ is a hole that only derives sequential code. We also write $?$ to denote a hole of either type. Finally, we say a sketch is* complete *if it has no holes, and it is* control-flow complete *if it has no control-flow holes.*

Example 4.1. *The following is a sketch of a program containing a loop:*

$$?^c_1 \,;\, \texttt{while(x > y, } ?^s\texttt{)} \,;?^c_2$$

Because the sketch contains control-flow holes, it is not *a control-flow complete sketch.*

As discussed earlier, our technique decomposes the original synthesis problem into smaller subproblems by mapping each hole $?$ in the sketch to a set of subtraces. We introduce *trace decomposition* to capture this idea:

Definition 4.2 (**Trace Decomposition**). *Given a sketch $\mathcal{S}$ and set of traces T, a trace decomposition Δ is a mapping from each hole $?_i$ in $\mathcal{S}$ to a set of traces T_i such that each trace $t_i \in T_i$ is a subsequence of some trace in T.*

Example 4.2. *Consider the sketch from Example 4.1 and the following trace:*

1. `y := 10`	$\{x \mapsto 14, y \mapsto 10\}$	5. `y += 2`	$\{x \mapsto 14, y \mapsto 14\}$
2. `x > y`	$\{x \mapsto 14, y \mapsto 10\}$	6. `x > y`	$\{x \mapsto 14, y \mapsto 14\}$
3. `y += 2`	$\{x \mapsto 14, y \mapsto 12\}$	7. `return y`	$\{x \mapsto 14, y \mapsto 14\}$
4. `x > y`	$\{x \mapsto 14, y \mapsto 12\}$		

The following is a decomposition is this trace:

$$\begin{aligned}
?^c_1 &\mapsto \quad \{[\ (\texttt{y := 10}, \{x \mapsto 14, y \mapsto 10\})\ ]\} \\
?^s &\mapsto \quad \{[\ (\texttt{y += 2}, \{x \mapsto 14, y \mapsto 12\})\ ], [\ (\texttt{y += 2}, \{x \mapsto 14, y \mapsto 14\})\ ]\} \\
?^c_2 &\mapsto \quad \{[\ (\texttt{return y}, \{x \mapsto 14, y \mapsto 14\})\ ]\}
\end{aligned}$$

Definition 4.3 (**Trace-augmented sketch**). *Given a program with traces T, a t-sketch Θ is a pair $(\mathcal{S}, \Delta)$ where $\mathcal{S}$ is a sketch and Δ is a trace decomposition of T with respect to $\mathcal{S}$. We write $\Theta.\mathcal{S}$ and $\Theta.\Delta$ to indicate each component of the t-sketch.*

Finally, because we are interested in t-sketches that are control-flow complete, we refer to them as *control flow skeletons:*

```
1:  procedure Deobfuscate(P̂, T)
2:      input: Obfuscated program P̂
3:      input: Trace set T of P̂
4:      output: Deobfuscated program P
5:      while true do
6:          Θ ← GetNextCFS(T)
7:          if Θ = ⊥ then return ⊥
8:          V ← GetInputVars(P̂)
9:          while true do
10:             P ← CompleteCFS(Θ, P̂, V)
11:             if isReduction(P, P̂) then return P
12:             V′ ← GetNextVocabulary(Θ)
13:             if V = V′ then break;
14:             else V ← V′
```

Algorithm 1. Top-level deobfuscation algorithm. We note isReduction returns false when P is $\bot$.

Definition 4.4 (**Control-flow skeleton**). *A* control-flow skeleton (CFS) *is a t-sketch where $\Theta.\mathcal{S}$ is control-flow complete.*

In other words, a CFS *completely* fixes the control flow of the target program.

4.2 Top-Level Algorithm

Algorithm 1 shows our top-level deobfuscation algorithm. As discussed earlier, the key idea is to use dynamic traces of the obfuscated program to make synthesis more tractable. The algorithm consists of two phases that are repeatedly executed until a solution is found:

- **Phase 1:** This phase corresponds to the body of the outer loop and identifies the *most promising* control-flow skeleton by calling GetNextCFS at line 6. As discussed in the next subsection, we use the provided traces to determine how promising a control flow skeleton is.
- **Phase 2:** This phase corresponds to the inner loop of Algorithm 1 and tries to synthesize all basic blocks in a given CFS such that the resulting program P is a control flow reduction of the input program $\hat{P}$. Specifically, given a CFS and a "vocabulary" V, CompleteCFS (invoked at line 10) returns a complete program that only uses variables in V. This vocabulary is initialized to the input variables (line 8) and iteratively updated (via the call to GetNextVocabulary at line 12) until a fixed point is reached. The idea behind the vocabulary is that the obfuscation may introduce many spurious variables, so we wish to find a semantically equivalent program that only uses a minimal set of variables. This is why Algorithm 1 only updates the vocabulary if it fails to find a completion of the CFS with the existing vocabulary.

In the next two subsections, we present the GetNextCFS and CompleteCFS procedures in more detail and provide information about growing the deobfuscation vocabulary in Section 5.

4.3 Synthesizing Control Flow Skeletons

Algorithm 2 shows the GetNextCFS procedure that lazily enumerates control flow skeletons in decreasing order of likelihood with respect to the provided traces T. The core idea is to pattern match a given sketch against the dynamic program traces to determine whether this control flow structure is probable with respect to the observed program executions. For a given sketch $\mathcal{S}$, the algorithm also uses T to deduce a trace decomposition and refines the current sketch into a

1: **procedure** GetNextCFS(T)
2: **input:** Trace set T of $\hat{P}$
3: **output:** Next most likely t-sketch given T
4: $\mathcal{W} \leftarrow$ PriorityQueue($(?^c_P, \{?^c_P \mapsto T\}), 1$)
5: **while** $\neg$Empty($\mathcal{W}$) **do**
6: $(\Theta, w) \leftarrow \mathcal{W}$.dequeue()
7: **if** ControlFlowComplete($\Theta.\mathcal{S}$) **then**
8: **yield** Θ
9: $\mathcal{F} \leftarrow$ Expand(Θ)
10: **for** $\mathcal{E}_i \in \mathcal{F}$ **do**
11: $\Upsilon \leftarrow$ Decompose($\Theta, \mathcal{E}_i$)
12: $C \leftarrow \{(\Theta'_i, w \times w_i) \mid (\Theta_i, w_i) \in \Upsilon\}$
13: $\mathcal{W} \leftarrow \mathcal{W}$.addAll($C$)
14: **return** $\perp$

Algorithm 2. Algorithm for getting next CFS. The Expand procedure replaces one of the control flow holes in the sketch with the right hand side of all possible productions in the grammar.

t-sketch. This process of repeatedly refining t-sketches continues until the algorithm finds one that is control-flow complete.

In more detail, the algorithm maintains a priority queue of t-sketches along with a corresponding score. Intuitively, the higher the score, the more likely we consider this sketch to be with respect to the observed execution traces. The queue is initialized to be a singleton set containing the trivial sketch $?^c_P$ (i.e., a single hole corresponding to the start symbol of the grammar) and the decomposition $\{?^c_P \mapsto T\}$ which just maps that hole to the input trace set T (line 4).

The main loop (lines 5-13) starts by dequeueing the next most likely t-sketch $\Theta = (\mathcal{S}, \Delta)$ from the priority queue. If $\mathcal{S}$ is control-flow complete, then Θ is returned as the next most promising CFS. Otherwise, the algorithm invokes Expand to refine the current t-sketch by choosing one of the control-flow holes in $\mathcal{S}$ and finding possible ways to fill that hole. In particular, Expand returns a set of mappings from holes in $\mathcal{S}$ to a candidate *expansion*. Here, an expansion for hole $?^c$ (associated with non-terminal N) is a string s over the alphabet $(\mathcal{V} \cup \Sigma)$ such that $N \Rightarrow^* s$. Thus, hole $?^c$ can be replaced with s to generate a refinement of $\mathcal{S}$. For example, given the empty program ?, Expand returns a set of possible refinements, such as ite($x < 0, ?, ?$). Note that, when filling a hole using a conditional or loop, the Expand procedure fixes their guards but not their body (as in the previous example). When expanding a hole ?, guards are chosen based on which guards appear in traces associated with ? in the decomposition $\Theta.\Delta$.

Each candidate expansion $\mathcal{E}_i$ produced by Expand will contain holes that are not present in $\mathcal{S}$, meaning that we must infer subtraces for the new holes. Thus, lines 10-13 process each candidate

Table 1. Examples of Trace Matching Rules.

Trace Matching Rules			
Name	"Regex" Structure	"Regex" Parts	Control-flow Structure
basic-loop	(pre) (body)* [!g] (post)	pre: .* body: g[^g]* post: .*	(pre) while(g) do (body) (post)
break-loop	(pre) (body)* [g2] (post)	pre: .* body: g[^g]* post: .*	while(g) do (body)
ite-true	(pre) [g] (body) (post)	pre: [^g]* body: ^s* post: [s] (.)*	(pre) if (g) {(body)} else {*} (post)

proposed by Expand to generate a corresponding new t-sketch. Specifically, the Decompose procedure (discussed next) infers a set of new trace decompositions, along with their corresponding scores, for each $\mathcal{E}_i$. Finally, the resulting t-sketches are added to the priority queue.

4.3.1 Rule-based Decomposition. The basic idea underlying trace decomposition is to pattern match the given trace against regular expressions that encode common program structures. If there is a match, we decompose the trace into subtraces based on which parts of the regex correspond to which subtraces. While our algorithm considers multiple decompositions, only those t-sketches that have a match are assigned a high score, allowing them to be prioritized by the search algorithm. It should be noted that our regex matching is *not* greedy in the sense that it will assign a high score to *all* decompositions matching one of the regular expression rules.

Table 1 presents examples of regexes used for decomposing traces. Naturally, these rules capture common control flow patterns. For example, the first rule describes traces that correspond to basic while loops (without break). According to this rule, a trace t of the obfuscated program is likely to correspond to a while loop in the original unobfuscated program if t exhibits the pattern "*Pre* followed by any number of occurrences of *Body*, followed by !g and then *Post*". Here, *Pre* and *Post* can include any arbitrary sequence of statements, but *Body* must start with guard g (evaluating to true) and must be followed by any sequence of statements except for !g (denoting g evaluating to false). Similarly, the second regex captures common looping patterns that involve a break statement. This regex is similar to the first one except that the guard $g2$ that terminates the loop can be different from g. The last rule captures common if-then-else patterns: here, *Pre*, *Post* indicate the part of the trace before and after the if statement respectively, and *body* is the subtrace for the true branch.

Example 4.3. *Consider the trace from Example 4.2 and let* `x > y` *be our guard g. Then, the trace will be accepted by the basic-loop rule with the first statement* `y := 10` *matching pre, states 2-5 matching body, state 6 matching* !g*, and state 7 matching post.*

Figure 6 shows how the rules from Table 1 are used to assign scores to t-sketches using judgments $\Theta, \mathcal{E}_i \rightsquigarrow \Theta_i, w_i$. The meaning of this judgment is that, given a t-sketch Θ and candidate expansion $\mathcal{E}_i$, we refine Θ to Θ_i with score w_i. The Decompose procedure is defined using these rules as:

$$\textsc{Decompose}(\Theta, \mathcal{E}_i) = \{(\Theta_i, w_i) \mid \Theta, \mathcal{E}_i \rightsquigarrow \Theta_i, w_i\}$$

We explain the rules from Figure 6 in more detail below.

While. This rule decomposes candidate expansions involving a while loop with guard g and body $?_{body}$ where the loop has prefix $?_{pre}$ and suffix $?_{post}$. We first check whether all traces in $\Theta.\Delta[?]$ match one of the loop patterns, such as basic-loop or break-loop from Table 1. If so, the candidate expansion is assigned a high score w_γ and a new sketch $\mathcal{S}_i$ is obtained by replacing hole ? with $\mathcal{E}_i[?]$. Finally, we obtain a decomposition by associating each of the holes $?_{pre}$, $?_{body}$, and $?_{post}$ with the sub-traces that are matched by *pre*, *body*, and *post* in the loop rules from Table 1.

ITE. Similar to While, this checks whether all traces match one of the ITE patterns, such as the ite-true regex from Table 1. Then, it performs the trace decomposition based on which parts of the trace match the ITE regexes.

Block. This rule considers the case where the expansion is straight-line code. It first checks if there is a guard for which a loop or if-then-else pattern that could match with all the traces. If the check fails, then no viable control flow pattern is present and so it proceeds to decompose the trace by removing all control flow statements from $\Theta.\Delta[?]$. Since the rule assumes that the original program fragment under consideration does not have nested control-flow statements, any

Rules for constructing trace decomposition Δ

$$\frac{\begin{array}{c}\mathcal{E}_i[?] = ?_{pre}\ ;\ \text{while}(g, ?_{body})\ ;\ ?_{post} \quad \mathcal{S}_i = \Theta.\mathcal{S}[? \mapsto \mathcal{E}_i[?]] \\ \forall t_j \in \Theta.\Delta[?].\ \text{MatchLoop}(t_j, g) = (P_j, B_j, C_j) \\ \Delta_i = \Delta[?_{pre} \mapsto \cup_j P_j, ?_{body} \mapsto \cup_j B_j, ?_{post} \mapsto \cup_j C_j]\end{array}}{\Theta, \mathcal{E}_i \rightsquigarrow (\mathcal{S}_i, \Delta_i), w_\gamma}\ \text{WHILE}$$

$$\frac{\begin{array}{c}\mathcal{E}_i[?] = ?_{pre}\ ;\ \text{ite}(g, ?_{true}, ?_{false})\ ;\ ?_{post} \quad \mathcal{S}_i = \Theta.\mathcal{S}[? \mapsto \mathcal{E}_i[?]] \\ \forall t_j \in \Theta.\Delta[?].\ (P_j, T_j, F_j, C_j) = \text{MatchITE}(t_j, g) \\ \Delta_i = \Delta[?_{pre} \mapsto \cup_j P_j, ?_{true} \mapsto \cup_j T_j, ?_{false} \mapsto \cup_j F_j, ?_{post} \mapsto \cup_j C_j]\end{array}}{\Theta, \mathcal{E}_i \rightsquigarrow (\mathcal{S}_i, \Delta_i), w_\gamma}\ \text{ITE}$$

$$\frac{\begin{array}{c}T := \Theta.\Delta[?] \quad G := \cup_{t \in T} \text{Guards}(t). \\ \neg(\exists g \in G.\ \forall t \in T.\ \text{MatchLoop}(t, g) \vee \text{MatchITE}(t, g)) \\ \mathcal{E}_i[?] = ?_B \quad \mathcal{S}_i = \Theta.\mathcal{S}[h \mapsto \mathcal{E}_i[h]] \\ \Delta_i = \Theta.\Delta[?_B \mapsto \text{RemoveControlFlowStmts}(\Theta.\Delta[?])]\end{array}}{\Theta, \mathcal{E}_i \rightsquigarrow (\mathcal{S}_i, \Delta_i), w_\gamma}\ \text{BLOCK}$$

$$\frac{\begin{array}{c}\mathcal{E}_i[?] = ?_{pre}\ ;\ \text{while}(g, ?_{body})\ ;\ ?_{post} \quad \mathcal{S}_i = \Theta.\mathcal{S}[h \mapsto \mathcal{E}_i[h]] \\ \exists t_j \in \Theta.\Delta[?].\ \neg\text{MatchLoop}(t_j, g) \\ \Delta_i = \Theta.\Delta[?_{pre} \mapsto *, ?_{body} \mapsto *, ?_{post} \mapsto *]\end{array}}{\Theta, \mathcal{E}_i \rightsquigarrow (\mathcal{S}_i, \Delta_i), w_\alpha}\ \text{NO-WHILE}$$

Fig. 6. Rules describing Decompose algorithm

control-flow elements in the trace must be due to the deobfuscation and are therefore removed from the decomposed trace for $?_B$.

No-While. This rule exemplifies how we deprioritize unlikely sketches. The proposed expansion for hole ? involves a loop with guard g, but there exists at least one trace that does not match a loop pattern for guard g. Therefore, it is unlikely that $\mathcal{E}_i$ is a correct expansion of ?; hence, we assign it a low score w_α. Furthermore, the decomposition does not carry any information and allows any sub-trace (indicated by $*$).

Example 4.4. *Consider the trace t from Example 4.2, let $\Theta = (?^c_p, \{t\})$ (where $?^c_p$ is simply the starting hole), and let $\mathcal{E}_i[?^c_p] = ?_{pre}\ ;\ \text{while}(g, ?_{body})\ ;\ ?_{post}$. In this case, t matches the basic-loop rule (as mentioned in Example 4.3) and thus $\Delta, \mathcal{E}_i \rightsquigarrow (\mathcal{S}_i, \Delta_i), w_\gamma$ is satisfied where $\mathcal{S}_i = \mathcal{E}_i[?^c_p]$ and $\Delta_i = \{?_{pre} \mapsto \{[(s_1, \sigma_1)]\}, ?_{body} \mapsto \{[(s_2, \sigma_2), \ldots, (s_5, \sigma_5)]\}, ?_{post} \mapsto \{[(s_7, \sigma_7)]\}\}$.*

4.4 Synthesizing Basic Blocks of a Control Flow Skeleton

We now turn our attention to Algorithm 3 for synthesizing basic blocks in a CFS. The CompleteCFS procedure takes as input a CFS Θ (i.e., Θ has no control flow holes) and a candidate vocabulary V for the deobfuscated program. It either returns $\perp$ to indicate that Θ is infeasible (under vocabulary V) or returns a deobfuscated program over vocabulary V. Recall that the explicit vocabulary allows us to disregard some of the variables in the obfuscated program, as program obfuscations typically

1: **procedure** CompleteCFS($\Theta, \hat{P}, V$)
2: **input:** CFS Θ; deobfuscation vocabulary V
3: **output:** Completion P of Θ
4: $C \leftarrow \emptyset$
5: **for** $?^s \in \text{GetHoles}(\Theta.\mathcal{S})$ **do**
6: $\hat{P}_{?^s} \leftarrow \text{GetObfuscatedCode}(\hat{P}, \Theta.\Delta[?^s])$
7: $P_{?^s} \leftarrow \text{SynthesizeBasicBlock}(?^s, \Theta.\Delta[?^s], \hat{P}_{?^s}, V)$
8: **if** $P_{?^s} = \bot$ **then return** $\bot$
9: $C[?^s] \leftarrow P_{?^s}$
10: **return** $\Theta.\mathcal{S}[C]$

Algorithm 3. Algorithm for completing a control flow skeleton

introduce many redundant variables. To this end, we introduce the notion of *equivalence modulo a set of variables* as follows:

Definition 4.5 (**Equivalence modulo V**). *Programs P and P' are equivalent modulo variables V, written $P \equiv_V P'$, if:*

$$\forall \sigma_{in}, \sigma_{out}, \sigma'_{out}.\ [\![P]\!]_{\sigma_{in}} = \sigma_{out} \wedge [\![P']\!]_{\sigma_{in}} = \sigma'_{out} \implies \forall v \in V.\ \sigma_{out}[v] = \sigma'_{out}[v]$$

In other words, P and P' are equivalent modulo variables V if, given the same input, their outputs agree over V.

With this definition in place, we now explain the CompleteCFS procedure presented in Algorithm 3. At a high level, this algorithm performs synthesis in a *modular* way in that it tries to synthesize each basic block *independently*. For each statement hole $?^s$, it first retrieves the obfuscated version of the code via the call to GetObfuscatedCode at line 6.[2] Then, it invokes the SynthesizeBasicBlock procedure (discussed next) to find a completion $P_{?^s}$ of the hole such that $P_{?^s}$ is equivalent to its obfuscated counterpart $\hat{P}_{?^s}$ modulo variables V. If the algorithm fails to find an equivalent program for any hole in the sketch, either the sketch or the trace decomposition is wrong, and the algorithm returns $\bot$. The final deobfuscated program is obtained by replacing all the holes in the sketch with their completions as given by mapping C.

We next explain SynthesizeBasicBlock (see Algorithm 4), which performs top-down enumerative search for synthesizing straight-line code. The key difference of this algorithm from standard top-down enumerative synthesis is that it utilizes a notion of *trace extensibility* to identify dead ends. In more detail, it takes as input a hole $?^s$ to be filled in, subtraces T associated with $?^s$, deobfuscation vocabulary V, and code $\hat{P}_{?^s}$ corresponding to $?^s$ in the obfuscated program. Similar to any top-down synthesis algorithm, it maintains a priority queue over partial programs, sorted according to program size. In each iteration, it dequeues the smallest program and checks whether it is (a) complete (meaning there are no holes) and (b) equivalent to the obfuscated code modulo vocabulary V. If so, this program is returned as a solution (line 10). Otherwise, $P_{?^s}$ is refined by choosing a statement hole and replacing it with either an atomic statement or a sequence of statements (line 12). For each possible refinement $P'_{?^s}$ of $P_{?^s}$, the algorithm checks its feasibility by calling the TraceExtensible procedure, defined as follows:

[2]Since the obfuscated version of the code can be inferred from the trace in a straightforward way, we do not discuss the implementation of GetObfuscatedCode in detail.

```
1:  procedure SynthesizeBasicBlock(?^s, T, P̂_?s, V)
2:      input: Hole ?^s to be completed
3:      input: Traces T for ?^s
4:      input: Obfuscated code snippet P̂_?s for ?^s
5:      input: Variable set V
6:      output: Deobfuscated version of P̂
7:      W ← PriorityQueue(?^s, 1)
8:      while ¬Empty(W) do
9:          P_?s ← W.dequeue()
10:         if Complete(P_?s) ∧ P_?s ≡_V P̂_?s then return P_?s
11:         for P'_?s ∈ ExpandStmtHole(P_?s, T, V) do
12:             if TraceExtensible(P'_?s, T) then
13:                 W ← W.add(P'_?s, |P'_?s|)
14:     return ⊥
```

Algorithm 4. Algorithm for synthesizing a statement hole

Definition 4.6 (**Trace extensibility**). *A code fragment P is extensible with respect to trace set T if, for every trace $t \in T$, t is a trace-extension of $Trace(P, t.\sigma_{in})$.*

Intuitively, if $P'_{?s}$ is *not* trace extensible with respect to T, $P'_{?s}$ cannot be completed in a way that will yield an equivalent deobfuscated program. Hence, the algorithm discards all programs that are *not* extensible with respect to T.

4.5 Properties of the Algorithm

In this section, we prove our algorithm is sound and complete and briefly discuss the assumptions under which our algorithm returns the *minimum* control-flow reduction.

Theorem 4.1 (**Soundness**). *If $P = Deobfuscate(\hat{P}, T)$ and T are traces of $\hat{P}$, P is either $\bot$ or P is a control-flow reduction of $\hat{P}$.*

Proof. Follows from the correctness of isReduction$(P, \hat{P})$. □

Theorem 4.2 (**Completeness**). *For traces T of $\hat{P}$, $Deobfuscate(\hat{P}, T)$ eventually returns a control-flow reduction P of $\hat{P}$ if one exists.*

Proof. Please see the extended version of the paper [46] for full proofs. □

Our algorithm is guaranteed to return a minimum control-flow reduction under certain assumptions about GetNextCFS and GrowVocabulary. For more information on these assumptions and a proof of Theorem 4.2, see the extended version of the paper [46].

5 Implementation

We implemented our algorithm in a tool called Chisel for deobfuscating C programs. Chisel is written in just over 8000 lines of Python and uses GDB [26] for generating traces. In this section, we describe a number of important optimizations used in our implementation.

Exploration of t-sketches. In Section 4, we compute the likelihood of a new t-sketch Θ' given a t-sketch Θ and hole expansion $\mathcal{E}_i$ via the rules shown in Figure 6. For each expansion $\mathcal{E}_i$, Algorithm 2 adds all t-sketches produced by the rules to the queue. In practice, eagerly adding all t-sketches

can be prohibitively expensive. We address this issue by discarding very low probability t-sketches. In other words, if the probability of a t-sketch falls below a certain threshold, we do not add it to the priority queue used in the GetNextCFS procedure. It should be noted that this decision sacrifices the theoretical completeness guaranteed by our algorithm. However, in practice, we did not encounter any cases in our experimental evaluation where this incompleteness resulted in actually missing the desired control-flow sketch.

Memoizing synthesis results. Algorithm 1 iteratively adjusts the vocabulary V when a sketch cannot be completed. As presented, every time V is adjusted, CompleteCFS resumes synthesis from scratch. This significantly slows down the performance of Chisel when several vocabulary sets are explored. We address this by memoizing all programs that have been enumerated. For each completion of a basic block, we save the completion, execute the code in the blocks using the inputs from the associated traces to derive values for each variable, and then save those variables whose values agreed with the trace. When the vocabulary is adjusted, we check the cache for programs that satisfy the variables in the vocabulary and return the smallest one. Otherwise, we resume synthesis, skipping programs that are already in the cache.

GetNextVocabulary details. GetNextVocabulary internally tracks the vocabulary sets used for a given t-sketch and, when called, returns the smallest vocabulary set that has not been used thus far. However, many vocabulary sets are infeasible in that there is no valid completion of the CFS given that set. For example, if the vocabulary set does not include the return variables then no completion of the sketch can be a control-flow reduction of the obfuscated program. GetNextVocabulary rules out these infeasible sets by only considering vocabulary sets that include the input variables, the return variables and the variables in the guards of the CFS as those are expected to appear in the deobfuscated program. Another way a vocabulary set can be infeasible is if it contains a variable but none of its data dependencies. To avoid such cases, GetNextVocabulary tracks the data dependencies for each variable across all the supplied traces and prunes any vocabulary set which includes a variable but none of its dependencies.

Checking control flow reduction. Checking if the program P is a control flow reduction of $\hat{P}$ is a highly nontrivial relational verification task. Rather than perform this expensive check for every complete CFS, we instead only check if the synthesized program is a reduction with respect to the provided traces. More precisely, for each trace t in the input trace set, we check that $Trace(P, t.\sigma_{in})$ is a trace extension of $Trace(\hat{P}, t.\sigma_{in})$. While such a check is unsound, we have not encountered a case where our algorithm returned a program that was not a reduction.

6 Experiments

In this section, we present the results of an experimental evaluation that is designed to address the following research questions:

RQ1. Can Chisel deobfuscate a variety of control-flow obfuscations?
RQ2. How does Chisel compare against other baselines?
RQ3. How does Chisel scale with respect to the size of the input program?
RQ4. How important is trace-informed decomposition in practice?
RQ5. Does Chisel generalize to new obfuscations?
RQ6. Can Chisel be useful for binary deobfuscation?

6.1 Benchmarks

To answer these research questions, we start with a set of 91 unobfuscated C programs used in prior work on control-flow obfuscation [4–7]. Among these, 51 are classical algorithms covered in

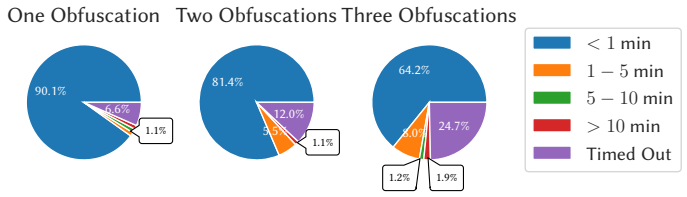


Fig. 7. Percentage of programs for which Chisel finds a minimum control-flow reduction (per number of obfuscations)

an undergraduate curriculum (e.g., sorting, search, greatest common divisor, hashing) and 40 are artificial programs which use a diverse set of control flow constructs like for-loops, while-loops, nested if-statments, etc.[3] To evaluate our approach, we obfuscate these programs using six different techniques: flattening, dead-code insertion, and basic-block splitting supported by Tigress [14] and loop-unrolling, loop-fission, and irrelevant code insertion by C-Obfuscator.[4] For Tigress, we used parameters from the recommended recipes in the documentation. Please refer to the extended version of this paper [46] for the detailed parameters we used to produce the benchmarks. Because generating all combinations of obfuscations for all programs is infeasible, we obtain the obfuscated benchmarks using the following methodology:

(1) For each original program, we obtain two obfuscated versions by randomly choosing two obfuscations and applying each to the unobfuscated benchmark.
(2) We construct two additional benchmarks (per original program) by randomly choosing two *pairs* of obfuscations and applying each to the unobfuscated benchmark.
(3) Finally, we obtain yet another two benchmarks by randomly choosing two *triples* of obfuscations and applying each triple to the unobfuscated benchmark.

Following this procedure, we generate a total of 546 obfuscated benchmarks.

6.2 Experimental Set-up

Recall that Chisel requires the user to provide a set of inputs that can be used to generate traces. Because dynamic traces are used to infer the original program's control flow, Chisel's ability to successfully deobfuscate may depend on the quality of the inputs. For our evaluation, we generate inputs using the KLEE symbolic execution engine [11], along with randomly sampled inputs. To collect traces of programs, we use gdb's stock Python API, which automates single-stepping in a C program and extracting local variable valuations.

All of the experiments reported in the following subsections are run on a 56 core machine with 264 GB RAM, running Debian 12.[5] For each benchmark, we use a time limit of 20 minutes to allow completing the experiments in a reasonable amount of time.

6.3 Main Results

The results of our evaluation are summarized in Figure 7, which shows the distribution of synthesis times for different number of obfuscations as a pie chart. Note that Chisel can successfully deobfuscate 93% of the benchmarks when only one obfuscation is applied. If we increase the number

[3]We note that the original benchmark set had 48 such programs but we removed 8 because they were duplicates.
[4]C-Obfuscator is built by the authors for the purpose of this evaluation due to a lack of open-source options. It will be made publicly available upon publication of the paper.
[5]The large machine is used to run experiments in parallel – we have recorded comparable results on machines with 16GB of RAM and 8 cores.

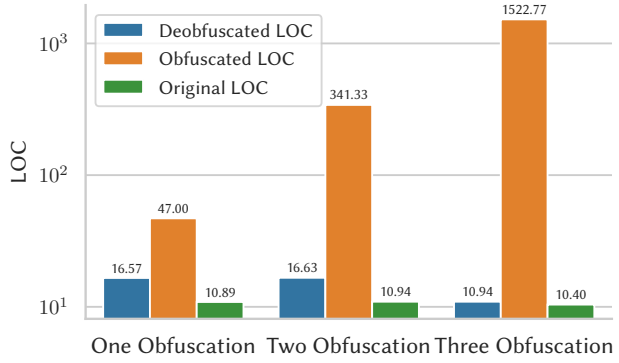


Fig. 8. Average lines of code for programs before and and deobfuscation by Chisel

of obfuscations to 2 (resp. 3), Chisel can still reverse engineer the code for 88% (resp. 75%) of the benchmarks within a 20 minute time limit.

To gain more intuition about the quality of the deobfuscated code, Figure 8 shows LOC in *log scale* for (a) the obfuscated code, (b) the original program, and (c) the code synthesized by Chisel. As is evident from this bar chart, Chisel almost always retrieves a program of comparable size to the original program before obfuscation, even when the obfuscated programs contain thousands of LOC. Furthermore, we analyzed the deobfuscated programs and found that 81% of "solved" benchmarks are reduced to within 5 LOC of the source program, with an average reduction in size of 63%, or 555 lines of code, compared to the obfuscated program.

Finally, we performed a manual analysis of the results. We found that most deobfuscated programs are minimum control-flow reductions. For those programs which are not, we found a common pattern: original programs containing a single while loop with a fixed number of iterations N. These programs can be synthesized by copying the body of the loop N times. For example, `while(i < 3, i++)` is equivalent to `i++` repeated 3 times when $i = 0$. In this case, because the control-flow sketch of the while loop is larger than the control-flow sketch of the straight-line program, we will synthesize the straight-line program first. We found the only time this occurs is when there is a fixed-iteration loop whose body contains no control-flow operators.

> **Result for RQ1:** Given a 20 minute time limit, Chisel can deobfuscate 86% of the benchmarks. On average, Chisel results in an average reduction in LOC of 63% (which is on average 555 LOC) compared to the obfuscated code. Furthermore, the deobfuscation result is within 5 (resp. 10) LOC of the original program for 81% (resp. 92%) of the solved benchmarks.

6.4 Baseline Comparison

In this section, we compare Chisel to three different baselines inspired by the deobfuscation literature. First, we compare to Sketch [60], a CEGIS-based program synthesis tool in which we can encode the deobfuscation problem. Second, given the recent successes of large language models (LLMs) across a variety of tasks, including deobfuscation (e.g., [38]), we compare Chisel to OpenAI's GPT-4 [51]. Finally, we compare with `opt`, a binary optimization tool for LLVM bitcode which has been shown to be effective for deobfuscation [25].

Table 2. Comparison of Chisel with Sketch.

Sketch Comparison			
Tool	1 Obfuscation	2 Obfuscations	3 Obfuscations
Sketch	0%	0%	0%
Sketch+CFS	56%	33%	32%

6.4.1 Sketch comparison. Sketch [59, 60] is a well-known program synthesis tool which uses counterexample-guided inductive synthesis (CEGIS) to solve synthesis problems in a C-like language. Among existing synthesis tools, we choose Sketch as a baseline because it is the only synthesizer that has a C-like grammar. In contrast, evaluating against other SyGuS baselines requires writing a custom C interpreter, which is a non-trivial task.

To compare against Sketch, we encode the deobfuscation task as a sketch-completion problem; details of our encoding can be found in the extended version of this paper [46]. We compare Chisel against two different uses of Sketch. In the first case, Sketch is provided the obfuscated program as a specification and must synthesize the entire deobfuscation (as is the case for Chisel). We refer to this variant as Sketch in our comparison. In the second case, we provide Sketch with the ground-truth control flow skeleton and only require it to complete the given CFS. This is in line with Sketch's original usage scenario where the user provides a sketch as input – we refer to this variant as Sketch+CFS. However, please note that Sketch+CFS solves a much easier problem compared to Chisel.

Benchmarks. We evaluate Sketch on a subset of the original benchmarks for two reasons: First, to evaluate Sketch-CFS, we need access to a CFS (including the trace decomposition), as the trace decomposition defines the specification for each subtask provided to Sketch. However, note that we do not have access to such a ground truth CFS unless Chisel produces them[6]; thus, we restrict ourselves to the subset of benchmarks for which Chisel produces a CFS. Second, many of the benchmarks contain C features that are not supported natively in Sketch, so, to allow for a fair comparison, we restrict ourselves to those benchmarks that only contain Sketch-supported features. Hence, for this evaluation, we only consider a total of 65 benchmarks.

Results. Table 2 shows the results of this experiment for Sketch; here, we do not report the results for Chisel because it solves *all* of these benchmarks. As shown in Table 2, Sketch cannot solve any of the benchmarks without having access to the control flow skeleton. However, even when it is provided the CFS, it can only solve 41% of the benchmarks overall. Furthermore, Sketch solves each benchmark in an average of 77.01 seconds as compared to Chisel which takes only 63.38 seconds on average.[7] We believe Chisel outperforms Sketch even when it is provided the CFS because Chisel can prune many CFS completions using trace-based decomposition unlike Sketch.

6.4.2 Large Language Model Comparison. Given the recent successes of large language models (LLMs) across a variety of tasks, we next compare Chisel to OpenAI's GPT-4 [51]. To perform this comparison, we sample 150 benchmarks from our benchmark set (50 from each of 1, 2, and 3 obfuscations). For each benchmark, we ask GPT-4 to deobfuscate the program and take the top 3

[6]Obtaining the CFS from the original program is difficult because applying obfuscation alters program statements of a trace in a non-obvious way, preventing us from directly associating each obfuscated statement to its ground-truth counterpart.
[7]As with all experiments we have reported, both Sketch and Chisel were run on the same machine with the same specs for this comparison.

Table 3. Comparison of Chisel with GPT-4.

LLM Comparison						
Tool	1 Obfuscation		2 Obfuscations		3 Obfuscations	
	Equiv	Minimum	Equiv	Minimum	Equiv	Minimum
GPT-4	72%	54%	58%	46%	58%	46%
Chisel	**94%**	**94%**	**92%**	**92%**	**80%**	**80%**

results.[8] When developing our prompting strategy for the ChatGPT comparison, we experimented with different numbers and types of prompts and chose the strategy that performed the best across a small set of representative examples from our benchmark set. The specific prompt we used can be found in the extended version of this paper [46].

The results of this experiment are summarized in Table 3, which shows the percentage of benchmarks for which the deobfuscation approach returned (1) an equivalent program and (2) a minimum control flow reduction. As shown, Chisel finds a minimum program much more often than GPT-4 (34-46% more benchmarks depending on the number of obfuscations). GPT-4 is able to find an equivalent (but not necessarily minimal program) in a high number of cases (72%) for 1 obfuscation, but this drops significantly with multiple obfuscations.

To better understand the performance of GPT-4, we manually investigated the programs it produced. In 80% of cases, GPT-4 could at least produce compilable code – code that could not compile usually contained type errors or left off return values. In cases where GPT-4 produced an equivalent but non-minimum program, it was often that some (but not all of) the obfuscations had been removed; in fact, it was not unusual for GPT-4 to simply copy the obfuscated code. In cases where GPT-4 produced the minimum program, we noticed that the program usually corresponded to common programming assignments like factorial or bubblesort (as opposed to one of the artificial programs from the benchmark set). This observation suggests that our results may slightly overstate the abilities of GPT-4 for deobfuscation, given that it seems to do best on the programs which (likely) occur most frequently in its training data.

6.4.3 Compiler Optimization Comparison. Compiler optimizations have been found to be effective for code deobfuscation in a variety of domains [25, 28, 41, 72]. In this experiment, we compare Chisel to `opt`, LLVM's binary optimizer. Because Chisel operates at the source-code level while `opt` simplifies LLVM bitcode, we perform the comparison by compiling the results of Chisel and comparing the resulting binary size. Table 4 shows the results of this experiment. In general, we can see that binaries produced from Chisel are significantly smaller (30% smaller binary size on average) and are much closer in size to the compiled minimum program.

To give further context to these results, we sampled 30 benchmarks and used the Hex-Rays decompiler [29] to compare the results of `opt` and Chisel at the source-code level. For those benchmarks sampled, `opt` could produce the minimum reduction in only 7 cases (23%) as compared to Chisel which can find the minimum reduction in all 30 cases. Furthermore, decompiled code from Chisel was on average 150 LOC (or about 42%) smaller as compared to decompiled code from `opt`. On manual inspection of the decompiled code from `opt`, we found that in most cases most of the obfuscations still remained.

[8]We tested with between the top 1 and 10 results on a small sample of the benchmarks and found no difference with more than the top 3 results.

Table 4. Comparison of Chisel with LLVM opt.

Compiler Optimization Comparison				
Measurement	Chisel		opt	
	mean	median	mean	median
Binary Size (in KB)	**4.5**	**4.1**	6.4	5.9
Size Compared to Minimum Program	**1.1**	**1.0**	1.5	1.4

6.5 Related Tool Comparison

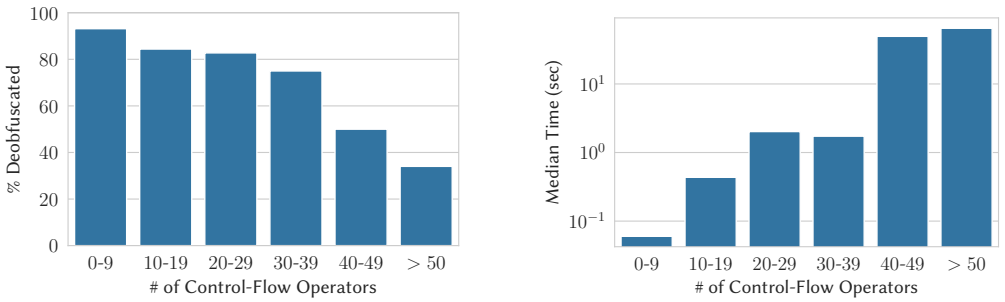
In addition to the general deobfuscation baselines compared to in Section 6.4, we also compare to Yadegari et al. [68] which construct a simplified control-flow graph (CFG) from a trace of a control-flow obfuscated program. In their algorithm, Yadegari et al. apply simplification rules to the input trace, construct a CFG based on that simplified trace, and apply transformations to the CFG to further simplify it. This is conceptually very different from our synthesis-based approach which uses traces as a mechanism for pruning the search space.

Thus, to compare and contrast these different approaches, we perform a small comparison with their tool on 5 simple benchmarks which Chisel can solve in seconds. The benchmarks are obfuscated with 3 different control-flow extensions: branch insertion, dead-code insertion (via opaque predicates), and flattening. It should be noted that a key difference between Chisel and the tool from Yadegari et al. is that their tool uses a *single* trace to construct a simplified program while Chisel uses multiple. Thus, for each example, we tried each trace used by Chisel as an input to the tool from Yadegari et al. and chose whichever produced the best result.

We found that their implementation does not produce an equivalent program on any benchmark we tried. For instance, we considered a simple program that returns the parity of an integer and obfuscated it by adding irrelevant branches. Not only does Yadegari's tool not remove the comparison instructions introduced by obfuscation, but it also incorrectly removes the parity check computation. This trend holds for every benchmark we tested, i.e., Yadegari's tool *never* removed all of the obfuscated code nor did it ever even produce equivalent code. In the extended version of this paper [46], we give a full explanation of each of the benchmarks considered and how the tool from Yadegari et al. failed to deobfuscate that benchmark.

We conjecture two reasons as to why the tool from Yadegari et al. performed so poorly on these simple benchmarks. First, as mentioned above, their work makes the strong assumption that an equivalent program can be recovered from a single trace. We believe this assumption is too strong in many cases, especially when programs contain complex looping and conditional behavior. Chisel overcomes this limitation by considering multiple traces at once while deobfuscating. Second, the tool from Yadegari et al. was analyzed and evaluated on virtualization and return-oriented-programming (ROP) obfuscations, which have very different behavior from the control-flow extensions considered by Chisel. For instance, branch insertion *statically* adds extra control-flow into a program, while virtualization encodes a program into a new IR which is *dynamically* decoded and executed at runtime. It is not surprising that an algorithm designed to deobfuscate techniques like virtualization and ROP might not be as effective when facing the static control-flow extensions considered in this paper.

While this evaluation is not a completely apples-to-apples comparison, we believe it sheds light on the different intended usage scenarios of these tools. In particular, our comparison indicates that Chisel may be more effective for static obfuscations like branch insertion while the tool from Yadegari et al. can handle obfuscations like virtualization and ROP which are out of the scope of obfuscations considered by Chisel. Because of this, we believe it is feasible that the two techniques



(a) Percentage of Synthetic Program Deobfuscated by Control-Flow Complexity

(b) Median Deobfuscation Time of Synthetic Programs by Control-Flow Complexity

Fig. 9. Chisel performance by code complexity

Table 5. Highlights from scalability evaluation.

Deobfuscation Highlights			
Max # ITE	33,996	Max # Loops	6,556
Max LOC	147,923	Max # of Variables	4,384

could be combined to handle a wider variety of obfuscations than those considered by either tool individually.

> **Result for RQ2:** Chisel is able to deobfuscate a significantly larger portion of the benchmarks compared to three different baselines, including (1) Sketch, a program synthesis tool, (2) GPT-4, a state-of-the-art large language model, and (3) LLVM's binary optimizer. Chisel is also able to handle obfuscations like branch insertion which are not well supported by Yadegari et al. [68], a state-of-the-art control-flow deobfuscator.

6.6 Scalability Evaluation

To stress test the scalability of our approach, we perform an additional experiment on 50 synthetic benchmarks that intentionally vary in size and complexity. In more detail, we sampled 10 programs for each AST-size of 5, 10, 15, 20, and 30 and obfuscate these synthetic source programs using the exact same methodology described in Section 6.1. To evaluate the scalability of Chisel, we plot the number of benchmarks solved and time taken to solve benchmarks vs. the control-flow complexity of the program, measured as the number of control-flow operators in the program.[9]

The results of this evaluation are shown in Figures 9a and 9b. As expected, the more complicated the program, the fewer benchmarks Chisel can solve. Similarly, the more complicated the program, the slower the deobfuscation. However, it should be noted that Chisel can still solve over half of benchmarks containing between 40 and 50 control-flow operators, which comprise programs of thousands of lines of code. Furthermore, the largest programs that Chisel can deobfuscate contain up to 33,996 if statements, 6,556 loops, and 147,923 lines of code.

[9]It should be noted that this is a better metric of complexity than LOC as some obfuscations create large amounts of dead code that are easily eliminated.

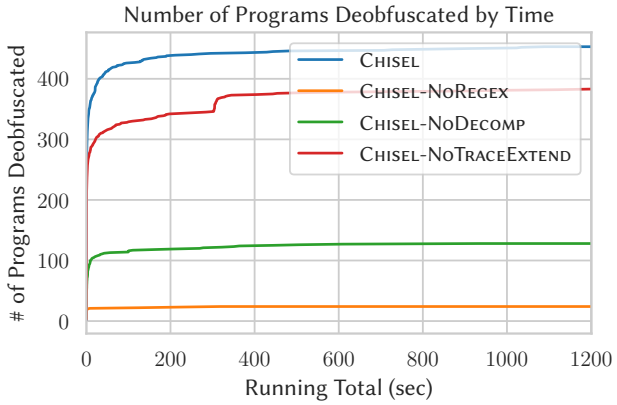


Fig. 10. Comparison of Chisel and ablations.

Result for RQ3: Given a 20-minute timeout,Chisel is able to deobfuscate programs containing up to 6,556 loops, up to 33,996 conditionals and more than one hundred thousand lines of code.

6.7 Ablation Study

Next, we evaluate the relative importance of the key ideas underlying our approach through an ablation study. Specifically, we consider the following variants of Chisel:

- Chisel-NoRegex: This variant does not use regex pattern matching to assign scores to t-sketches. It considers all t-sketches to be equally likely and enumerates them according to size.
- Chisel-NoDecomp: This variant does not performs trace decomposition. As a result, it cannot complete holes in the sketch in a modular way.
- Chisel-NoTraceExtend: This variant is identical to Chisel except it does not use trace extensibility for pruning the search space. This variant essentially corresponds to removing the call to TraceExtensible in Algorithm 4.

The results of this ablation study are presented as a cumulative distribution function (CDF) in Figure 10 where the x-axis shows cumulative running time and the y-axis shows the number of benchmarks solved. Chisel outperforms all ablations – the next best ablation (Chisel-NoTraceExtend) can solve 85% of the benchmarks solved by Chisel, while the other two solve only 28% and 6% respectively.

Result for RQ4: Trace-informed compositional synthesis is crucial for the effectiveness of our approach: Without using traces for CFS inference, Chisel solves 94% fewer benchmarks, and, without decomposition, Chisel solves 72% fewer benchmarks.

6.8 Generalizability of Chisel

To assess if Chisel can be applied to new obfuscation techniques, we performed two experiments. In the first, we chose an obfuscation technique from the literature called the branch insertion transformation [15] which is different from all obfuscations considered when creating Chisel. The branch insertion transformation statically replaces each basic block B in the program with a non-deterministic choice between two new basic-blocks B_1 and B_2, each of which is an obfuscated version of B. A detailed description of the technique can be found in the extended version of this

paper [46]. We used this technique to obfuscate all 91 benchmarks and found that Chisel could find the minimum program for 85 of them (93%) in under 20 minutes.

In the second experiment, we disabled each trace-matching rule from Table 1 individually to assess their impact on the performance of Chisel *per obfuscation type* – if a rule is general, one would expect its absence to affect performance for multiple obfuscation types. We found that the average difference in performance between obfuscation types when disabling a rule was 8.4%. In other words, disabling any of our rules leads to a similar decrease in performance across all obfuscation types, indicating the rules generalize beyond the specific obfuscations considered in this evaluation. The full results can be seen in the extended version of this paper [46].

> **Result for RQ5:** Chisel is able to deobfuscate 93% of the programs obfuscated with the branch insertion transformation which was not considered when developing the tool. Furthermore, disabling any trace-matching rule results in a performance decrease across most of the obfuscation types we consider, indicating they are not overfit to handle specific obfuscations.

6.9 Binary Evaluation

Our experiments in the preceding sections demonstrate that Chisel can be useful across a variety of obfuscations and programs. However, all of these evaluate Chisel's usefulness on source-level deobfuscation, while many deobfuscation tasks are at the binary level. To address this concern, we evaluate Chisel's binary deobfuscation capabilities by combining it with IDA-Pro [29], a state-of-the-art decompiler. To do this, we first compiled 60 random obfuscated benchmarks that Chisel was able to solve and then used IDA-Pro to decompile them. As noted by [42], most decompilers (IDA-Pro included) do not produce compilable code. Thus, we perform a number of small changes to the output of the decompiler, similar to those described in [42], such as adding back in necessary keywords, adjusting illegal variable names, and removing keywords that are not supported by `gcc`. A full description of the changes can be found in the extended version of this paper. To recompile the programs, we use `gcc` with default options. Finally, we used Chisel to deobfuscate the resulting program with a 20-minute timeout. For these benchmarks, Chisel finds the minimum program for 81% and finds a program smaller than the original for 91%.

Failure analysis. We conducted a manual analysis of the remaining 9% of programs for which Chisel could not find a smaller version of the obfuscated program and found that all failures were due to one of two different code changes introduced by the decompiler. The first is that some assignments are merged into other expressions – for instance, `x < y; x++` would be merged into `x++ < y`. Chisel does not currently support this form of updating in its trace collection. When these are split out into separate statements (as can be achieved with lightweight static analysis) Chisel succeeds on these benchmarks. The second cause of failures is that the decompiler splits a single variable into multiple copies of the same variable. This change causes some of our regular expression matching rules to fail (as a guard may appear using various forms of the same variable), slowing Chisel's deobfuscation process. Again, however, we believe this could be easily resolved via lightweight static analysis to merge duplicate variables.

> **Result for RQ6:** When combined with a state-of-the-art decompilation technique, Chisel was able to successfully deobfuscate 55 out of 60 obfuscated binaries, resulting in an average LOC reduction of 85%.

7 Limitations

Our experiments demonstrate that Chisel is able to deobfuscate a wide variety of control-flow obfuscations across different programs, even when multiple obfuscations are applied. However, there are some limitations to both our approach, as well as our prototype, which we discuss next.

First, our method relies on dynamic traces for deobfuscation. We use dynamic analysis because static analyses are notoriously ineffective for reasoning about obfuscated code [16, 50]. We are not alone in making this decision – a number of deobfuscation techniques rely on dynamic traces [9, 17, 25, 67, 68]. However, it should be noted that this decision is a tradeoff. While it avoids many of the pitfalls faced by static analyses, low coverage can reduce the speed of our synthesis algorithm as it uses traces to speed up the synthesis process and reduce the search space. Although trace coverage can impact our algorithm's ability to retrieve the *complete* deobfuscated program, per Theorem 4.2, it can always eventually recover the portions of the original program exercised by the traces. This is a valuable property for real-world reverse engineering, where perfect deobfuscation is not always necessary.

Second, programs generated by Chisel intentionally do not contain `gotos` (as they often make code less readable) and only use `while` for looping (with `break` statements). Hence, if the original program contains alternative looping mechanisms such as a `for` loop, the code synthesized by Chisel would instead have a `while` loop. Similarly, `switch` statements deobfuscate to `if-then-else` statements.

Third, our implementation uses testing (rather than full-fledged verification) to check that the deobfuscated program is equivalent to the obfuscated one, which means it is possible for Chisel to produce a program that is not equivalent. While this outcome is certainly possible, we found no instances of this when manually reviewing the results of our experiments, which included hundreds of deobfuscated programs.

Finally, Chisel only handles control-flow extensions, which do not include some well-known control-flow obfuscations, including virtualization [66], return-oriented programming [10], and exception-based obfuscations [69]. In addition, popular non-control-flow based obfuscations like mixed boolean arithmetic [77] are not supported by Chisel. However, there is no fundamental reason why Chisel cannot be used in tandem with techniques designed to handle such obfuscations.

8 Related Work

Chisel is part of a long line of work on program deobfuscation. Broadly speaking, prior work can be bucketed into two categories depending on whether they target data-flow [9, 18, 77] or control-flow [37] obfuscations. Recall that data-flow obfuscations focus on obfuscating constants and expressions in the program, whereas control-flow obfuscations focus on complicating the overall control flow of the program. Chisel falls under the latter category, so we first describe prior work on control-flow debofuscation.

Control-flow deobfuscation. In recent years, many control flow deobfuscators have been proposed that target specific control flow deobfuscations or limited combinations of them [22, 70, 71]. For example, the ReDex code optimizer [71] and Deoptfuscater [70] remove target variable renaming, call indirection, and opaque predicates in Android applications. CaDeCFF [22] uses data-flow analysis and tree-based code generation to deobfuscate flattening. DiANa [34] combines taint analysis and symbolic execution to deobfuscate Android binaries obfuscated using O-LLVM [33]. BinRec [1] deobfuscates a binary by lifting dynamic traces of the binary to an intermediate representation that can be lowered back into a "recovered" binary after simplification. Their approach is similar in spirit to compiler optimization for code deobfuscation [25, 28, 41, 72] which we compare to in Section 6.4.3. Unlike these prior works, Chisel can handle a broad class of control flow obfuscations

including arbitrary combinations of them. One prior work that also targets a broad class of control flow obfuscations is Yadegari et al., [68]. They perform dynamic taint analysis to identify control dependencies, use the dependencies to apply semantics-preserving simplifications, and finally construct a control-flow graph (CFG) representing the deobfuscated code. However, the effectiveness of this technique is dependent on manually-crafted simplification rules that may or may not apply for some types of obfuscations. In contrast to their approach, our method uses syntax-guided synthesis rather than rule-based simplification. Please see Section 6.5 for a comparison between their approach and Chisel. Additionally, please see the extended version of this paper [46] for a detailed discussion on our attempts to evaluate against other control-flow deobfuscators, and why we believe Chisel will outperform them.

Trace-guided synthesis. A number of synthesizers use traces to guide their search procedure [20, 30, 45, 52, 54, 58, 74], but they do so in different ways from Chisel. In particular, prior work on *recursive program synthesis* requires users to provide "recursion traces" to effectively augment the set I/O examples for the synthesis task. For example, Myth [52] requires users to provide a complete trace of the call stack of the desired program for a given I/O example, and Syrup [74] utilizes a novel version-space algebra based on recursion traces to compactly represent solutions to recursive synthesis problems. Traces are also used in *programming from demonstration* tasks [20, 54] where a program needs to be synthesized that matches a trace of actions performed by the expert. For example, Patton et al [54], use traces to synthesize robot controllers. Their idea is to view demonstrations as positive string examples over a language and infer a program sketch by generalizing from those positive examples to a regular expression. They complete the inferred sketches using LLM guidance. Prior work by Mariano et al [45] leverages traces to transpile imperative to functional programs. Similar to Chisel, they use traces to prune infeasible partial programs using a notion of trace-compatibility, which is similar to our trace-extensibility. Their notion of trace compatibility requires agreement between values of *shared expressions* in a pair of so-called *cognate grammars*, whereas our notion of trace extensibility requires agreement between a subset of the variables. However, unlike [45], we additionally use traces to guide the generation of control flow skeletons and decompose the target program into independent sub-problems. Finally, Konure [57] uses active learning to infer models of applications which access relational databases. Like Chisel, their approach observes database interactions (a sort of trace) and uses program synthesis to generate a matching program from those interactions. However, unlike Chisel, which is designed for the problem of deobfuscation, Konure's DSL and inference algorithm are designed specifically for database applications.

Compositional synthesis. Many synthesizers try to decompose a high level synthesis task into independent subtasks to improve scalability [2, 3, 24, 27, 52, 55, 75]. For example, λ^2 [24] uses the semantics of list combinators to infer new I/O examples for sub-problems and type-directed synthesizers like Synquid [55] propagate goal types for holes in the partial program which, in many cases, can be solved independently. CLIS [75] performs UDF to SQL translation using a notion of lazy inductive synthesis where each iteration of the synthesis procedure generates increasingly hard synthesis sub-problems given a dataflow graph of the UDF program. Unlike all these approaches, our work performs modular synthesis using traces from the obfuscated program along with a candidate control-flow sketch.

Synthesis for deobfuscation. We are not the first to apply program synthesis techniques to deobfuscate programs. Syntia [9] uses SMT-solving and stochastic program synthesis to deobfuscate Mixed Boolean-Arithmetic (MBA) expressions [77]. Xyntia [47] proposes a new AI-based blackbox method for deobfuscating MBA expressions. AutoSimpler [76] deobfuscates MBA expressions via a

heuristic Nested Monte Carlo Search. QSynth [18] deobfuscates MBA expressions and virtualization expressions using enumerative synthesis. The work described in [32] deobfuscates loop-free code by reducing to SMT solving. These deobfuscators target either data-flow obfuscations or straight line programs, and thus are not applicable to our setting where entire functions with loops and conditionals need to be synthesized.

9 Conclusion and Future Work

In this paper, we proposed a new technique for reverse engineering control-flow obfuscations using program synthesis. We have performed an extensive empirical evaluation of our tool, Chisel, on more than 500 obfuscated programs, including those that have been obfuscated using multiple different techniques. Our evaluation shows that Chisel is able to successfully recover the original program in 86% of the cases. We compare Chisel against three different general deobfuscation baselines, including a program synthesizer, a state-of-the-art LLM, and a binary optimizer, and show that Chisel is much more effective at the deobfuscation task. We also compare Chisel with a state-of-the-art control-flow deobfuscator from the literature [68] and conclude that their approach is potentially complimentary with ours as they focus on a different set of obfuscation techniques than those considered by Chisel.

Acknowledgments

This works was supported in part by National Science Foundation grant 2040206. This work was conducted in a research group supported by NSF awards CCF-1762299, CCF-1918889, CNS-1908304, CCF-1901376, CNS-2120696, CCF-2210831, and CCF-2319471.

References

[1] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 36, 16 pages. https://doi.org/10.1145/3342195.3387550

[2] Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336. https://doi.org/10.1007/978-3-319-21668-3_10

[4] Sebastian Banescu. 2017. *Characterizing the strength of software obfuscation against automated attacks.* Ph. D. Dissertation. Technische Universität München.

[5] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation against Symbolic Execution Attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2991079.2991114

[6] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. 2017. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning.. In *USENIX Security Symposium*. USENIX, USA, 661–678.

[7] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner. 2015. A framework for measuring software obfuscation resilience against automated attacks. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, IEEE, USA, 45–51.

[8] Chandan Kumar Behera and D Lalitha Bhaskari. 2015. Different obfuscation techniques for code protection. *Procedia Computer Science* 70 (2015), 757–763.

[9] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *USENIX Security Symposium*. USENIX, USA, 643–659.

[10] Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. The ROP needle: hiding trigger-based injection vectors via code reuse. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol,

Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 1962–1970. https://doi.org/10.1145/3297280.3297472

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[12] Jien-Tsai Chan and Wuu Yang. 2004. Advanced obfuscation techniques for Java bytecode. *Journal of Systems and Software* 71, 1 (2004), 1–10. https://doi.org/10.1016/S0164-1212(02)00066-3

[13] Christian Collberg. 2023. Flatten. https://tigress.wtf/flatten.html. Accessed: 2023-04-10.

[14] Christian Collberg. 2023. The Tigress C Obfuscator. https://tigress.wtf/. Accessed: 2023-04-09.

[15] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[16] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 184–196. https://doi.org/10.1145/268946.268962

[17] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of Virtualization-Obfuscated Software A Semantics-Based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, USA, 275–284. https://doi.org/10.1145/2046707.2046739

[18] Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. QSynth - A Program Synthesis based approach for Binary Code Deobfuscation. *Proceedings 2020 Workshop on Binary Analysis Research* 0, 0 (2020), 42–49.

[19] Stephen Dolan. 2013. mov is Turing-complete. https://drwho.virtadpt.net/files/mov.pdf

[20] Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: web robotic process automation using interactive programming-by-demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 152–167. https://doi.org/10.1145/3519939.3523711

[21] Weiyu Dong, Jian Lin, Rui Chang, and Ruimin Wang. 2022. CaDeCFF: Compiler-Agnostic Deobfuscator of Control Flow Flattening. In *Internetware 2022: 13th Asia-Pacific Symposium on Internetware, Hohhot, China, June 11 - 12, 2022*. ACM, China, 282–291. https://doi.org/10.1145/3545258.3545269

[22] Weiyu Dong, Jian Lin, Rui Chang, and Ruimin Wang. 2022. CaDeCFF: Compiler-Agnostic Deobfuscator of Control Flow Flattening. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware* (Hohhot, China) *(Internetware '22)*. Association for Computing Machinery, New York, NY, USA, 282–291. https://doi.org/10.1145/3545258.3545269

[23] Stephen Drape. 2010. Intellectual property protection using obfuscation.

[24] John Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *ACM SIGPLAN Notices* 50 (06 2015), 229–239. https://doi.org/10.1145/2813885.2737977

[25] Peter Garba and Matteo Favaro. 2019. SATURN - Software Deobfuscation Framework Based on LLVM. *CoRR* abs/1909.01752 (2019), 27–38. arXiv:1909.01752 http://arxiv.org/abs/1909.01752

[26] GNU. 2023. GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/. Accessed: 2023-04-09.

[27] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and Effect-Guided Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3453483.3454048

[28] Adrian Herrera. 2020. Optimizing Away JavaScript Obfuscation. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, USA, 215–220. https://doi.org/10.1109/SCAM51674.2020.00029

[29] SA Hex-Rays. 2013. Hex-Rays Decompiler.

[30] Martin Hofmann. 2010. IGOR2 - an analytical inductive functional programming system: tool demo. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Madrid, Spain) *(PEPM '10)*. Association for Computing Machinery, New York, NY, USA, 29–32. https://doi.org/10.1145/1706356.1706364

[31] Anusthika Jeyashankar. 2023. Most Common Malware obfuscation Techniques. https://www.socinvestigation.com/most-common-malware-obfuscation-techniques/. Accessed: 2023-04-09.

[32] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[33] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, USA, 3–9. https://doi.org/10.1109/SPRO.2015.10

[34] Zeliang Kan, Haoyu Wang, Lei Wu, Yao Guo, and Guoai Xu. 2019. Deobfuscating Android Native Binary Code. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, USA, 322–323. https://doi.org/10.1109/ICSE-Companion.2019.00135

[35] Seoyeon Kang, Jeongwoo Kim, Eun-Sun Cho, and Seokwoo Choi. 2022. Program Synthesis-based Simplification of MBA Obfuscated Malware with Restart Strategies. In *Proceedings of the 2022 ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks* (Los Angeles, CA, USA) *(Checkmate '22)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3560831.3564258

[36] Sangjun Ko, Jusop Choi, and Hyoungshick Kim. 2017. COAT: Code Obfuscation Tool to Evaluate the Performance of Code Plagiarism Detection Tools. In *2017 International Conference on Software Security and Assurance (ICSSA)*. IEEE, USA, 32–37. https://doi.org/10.1109/ICSSA.2017.29

[37] Renuka Kumar and Anjana Mariam Kurian. 2018. A Systematic Study on Static Control Flow Obfuscation Techniques in Java. *ArXiv* abs/1809.11037 (2018), 1–20.

[38] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems* 34 (2021), 14967–14979.

[39] Jaehyung Lee and Woosuk Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (, Copenhagen, Denmark,) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2351–2365. https://doi.org/10.1145/3576915.3623186

[40] Yuhan Li, Bin Wen, and Haixiao Zheng. 2023. Generic O-LLVM Automatic Multi-Architecture Deobfuscation Framework Based on Symbolic Execution. In *Proceedings of the 4th International Conference on Advanced Information Science and System* (Sanya, China) *(AISS '22)*. Association for Computing Machinery, New York, NY, USA, Article 59, 6 pages. https://doi.org/10.1145/3573834.3574541

[41] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. 2018. Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization. In *Information and Communications Security*, Sihan Qing, Chris Mitchell, Liqun Chen, and Dongmei Liu (Eds.). Springer International Publishing, Cham, 313–324.

[42] Zhibo Liu and Shuai Wang. 2020. How Far We Have Come: Testing Decompilation Correctness of C Decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 475–487. https://doi.org/10.1145/3395363.3397370

[43] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 389–400. https://doi.org/10.1145/2635868.2635900

[44] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers and Security* 51 (2015), 16–31. https://doi.org/10.1016/j.cose.2015.02.007

[45] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. 2022. Automated Transpilation of Imperative to Functional Code Using Neural-Guided Program Synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 71 (apr 2022), 27 pages. https://doi.org/10.1145/3527315

[46] Benjamin Mariano, Ziteng Wang, Shankara Pailoor, Christian Collberg, and Isil Dillig. 2024. Control-Flow Deobfuscation Using Trace-Informed Compositional Program Synthesis (extended version). https://bmarwritescode.github.io/assets/pdf/chisel.pdf.

[47] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. 2021. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2513–2525. https://doi.org/10.1145/3460120.3485250

[48] Uwe Meyer-Bäse, Encarni Castillo, Guillermo Botella, L. Parrilla, and Antonio García. 2011. Intellectual property protection (IPP) using obfuscation in C, VHDL, and Verilog coding. In *Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX*, Harold Szu (Ed.), Vol. 8058. International Society for Optics and Photonics, SPIE, USA, 80581F. https://doi.org/10.1117/12.884142

[49] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security and Privacy* 9, 5 (2011), 41–47. https://doi.org/10.1109/MSP.2011.98

[50] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/3359789.3359812

[51] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[52] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *SIGPLAN Not.* 50, 6 (jun 2015), 619–630. https://doi.org/10.1145/2813885.2738007
[53] Colby Parker, Jeffrey Todd McDonald, and Dimitrios Damopoulos. 2021. Machine Learning Classification of Obfuscation using Image Visualization. In *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, July 6-8, 2021*, Sabrina De Capitani di Vimercati and Pierangela Samarati (Eds.). SCITEPRESS, USA, 854–859. https://doi.org/10.5220/0010607408540859
[54] Noah Patton, Kia Rahmani, Meghana Missula, Joydeep Biswas, and Işıl Dillig. 2024. Programming-by-Demonstration for Long-Horizon Robot Tasks. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 512–545. https://doi.org/10.1145/3632860
[55] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *SIGPLAN Not.* 51, 6 (jun 2016), 522–538. https://doi.org/10.1145/2980983.2908093
[56] Aleieldin Salem and Sebastian Banescu. 2016. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW@ACSAC 2016, Los Angeles, California, USA, December 5-6, 2016*, Mila Dalla Preda, Natalia Stakhanova, and Jeffrey Todd McDonald (Eds.). ACM, USA, 1:1–1:11. https://doi.org/10.1145/3015135.3015136
[57] Jiasi Shen and Martin C. Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3314221.3314591
[58] Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving Neural Program Synthesis with Inferred Execution Traces. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc., USA. https://proceedings.neurips.cc/paper_files/paper/2018/file/7776e88b0c189539098176589250bcba-Paper.pdf
[59] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*. ACM, USA, 136–148. https://doi.org/10.1145/1375581.1375599
[60] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, USA, 404–415. https://doi.org/10.1145/1168857.1168907
[61] PreEmptive Solutions. 2023. Control Flow Obfuscation. https://www.preemptive.com/dasho/pro/userguide/en/understanding_obfuscation_control.html. Accessed: 2023-04-11.
[62] Ramtine Tofighi-Shirazi, Irina Mariuca Asavoae, and Philippe Elbaz-Vincent. 2019. Fine-Grained Static Detection of Obfuscation Transforms Using Ensemble-Learning and Semantic Reasoning. *CoRR* abs/1911.07523 (2019), 1–12. arXiv:1911.07523 http://arxiv.org/abs/1911.07523
[63] Ramtine Tofighi-Shirazi, Irina-Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. 2019. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection* (London, United Kingdom) *(SPRO'19)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/3338503.3357719
[64] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh-ha Le. 2018. DoSE: Deobfuscation Based on Semantic Equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop* (San Juan, PR, USA) *(SSPREW-8)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3289239.3289243
[65] S.K. Udupa, S.K. Debray, and M. Madou. 2005. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, USA, 10 pp.–54. https://doi.org/10.1109/WCRE.2005.13
[66] Code Virtualizer. 2023. Total obfuscation against reverse engineering.
[67] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 732–744. https://doi.org/10.1145/2810103.2813663
[68] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy*. IEEE, USA, 674–691. https://doi.org/10.1109/SP.2015.47
[69] Xinlei Yao, Jianmin Pang, Yichi Zhang, Yong Yu, and Jianping Lu. 2012. A Method and Implementation of Control Flow Obfuscation Using SEH. In *2012 Fourth International Conference on Multimedia Information Networking and Security*. IEEE, USA, 336–339. https://doi.org/10.1109/MINES.2012.25
[70] Geunha You, Gyoosik Kim, Sangchul Han, Minkyu Park, and Seong-Je Cho. 2022. Deoptfuscator: Defeating Advanced Control-Flow Obfuscation Using Android Runtime (ART). *IEEE Access* 10 (2022), 61426–61440. https://doi.org/10.1109/

ACCESS.2022.3181373

[71] Geunha You, Gyoosik Kim, Jihyeon Park, Seong-Je Cho, and Minkyu Park. 2020. Reversing obfuscated control flow structures in android apps using redex optimizer. In *The 9th International Conference on Smart Media and Applications*. ACM, USA, 272–276.

[72] Geunha You, Gyoosik Kim, Jihyeon Park, Seong-Je Cho, and Minkyu Park. 2021. Reversing Obfuscated Control Flow Structures in Android Apps Using ReDex Optimizer. In *ACM* (Jeju, Republic of Korea) *(SMA 2020)*. Association for Computing Machinery, New York, NY, USA, 272–276. https://doi.org/10.1145/3426020.3426089

[73] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. IEEE, USA, 297–300. https://doi.org/10.1109/BWCCA.2010.85

[74] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 141 (jun 2023), 24 pages. https://doi.org/10.1145/3591255

[75] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 112 (oct 2021), 26 pages. https://doi.org/10.1145/3485489

[76] Yujie Zhao, Zhanyong Tang, Guixin Ye, Xiaoqing Gong, and Dingyi Fang. 2021. Input-output example-guided data deobfuscation on binary. *Security and Communication Networks* 2021 (2021), 1–16.

[77] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications*, Sehun Kim, Moti Yung, and Hyung-Woo Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–75.

Received 2024-04-05; accepted 2024-08-18