Syntia: Breaking State-of-the-Art Binary Code Obfuscation via Program Synthesis

Tim Blazytko, Moritz Contag, Cornelius Aschermann, Thorsten Holz

Ruhr-Universität Bochum, Germany
{firstname.lastname}@rub.de

Abstract

In modern businesses, code obfuscation has become a vital tool to protect, for example, intellectual property against competitors. In general, it attempts to impede program understanding by making the to-be-protected program more complex. In this paper, we will give an overview of contemporary (binary) code obfuscation techniques, including Mixed Boolean-Arithmetic and Virtual Machines. We further note a common theme in state-of-the-art deobfuscation techniques: They mostly use a mixed approach of symbolic execution and taint analysis; two techniques that require precise analysis of the underlying code. Also, these techniques require a non-trivial amount of domain knowledge. This limits the applicability of these techniques and hints at the necessity of finding alternative approaches to tackle the problem of code obfuscation.

Consequently, we introduce program synthesis as a promising technique that is orthogonal to traditional deobfuscation techniques. As program synthesis can synthesize code of arbitrary code complexity, it is only limited by the complexity of the underlying code's semantic and thus overcomes some of the limitations traditional approaches suffer from. Our prototype implementation, Syntia, is guided by Monte Carlo Tree Search (MCTS) and simplifies execution traces by dividing them into distinct trace windows whose semantics are then "learned" by the synthesis. To demonstrate the practical feasibility of our approach, we apply it to modern, commercial protection systems and emerging techniques such as Mixed Boolean-Arithmetic. Syntia automatically learns the semantics of 489 out of 500 random expressions obfuscated via Mixed Boolean-Arithmetic and achieves a success rate of more than 94% when synthesizing the semantics of arithmetic instruction handlers in two state-of-the-art virtualization-based protection systems, Themida and VMProtect. Finally, we discuss the role of program synthesis in the landscape of modern deobfuscation techniques.

1 Introduction

Code obfuscation describes the process of applying an obfuscating transformation to an input program to obtain an obfuscated copy of the program. Said copy should be more complex than the input program such that an analyst cannot easily reason about it. An obfuscating transformation is further desired to be *semantics-preserving*, i. e., it must not change observable program behavior [12]. Code obfuscation can be leveraged in many application domains, for example in software protection solutions to prevent illegal copies, or in malicious software to impede the analysis process. In practice, different kinds of obfuscation techniques are used to hinder the analysis process. Most notably, industry-grade obfuscation solutions are typically based on *Virtual Machine (VM)*-based transformations [39, 56, 58, 59], which are considered one of the strongest obfuscating transformations available [2]. While these protections are not perfect and in fact are broken regularly, attacking them is still a time-consuming task that requires highly specific domain knowledge of the individual Virtual Machine implementation. Consequently, for example, this gives game publishers a head-start in which enough revenue can be generated to stay profitable. On the other hand, obfuscated malware stays under the radar for a longer time, until concrete analysis results can be used to effectively defend against it.

To deal with this problem, prior research has explored many different approaches to enable deobfuscation of obfuscated code. For example, Rolles proposes static analysis to aid in deobfuscation of VM-based obfuscation schemes [45]. However, it incorporates specific implementation details an attacker has to know a priori. Further, static analysis of obfuscated code is notoriously known to be intractable in the general case [12]. Hence, recent deobfuscation proposals have shifted more towards dynamic analysis [13, 62, 63]. Commonly, they produce an execution trace and use techniques such as (dynamic) taint analysis or symbolic execution to distinguish input-dependent instructions. Based on their results, the program code can be reduced to only include relevant, input-dependent instructions. This effectively strips the obfuscation layer. Even though such deobfuscation approaches sound promising, recent work proposes several ways to effectively thwart underlying techniques, such as symbolic execution [2]. For this reason, it suggests itself to explore distinct techniques that may be leveraged for code deobfuscation.

In this paper, we propose an approach orthogonal to prior work on approximating the underlying semantics of obfuscated code. Instead of manually analyzing the instruction handlers used in virtualization-based (VM) obfuscation schemes in a complex and tedious manner [45] or learning merely the bytecode decoding (not the semantics) of these instruction handlers [54], we aim at learning the *semantics* of VM-based instruction handlers in an automated way. Furthermore, our goal is to develop a generic framework that can deal with different use cases. Naturally, this includes constructs close to obfuscation, such as Mixed Boolean-Arithmetic (MBA), different kinds of VM-based obfuscation schemes, or even analysis of code chunks (so called *gadgets*) used in Return-oriented Programming (ROP) exploits.

To this extend, we explore how *program synthesis* can be leveraged to tackle this problem. Broadly speaking, program synthesis describes the task of automatically constructing programs for a given specification. While there exists a variety of program synthesis approaches [21], we focus on SMT-based and stochastic program synthesis in the following, given its proven applicability to problem domains close to trace simplification and deobfuscation. SMT-based program synthesis constructs a loop-free program based on first-order logic constraints whose satisfiability is checked by an SMT solver. For *component-based* synthesis, components are described that build the instruction set of a synthesized program; for instance, components may be bitwise addition or arithmetic shifts. The characteristics of a well-formed program such as the interconnectivity of components are defined and the semantics of the program are described as a logical formula. Then, an SMT solver returns a permutation of the components that forms a well-encoded program following the previously specified intent [22, 24], if it is satisfiable, i. e., such a permutation *does* exist.

Instead of relying on a logical specification of program intent, *oracle-guided* program synthesis uses an input-output (I/O) oracle. Given the outputs of an I/O oracle for arbitrary program inputs, program synthesis learns the oracle's semantics based on a finite set of I/O samples. The oracle is iteratively queried with *distinguishing* inputs that are provided by an SMT solver. Locating distinguishing inputs is the most expensive task in this approach. The resulting synthesized program has the same input-output behavior as the I/O oracle [24]. Contrary to SMT-based approaches that only construct semantically correct programs, stochastic synthesis *approximates* program equivalence and thus remains faster. In addition, it can also find partial correct programs. Program synthesis is modeled as heuristic optimization problem, where the search is guided by a cost function. It determines, for instance, output similarity of the synthesized expression and the I/O oracle for same inputs [51].

As program synthesis is indifferent to code complexity, it can synthesize arbitrarily obfuscated code and is only limited by the underlying code's *semantic* complexity. We demonstrate that a stochastic program synthesis algorithm based on Monte Carlo Tree Search (MCTS) achieves this in a scalable manner. To show feasibility of our approach, we automatically learned the semantics of 489 out of 500 MBA-obfuscated random expressions. Furthermore, we synthesize the semantics of arithmetic instruction handlers in two state-of-the art commercial virtualization-based obfuscators with a success rate of more than 94%. Finally, to show applicability to areas more focused on security aspects, we further automatically learn the semantics of ROP gadgets.

Contributions In summary, we make the following contributions in this paper:

- We introduce a generic approach for trace simplification based on program synthesis to obtain the semantics of different kinds of obfuscated code. We demonstrate how Monte Carlo Tree Search (MCTS) can be utilized in program synthesis to achieve a scalable and generic approach.

- We implement a prototype of our method in a tool called Syntia. Based on I/O samples from assembly code as input, Syntia can apply MCTS-based program synthesis to compute a simplified expression that represents a deobfuscated version of the input.
- We demonstrate that Syntia can be applied in several different application domains such as simplifying MBA expressions by learning their semantics, learning the semantics of arithmetic VM instruction handlers and synthesizing the semantics of ROP gadgets.

2 Technical Background

Before presenting our approach to utilize program synthesis for recovering the semantics of obfuscated code, we first review several concepts and techniques we use throughout the rest of the paper.

2.1 Obfuscation

In the following, we discuss several techniques that qualify as an obfuscating transformation, namely virtualization-based obfuscation, Return-oriented Programming and Mixed Boolean-Arithmetic.

2.1.1 Virtualization-based Obfuscation

Contemporary software protection solutions such as VMProtect [59], Themida [39], and major game copy protections such as SecuROM base their security on the concept of *Virtual Machine-based* obfuscation (also known as *virtualization-based* obfuscation [45]).

Similar to system-level Virtual Machines (VMs) that emulate a whole system platform, process-level VMs emulate a foreign instruction set architecture (ISA). The core idea is to translate parts of a program, e. g., a function f containing intellectual property, from its native architecture—say, Intel x86—into a custom VM-ISA. The obfuscator then embeds both the *bytecode* of the virtualized function (its instructions encoded for the VM-ISA) along with an *interpreter* for the new architecture into the target binary whilst removing the function's original, native code. Every call to f is then replaced with an invocation of the interpreter. This effectively thwarts any naive reverse engineering tool operating on the native instruction set and forces an adversary to analyze the interpreter and re-translate the interpreted bytecode back into native instructions. Commonly, the interpreter is heavily obfuscated itself. As VM-ISAs can be arbitrarily complex and generated uniquely upon protection time, this process is highly time-consuming [45].

Components. The *(VM) context* holds internal variables of the VM-ISA such as general-purpose registers or the virtual instruction pointer. It is initialized by

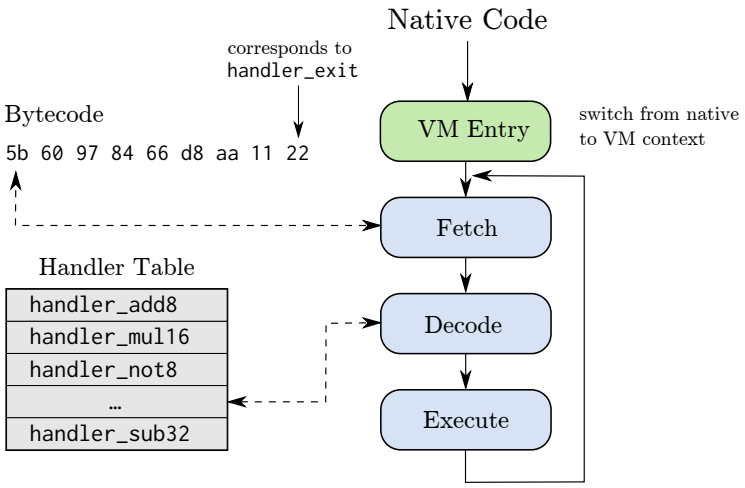


Figure 1: The Fetch–Decode–Execute cycle of a Virtual Machine. Native code calls into the VM, upon which startup code is executed (VM entry). It performs the context switch from native to VM context. Then, the next instruction is fetched from the bytecode stream, mapped to the corresponding handler using the handler table (decoding) and, finally, the handler is executed. The process repeats for subsequent VM instructions in the bytecode until the exit handler is executed, which returns back to native code.

sequence called *VM entry*, which handles the context switch from native code to bytecode.

After initialization, the *VM dispatcher* fetches and decodes the next instruction and invokes the corresponding *handler* function by looking it up in a global *handler table* (depicted in Figure 1). The latter maps indices, obtained from the instruction's bytecode in the decoding step, to handlers addresses. In its most simple implementation, all handler functions return to a central dispatching loop which then dispatches the next handler. Eventually, execution flow reaches a designated handler, *VM exit*, which performs the context switch back to the native processor context and transfers control back to native code.

Custom ISA. The design of the target VM-ISA is entirely up to the VM designer. Still, to maximize the amount of handlers an analyst has to reverse engineer, VMs often opt for reduced complexity for the individual handlers, akin to the RISC design principle. To exemplify, consider the following Intel x86 code:

```
1 mov eax, dword ptr [0x401000 + ebx * 4]
2 pop dword ptr [eax]
```

This might get translated into VM-ISA as follows:

```
 1 vm_mov  T0, vm_context.real_ebx
 2 vm_mov  T1, 4
 3 vm_mul  T2, T0, T1
 4 vm_mov  T3, 0x401000
 5 vm_add  T4, T2, T3
 6 vm_load T5, dword(T4)
 7 vm_mov  vm_context.real_eax, T5
 8 vm_mov  T6, T5
 9 vm_mov  T7, vm_context.real_esp
10 vm_add  T8, T7, T1
11 vm_mov  vm_context.real_esp, T8
12 vm_load T9, dword(T7)
13 vm_store dword(T6), T9
```

It favors many small, simple handlers over fewer more complicated ones.

Bytecode Blinding. In order to prevent global analysis of instructions, the bytecode *bc* of each VM instruction is blinded based on its instruction type, i. e., its corresponding handler h, at protection time. Likewise, each handler *unblinds* the bytecode before decoding its operands: $(bc, vm\_key) \leftarrow \mathsf{unblind}_h(blinded\_bc, vm\_key)$.

The routine is parameterized for each handler h and updates a global key register in the VM context. Consequently, instruction decoding can be *flow-sensitive*: An adversary is unable to patch a single VM instruction without re-blinding all subsequent instructions. This, in turn, requires her to extract the unblinding routines from every handler involved. The individual unblinding routines commonly consist of a combination of arithmetic and logical operations.

Handler Duplication. In order to easily increase analysis complexity, common VMs *duplicate* handlers such that the same virtual instruction can be dispatched by multiple handlers. In presence of bytecode blinding, these handlers' semantics only differ in the way they unblind the bytecode, but perform the same operation on the VM context.

Architectures. In his paper about interpretation techniques, Klint denotes the aforementioned concept using a central decoding loop as the "classical interpretation method" [28]. An alternative is proposed by Bell with *Threaded Code* (TC) [4]: He suggests inlining the dispatcher routine into the individual handler functions such that handlers execute in a chained manner, instead of returning to a central dispatcher. Nevertheless, the dispatcher still indexes a global handler table. In the context of interpreters for software protection, this prevents the attacker from readily investigating each atomic step of the VM. Further, since the dispatcher still indexes the global handler table, the adversary immediately obtains the addresses of all handler functions.

In Klint's paper, however, he describes an extension of TC, *Direct Threaded Code* (DTC). As in the TC approach, the dispatcher is appended to each handler. The handler table, though, is inlined into the *bytecode* of the instruction. Each

instruction now directly specifies the address of its corresponding handler. This way, in presence of bytecode blinding, not all handler addresses are exposed immediately, but only those used on a certain path in the bytecode.

Attacks. Several academic works have been published that propose novel attacks on virtualization-based obfuscators [13, 45]. Section 6.2 discusses and classifies them. In addition, it draws a comparison to our approach.

2.1.2 Return-oriented Programming

In *Return-oriented Programming* (ROP) [30, 53], shellcode is expressed as a so-called ROP chain, a list of references to *gadgets* and parameters for those. In the preliminary step of an attack, the adversary makes `esp` point to the start of the chain, effectively triggering the chain upon function return. Gadgets are small, general instruction sequences ending on a `ret` instruction; other flavors propose equivalent instructions. Concrete values are taken from the ROP chain on the stack. As an example, consider the gadget `pop eax; ret`: It takes the value on top of the stack, places it in `eax` and, using the `ret` instruction, dispatches the next gadget in the chain. By placing an arbitrary immediate value `imm32` next to this gadget's address in the chain, an attacker effectively encodes the instruction `mov eax, imm32` in her ROP shellcode. Depending on the gadget space available to the attacker, this technique allows for arbitrary computations [40, 52].

Automated analysis of ROP exploits is a desirable goal. However, its unique structure poses various challenges compared to traditional shellcode detection. In their paper, Graziano et al. outline them and propose an analysis framework for code-reuse attacks [19]. Amongst others, they mention challenges such as verbosity of the gadgets, stack-based chaining, lack of immediates, and the distinction of function calls and regular control flow. Further, they stress how an accurate emulation of gadgets is important for addressing these challenges. Considering the aforementioned challenges, at its core, Return-oriented Programming can be seen as an albeit weaker flavor of obfuscated code. In particular, the chained invocation of gadgets is reminiscent of handlers in VM-based obfuscation schemes following the threaded code principle.

In addition to its application to exploitation, ROP has seen other fields of applications such as rootkit development [60], software watermarking [34], steganography [33], and code integrity verification [1], which reinforces the importance of automatic ROP chain analysis.

2.1.3 Mixed Boolean-Arithmetic

Zhou et al. propose transformations over Boolean-arithmetic algebras to hide constants by turning them into more complex, but semantically equivalent expressions, so called MBA expressions [14, 64]. In Section 6.1, we provide details on their proposal of MBA expressions and show how our approach is still able to simplify them.

2.2 Trace Simplification

Due to the complexity of static analysis of obfuscated code, many deobfuscation approaches proposed recently make use of dynamic analysis [13, 19, 19, 54, 63]. Notably, they operate on *execution traces* that record instruction addresses and accompanying metadata, e. g., register content, along a concrete execution path of a program. Subsequently, trace simplification is performed to strip the obfuscation layer and simplify the underlying code. Depending on the approach, multiple traces are used for simplification or one single trace is reduced independently.

Coogan et al. [13] propose *value-based dependence* analysis of a trace in order to track the flow of values into system calls using an equational reasoning system. This allows them to reduce the trace to those instructions *relevant* to the previously mentioned value flow.

Graziano et al. [19] mainly apply standard compiler transformations such as dead code elimination or arithmetic simplifications to reduce the trace.

Yadegari et al. [63] use bit-level taint analysis to identify instructions relevant to the computation of outputs. For subsequent simplification, they introduce the notion of *quasi-invariant* locations with respect to an execution. These are locations that hold the same value at every use in the trace and can be considered constants when performing constant propagation. Similarly, they use several other compiler optimizations and adapt them to make use of information about *quasi-invariance* to prevent over-simplification.

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a stochastic, best-first tree search algorithm that directs the search towards an optimal decision, without requiring much domain knowledge. The algorithm builds a search tree through reinforcement learning by performing random simulations that estimate the quality of a node [5]. Hence, the tree grows asymmetrically. MCTS has had significant impact in artificial intelligence for computer games [16, 35, 50, 57], especially in the context of Computer Go [17, 55].

In an MCTS tree, each node represents a game state; a directed link from a parent node to its child node represents a move in the game's domain. The core algorithm iteratively builds the decision tree in four main steps that are also illustrated in Figure 2: (1) The *selection* step starts at the root node and successively selects the most-promising child node, until an *expandable* leaf (i. e., a non-terminal node that has unvisited children) is reached. (2) Following, one or more unvisited child nodes are added to the tree in the *expansion* step. (3) In the *simulation* step, node rewards are determined for the new nodes through random playouts. For this, consecutive game states are randomly derived until a terminal state (i. e., the end of the game) is reached; the game's outcome is represented by a reward. (4) Finally, the node rewards are propagated backwards through the selected nodes to the root in the *backpropagation* step. The algorithm terminates

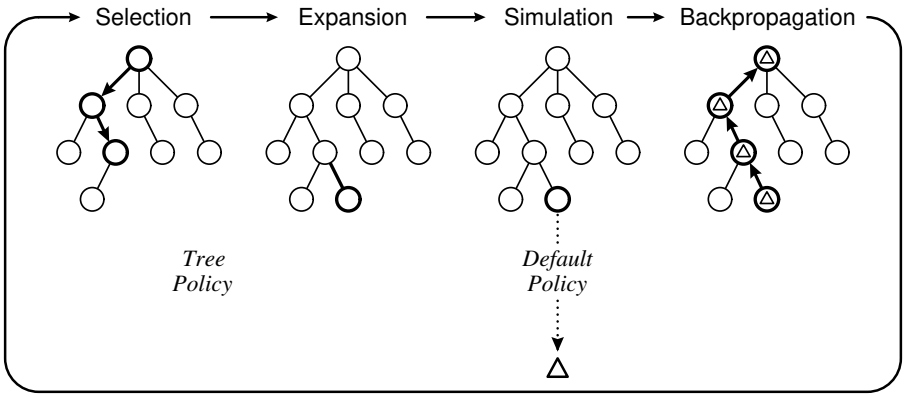


Figure 2: Illustration of a single MCTS round (taken from Browne et al. [5]).

if either a specified time/iteration limit is reached or an optimal solution is found [5, 8].

Selecting the most-promising child node can be treated as a so called *multi-armed bandit problem*, in which a gambler tries to maximize the sum of rewards by choosing one out of many slot machines with an unknown probability distribution. Applied to MCTS, the *Upper Confidence Bound for Trees* (UCT) [5, 17, 29] provides a good balance between exploration and exploitation. It is obtained by

$$\overline{X}_j + C\sqrt{\frac{\ln n}{n_j}}, \tag{1}$$

where $\overline{X}_j$ represents the average reward of the child node j, n the current node's number of visits, n_j the visits of the child node and C the exploration constant. The average reward is referred to as *exploitation parameter*: if C is decreased, the search is directed towards nodes with a higher reward. Increasing C, instead, leads to an intensified exploration of nodes with few simulations.

2.4 Simulated Annealing

Simulated Annealing is a stochastic search algorithm that has been used to effectively solve NP-hard combinatorial problems [27]. The main idea of Simulated Annealing is to approximate a global optimum by iteratively improving an initial candidate and exploring the local neighborhood. To avoid a convergence to local optima, the search is guided by a falling temperature T that decreases the probability of accepting worse candidates over time [25]; in the following, we assume that a falling temperature depends on a decreasing loop counter.

Figure 3 illustrates this process on the example of finding the darkest area in a given map. Starting in an initial state (s_0), the algorithm always accepts a candidate that has a better score than the current one (green arrows). If the score is worse, we accept the worse candidate with some probability (the red arrow from s_2 to s_3) that depends on the temperature (loop counter) and

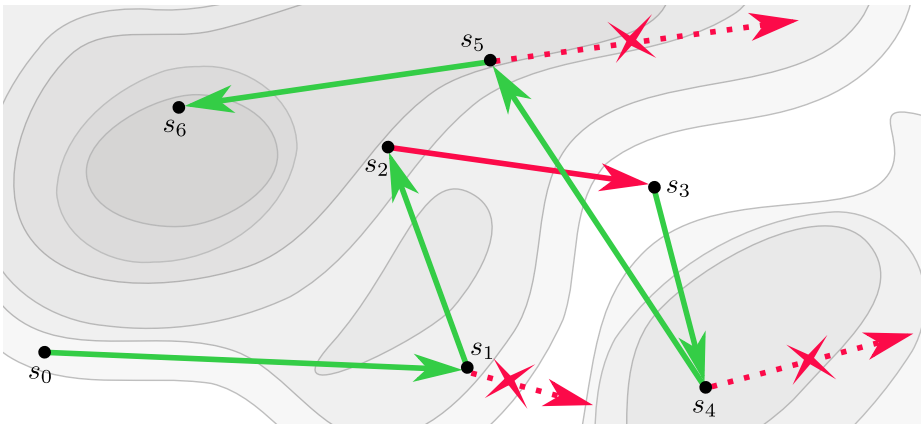


Figure 3: Simulated Annealing approximates a global optimum (the darkest area in the map).

how much worse the candidate is. The higher the temperature, the more likely the algorithm accepts a significantly worse candidate solution. Otherwise, the candidate is discarded (e. g., the crossed out red arrow at s_4); in this case, we pick another one in the local neighborhood. This allows the algorithm to escape from local optima while the temperature is high; for low temperatures (loop counters closer to 0), it mainly accepts better candidate solutions. The algorithm terminates after a specified number of iterations.

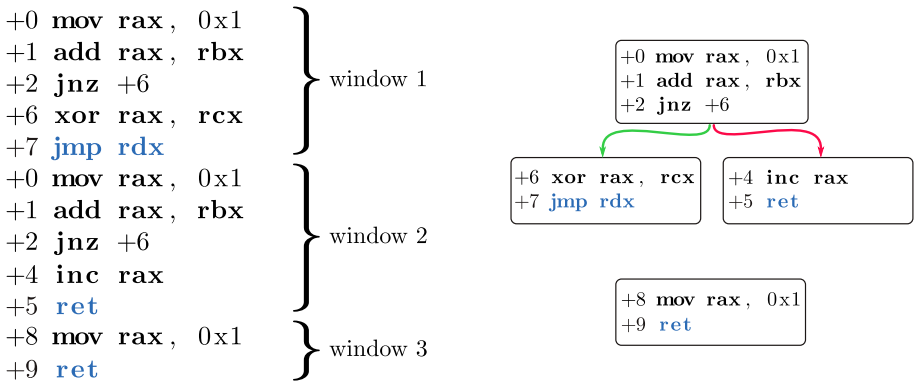
3 Approach

Given an instruction trace, we dissect the instruction trace into *trace windows* (i. e., subtraces) and aim at learning their high-level semantics which can be used later on for further analysis. In the following, we describe our approach which is divided into three distinct parts:

1. *Trace Dissection.* The instruction trace is partitioned into unique sequences of assembly instructions in a (semi-)automated manner.
2. *Random Sampling.* We derive random input-output pairs for each trace window. These pairs describe the trace window's semantics.
3. *Program Synthesis.* Expressions that map all provided inputs to their corresponding outputs are synthesized.

3.1 Trace Dissection

The choice of trace window boundaries highly impacts later analysis stages. Most notably, it affects synthesis results: if a trace window ends at an *intermediary* computation step, the synthesized formula is not necessarily succinct or meaningful at all, as it includes spurious semantics.



(a) Instruction Trace (b) Trace Windows (c) Reconstructed CFG

Figure 4: Dissecting a given trace (a) into several trace windows (b). The trace windows can be used to reconstruct a (possibly disconnected) control-flow graph (c).

Yet, we note how trace dissection of ROP chains and VM handlers lends itself to a simple heuristic. Namely, we split traces at indirect branches. In the ROP case, this describes the transition between two gadgets (commonly, on a `ret` instruction), whereas for VM handlers it distinguishes the invocation of the next handler (cf. Section 6.2). Figure 4 illustrates the approach. Given concrete trace window boundaries, we can reconstruct a control-flow graph consisting of multiple connected components. A *trace window* then describes a particular path through a connected component.

3.2 Random Sampling

The goal of random sampling is to derive input-output relations that describe the semantics of a trace window. This happens in two steps: First, we determine the inputs and outputs of the trace window. Then, we replace the inputs with random values and obverse the outputs.

Generally speaking, we consider register and memory reads as inputs and register and memory writes as outputs. For inputs, we apply a read-before-write principle: inputs are only registers/memory locations that are read before they have been written; for outputs, we consider the last writes of a register/memory location as output.

```
1 mov rax, [rbp + 0x8]
2 add rax, rcx
3 mov [rbp + 0x8], rax
4 add [rbp + 0x8], rdx
```

Following this principle, the code above has three inputs and two outputs: The inputs are the memory read M_0 in line 1, `rcx` (line 2) and `rdx` (line 4). The two outputs are o_0 (line 2) and o_1 (line 4).

In the next step, we generate random values and obverse the I/O relationship. For instance, we obtain the outputs $(7, 14)$ for the input tuple $(2, 5, 7)$; for the inputs $(1, 7, 10)$, we obtain $(8, 18)$.

By default, we use register locations as well as memory locations as inputs and outputs. However, we support the option to reduce the inputs and outputs to either register or memory locations. For instance, if we know that registers are only used for intermediate results, we may ignore them since it reduces the complexity for the synthesis.

3.3 Synthesis

This section demonstrates how we synthesize the semantics of assembly code; we discuss the inner workings of our synthesis approach in the next section.

After we obtained the I/O samples, we combine the different samples and synthesize each output separately. These synthesis instances are mutually independent and can be completely parallelized.

To exemplify, for the I/O pairs above, we search an expression that transforms $(2, 5, 7)$ to 7 and $(1, 7, 10)$ to 8 for o_0; for o_1, the expression has to map $(2, 5, 7)$ to 14 and $(1, 7, 10)$ to 18. Then, the synthesizer finds $o_0 = M_0 + \texttt{rcx}$ and $o_1 = M_0 + \texttt{rcx} + \texttt{rdx}$.

4 Program Synthesis

In the last section, we demonstrated how we obtain I/O samples from assembly code and apply program synthesis to that context. This section describes our algorithm in detail; we show how we find an expression that maps all inputs to their corresponding outputs for all observed samples. We use Monte Carlo Tree Search, since it has been proven to be very effective when working on infinite decision trees without requiring much domain knowledge.

We consider program synthesis as a single-player game whose purpose is to synthesize an expression whose input-output behavior is as close as possible to given I/O samples. In essence, we define a context-free grammar that consists of terminal and non-terminal symbols. (Partially) derived words of the grammar are *game states*; the grammar's production rules represent the *moves* of the game. *Terminal nodes* are expressions that contain only terminal symbols; these are *end states* of the game.

Given a maximum number of iterations and I/O samples, we iteratively apply the four MCTS steps (cf. Section 2.3), until we find a solution or we reach the timeout. Starting with a non-terminal expression as *root node*, we *select* the most-promising expandable node. A node is *expandable*, if there still exist production rules that have not been applied to this node. We choose a production rule randomly and expand the selected node. To evaluate the quality of the new node, we perform a *random playout*: First, we randomly derive a terminal expression by successively applying random production rules. Then, we evaluate the expressions based on the inputs from the I/O pairs and compare

the output similarity. The similarity score is the node *reward*. A reward of 1 ends the synthesis, since the input-output behavior is the same for the provided samples. Finally, we *propagate* the reward back to the root.

In the following, we give details on node selection, our grammar, random playouts and backpropagation. Finally, we discuss the algorithm configuration and parameter tuning. To demonstrate the different steps of our approach, we use the following running example throughout this section:

Example 1 (I/O relationship)**.** *Working with bit-vectors of size* 3 *(i. e., modulo* 2^3*), we observe for an expression with two inputs and one output the I/O relations:* $(2, 2) \rightarrow 4$ *and* $(4, 5) \rightarrow 1$*. A synthesized expression that maps the inputs to the corresponding outputs is* $f(a, b) = a + b$*.*

4.1 Node Selection

Since we have an infinite search space for program synthesis, node selection must be a trade-off between exploration and exploitation. The algorithm has to explore different nodes such that several promising and non-promising candidates are known. On the other hand, it has to follow more promising candidates to find deeper expressions. As described in Section 2.3, the UCT (cf. Equation 1) provides a good balance between exploitation and exploration for many MCTS applications.

However, we observed that it does not work for our use case: if we set the exploration constant C to a higher value (focus on exploration), it does not find deeper expressions; if we set C to a lower value, MCTS gets lost in deep expressions. To solve this problem, we use an adaption of UCT that is known as *Simulated Annealing UCT* (SA-UCT) [48]. The main idea of SA-UCT is to use the characteristics of Simulated Annealing (cf. Section 2.4) and apply it to UCT. SA-UCT is obtained by replacing the exploration constant C by a variable T with

$$T = C\frac{N - i}{N}, \tag{2}$$

where N is the maximum number of MCTS iterations and i the current iteration. Then, SA-UCT is defined as

$$\overline{X}_j + T\sqrt{\frac{\ln n}{n_j}}. \tag{3}$$

T decreases over time, since $\frac{N-i}{N}$ converges to 0 for increasing values of i. As a result, MCTS places the emphasis on exploration in the beginning; the more T decreases, the more the focus shifts to exploitation.

4.2 Grammar

Game states are represented by sentential forms of a context-free grammar that describes valid expressions of our high-level abstraction. We introduce a terminal

symbol for each input (which corresponds to a variable that stores this input) and each valid operator (e. g., addition or multiplication). For every data type that can be computed we introduce one non-terminal symbol (in our running example, we only use a single non-terminal value U that represents an unsigned integer). The production rules describe how we can derive expressions in our high-level description. Since the sentential forms represent partial expressions, we will use the term expression to denote the (partial) expression that is represented by a given sentential form. Sentences of the grammar are final states in the game since they do not allow any further moves (derivations). They represent expressions that can be evaluated. We represent expressions in *Reverse Polish Notation* (RPN).

Example 2. *The grammar in our previous example has two input symbols* $V = \{a, b\}$*, since each I/O sample has two inputs. If the grammar supports addition and multiplication* $O = \{+, *\}$*, there are four production rules:* $R = \{U \to U\ U\ +\ |\ U\ U\ *\ |\ a\ |\ b\}$*. An unsigned integer expression* U *can be mapped to an addition or multiplication of two such expressions or a variable. The final grammar is* $(\{U\}, \Sigma = V \cup O, R, U)$*.*

Synthesis Grammar. Our grammar is designed to synthesize expressions that represent the semantics of bit-vector arithmetic, especially for the x86 architecture. For every data type (U_8, U_{16}, U_{32} and U_{64}), we define the set of operations as $O = \{+, -, *, /_s, /, \%_s, \%, \wedge, \vee, \oplus, \ll, \gg, \gg_a, -_1, \neg,$ `sign_ext`, `zero_ext`, `extract`, $++$, $1\}$, where the operations are binary addition, subtraction, multiplication, signed/unsigned division, signed/unsigned remainder, bitwise and/or/xor, logical left shift, logical/arithmetic right shift as well as unary minus and complement. The unary operations `sign_ext` and `zero_ext` extend smaller data types to signed/unsigned larger data types. Conversely, the unary operator `extract` transforms larger data types into smaller data types by extracting the respective least significant bits. Since the x86 architecture allows register concatenation (e. g., for division), we employ the binary operator $++$ to concatenate two expressions of the same data type. Finally, to synthesize expressions such as increment and decrement, we use the constant 1 as niladic operator. The input set V consists of $|V| = n$ variables, where n represents the number of inputs.

Tree Structure. The sentential form U is the root node of the MCTS tree. Its child nodes are other expressions that are produced by applying the production rules to a single non-terminal symbol of the parent. The expression depth (referred to as *layer*) is equivalent to the number of derivation steps, as depicted in Figure 5.

Example 3. *The root node* U *is an expression of layer* 0*. Its children are* a*,* b*,* $U\ U\ +$*, and* $U\ U\ *$*, where* a *and* b *are terminal expressions of layer* 1*. Assuming that the right-most* U *in an expression is replaced, the children of* $U\ U\ +$ *are* $U\ b\ +$*,* $U\ a\ +$*,* $U\ U\ U\ +\ +$*, and* $U\ U\ U\ *\ +$*. To obtain the layer* 3 *expression* $b\ a\ +$*, the following derivation steps are applied:* $U \Rightarrow U\ U\ + \Rightarrow U\ a\ + \Rightarrow b\ a\ +$*.*

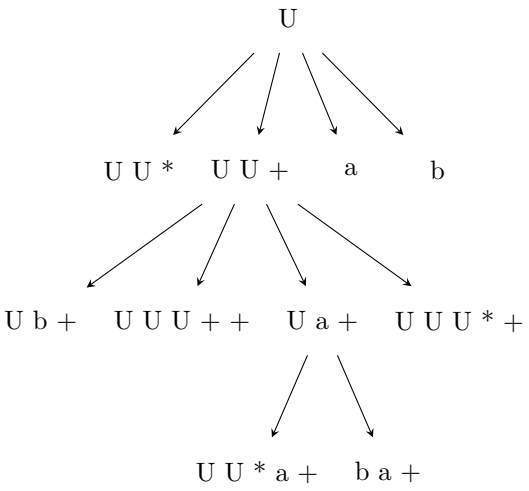


Figure 5: An MCTS tree for program synthesis that grows towards the most-promising node $b\ a\ +$, the right-most leaf in layer 3.

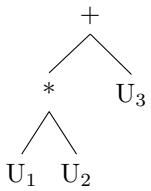


Figure 6: The left-most U in $U_3\ U_2\ U_1\ *\ +$ is the top-most-right-most non-terminal in the abstract syntax tree. (The indices are provided for illustrative purposes only.)

To direct the search towards outer expressions, we replace the *top-most-right-most* non-terminal. If we, instead, substitute always the *right-most* non-terminal only, then the search would be guided towards most-promising subexpressions. If the expression is too nested, the synthesizer would find the partial subexpression but not the whole expression. The top-most-right-most derivation is illustrated in Figure 6, which shows the abstract syntax tree (AST) of an expression.

Example 4. *The expression $(U + (U * U))$ is represented as $U\ U\ U\ *\ +$. If we successively replace the right-most U, the algorithm is unlikely to find expressions such as $((a + b) + (b * (b * a)))$, since it is directed into the subexpression with the multiplication. Instead, replacing the top-most-right-most non-terminal directs the search to the top-most addition and then explores the subexpressions.*

4.3 Random Playout

One of the key concepts of MCTS are random playouts. They are used to determine the outcome of a node; this outcome is represented by a reward. In the first step, we randomly apply production rules to the current node, until we obtain a terminal expression. To avoid infinite derivations, we set a maximum playout depth. This maximum playout depth defines how often a non-terminal symbol can be mapped to rules that contain non-terminal symbols; at the latest we reached the maximum, we map non-terminals only to terminal expressions. This happens in a top-most-right-most manner. Afterwards, we evaluate the expression for all inputs from the I/O samples.

Example 5. *Assuming a maximum playout depth of* 2 *and the expression* $U\ U\ *$*, the first top-most-right-most* U *is randomly substituted with* $U\ U\ *$*, the second one with* $U\ U\ +$*. After that, the remaining non-terminal symbols are randomly replaced with variables:* $U\ U\ * \Rightarrow U\ U\ U\ *\ * \Rightarrow U\ U\ +\ U\ U\ *\ * \Rightarrow \cdots \Rightarrow a\ a\ +\ b\ a\ *\ *$*. A random playout for* $U\ U\ +$ *is* $a\ b\ b\ +\ +$*.*

For the I/O sample $(2, 2) \to 4$*, we evaluate* $g(2, 2) = 0$ *for* $g(a, b) = ((a + a) * (b * a)) \mod (2^8)$ *and* $h(2, 2) = 6$ *for* $h(a, b) = (a + (b + b)) \mod 2^8$*.*

We set terminal nodes to inactive after their evaluation, since they already are end states of the game; there is no possibility to improve the node's reward by random playouts. As a result, MCTS will not take these nodes into account in further iterations. The node's reward is the similarity of the evaluated expressions and the outputs from the I/O samples. We describe in the following section how to measure the similarity to the outputs.

4.4 Measuring Similarity of Outputs

To measure the similarity of two outputs, we compare values with different metrics: arithmetic distance, Hamming distance, count leading zeros, count trailing zeros, count leading ones and count trailing ones. While the numeric distance is a reliable metric for arithmetic operations, it does not work well with overflows and bitwise operations (e. g., xor and shifts). In turn, the Hamming distance addresses these operations since it states in how many bits two values differ. Finally, the leading/trailing zeros/ones are strong indicators that two values are in the same range. We scale each result between a value of 0 and 1. Since the different metrics compensate each other, we set the total similarity reward to the average reward of all metrics.

Example 6. *Considering I/O pair* $(2, 2) \to 4$*, the output similarities for* g *and* h *(as defined in Example 5) are* $\text{similarity}(4, 0)$ *and* $\text{similarity}(4, 6)$*. Limiting to the metrics of Hamming distance and count leading zeros (*`clz`*), we obtain* $\text{hamming}(4, 0) = \text{hamming}(4, 6) = 0.67$*,* $\text{clz}(4, 0) = 0$ *and* $\text{clz}(4, 6) = 1.0$*. Therefore, the average similarities are* $\text{similarity}(4, 0) = 0.335$ *and* $\text{similarity}(4, 6) = 0.835$*. Related to the random playouts, the evaluated node* $U\ U\ +$ *has a higher reward than* $U\ U\ *$*.*

During a random playout, we calculate the similarity for all I/O samples. The final node reward is the average score of all similarity rewards. A reward of 1 finishes program synthesis, since the evaluated expression produces exactly the outputs from the I/O samples.

4.5 Backpropagation

After obtaining a score by random playout, we do the following for the selected node and all its parents, up to the root: (1) We update the node's average reward. This reward is averaged based on the node's and its successors' total number of random playouts. (2) If the node is fully expanded and its children are all inactive, we set the node to inactive. (3) Finally, we set the current node to its parent node.

4.6 Expression Simplification

Since MCTS performs a stochastic search, synthesized expressions are not necessary in their shortest form. Therefore, we apply some basic standard expression simplification rules. For example, if the synthesizer constructs integer values as $((1 \ll 1) \ll (1 + (1 \ll 1)))$, we can reduce them to the value 16.

5 Implementation

We implemented a prototype implementation of our approach in our tool *Syntia*, which is written in Python. For trace generation and random sampling, we use the *Unicorn Engine* [44], a CPU emulator framework. To analyze assembly code (e. g., trace dissection), we utilize the disassembler framework *Capstone* [43]. Furthermore, we use the SMT solver *Z3* [36] for expression simplification.

Initially, Syntia expects a memory dump, a start and an end address as input. Then, it emulates the program and outputs the instruction trace. Then, the user has the opportunity to define its own rules for trace dissection; otherwise, Syntia dissects the trace at indirect control transfers. Additionally, the user has to decide if register and/or memory locations are used as inputs/outputs and how many I/O pairs shall be sampled. Syntia traces register and memory modifications in each trace window, derives the inputs and outputs and generates I/O pairs by random sampling. The last step can be parallelized for each trace window. Finally, the user defines the synthesis parameters. Syntia creates a synthesis tasks for each (trace window, output) pair. The synthesis tasks are performed in parallel. The synthesis results are simplified by Z3's term-rewriting engine.

6 Experimental Evaluation

In the following, we evaluate our approach in three areas of application. The experiments have been evaluated on a machine with two Intel Xeon E5-2667

CPUs (in total, 12 cores and 24 threads) and 96 GiB of memory. However, we never have used more than 32 GiB of memory even though parallel I/O sampling for many trace windows can be memory intensive; synthesis itself never used more than 6 GiB of memory.

6.1 Mixed Boolean-Arithmetic

Mixed Boolean-Arithmetic (MBA) describes—informally speaking—the mixture of arithmetic and logical bit-vector operations to build semantically equivalent expressions that are harder to understand. In the following, we first give a more formal definition of MBA-obfuscation. Afterwards, we describe the MBA-based obfuscation in the Tigress Obfuscator and in a real-world DRM system. Finally, we deobfuscate their obfuscated expressions via program synthesis.

Zhou et al. proposed the concept of *MBA expressions* [64]. By transforming simpler expressions and constants into MBA expressions over Boolean-arithmetic algebras, they can generate semantically-equivalent, but much more complex code which is arguably hard to reverse engineer. Effectively, this obfuscating transformation allows them to hide formulas and constants in plain code. In their paper, they define a Boolean-arithmetic algebra as follows:

Definition 1 (Boolean-arithmetic algebra [64])**.** *With n a positive integer and $\mathsf{B} = \{0, 1\}$, the algebraic system $(\mathsf{B}^n, \wedge, \vee, \oplus, \neg, \leq, \geq, >, <, \leq^s, \geq^s, >^s, <^s, \neq, =, \gg^s, \gg, \ll, +, -, \cdot)$, where $\ll, \gg$ denote left and right shifts, $\cdot$ (or juxtaposition) denotes multiply, and signed compares and arithmetic right shift are indicated by ^{s}, is a* Boolean-arithmetic algebra (BA-algebra), $\mathsf{BA}[n]$*. n is the* dimension *of the algebra.*

Specifically, they highlight how $\mathsf{BA}[n]$ includes, amongst others, the Boolean algebra $(\mathsf{B}^n, \wedge, \vee, \neg)$ as well as the integer modular ring $\mathbb{Z}/(2^n)$. As a consequence, Mixed Boolean-Arithmetic (MBA) expressions over B^n are hard to simplify in practice. In general, we note that reducing a complex expression to an equivalent, but simpler one by, e. g., removing redundancies, is considered NP-hard [31].

Zhou et al. represent MBA expressions as polynomials over $\mathsf{BA}[n]$. While polynomial MBA expressions are conceptually not restricted in terms of complexity, Zhou et al. define *linear MBA expressions* as those polynomials with degree 1. In particular, $f(x, y) = x - (x \oplus y) - 2(x \vee y) + 12564$ is a linear MBA expression, whereas $f(x, y) = x + 9(x \vee y)yx^3$ is not.

Implementation in Tigress. In practice, MBA expressions are used in the Tigress C Diversifier/Obfuscator by Collberg et al. [9] which uses the technique to encode integer variables and expressions in which they are used [11]. Further, Tigress also supports common arithmetic encodings to increase an expression's complexity, albeit not based on MBAs [10].

For example, the rather simple expression $x + y + z$ is transformed into the layer 23 expression $(((x \oplus y) + ((x \wedge y) \ll 1)) \vee z) + (((x \oplus y) + ((x \wedge y) \ll 1)) \wedge z)$ using its arithmetic encoding option. In a second transformation step, Tigress

Table 1: Trace window statistics and synthesis performance for Tigress (MBA), VMProtect (VMP), Themida (flavor Tiger White, TM), and ROP gadgets.

	MBA	**VMP**	**TM**	**ROP**
#trace windows	500	12,577	2,448	78
#unique windows	500	449	106	78
#instructions per window	116	49	258	3
#inputs per window	5	2	15	3
#outputs per window	1	2	10	2
#synthesis tasks	500	1,123	1,092	178
I/O sampling time (s)	110	118	60	17
overall synthesis time (s)	2,020	4,160	9,946	829
synthesis time per task (s)	4.0	3.7	9.1	4.7

encodes it into a linear MBA expression of layer 383 (omitted due to complexity). Such expressions are hard to simplify symbolically.

Evaluation Results for Tigress. We evaluated our approach to simplify MBA expressions using Syntia. As a testbed, we built a C program which calls 500 *randomly* generated functions. Each of these random functions takes 5 input variables and returns an expression of layer 3 to 5. Then, we applied the arithmetic encoding provided by Tigress, followed by the linear MBA encoding. The resulting program contained expressions of up to $2,821$ layers, the average layer being 156. The arithmetic encoding is applied to highlight that our approach is invariant to the code's increased *symbolic* complexity and is only concerned with semantical complexity.

Based on a concrete execution trace it can be observed that the 500 functions use, on average, 5 memory inputs (as parameters are passed on the stack) and one register output (the register containing the return value). Table 1 shows statistics for the analysis run using the configuration vector $(1.5, 50000, 50, 0)$. The first two components indicate a strong focus on *exploration* in favor of exploitation; due to the small number of synthesis tasks, we used 50 I/O samples to obtain more precise results.

The sampling phases completed in less than two minutes. Overall, the 500 synthesis tasks were finished in about 34 minutes, i. e., in 4.0 seconds per expression. We were able to synthesize 448 out of 500 expressions (89.6%). The remaining expressions are not found due to the probabilistic nature of our algorithm; after 4 subsequent runs, we synthesized 489 expressions (97.8%) in total.

To get a better feeling for this probabilistic behavior, we compared the cumulative numbers of synthesized MBA expressions for 10 subsequent runs. Figure 7 shows the results averaged over 15 separate experiments. On average, the first run synthesizes 89.6% (448 expressions) of the 500 expressions. A second run yields 22 new expressions (94.0%), while a third run reveals 10 more

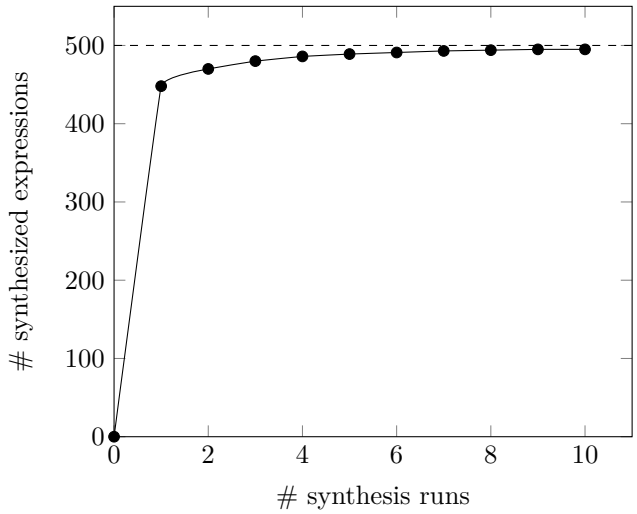


Figure 7: Subsequent synthesis runs increase the number of synthesized MBA expressions. Each point represents the average cumulative number of synthesized expressions from 15 separate experiments.

expressions (96.0%). While converging to 500, the number of newly synthesized expressions decreases in subsequent runs. Comparing the fifth and the eighth run, we only found 5 new expressions (from 489 to 494). After the ninth run, Syntia synthesized 495 (99.0%) of the MBA expressions.

MBA-based Obfuscation in a Commercial DRM System. In their talk at REcon 2014, Mougey and Gabriel [37] presented their results after reverse engineering a real-world *Digital Rights Management (DRM)* system. They found a complex MBA-obfuscated expression that is shown in Figure 8. This expression is equivalent to $x \oplus 92$, where x is a byte value (data type U_8).

To synthesize this expression, Syntia's grammar must include some concept of constants, since 92 has an enormous impact on the output. For this, we add add all 256 possible byte values as terminal symbols to the grammar. Then, Syntia synthesizes the expression within 4 seconds.

This approach cannot be adapted for larger data types than U_8, since the set of production rules becomes too large. Instead, constants can be synthesized over several layers; the grammar can be extended by a non-terminal symbol for constants and a set of rules that assembles constants gradually (e.g., one byte per layer).

6.2 VM Instruction Handler

As introduced in Section 2.1.1, an instruction handler of a Virtual Machine implements the effects of an atomic instruction according to the custom VM-ISA. It operates on the VM context and can perform arbitrarily complex tasks. As

$$
\begin{aligned}
a &:= 229x + 247 \\
b &:= 237a + 214 + ((38a + 85) \wedge 254) \\
c &:= (b + ((-2b + 255) \wedge 254)) \cdot 3 + 77 \\
d &:= ((86c + 36) \wedge 70) \cdot 75 + 231c + 118 \\
e &:= ((58d + 175) \wedge 244) + 99d + 46 \\
f &:= (e \wedge 148) \\
g &:= (f - (e \wedge 255) + f) \cdot 103 + 13 \\
result &:= (237 \cdot (45g + (174g \vee 34) \cdot 229 + 194 - 247) \wedge 255)
\end{aligned}
$$

Figure 8: MBA-based obfuscation in a real-wold DRM system [37] (adapted from Eyrolles et al. [15]). The expression is equivalent to $x \oplus 92$ for $x \in \{0, 1, \ldots, 2^8 - 1\}$.

handlers are heavily obfuscated, manual analysis of a handler's semantics is a time-consuming task.

Attacking VMs. When faced with virtualization-based obfuscations, an attacker typically has two options. For one, she can analyze the interpreter and, for each handler, extract all information required to *re-translate* the bytecode back to native instruction. Especially in face of handler duplication and bytecode blinding, this requires her to precisely capture all effects produced by the handlers. This includes both the high-level semantics with regard to input and output variables *as well as* the individual unblinding routines. In his paper, Rolles discusses how this type of attack requires complete understanding of the VM and therefore has to be repeated for each virtualization obfuscator [45]. Thus, we note that this attack does not lend itself easily to full automation. Another approach is to perform analyses on the *bytecode level*. The idea is that while an attacker cannot learn the full semantics of the original code, the analysis of the interaction of handlers itself reveals enough information about the underlying code. This allows the attacker to skip details like bytecode blinding as she only requires the high-level semantics of a handler. Sharif et al. successfully mounted such an attack to recover the CFG of the virtualized function [54], but do not take semantics other than virtual instruction pointer updates into account.

We recognize the latter approach as promising and note how Syntia allows us to *automatically* extract the high-level semantics of arithmetical and logical instruction handlers. This is achieved by operating on an execution trace through the interpreter and simplify its individual handlers—as distinguished by trace window boundaries—using program synthesis. Especially, we highlight how obtaining the semantics of *one* handler automatically yields information about the underlying native code at *all points* of the trace where this specific handler is used to encode equivalent virtualized semantics.

Evaluation Setup. We evaluated Syntia to learn the semantics of arithmetic and logical VM instruction handlers in recent versions of VMProtect [59] (v3.0.9) and Themida [39] (v2.4.5.0). To this end, we built a program that covers bit-vector arithmetic for operand widths of 8, 16, 32, and 64 bit. Since we are interested in analyzing effects of the VM itself, using a synthetic program does not distort our results. For verification, we manually reverse engineered the VM layouts of VMProtect and Themida. Note that the commercial versions of both protection systems have been used to obfuscate the program. These are known to provide better obfuscation strength compared to the evaluation versions.

We argue that our evaluation program is representative of *any* program obfuscated with the respective VM-based obfuscating scheme. As seen in Section 2.1.1, common instructions map to a plethora of VM handlers. Consequently, if we succeed in recovering the semantics of these integral building blocks, we are at the same time able to recover other variations of native instructions using these handlers as well.

This motivates the design of our evaluation program, which aims to have a wide coverage of all possible arithmetic and logical operations. We note that this may not be the case for *real-world* test cases, which may not trigger all interesting VM handlers. To this extent, our evaluation program is, in fact, *more representative* than, e. g., malware samples.

6.2.1 VMProtect

In its current version, VMProtect follows the *Direct Threaded Code* design principle (cf. Section 2.1.1). Each handler directly invokes the next handler based on the address encoded directly in the instruction's bytecode. Hence, reconstructing the handlers requires an instruction trace. Also, this impacts trace dissection: since VM handlers dispatch the next handler, they end with an indirect jump. Unsurprisingly, Syntia could automatically dissect the instruction trace into trace windows that represent a single VM handler. As evident from Table 1, there are 449 *unique* trace windows out of a total of 12,577 in the instruction trace.

Further, VMProtect employs *handler duplication*. For example, the 449 instruction handlers contain 12 instances performing 8-bit addition, 11 instances for each of addition (for each flavor of 16-, 32-, 64-bit), `nor` (8-, 64-bit), left and right shift (32-, 64-bit); amongst multiple others. If Syntia is able to learn one instance in each group, it is safe to assume that it will successfully synthesize the full group, as supported by our results.

Similarly, the execution trace is made up of all possible handlers and some of them occur multiple times. Hence, if we correctly synthesize semantics for, e. g., a 64-bit addition, this immediately yields semantics for 772 trace windows (6.2% of the full trace, 32.0% of all arithmetic and logical trace windows in the trace). Equivalent reasoning applies to 16-bit `nor` operations in our trace (3.6% of the full trace, 18.8% of all arithmetic and logical trace windows). In total, our results reveal semantics for 19.7% of the full execution trace (2,482 out of 12,577 trace windows). Manual analysis suggests that the remaining trace semantics mostly

consists of control-flow handling and stack operations. These are especially used when switching from the native to the VM context and amount for a large part of the execution trace.

On average, an individual instruction handler consists of 49 instructions. As VMProtect's VM is stack-based, binary arithmetic handlers pop two arguments from the stack and push the result onto the stack. This tremendously eases identification of inputs and outputs. Therefore, we mark memory operands as inputs and outputs and use the configuration vector $(1.5, 30000, 20, 0)$ for the synthesis. The sampling phase finished in less than two minutes. Overall, the $1,123$ synthesis tasks completed in less than an hour, which amounts to merely 3.7 seconds per task. In total, in our first run, we automatically identified 190 out of 196 arithmetical and logical handlers (96.9%). The remaining 6 handlers implement 8-bit divisions and shifts. Due to their representation in x86 assembly code, Syntia needs to synthesize more complex expressions with nested data type conversions. As the analysis is probabilistic in nature, we scheduled five more runs which yielded 4 new handlers. Thus, we are able to automatically pinpoint 98.9% of all arithmetic and logical instruction handlers in VMProtect.

6.2.2 Themida

The protection solution Themida supports three basic VM flavors, namely, Tiger, Fish, and Dolphin. Each flavor can further be customized to use one of three obfuscation levels, in increasing complexity: White, Red, and Black. We note that related work on deobfuscation does not directly mention the exact configuration used for Themida. In hopes to be comparable, we opted to use the default flavor Tiger, using level White, in our evaluation. Unlike VMProtect, Tiger White uses an explicit handler table while inlining the dispatcher routine; i. e., it follows the *Threaded Code* design principle (cf. Section 2.1.1). Consequently, trace dissection again yields one trace window per instruction handler. Even though the central handler table lists $1,111$ handlers, we identified 106 *unique* trace windows along the concrete execution trace.

Themida implements a register-based architecture and stores intermediate computations in one of many register available in the VM context. This, in turn, affects the identification of input and output variables. While in the case of VMProtect, inputs and outputs are directly taken from two slots on the stack, Themida has a significantly higher number of potential inputs and outputs (i. e., all virtual registers in the VM context, 10 to 15 in our case).

Tiger White supports handlers for addition, subtraction, multiplication, logical left and right shift, bitwise operations and unary subtraction; each for different operand widths. In contrast to VMProtect, handlers are neither duplicated nor do they occur multiple times in the execution trace. Hence, the trace itself is much more compact, spanning $2,448$ trace windows in total; roughly 5 times shorter than VMProtect's. Still, Themida's handlers are much longer, with 258 instructions on average.

We ran the analysis using the configuration vector $(1.8, 50000, 20, 0)$. Due to the higher number of inputs, this configuration—in comparison to the previous

section—sets a much higher focus on exploration as indicated by higher values chosen for the first two parameters. Sampling finished in one minute, whereas the synthesis phase took around 166 minutes. At $1,092$ synthesis tasks, this amounts to roughly 9.1 seconds per task. Eventually, we automatically learned the semantics of 34 out of 36 arithmetic and logical handlers (94.4%). The remaining handlers (8-bit subtraction and logical or) were not found as we were unable to complete the sampling phase due to crashes in Unicorn engine.

6.3 ROP Gadget Analysis

We further evaluated Syntia on ROP gadgets, specifically, on four samples that were thankfully provided by Debray [63]. They implement bubble sort, factorials, Fibonacci, and matrix multiplication in ROP. To have a larger set of samples, we also used a CTF challenge [42] that has been generated by the ROP compiler Q [52] and another Fibonacci implementation that has been generated with ROPC [40].

Syntia automatically dissected the instruction traces into 156 individual gadgets. Since many gadgets use exactly the same instructions, we unified them into 78 unique gadgets. On average, a gadget consists of 3 instructions with 3 inputs and 2 outputs (register and memory locations).

Due to the small numbers of inputs and synthesis tasks, we chose the configuration vector $(1.5, 100000, 50, 0)$ that sets a very strong focus on exploration while accepting a higher running time. Especially, we experienced both effects for the maximum number of MCTS iterations.

Syntia synthesized partial semantics for 97.4% of the gadgets in less than 14 minutes; in total, we were successful in 163 out of the 178 (91.5%) synthesis tasks. Our synthesis results include 58 assignments, 17 binary additions, 5 ternary additions, 4 unary minus, 4 binary subtractions, 4 register increments/decrements, 2 binary multiplications and 1 bitwise and. In addition, we found 68 stack pointer increments due to `ret` statements. The results do not include larger constants or operations such as `ror` as they are not part of our grammar.

7 Discussion

In the following, we discuss different aspects of program synthesis for trace simplification and MCTS-based program synthesis. Furthermore, we point out limitations of our approach as well as future work.

Program Synthesis for Trace Simplification. Current research on deobfuscation [13, 54, 62, 63] operates on instruction traces and uses a mixed approach consisting of symbolic execution [62] and taint analysis [61]; two approaches that require a precise analysis of the underlying code. While techniques exist that defeat taint analysis [6, 49], recent work shows that symbolic execution can similarly be attacked [2].

Program synthesis is an orthogonal approach that operates on a purely semantical level as opposed to (binary) code analysis; it is oblivious to the underlying code constructs. As a result, syntactical aspects of code complexity such as obfuscation or instruction count do not influence program synthesis negatively. It is merely concerned with the complexity of the code's *semantics*. The only exception where code-level artifacts matter is the generation of I/O samples; however, this can be realized with small overhead compared to regular execution time using dynamic binary instrumentation [38, 41].

Commonly, instruction traces contain repetitions of unique trace windows that can be caused by loops or repeated function calls to the same function. By synthesizing these trace windows, the synthesized semantics pertain for all appearances on the instruction trace; the more frequently these trace windows occur in the trace, the higher the percentage of *known* semantics in the instruction trace. We stress how VM-based obfuscation schemes do this to the extreme: a relatively small number of unique trace windows are used over the whole trace.

In general, the synthesis results may not be precise semantics since we approximate them based on I/O samples. If these do not reflect the full semantics, the synthesis misses edge cases. For instance, we sometimes cannot distinguish between an arithmetic and a logical right shift if the random inputs are no *distinguishing* inputs. We point out that this is not necessarily a limitation, since a human analyst might still get valuable insights from the approximated semantics.

As future work, we consider improving trace simplification by a stratified synthesis approach [23]. The main idea is to incrementally synthesize larger parts of the instruction trace based on previous results and successively approximate high-level semantics of the entire trace. Further, we note that the work by Sharif et al. [54] is complementary to our synthesis approach and would also allow us to identify control flow. Likewise, extending the grammar by control-flow operations is another viable approach to tackle this limitation.

MCTS-based Program Synthesis. Compared to SMT-based program synthesis, we obtain *candidate* solutions, even if the synthesizer does not find an *exact* result. This is particularly beneficial for applications such as deobfuscation, since a human analyst can sometimes infer the full semantics. We decided to utilize MCTS for program synthesis since it has been proven very effective when operating on large search trees without domain knowledge. However, our approach is not limited to MCTS, other stochastic algorithms are also applicable.

Drawn from the observations made in Section 6, we infer that the MCTS approach is much more effective with a configuration that focuses on exploration instead of exploitation. The SA-UCT parameter ensures that paths with a higher reward are explored in-depth in later stages of the algorithm. We still try to improve exploration strategies, for instance with *Nested Monte Carlo Tree Search* [35] and *Monte Carlo Beam Search* [7].

Limitations. In general, limits of program synthesis apply to our approach as well. Non-determinism and point functions—Boolean functions that return 1 for exactly one input out of a large input domain—cannot be synthesized practically. This also holds for semantics that have strong confusion and diffusion properties, such as cryptographic algorithms. These are inherently very complex, non-linear expressions with a deep nesting level. Our approach is also limited by the choice of trace window boundaries; ending a trace window in intermediate computation steps may produce formulas that are not *meaningful* at all.

8 Related Work

We now review related work for program synthesis, Monte Carlo Tree Search and deobfuscation. Furthermore, we describe how our work fits into these research areas.

Program Synthesis. Gulwani et al. [22] introduced an SMT-based program synthesis approach for loop-free programs that requires a logical specification of the desired program behavior. Building on this, Jha et al. [24] replaced the specification with an I/O oracle. Upon generation of *multiple* valid program candidates, they derive *distinguishing inputs* that are used for subsequent oracle queries. They demonstrated their use case by simplifying a string obfuscation routine of MyDoom. Godfroid and Taly [18] used an SMT-based approach to learn the formal semantics of CPU instruction sets; for this, they use the CPU as I/O oracle.

Schkufza et al. [51] proved that stochastic program synthesis often outperforms SMT-based approaches. This is mostly due to the fact that common SMT-based approaches effectively enumerate *all* programs of a given size or prove their non-existence. On the other hand, stochastic approaches focus on promising parts of the search space without searching exhaustively. Schkufza et al. use this technique for stochastic superoptimization on the basis of their tool *STOKE*. Recent work by Heule et al. [23] demonstrates a stratified approach to learn the semantics of the x86-64 instruction set, based on STOKE. Their main idea is to re-use synthesis results to synthesize more complex instructions in an iterative manner. To the best of our knowledge, STOKE is the only other stochastic synthesis tool that is able to synthesize low-level semantics. By design, their code only produces Intel x86 code.

In our case, stochastic techniques have additional properties that are not achieved by previous tools: we obtain partial results that are often already "close" to a real solution and might be helpful for a human analyst who tries to understand obfuscated code. Furthermore, we can encode arbitrary complex function symbols in our grammar (e. g., complex encoding schemes or hash functions); a characteristic that is not easily reproduced by SMT-based approaches.

In the context of non-academic work, Rolles applied some of the above mentioned SMT-based approaches to reverse engineering and deobfuscation [46].

Amongst others, he learned obfuscation rules by adapting peephole superoptimization techniques [3] and extracted metamorphic code using an oracle-guided approach. In his recent work, he performs SMT-based shellcode synthesis [47].

Monte Carlo Tree Search. MCTS has been widely studied in the area of AI in games [16, 35, 50, 57]. Ruijl et al. [48] combine Simulated Annealing and MCTS by introducing SA-UCT for expression simplification. Lim and Yoo [32] describe an early exploration on how MCTS can be used for program synthesis and note that it shows comparable performance to genetic programming. We extend the research of MCTS-based program synthesis by applying SA-UCT and introducing node pruning. For our synthesis approach, we designed a context-free grammar that learns the semantics of Intel x86 code.

Deobfuscation. Rolles provides an academic analysis of a VM-based obfuscator and outlines a possible attack on such schemes in general [45]. He proposes using static analysis to re-translate the VM's bytecode back into native instructions. This, however, requires minute analysis of *each* obfuscator and hence is time-consuming and prone to minor modifications of the scheme. Kinder is also concerned with (static) analysis of VMs [26]. Specifically, he lifts a *location-sensitive* analysis to be usable in presence of virtualization-based obfuscation schemes. His work highlights how the execution trace of a VM, while performing various computations, always exhibits a recurring set of addresses. As seen in Section 6, our approach actually benefits from this side effect. In contrast, Sharif et al. [54] analyze VMs in a *dynamic* manner and record execution traces. In contrast to the work of Rolles, their goal is not to re-translate, but to directly analyze the bytecode itself. Specifically, they aim to reconstruct parts of the underlying code's control flow from the bytecode. This approach is closest to our work as we are, in turn, mostly concerned with arithmetic and logical semantics of a handler.

More recent results include work by Coogan et al. [13] as well as Yadegari et al. [63]. Both approaches seek to deobfuscate code based on execution traces by further making use of symbolic execution and taint tracking. The former approach is focused on the *value flow* to system calls to reduce a trace whereas Yadegari et al. propose a more general approach and aim to produce fully deobfuscated code. However, to counteract symbolic execution-based deobfuscation approaches, Banescu et al. propose novel obfuscating transformations that specifically target their deficiencies [2]. For one, they propose a construct akin to *random opaque predicates* [12] that deliberately explodes the number of paths through a function. A second technique preserves program behavior of the obfuscated program for specific input invariants *only*, effectively increasing the input domains and thus the search space for symbolic executors.

Guinet et al. present *arybo*, a framework to simplify MBA expressions [20]. In essence, they perform bit-blasting and use a Boolean expression solver that tries to simplify the expression symbolically. Eyrolles [15] describes a symbolic approach that uses pattern matching. Furthermore, she suggests improvements of current

MBA-obfuscated implementations that impede these symbolic deobfuscation techniques [14]. To this effect, we also argue that symbolic simplification is inherently limited by the complexity of the input expression. However, we demonstrated that a synthesis-based approach allows fine-tuned simplification, irrespective of *syntactical* complexity, while producing approximate intermediate results.

9 Conclusion

With our prototype implementation of Syntia we have shown that program synthesis can aid in deobfuscation of real-world obfuscated code. In general, our approach is vastly different in nature compared to proposed deobfuscation techniques and hence may succeed in scenarios where approaches requiring precise code semantics fail.

References

[1] Andriesse, D., Bos, H., and Slowinska, A. Parallax: Implicit Code Integrity Verification using Return-Oriented Programming. In *Conference on Dependable Systems and Networks (DSN)* (2015).

[2] Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., and Pretschner, A. Code Obfuscation against Symbolic Execution Attacks. In *Annual Computer Security Applications Conference (ACSAC)* (2016).

[3] Bansal, S., and Aiken, A. Automatic Generation of Peephole Superoptimizers. In *ACM Sigplan Notices* (2006).

[4] Bell, J. R. Threaded Code. *Communications of the ACM* (1973).

[5] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* (2012).

[6] Cavallaro, L., Saxena, P., and Sekar, R. Anti-Taint-Analysis: Practical Evasion Techniques against Information Flow based Malware Defense. *Secure Systems Lab at Stony Brook University, Tech. Rep* (2007).

[7] Cazenave, T. Monte carlo beam search. *IEEE Transactions on Computational Intelligence and AI in Games* (2012).

[8] Chaslot, G. *Monte-Carlo Tree Search*. PhD thesis, Universiteit Maastricht, 2010.

[9] Collberg, C., Martin, S., Myers, J., and Nagra, J. Distributed Application Tamper Detection via Continuous Software Updates. In *Annual Computer Security Applications Conference (ACSAC)* (2012).

[10] Collberg, C., Martin, S., Myers, J., and Zimmerman, B. Documentation for Arithmetic Encodings in Tigress. `http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic`.

[11] Collberg, C., Martin, S., Myers, J., and Zimmerman, B. Documentation for Data Encodings in Tigress. `http://tigress.cs.arizona.edu/transformPage/docs/encodeData`.

[12] Collberg, C., Thomborson, C., and Low, D. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM Symposium on Principles of Programming Languages (POPL)* (1998).

[13] Coogan, K., Lu, G., and Debray, S. Deobfuscation of Virtualization-obfuscated Software: A Semantics-Based Approach. In *ACM Conference on Computer and Communications Security (CCS)* (2011).

[14] Eyrolles, N. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2017.

[15] Eyrolles, N., Goubin, L., and Videau, M. Defeating MBA-based Obfuscation. In *ACM Workshop on Software PROtection (SPRO)* (2016).

[16] Finnsson, H. Generalized Monte-Carlo Tree Search Extensions for General Game Playing. In *AAAI Conference on Artificial Intelligence* (2012).

[17] Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., and Teytaud, O. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM* (2012).

[18] Godefroid, P., and Taly, A. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *ACM SIGPLAN Notices* (2012).

[19] Graziano, M., Balzarotti, D., and Zidouemba, A. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2016).

[20] Guinet, A., Eyrolles, N., and Videau, M. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *GreHack Conference* (2016).

[21] Gulwani, S. Dimensions in Program Synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming* (2010).

[22] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. Synthesis of Loop-free Programs. *ACM SIGPLAN Notices* (2011).

[23] Heule, S., Schkufza, E., Sharma, R., and Aiken, A. Stratified synthesis: Automatically Learning the x86-64 Instruction Set. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2016).

[24] Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided Component-based Program Synthesis. In *ACM/IEEE 32nd International Conference on Software Engineering* (2010).

[25] Kim, D.-W., Kim, K.-H., Jang, W., and Chen, F. F. Unrelated Parallel Machine Scheduling with Setup Times using Simulated Annealing. *Robotics and Computer-Integrated Manufacturing* (2002).

[26] Kinder, J. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *IEEE Working Conference on Reverse Engineering (WCRE)* (2012).

[27] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by Simulated Annealing. *Science* (1983).

[28] Klint, P. Interpretation Techniques. *Software, Practice and Experience* (1981).

[29] Kocsis, L., and Szepesvári, C. Bandit based Monte-Carlo Planning. In *European Conference on Machine Learning* (2006).

[30] Krahmer, S. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique, 2005.

[31] Liberatore, P. The Complexity of Checking Redundancy of CNF Propositional Formulae. In *International Conference on Agents and Artificial Intelligence* (2002).

[32] Lim, J., and Yoo, S. Field Report: Applying Monte Carlo Tree Search for Program Synthesis. In *International Symposium on Search Based Software Engineering* (2016).

[33] Lu, K., Xiong, S., and Gao, D. RopSteg: Program Steganography with Return Oriented Programming. In *ACM Conference on Data and Application Security and Privacy (CODASPY)* (2014).

[34] Ma, H., Lu, K., Ma, X., Zhang, H., Jia, C., and Gao, D. Software Watermarking using Return-Oriented Programming. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2015).

[35] Marc, Sebag, M., Silver, D., Szepesvári, C., and Teytaud, O. Nested Monte-Carlo Search. *Communications of the ACM* (2012).

[36] Microsoft Research. The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[37] Mougey, C., and Gabriel, F. DRM obfuscation versus auxiliary attacks. REcon conference, 2014. https://recon.cx/2014/schedule/events/44.html.

[38] Nethercote, N., and Seward, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Sigplan Notices* (2007).

[39] Oreans Technologies. Themida – Advanced Windows Software Protection System. http://oreans.com/themida.php.

[40] pakt. ROPC: A Turing complete ROP compiler. https://github.com/pakt/ropc.

[41] Pewny, J., Garmany, B., Gawlik, R., Rossow, C., and Holz, T. Cross-architecture Bug Search in Binary Executables. In *IEEE Symposium on Security and Privacy* (2015).

[42] Plaid CTF. ROP Challenge "quite quixotic chest". https://ctftime.org/task/2305, 2016.

[43] Quynh, N. A., Di, T. S., Nagy, B., and Vu, D. H. Capstone Engine. http://www.capstone-engine.org.

[44] Quynh, N. A., and Vu, D. H. Unicorn – The Ultimate CPU Emulator. http://www.unicorn-engine.org.

[45] Rolles, R. Unpacking Virtualization Obfuscators. In *USENIX Workshop on Offensive Technologies (WOOT)* (2009).

[46] Rolles, R. Program Synthesis in Reverse Engineering. http://www.msreverseengineering.com/blog/2014/12/12/program-synthesis-in-reverse-engineering, 2014.

[47] Rolles, R. Synesthesia: A Modern Approach to Shellcode Generation. http://www.msreverseengineering.com/blog/2016/11/8/synesthesia-modern-shellcode-synthesis-ekoparty-2016-talk, 2016.

[48] Ruijl, B., Vermaseren, J. A. M., Plaat, A., and van den Herik, H. J. Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. In *International Conference on Agents and Artificial Intelligence* (2014).

[49] Sarwar, G., Mehani, O., Boreli, R., and Kaafar, D. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices. *Nicta* (2013).

[50] Schadd, M. P., Winands, M. H., Tak, M. J., and Uiterwijk, J. W. Single-player Monte-Carlo Tree Search for SameGame. *Knowledge-Based Systems* (2012).

[51] Schkufza, E., Sharma, R., and Aiken, A. Stochastic Superoptimization. *ACM SIGPLAN Notices* (2013).

[52] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium* (2011).

[53] Shacham, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[54] Sharif, M., Lanzi, A., Giffin, J., and Lee, W. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy* (2009).

[55] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* (2016).

[56] Sony DADC. SecuROM Software Protection. https://www2.securom.com/Digital-Rights-Management.68.0.html.

[57] Szita, István and Chaslot, Guillaume and Spronck, Pieter. Monte-Carlo Tree Search in Settlers of Catan. In *Advances in Computer Games* (2009).

[58] Tages SAS. SolidShield Software Protection. https://www.solidshield.com/software-protection-and-licensing.

[59] VMProtect Software. VMProtect Software Protection. http://vmpsoft.com.

[60] Vogl, S., Pfoh, J., Kittel, T., and Eckert, C. Persistent Data-only Malware: Function Hooks without Code. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[61] Yadegari, B., and Debray, S. Bit-level Taint Analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation* (2014).

[62] Yadegari, B., and Debray, S. Symbolic Execution of Obfuscated Code. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[63] Yadegari, B., Johannesmeyer, B., Whitely, B., and Debray, S. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy* (2015).

[64] Zhou, Y., Main, A., Gu, Y. X., and Johnson, H. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop on Information Security Applications (WISA)* (2007).