Loki: Hardening Code Obfuscation Against Automated Attacks

Moritz Schloegel[*], Tim Blazytko[*], Moritz Contag[*], Cornelius Aschermann[*]
Julius Basler[*], Thorsten Holz[†], Ali Abbasi[*]

[*]*Ruhr-Universität Bochum*
[†]*CISPA Helmholtz Center for Information Security*

Abstract

Software obfuscation is a crucial technology to protect intellectual property and manage digital rights within our society. Despite its huge practical importance, both commercial and academic state-of-the-art obfuscation methods are vulnerable to a plethora of automated deobfuscation attacks, such as symbolic execution, taint analysis, or program synthesis. While several enhanced obfuscation techniques were recently proposed to thwart taint analysis or symbolic execution, they either impose a prohibitive runtime overhead or can be removed in an automated way (e. g., via compiler optimizations). In general, these techniques suffer from focusing on a single attack vector, allowing an attacker to switch to other, more effective techniques, such as program synthesis.

In this work, we present LOKI, an approach for software obfuscation that is resilient against all known automated deobfuscation attacks. To this end, we use and efficiently combine multiple techniques, including a generic approach to synthesize formally verified expressions of arbitrary complexity. Contrary to state-of-the-art approaches that rely on a few hard-coded generation rules, our expressions are more diverse and harder to pattern match against. We show that even the state-of-the-art approach on Mixed-Boolean Arithmetic (MBA) deobfuscation fails to simplify them. Moreover, LOKI protects against previously unaccounted attack vectors such as program synthesis, for which it reduces the success rate to merely 19%. In a comprehensive evaluation, we show that our design incurs significantly less overhead while providing a much stronger protection level compared to existing works.

1 Introduction

Obfuscation describes the process of applying transformations to a given program with the goal of protecting the code from prying eyes. Generally speaking, obfuscation works by taking (parts of) a program and transforming it into a more complex, less intelligible representation, while at the same time preserving its observable input-output behavior [19]. Usually, such transformations come at the cost of increased program runtime and size, thus trading intelligibility for overhead. Although formal verification of code transformations is hard to achieve in practice [49, 85], obfuscation is used in a wide range of real-world scenarios. Examples include protection of intellectual property (IP), digital rights management (DRM), and concealment of malicious behavior in software. Generally speaking, obfuscation protects critical (often small) code parts against reverse engineering and, thus, misuse by competitors or other parties. For example, most contemporary DRM systems rely on some kind of obfuscation to prevent attackers from distributing unauthorized copies of their product [53]. License checks and cryptographic authentication schemes are examples for code that is commonly obfuscated in practice to prevent analysis. Most copy-protection schemes used by games use some kind of obfuscation to prevent unauthorized copies. As another example, market-leading companies, such as *Snapchat*, obfuscate how API calls to their backend are constructed, preventing abuse and access by competitors [28].

Among the countless obfuscation methods proposed in the literature [1, 8, 14, 19, 31, 35, 40, 50, 54, 56, 67, 74, 77, 80, 81, 87], one of the most promising techniques is Virtual Machine (VM)-based obfuscation [31, 58]. State-of-the-art, commercial obfuscators such as THEMIDA [55] and VMPROTECT [73], as well as most game copy-protection schemes used in practice [26, 66], make extensive use of VM-based obfuscation. They transform the to-be-protected code from its original Instruction Set Architecture (ISA) into a *custom* one, and bundle an interpreter with the program that emulates the new ISA. This effectively breaks any analysis tool unfamiliar with the new architecture. Attackers aiming to deobfuscate code affected by this scheme must first uncover the custom ISA before they can reconstruct the original code [58, 64]. Since the custom instruction sets are conceptually simple, VM-based obfuscation software usually applies additional obfuscating transformations to the interpreter such that it is harder to analyze. For example, dead code insertion or constant unfolding are often used. At their core, these transformations inflate the number of executed instructions and mainly add to the code's

syntactic complexity, but can be successful in thwarting manual attacks.

However, it is often sufficient to apply well-known compiler optimizations, such as *dead code elimination*, *constant folding*, or *constant propagation*, to reduce the code's *syntactic* complexity and enable subsequent analyses [34, 84]. We tested this hypothesis and observe that this applies to the state-of-the-art tools THEMIDA and VMPROTECT, for both their fastest and strongest protection configurations: We found that a simple dead code elimination manages to reduce the number of assembly instructions per handler by at least 50% for five different targets, tremendously simplifying both manual and automated analyses (cf. Table 1). Subsequently, the resulting code can be further simplified using a wide range of automated techniques, including taint analysis [82, 84], symbolic execution [83, 84], program synthesis [7, 25], and various others [5, 6, 21, 29, 30, 34, 36, 42, 47, 58, 59, 64].

The reliance on *syntactic* complexity in state-of-the-art obfuscation schemes and the broad arsenal of advanced deobfuscation techniques sparked further research in the construction of more resilient schemes that aim to impede these automated analyses. Proposals were made to hinder taint analysis [10, 60] and render symbolic execution ineffective [1, 54, 80, 87]. For example, the latter can be achieved by triggering a path explosion for the symbolic execution engine by artificially increasing the number of paths to analyze. Other promising obfuscation schemes emerged, including Mixed Boolean-Arithmetic (MBA) expressions [2, 29, 87] that offer a model to encode arbitrary arithmetic formulas in a complex manner. The expressions are represented in a domain that does not easily lend itself to simplification, effectively hiding the actual semantic operations. Usually, automated approaches to deobfuscate MBAs are based on symbolic simplification [6, 29, 30, 36]; they rely on certain assumptions about the expression's structure, making them unfit to simplify such expressions in the general case. Other approaches are based on program synthesis [7, 25, 51], which have been proven highly effective for most tasks. In general, synthesis-based deobfuscation techniques remain unchallenged to date and are valuable methods for automated analysis of obfuscated code. Recent works aiming at simplifying MBA turned towards machine learning [32] and algebraic simplification [48]. Especially the latter approach, relying on a hidden two-way feature between 1-bit and n-bit variables used within MBAs, provides an automated attacker with unprecedented MBA deobfuscation capabilities.

In this paper, we introduce a novel and comprehensive set of obfuscation techniques that can be combined to protect code against all known automated deobfuscation attacks, while imposing only reasonable overhead in terms of space and runtime. Our techniques are specifically designed such that a human analyst gains no significant advantage from employing automated deobfuscation techniques, including compiler optimizations (cf. Table 1), forward taint analysis, symbolic execution, and even program synthesis (cf. Section 6). We explicitly assume scenarios where these techniques are specifically tailored to our design *(white-box scenario)*.

To achieve such protection, we propose a generic algorithm to synthesize formally verified, arbitrarily complex MBA expressions. This is in strong contrast to state-of-the-art approaches that rely on a few handwritten rules, greatly limiting their effectiveness. For example, given $7,000$ VM handlers, TIGRESS—the state-of-the-art academic obfuscator—uses only 16 unique MBAs, while our design features ~$5,500$ unique MBAs. As a result, our MBAs are highly unlikely to be simplified: In fact, current state-of-the-art MBA deobfuscation tools such as MBA-BLAST [48] can only simplify 0.5% of LOKI's MBAs. Furthermore, we conduct the first conclusive analysis of the limits of program synthesis with regard to deobfuscation. Based on the resulting insights, we present a hardening technique capable of impeding program synthesis, reducing its success rate to 19%—for TIGRESS, it is 67%. In summary, we present a new design featuring both high diversity and resilience against static and dynamic, automated deobfuscation attacks. While providing more value, our design incurs significantly less overhead compared to commercial, state-of-the-art obfuscation schemes (up to 40 times). Moreover, we port modern testing techniques, i.e., formal verification and fuzzing, to our design and show that complex combinations of obfuscation transformations benefit from such methods to assert the correctness of complex and non-deterministic obfuscation transformations.

Contributions. We make the following contributions:

- We present the design, implementation, and evaluation of LOKI, a software obfuscation approach resilient against all known automated deobfuscation attacks, even in white-box scenarios.

- We introduce a generic approach to synthesize diverse and formally verified Mixed Boolean-Arithmetic (MBA) expressions of arbitrary complexity that withstand even current state-of-the-art deobfuscation attacks.

- We are the first to propose an approach resilient against program synthesis-based attacks and map out limits of program synthesis in an empirical evaluation.

We publish the source code of LOKI as well as all evaluation artifacts (including test cases, binaries, and evaluation tooling) at https://github.com/RUB-Syssec/loki. An extended version of this paper with more technical details is available as a technical report [62].

2 Technical Background

We start by providing an overview of the required technical information on obfuscation and deobfuscation techniques.

Table 1: VM handler statistics for THEMIDA, VMPROTECT, and our approach called LOKI. The two commercial obfuscators are configured in their fastest (*Virtualization* and *Tiger White*) and strongest configuration (*Ultra* and *Dolphin Black*), but without additional security features (e. g., anti-debug). We track their handlers' *average number* of assembly and intermediate language (IL) instructions before and after dead code elimination (denoted as the percentage-wise reduction in parentheses). All values are averaged over five cryptographic algorithms (AES, DES, MD5, RC4, and SHA-1).

	VMPROTECT		THEMIDA		LOKI
Statistics	Virtualization	Ultra	Tiger White	Dolphin Black	
Assembly instructions	$69 \, (-50.79\%)$	$73 \, (-51.58\%)$	$219 \, (-53.68\%)$	$243 \, (-56.01\%)$	$222 \, (-1.14\%)$
IL instructions	$75 \, (-50.88\%)$	$80 \, (-51.89\%)$	$221 \, (-53.76\%)$	$247 \, (-55.94\%)$	$234 \, (-1.44\%)$
Handlers executed	46,591	151,303	83,191	290,815	4,123
... of them unique	274	4,578	204	337	55

2.1 VM-based Obfuscation

Virtual machine-based obfuscation, also known as *virtualization*, protects code by translating it into an intermediate representation called *bytecode*. This bytecode is interpreted by a CPU implemented in software, adhering to a custom instruction set architecture (ISA). An attacker must first reverse engineer this software CPU, a tedious and time-consuming task [58, 64]. Only after understanding the VM, they can reconstruct the original high-level code.

VM Interpreter. The original, unprotected code is replaced with a call to the *VM entry* that invokes the interpreter. It sets up the initial context of the VM and points it to the bytecode that is to be interpreted. This is implemented by the *VM dispatcher* using a fetch-decode-execute loop: first, it fetches the next instruction, decodes its opcode, and then transfers execution to the respective *VM handler*. Often, the handler is determined via a *global handler table* that is indexed by the opcode. After handler execution, the control flow returns to the VM dispatcher. Eventually, execution finishes by invoking a special *VM exit* handler aborting the loop.

Abstraction of Handler Semantics. Handlers are often semantically simple [7, 58]; they perform a single arithmetic or logical operation on a number of operands, e. g., $x \odot y$. We call the semantic function of a handler, i. e., the underlying instruction it implements, its *core semantics*. We can represent core semantics as a function $f(x, y)$, or more general as $f(x, y, c)$ where c is a constant. To measure the *syntactic complexity* of the core semantics, we compute the (syntactic) expression depth of f as the sum of all variable occurrences and operators. In contrast, the *semantic depth* refers to the syntactic depth of the syntactically shortest equivalent expression. Intuitively, it can be understood as the number of nodes in an Abstract Syntax Tree (AST).

Example 1: *We can represent a VM handler's core semantics* $x + y$ *as* $f(x, y, c) := x + y$ *with a syntactic depth of* 3. *A syntactically more complex function* $g(x, y, c) := x + y - x + c - c$ *has a syntactic depth of* 9 *but a semantic depth of* 1, *since* g *can be simplified to* $g(x, y, c) := y$.

Superoperators. Superoperators [57] are an approach to make handlers semantically more complex. Intuitively, this is achieved by combining different instruction sequences from the unprotected code into a single VM handler. Usually, these sequences compute independent results such that this VM "su-

perhandler" computes multiple, independent VM handlers in a single step. As a consequence, superoperators often have multiple input and output tuples. Related to our function abstraction, we can say the function $f_s((x_0, y_0, c_0), \ldots, (x_n, y_n, c_n))$ computes an output tuple $(o_0, \ldots, o_n)$, where x_i, y_i, c_i and o_i represent the core semantics' inputs/output of a semantically simple VM handler. While originally developed to minimize the number of handlers executed to improve performance, superoperators have been used by obfuscators such as TIGRESS primarily for obtaining more complex VM handlers.

2.2 Mixed Boolean-Arithmetic

Mixed Boolean-Arithmetic (MBA) describes an approach to encode expressions in a syntactically complex manner. The goal is to hide underlying semantics in syntactically complex constructs. First described by Zhou et al. [87], MBA algebra connects arithmetic operations (e. g., addition) with bitwise operations (e. g., logical operations or bitshifts). The resulting expressions are usually hard to simplify symbolically [29, 78], since, for every expression, an infinite number of syntactic representations exists. In general, the task of reducing MBA expressions—known as *arithmetic encodings* [1]—to equivalent but simpler expressions is NP-hard [87].

Example 2: $f(x, y, c) := x + y$ *and* $g(x, y, c) := (x \oplus y) + 2 \cdot (x \wedge y)$ *are semantically equivalent. Both implement the same core semantics, but* g *uses a syntactically more complex representation, called MBA.*

3 Automated Deobfuscation Attacks

In the following, we detail common techniques used to analyze obfuscated code.

Forward Taint Analysis. *Forward taint analysis* follows the data flow of so-called *taint sources*, e. g., input variables, and marks all instructions as *tainted* that directly or indirectly depend on these sources [63, 64, 82, 84]. Taint analyses are implemented with varying granularity, referring to the smallest unit they can taint. Common approaches use either bit-level or byte-level granularity. Forward taint analysis can be used to reduce obfuscated code to the instructions depending on user input. The underlying idea is that important semantics rely only on the identified taint sources. All other code constructs,

```
1    mov edx, eax          ; edx1 := eax1
2    mov ecx, 0x20         ; ecx1 := 0x20
3    add edx, ecx          ; edx2 := edx1 + ecx1
4    add edx, 0x10         ; edx3 := edx2 + 0x10
```

Figure 1: An assembly code snippet used to illustrate forward taint analysis, backward slicing, and symbolic execution.

e. g., as added by an obfuscator, can be omitted in an automated matter. Still, if these constructs perform calculations on the user input, taint analysis can be mislead [10, 60].

Example 3: *In Figure 1, assume* eax *is a taint source. The analysis taints the first, third, and fourth instruction since they propagate a taint source. It does* not *taint the second instruction. While its value is later used in tainted instructions, it does not directly depend on* eax.

Backward Slicing. Contrary to forward taint analysis, backward slicing is a backward analysis. Starting from some output variable, it recursively backtracks and marks all input variables on which the output depends [24, 75, 84]. In code deobfuscation, slicing can be used to find all instructions that contribute to the output. Applied to VM handlers, it allows to strip all code not directly related to a handler's core semantics. Similar to forward taint analysis, increasing the number of dependencies (e. g., by inserting junk calculations to the output) reduces the usefulness of slicing.

Example 4: *When backtracking the value of* edx *(line 4 in Figure 1) by following each use and definition, each instruction is marked as they all contribute to the output.*

Symbolic Execution. Symbolic execution allows to summarize assembly code algebraically. Instead of using concrete values, it tracks symbolic assignments of registers and memory in a state map [63]. Often, it works on a verbose representation of code, called *intermediate language (IL)*. Symbolic executors usually know common arithmetic identities and can perform basic simplification, e. g., constant propagation. Applied to code obfuscation, symbolic execution is used to symbolically extract the core semantics of VM handlers [47], track user input in an execution trace [59, 83, 84], or detect opaque predicates (in combination with SMT solvers) [5]. Typically, techniques to impede symbolic execution aim at artificially increasing the syntactic complexity of arithmetic operations (via MBAs) or the number of paths to analyze (triggering a so-called *path explosion*) [1, 54].

Example 5: *After symbolic execution of Figure 1, we obtain the following mappings:* eax *maps to itself (it has not been modified),* ecx *maps to* 0x20 *(line 2). The formula for* edx *is* $eax + 0x20 + 0x10$. *Using arithmetic identities, the symbolic execution engine can simplify the expression to* $eax + 0x30$.

Program Synthesis. In contrast to other techniques that rely on syntactic analysis of obfuscated code, *program synthesis*-based approaches operate on the semantic level. They treat code as a black box and attempt to reconstruct the original code based on the observable behavior, often represented in the form of input-output samples. Approaches such as

SYNTIA [7] and XYNTIA [51] attempt to find an expression with equivalent behavior by relying on a stochastic algorithm traversing a large search space. Other approaches, e. g., QSYNTH [25], are based on enumerative synthesis: they compute large lookup tables of expressions which they use to simplify parts of an expression, reducing its overall complexity. For code deobfuscation, these approaches are used to simplify syntactically complex constructs (e. g., MBAs) or to learn semantics of VM handlers. However, program synthesis struggles with finding semantically complex expressions.

Example 6: *Consider the function* $f(x, y, c) := (x \oplus y) + 2 \cdot (x \wedge y)$. *To learn f's core semantics, we generate random inputs and observe* $f(2, 2, 2) = 4$, $f(10, 13, 10) = 23$, *and* $f(16, 3, 0) = 19$. *A synthesizer eventually produces a function* $g(x, y, c) := x + y$ *that has the same input-output behavior. Notably, it learns that parameter c is irrelevant.*

Ideally, superoperators provide such expressions. However, our experiments (cf. Section 6.3) demonstrate that current designs (e. g., as used by TIGRESS) are still vulnerable; since superoperators combine different core semantics (represented as individual inputs/output tuples), an attacker can synthesize each core semantics separately by targeting each output o_i.

Semantic Codebook Checks. A semantic codebook contains a list of expressions that an attacker expects to exist within obfuscated code. For a syntactically complex expression f, an attacker checks if f is semantically equivalent to an expression g in the codebook by using an SMT solver [69]. If the SMT solver cannot find an input *distinguishing f* and g, it *formally proved* they behave the same for all possible inputs. A typical application scenario are VM handlers: They often implement a simple core semantics (e. g., $x + y$) [7, 58]. Thus, an attacker can construct a codebook based on simple arithmetic and logical operations. As codebooks must contain the respective semantics, increasing the semantic complexity of expressions requires an (exponentially) larger codebook, making the approach infeasible for a practical application.

Example 7: *Consider a function* $f(x, y, c) := (x \oplus y) + 2 \cdot (x \wedge y)$ *and a codebook* $CB := \{x - y, x \cdot y, x + y, \dots\}$. *An attacker can consecutively pick an entry* $g(x, y, c) \in CB$ *and verify whether* $f = g$ *using an SMT solver. To this end, the solver searches an assignment that satisfies* $f(x, y, c) \neq g(x, y, c)$. *Only for* $g(x, y, c) := x + y$ *no solution can be found. Thus, the attacker proved that f can be reduced to a syntactically shorter expression, $x + y$.*

4 Design

We envision a combination of obfuscation techniques where the individual techniques harmonize and complement each other to thwart automated deobfuscation attacks. In line with this philosophy, we now present a set of generic techniques where each constitutes a defense in a particular domain. However, when these techniques are effectively combined, they exhibit comprehensive protection against automated attacks.

To achieve lasting resilience, we focus on inherent weaknesses underlying existing automated attack methods, instead of targeting specific shortcomings of a given implementation. We further underline our techniques' generic nature by discussing their application on an abstract function $f(x,y,c)$ as introduced in Section 2.1. Next, we first discuss the design principles of our approach, present the attacker model, and afterwards explain the individual techniques in detail.

4.1 Design Principles

We have seen outlined automated attack methods can succeed in extracting a function f's core semantics (cf. Section 3). To mitigate these attacks, our design is based on three principles: (1) merging core semantics, (2) adding syntactic complexity, and (3) adding semantic complexity. In the following, we present techniques incorporating these principles and discuss their purpose as well as synergy effects emerging for our overall design.

Merging Core Semantics. Our first technique extends f by merging different, independent core semantics to increase the complexity. This can be understood as combining different, independent VM handlers in a single handler, or—in a more generic setting—combining different semantic operations of an unprotected unit of code in a single function f. The merge is facilitated in such a way that each core semantics is always executed. Still, as these semantics are independent of each other, we must ensure they are individually addressable, i. e., f's output is equivalent to the result of a specific core semantics. To allow the selection of the desired semantics, we extend the function definition to $f(x,y,c,k)$, where k is a *key* selecting the targeted core semantics. Formally, the selection is realized by introducing a point function $e_i(k)$, called *key encoding*, that is associated with a specific core semantics and returns 1 only for its associated key, 0 for other valid keys. This guarantees that the original semantics are preserved. A consequence of this interlocked, "always-execute" nature is that taint analysis and backward slicing fail to remove all semantics in f not associated with a specific k.

Example 8: *We want to design a function f that returns, based on a distinguishing key, either $x+y$ or $x-y$. We write this as $f(x,y,c,k) := e_0(k)(x+y) + e_1(k)(x-y)$ where $e_i(k)$ can be any point function returning* 1 *for the associated k and* 0 *otherwise, for example $e_0(k) := k == 0xdead$. Assuming that $e_0(k)$ returns* 1 *and $e_1(k)$ yields* 0*, f returns $x+y$.*

Adding Syntactic Complexity. Assuming merged semantics using different key encodings, an attacker can still differentiate between key encoding and core semantics for a given function f, as $e_i(k)$ operates only on the key while core semantics use x, y, and c. At the same time, a dynamic attacker with knowledge of k can employ symbolic execution to simplify f to the core semantics associated with the known k by arithmetically nullifying operations not contributing to the result. To prevent such an attack, we increase the syntactic complexity by adding Mixed Boolean-Arithmetic (MBA) formulas to key encodings as well as core semantics.

Symbolically executing these syntactically complex formulas creates no meaningful expressions. Even though modern symbolic execution engines feature simplification rules for basic arithmetic identities and laws, there exists an unlimited number of MBA representations. In general, simplifying such an expression to its syntactically smallest representative is NP-hard [87].

Example 9: *For $f(x,y,c,k) := e_0(k)(x+y) + e_1(k)(x-y)$, we can replace $x+y$ with $(x \oplus y) + 2 \cdot (x \wedge y)$, $x-y$ with $x + \neg y + 1$, and replace the multiplication of $e_1(k)(x-y)$ with the rule $(a \wedge b) \cdot (a \vee b) + (a \wedge \neg b) \cdot (\neg a \wedge b)$ for $a \cdot b$, resulting in the final function $f(x,y,c,k) := e_0(k)((x \oplus y) + 2 \cdot (x \wedge y)) + (e_1(k) \wedge (x + \neg y + 1)) \cdot (e_1(k) \vee (x + \neg y + 1)) + (e_1(k) \wedge \neg (x + \neg y + 1)) \cdot (\neg e_1(k) \wedge (x + \neg y + 1))$.*

To exploit this weakness of symbolic execution and provide a high diversity, we *synthesize* and *formally verify* MBAs instead of using hardcoded rules. This additionally complicates pattern matching and increases the number of instructions marked by forward taint analysis and backward slicing.

Adding Semantic Complexity. One of the remaining problems are semantic attacks, for example, a dynamic attacker that uses input-output behavior to learn an expression equivalent to the core semantics (e. g., via program synthesis). Therefore, we increase the core semantics' complexity by applying the concept of superoperators (cf. Section 2.1). These superoperators make core semantics arbitrarily long and increase the search space for semantic attacks drastically.

Example 10: *Instead of using core semantics of depth 3 (e. g., $x+y$), we apply more advanced core semantics such as $(x+y) \cdot (x \oplus y))$ with depth 7 or $((x \cdot c) \ll (y \vee (x \oplus c)))$ with depth 9, resulting in $f(x,y,c,k) := e_0(k)((x+y) \cdot (x \oplus y)) + e_1(k)((x \cdot c) \ll (y | (x \oplus c)))$.*

While superoperators increase the semantic and syntactic complexity of core semantics, we further extend their syntactic complexity using MBAs. Their synergy additionally diminishes the effect of automated attacks.

4.2 Attacker Model

Intuitively, we envision a strong attacker to measure how our obfuscation scheme fares under worst-case conditions. For this purpose, we assume that an attacker has access to all automated attacks (cf. Section 3).

We assume an attacker has access to the target binary that includes at least one well-defined unit of obfuscated code at a known location. In line with our previous abstraction, we say this code unit can be represented by a function $f(x,y,c,k)$. The attacker's goal is to reconstruct the core semantics of f associated with a specific k. We require the reconstructed semantics to (1) contain *only* the core semantics associated with the specified k and (2) be comparable to the original code's semantics in terms of syntactic complexity. The intuition be-

hind these constraints is to exclude trivial solutions such as providing the unmodified function f itself (which contains, amongst others, the core semantics for the required k).

Further, we assume two *types* of attackers, a static and a dynamic one. The *static attacker* knows the precise code locations of x, y, c, and k as well as the location of function f's output. As a result, they can enrich static analyses, e. g., by defining these code locations as taint sources. A *dynamic attacker* extends the former by the ability to inspect and modify the values at these code locations. In particular, they can observe any key k and propagate it to remove core semantics not associated with this k. While a dynamic attacker is more powerful (in terms of accessible information), certain analysis scenarios such as code running on specific hardware (e. g., embedded devices), analysis on function-level without context, or the presence of techniques like anti-debugging [12,13] may rule out dynamic analysis in practice.

4.3 Key Selection Diversification

We want to prevent static attackers from learning the core semantics via semantic codebook checks and prevent identification of patterns in the key selection. To do so, we employ two different key encoding schemes: Key selection based on (1) the factorization problem, and (2) synthesized partial point functions. To conduct a semantic codebook check, an attacker uses an SMT solver to check for each entry of the codebook whether it is semantically equivalent to f. Assuming that f indeed includes a matching core semantics, the SMT solver has to find a value for k such that the corresponding $e_i(k)$ evaluates to 1. One way to prevent this is to design a key encoding that relies on inherently hard problems for the SMT solver, such as factorization.

Factorization-based Key Encoding. Factorization of a semiprime n (the product of two primes, p and q) is an inherently hard problem *as long as* the size of the factors are large enough (commonly, a few thousand bits). SMT solvers prune the search space by learning partial solutions for a given problem [45], but since no partial solutions exist for factorization, they are forced to perform an exhaustive search.

We define our factorization-based key encoding as $e_i(k) := (n \bmod k) \equiv 0$ where k is a valid 32-bit integer representing one of the two factors ($k \notin \{1, n\}$). As our evaluation shows, this encoding suffices to stall SMT solvers. However, its distinct structure makes pattern matching attempts easy. To increase diversity, we use MBAs and a second key encoding.

Partial Point Functions. Instead of restraining our set of key encodings to a specific type, we synthesize generic point functions without any predefined structure. This is based on the insight that the $e_i(k)$ impose only a single constraint: they must be defined for all valid keys (returning 1 for their associated one, 0 for others). Invalid keys may return arbitrary values, making our synthesized functions *partial point functions*. Consequently, we are not restricted to specific point

functions, such as the factorization-based encoding, but can use arbitrary point functions fulfilling this constraint.

Given a grammar containing ten different arithmetic and logical operations (such as addition, multiplication, and logical and bitwise operations), we generate expressions by chaining a randomly selected operation with random operands. This operand is either an arbitrary key byte or a random 64-bit constant. We chain at most 15 operations to limit the overhead resulting from this expression. Finally, we check if the synthesized expression satisfies the point function's constraint.

Example 11: *Let* $(k_0, k_1, k_2) := (0x1336, 0xabcd, 0x11cd)$ *be a set of keys. Then, we synthesize the point function* $e_0(k) := ((0xff \land k) \oplus 0xcd) \cdot 0x28cbfbeb9a020a33$ *for a 64-bit vector k. $e_0(k)$ evaluates to 1 for k_0 and to 0 for k_1 and k_2. For all other keys, it returns arbitrary values.*

4.4 Syntactic Complexity: MBA Synthesis

To thwart symbolic execution and pattern matching, we use MBAs for all components, including core semantics and key encodings. As hardcoded rules only provide low diversity, we precompute large classes of semantically equivalent arithmetic expressions and combine them through recursive, randomized expression rewriting. We now detail the creation of the equivalence classes and discuss our term rewriting.

Equivalence Class Synthesis. To create semantic equivalence classes for expressions, we rely on enumerative program synthesis [4, 37]. To this end, we first define a context-free grammar with a single non-terminal symbol S as start symbol and two terminal symbols, x and y, representing variables. For each arithmetic operation, we define a production rule that maps the non-terminal symbol to arithmetic operations (e. g., addition) or terminal symbols. To apply a specified production rule to a non-terminal expression, we replace the left-most S with the rule. Expressions without a non-terminal symbol can be evaluated by assigning concrete values to x and y. We say that the *depth* of an expression represents the number of times a non-terminal symbol was replaced by a production rule.

Example 12: *The grammar* $(\{S\}, \Sigma = V \cup O, P, S)$ *with the variables* $V = \{x, y\}$, *the set of arithmetic symbols* $O = \{+, -\}$ *and the production rules* $P = \{S \to x \mid y \mid (S + S) \mid (S - S)\}$ *defines the syntax of how to generate terminal expressions. To derive the expression $x + y$ of depth 3, we apply the following rules:* $S \to (S + S) \to (x + S) \to (x + y)$. *With a mapping of $\{x \mapsto 2, y \mapsto 6\}$, we can evaluate the terminal expression to 8.*

We now describe how we use our context-free grammar in combination with Algorithm 1, which illustrates the high-level approach of equivalence class synthesis. Starting with a worklist of non-terminal states (initialized with the start symbol S), we iteratively process all expressions for a certain depth until we reach a specified upper bound depth N. For a given depth, we derive all terminal and non-terminal

Algorithm 1: Computing equivalence classes.

Data: n is the maximum depth.

```
1 states ← {S}
2 for d ← 1 to n do
3     terminals ← derive_terminals(states)
4     process_terminals(terminals)
5     non_terminals ← derive_non_terminals(states)
6     states ← non_terminals
```

expressions (also referred to as *states*) before processing the terminals and then repeating the process for the next depth. The call to `process_terminals` is responsible for sorting the expressions into the respective equivalence classes. To this end, we evaluate all expressions for a high number of different inputs (e. g., $1,000$), recording their output. Expressions with the same output behavior for all provided inputs are sorted into the same equivalence class. This provides an effective but coarse-grained sorting of expressions into potential equivalence classes. In a final step, we verify that these classes are semantically correct. For this, we choose the member with the smallest depth as representative and check with an SMT solver that all other members are semantically equivalent to this representative. Expressions failing this check are removed from the equivalence class. All remaining expressions are formally proven to not alter the original semantics.

To prune the search space and avoid trivial expressions (e. g., $x + 0$), we symbolically simplify each terminal and nonterminal expression. For this purpose, we apply a normalization step to commutative operators, perform constant propagation, and simplify based on common arithmetic identities (e. g., $x + y - y$ becomes x).

Expression Rewriting. So far, we generated a large set of diverse equivalence classes we can use for replacing syntactically simple expressions with more complex ones. A naive approach replacing expressions with MBAs from the equivalence classes is bounded by the largest depth found in the respective class. To overcome this limitation, we propose a recursive expression rewriting approach using the equivalence classes as building blocks. This allows us to create expressions of *arbitrary* syntactical depth. Even assuming an attacker is in possession of all rewriting rules, it is difficult to invert an expression: Term rewriting is inherently *destructive* [76]. Without knowing the applied rewriting rules and their order, an attacker has to check all possibilities: n rewriting rules applied over m layers, resulting in the prohibitively large number of n^m candidates.

Given some expression e, we pick a random subexpression and check if it is a member of an equivalence class. If it is, we randomly choose another member from this class and use it to replace the picked subexpression within e. We recursively repeat this process for a randomly determined upper bound n. As all members within an equivalence class are proven to be pairwise equivalent, each replacement is guaranteed

to produce an equivalent expression. Consequently, the final expression is provably equivalent to the first.

Example 13: *Assume that we want to increase the syntactic complexity of $e := (x + y) + z$ with the upper bound $n = 2$. First, we randomly choose the subexpression $x + y$. We then pick another member of the same equivalence class—$(x \oplus y) + 2 \cdot (x \wedge y)$—and replace it in e. In this case, we obtain $e := ((x \oplus y) + 2 \cdot (x \wedge y)) + z$. In a second step, we choose $x \oplus y$, pick the semantically equivalent member $(x \vee y) - (x \wedge y)$ and replace it again. The final MBA-obfuscated expression is $e := (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y)) + z$.*

Empirical testing showed that for an initial expression the randomly picked subexpressions would often be short, causing the resulting recursive rewriting to be very local in nature rather than considering all of the expression. The previous example illustrates this behavior. Considering the expression as an *abstract syntax tree (AST)*, we twice replaced deeper parts of the AST while ignoring the top-level operation (addition with z). Consequently, subsequent iterations would be even less likely to pick the high-level operation, considering the wealth of other operations to pick from. Therefore, the AST would be significantly unbalanced. To avoid this, we prefer selecting top-level operations in the first loop iterations.

4.5 Semantic Complexity: Superoperators

Up to this point, f's core semantics have a rather low semantic complexity (e. g., $x + y$). To thwart semantic attacks, we use a variation of superoperators that increase the semantic complexity. The intention is to significantly increase the search space for an attacker: For example, assume a set of three variables V and a set of six binary operations O: For semantic depth 3 (e. g., $x + y$), an expression contains $m = 2$ variables and $n = 1$ operations, such that an attacker has to brute-force at most $|V|^m * |O|^n = 3^2 * 6^1 = 54$ possibilities. For depth 7 (e. g., $((x + y) \cdot (x \oplus c)))$, they must try up to $3^4 + 6^3 = 17,496$ different expressions (or $314,928$ for depth 9). In other words, the search space grows exponentially, making semantic code book checks as well as program synthesis infeasible.

However, common superoperator strategies, e. g., as used by Tigress [18], are not resilient against these attacks (cf. Section 6.3). They usually include independent core semantics, each having their own output; this causes the handler to have multiple, independent outputs, which an attacker can target individually. As each core semantics itself usually implements only a single operation [7, 58] (e. g., $x + y$ with semantic depth 3) attacking one such superoperator is similar to attacking a series of regular handlers. To avoid this pitfall, we design our superoperators to preserve the signature of f (a *single* output and x, y, c and k as inputs) while providing a high semantic depth. In other words, our superoperators consist of a chain of core semantics that depend on each other and must be executed sequentially: The output of the core semantics is used as input for subsequent core semantics; the last

```
1    d := a + b          ;  d1 := a  + b
2    b := a * d          ;  b1 := a  * d1
3    d := b | d          ;  d2 := b1 | d1
```

Figure 2: Three different core semantics, each implementing a simple operation. On the right-hand side, the SSA form of the respective expressions.

core semantics produces the output of the handler. Even if an attacker is aware of these superoperators, they cannot split a handler into multiple separate synthesis tasks and forces them to synthesize the whole expression.

On a technical level, we construct superoperators based on data-flow dependencies, more precisely *use-definition chains* based on *static single assignment (SSA)* [22]: Given an unprotected code unit in form of instructions in three-address code, we assign a unique variable to each variable definition and replace subsequent variable uses with its latest definition on the right-hand side (called *SSA form*). Then, we build superoperators by first randomly picking variables on the right-hand side and then replacing these *uses* by their respective variable *definitions* recursively. By choosing lower and upper limits for the recursion bound, we can control the superoperators' semantic depth. To further increase the syntactic complexity, we apply our MBAs.

Example 14: *Assume we have three sequential instructions (Figure 2, l. 1-3) implementing semantically simple operations; each represents an individual core semantics. Notably, the first instruction's output serves as input for the second and third. Similarly, the second instruction is an input to the third. To create a superoperator that implements a semantically more complex operation, we transform the code into SSA form, (randomly) pick* b1 *in the third instruction and replace this use by its definition (l. 2), yielding* d2 := (a * d1) | d1. *When picking* d1, *we replace it by its definition (l. 1) accordingly, transforming the expression into* d2 := (a * (a + b)) | (a + b). *While the initial expressions have a semantic depth of 3, the superoperator's depth is 9.*

Intuitively, replacing a *use* by its respective *definition* is guaranteed to preserve the semantics, as variable assignments are immutable in SSA form. Additionally, we prove the rewritten superoperator is equivalent to the original code with symbolic execution.

4.6 Synergy Effects

To summarize, each of our components thwarts specific deobfuscation attacks: MBAs tackle symbolic execution and pattern matching, while the nature of f with its multiple core semantics, selected via a key, prevents taint analysis and backward slicing from removing irrelevant semantics. Further, superoperators increase the semantic complexity, throwing off semantic attacks.

As indicated, especially the combination of our techniques prevents automated deobfuscation attacks: They do not only co-exist but have beneficial synergy effects, which in turn improve the overall resilience of the combination. For example,

our MBAs weaken pattern matching on all levels, including key encodings, and cause the differences between key encoding and core semantics to blur. Besides the syntactic confusion introduced, we can propagate the core semantics into the key encoding and vice versa. For instance, we may use MBAs that extend the key check with the variables x or y using arithmetic identities that do not alter the key check itself. At the same time, MBAs benefit from superoperators given they provide ample opportunity to pick and replace subexpressions.

4.7 Verification of Code Transformations

Obfuscation generally modifies the syntactic representation of code; thus, it is crucial to verify that it does not change the code's semantic behavior. One can achieve this by checking if the transformed code is semantically equivalent to the original one. While this works well for short sequences of instructions (e.g., by using SMT solvers) within a reasonable amount of time, it does not scale to complex programs. In such cases, the industrial state of the art approximates these guarantees by using extensive random testing [43, 86].

For our design, we choose the best applicable verification method to ensure correctness: For individual components, we use formal verification to prove their correctness (cf. Sections 4.3, 4.4, 4.5). To improve the confidence of the correctness of the combination, we use an approach similar to blackbox fuzzing [27, 38], where we compare the I/O behavior of the original and transformed code for a user-configurable number of random inputs, usually ranging from $1,000$ to $10,000$. These are randomly sampled depending on the type expected by the program (e.g., ASCII strings, random 64-bit integers, or known edge cases such as 0 or $0xff..ff$), which needs to be specified by the user. Crucially, we rely on human insight and careful specification of the input domain such that the sampled inputs cover the full program functionality. We apply this fuzzing both on the binary level as well as on the intermediate representation; for the former, we compare the compiled versions of the unprotected and protected programs, while we emulate the program's intermediate representation before and after transformations for the latter. As a consequence of our handlers' interlocked, always-execute nature, we achieve full code coverage and path coverage both on the intermediate representation as well as on the binary level for all handlers needed to represent the original code.

5 Implementation

To evaluate our techniques, we implement a VM-based obfuscation scheme named LOKI on top of LLVM [68] (version 9.0.0) and a code transformation component written in Rust. LOKI consists of ~3,100 LOC in C++ and ~8,700 LOC in Rust. In this scheme, each function $f(x, y, c, k)$ is represented by one of our 510 handlers. In other words, each handler can implement any semantic operation that requires no more than two input variables and one constant. Our handlers support

the inclusion of three to five core semantics (randomly chosen at creation time), which can be addressed by setting k accordingly. Besides these 510 handlers, we have a *VM exit* and a handler managing memory operations. The control flow between handlers is realized as *direct threaded code* [44], i. e., each handler inlines the VM dispatcher. Our VM assumes a 64-bit architecture. Code operating on smaller bit sizes is semantically upcasted to guarantee correctness.

Our approach to obfuscate real-world code consists of three major steps: Lifting, code transformation, and compilation. The *lifting* starts with a given C/C++ input program that contains a specified function to protect. We then translate this function to LLVM's intermediate representation (IR) and use various compiler passes to optimize the input and unroll loops as our prototype does not support control-flow to reduce engineering burden. Note that this is no inherent limitation of our approach, but a simplification we made as LLVM's passes sufficed in creating binaries that our prototype implementation can process. Finally, we lift the resulting LLVM IR to a custom IR which the code transformation component internally operates on. This component (a) parses the lifted representation of the targeted function, (b) creates superoperators based on this input (with recursion bound 3 to 12), (c) instantiates the VM handlers, applies our obfuscation techniques (e. g., MBAs), and verifies them. For MBAs, we use a random recursive expression rewriting bound between 20 and 30. We choose from a pre-computed database of $843,467$ MBAs (all expressions up to a depth of 9), split over 48 equivalence classes. In each class, there are roughly 17,500 entries on average. To exemplify the dimensions: An attacker has to try up to $n^m_{Loki} = 843,467^{30} = 6.1 * 10^{177}$ possibilities to simplify our MBAs; Based on our reverse engineering efforts, state-of-the-art obfuscator TIGRESS features only 47 hand-crafted rules (that are not applied recursively), such that an attacker has to evaluate $n^m_{Tig} = 47^1 = 47$ possibilities. (d) Finally, the Rust component generates the VM bytecode and translates the handlers back into LLVM IR. Then, obfuscated and original code are compiled with `-O3` and verified.

6   Experimental Evaluation

Based on our prototype implementation, we evaluate if our approach can withstand automated deobfuscation techniques (*resilience*), while maintaining *correctness* and imposing only acceptable overhead (*execution cost*). Overall, we follow the evaluation principles outlined by Collberg et al. [20].

All experiments were performed using Intel Xeon Gold 6230R CPUs at 2.10 GHz with 52 cores and 188 GiB RAM, running Ubuntu. Our obfuscation tooling uses LLVM [68] (v. 9.0) and the SMT solver Z3 [52] (v. 4.8.7). For tracing coverage, we rely on Intel Pin [39] (v. 3.23). Our deobfuscation tooling is based on MIASM [11] (commit 65ab7b8), TRITON [61] (v. 0.8.1), and SYNTIA [7] (commit e26d9f5). We use our prototype of LOKI and the academic state-of-the-art obfus-

cator, TIGRESS [18] (v. 3.1), to obfuscate five different programs, each implementing a cryptographic algorithm: AES, DES, MD5, RC4, and SHA1. This is a common approach: the first three are based on an obfuscation data set provided by Ollivier et al. [54]; the others are adapted from reference implementations [9, 70]. These algorithms are representative for real-world scenarios in which cryptographic algorithms are used to guard intellectual property (e. g., hash functions used for checksums in commercial DRM systems) [53]. In a case study, we obfuscate VLC's DVD decryption routine to show how LOKI can be applied onto real-world use cases. Where necessary, we adapt the programs slightly to allow LOKI to process them (cf. Section 5) without modifying their functionality. TIGRESS' configuration [62] resembles our design and works on the same source code files.

6.1   Benchmarking

Our goal is to benchmark the *correctness* and *cost* of our obfuscator. We do so by conducting a series of experiments, measuring the overhead in terms of runtime and disk size as well as verifying the correctness of transformed code. For each obfuscator, we create $1,000$ obfuscated instances for each of the five targeted programs and use them for all experiments. The overhead comparison is given as factor relative to the original, unobfuscated program compiled with `-O3`. To measure the MBA overhead, we create another $1,000$ obfuscated instances without any MBAs for LOKI.

Experiment 1: Correctness. *For each target, we verify that all $1,000$ obfuscated instances produce the same output as the original program for more than $1,000,000$ inputs. To obtain a uniform distribution over varying input lengths of our cryptographic targets, we create $10,000$ random inputs for each supported input length $l \in [16; 128]$. Additionally, we test a number of edge cases $\in \{0x0..0, 0xff..ff, 0x80..00, 0x00..01, 0xaa..aa, 0x55..55\}$ (or their cartesian product if two inputs are required). This amounts to a total of $1,134,068$ inputs, for which we assert equal input-output behavior.*

All obfuscated binaries (both those with and without MBAs) exhibit exactly the same behavior for the $1,134,068$ inputs tested.

Experiment 2: Code Coverage and Path Coverage. *To further increase confidence in our correctness tests, we measure both the code coverage and the path coverage that the inputs from Experiment 1 achieve on the to-be-protected code both for the original program and obfuscated instances.*

We find that each of the more than $1,000,000$ inputs from Experiment 1 achieves full code coverage and full path coverage. This ensures that our inputs cover the complete behavior of the code we obfuscate and that our obfuscation transformations have not altered this behavior.

Experiment 3: Overhead. *For each target, we measure the average execution runtime. To this end, each target wraps*

the to-be-protected code in a single function, which is called 10,000 *times per input. We then execute each obfuscated binary for* 1,000 *random inputs, recording the collected timings. We also compare the original program's disk size to the average of the obfuscated binaries.*

As evident from Table 2, the runtime overhead ranges from a factor of 301 to 482 compared to the original program's execution time. While this overhead may appear excessive—also in comparison to TIGRESS—state-of-the-art commercial obfuscation generally imposes an even larger slowdown, up to ten times more than LOKI (cf. Table 2, [15]). We re-run this experiment on the 1,000 binaries without MBAs to evaluate their impact. On average, they are responsible for ~39% of the overhead. Similar for the disk size, the obfuscated programs are 18 to 51 times larger than the original ones. Size-wise, MBAs cause ~33% of the overhead. For further details of our MBAs' overhead, we refer to the Technical Report [62]. Compared to THEMIDA and VMPROTECT, our obfuscating transformations generate almost always smaller programs, while TIGRESS always produces significantly smaller binaries.

Overall, we conclude that our overhead is moderate in comparison to commercial state-of-the-art obfuscators. For further discussion, we refer to Section 7. TIGRESS' overhead is impressively small, but it falls short in providing comprehensive protection as the following experiments show.

Case Study: VLC with LIBDVDCSS. To showcase the practical feasibility of LOKI in real-world scenarios, we obfuscate the DecryptKey function in LIBDVDCSS [71]; this component of VLC [72] is responsible for decrypting the multimedia content of DVDs keys. The underlying idea is to protect the decryption algorithm from the prying eyes of crackers and protect intellectual property. However, the vast majority VLC's code is irrelevant to content decryption, such that there is no need to obfuscate the whole LIBDVDCSS library or even the whole media player. After obfuscating the DecryptKey function, which is called *before* the actual media content is played, we measure the execution time of the function during initial startup, when the DVD is decrypted. We average the results over ten executions. We find that without obfuscation, the function is executed in 2,952 nanoseconds, while with obfuscation, the decryption lasts 937,606 nanoseconds. Overall, LOKI slows down the initialization by one millisecond, a negligible cost for protecting one's intellectual property, especially if the to-be-protected function is only called in the application's startup phase.

6.2 Resilience

We evaluate whether our techniques can withstand automated deobfuscation approaches. To this end, we analyze the impact of syntactic and semantic attacks against the obfuscated code in the presence of both static and dynamic attackers. We design all experiments by assuming the strongest attacker model. To this end, we test each component individually, therefore ignoring beneficial synergy effects. Where applicable, we first evaluate our techniques on a general design level before testing their concrete implementations. The former serves as universal evaluation of a technique's resilience, while the latter demonstrates that this also holds when actually implemented on the binary level.

LOKIATTACK. Fundamentally, attacking the obfuscated VM on the binary-level has two stages: (1) Identifying a specific handler within the VM, and (2) attacking (simplifying) this particular handler as far as possible. For our evaluation, especially (2) is interesting, as all our techniques focus on hardening individual handlers. As such, we develop a custom attack framework that we call LOKIATTACK. It is specifically tailored to the attacked obfuscators and automates the first stage: It identifies all VM handlers and provides the attacker (for each handler) with access to the handler parameters (x, y, c, and—for LOKI—k). For a dynamic attacker, it also provides concrete values for these parameters. Finally, LOKIATTACK uses symbolic execution to obtain all code paths through the intermediate representation (IR) of the O3-optimized VM code that depend on an *unknown* (static attacker) or *known* (dynamic attacker) value of k (for LOKI). For each such path, an attacker can launch the actual attack on the handler (stage 2), for which LOKIATTACK provides a number of techniques implemented as plugins, e. g., taint analysis, symbolic execution, or program synthesis. To implement LOKIATTACK, we use MIASM; the plugins for stage 2 are based on TRITON (byte-level taint analysis), MIASM (bit-level taint analysis, backward slicing, and symbolic execution), and SYNTIA (program synthesis). These plugins include costly operations (SMT solving, program synthesis, and symbolic execution), from which some may run for several days. As our evaluation consists of more than 300,000 analysis tasks, we limit each one to 1 hour to keep the analysis time manageable. This is a common use-case and in-line with previous work on deobfuscation [5,7,51].

6.3 Evaluation of Key Encodings

We evaluate whether a static attacker can obtain a specific core semantics using semantic codebook checks. Note this experiment is only applicable to LOKI as TIGRESS has no concept of key-based selection of core semantics. Assume that the function $f(x,y,c,k)$ includes $x+y$ as one of its core semantics. Then, an attacker can use an SMT solver to find a value for k such that f is semantically equivalent to $g(x,y,c) := x+y$. On a technical level, we employ an approach called *Counterexample-Guided Abstraction Refinement (CEGAR)* [16,17] that relies on two independent SMT solvers: While SMT solver A tries to find assignments for all variables (including k) such that f and g produce the same output, solver B tries to find a counterexample for this value of k such that f and g behave differently. Then, A uses the counterexample as guidance.

Table 2: Runtime and disk size overhead as factors relative to the non-obfuscated binaries (compiled with 03). *(w/o = without)*

		Time Factor					Size Factor				
		AES	DES	MD5	RC4	SHA1	AES	DES	MD5	RC4	SHA1
VMPROTECT	Virtualization	2,489	1,859	1,982	1,321	2,524	37	21	40	44	40
	Ultra	8,925	9,152	13,047	5,806	15,411	47	37	57	59	53
THEMIDA	Tiger White	1,388	622	203	240	552	58	38	58	58	59
	Dolphin Black	11,695	5,052	2,428	3,634	8,354	67	47	85	63	84
LOKI		386	301	357	482	386	33	18	39	37	51
	w/o MBA	236	185	204	315	233	21	13	25	26	32
TIGRESS		261	51	101	58	111	3	4	2	2	3

Experiment 4: Hardness of Key Encodings. *We generate* $1,000$ *random instances of our factorization-based key encoding and synthesize* $10,000$ *point functions. Then, we apply the CEGAR approach independently to both key encodings and check if the SMT solver finds a correct value for k.*

We observe that the SMT solver found no correct key for the factorization-based encoding, but hit the 1h timeout in all cases. Considering the point function-based key encoding, Z3 managed to find a value for k in $6,932$ cases (~69%). On average, it found the solution in 284s (excluding timeouts). We conclude that an SMT solver struggles with our factorization-based key encoding, while point functions often can be solved. Recall though that point functions primarily serve to diversify and erase discernible patterns to impede pattern matching.

Experiment 5: Key Encoding on Binary Level. *To verify if our implementation properly emits these key encodings, we generate* $1,000$ *binaries that contain one specific handler which includes $x+y$ as one of its core semantics. These binaries contain neither MBAs nor superoperators. Assuming a static attacker uses CEGAR, we check in how many cases the SMT solver finds a correct value for k.*

Using LOKIATTACK, we obtain the handler's instructions and use our CEGAR plugin based on Z3 to find a value for k, such that these instructions are semantically equivalent to $x+y$. While hitting the timeout in 690 cases, Z3 managed to find a correct value for k in 310 cases (31%). The SMT solver needed, on average, 444s to find the solution (excluding timeouts). Overall, we conclude that our key encodings indeed pose a challenge for a static attacker relying on SMT solvers.

Note that this component is special within our system, as its approach specifically targets only static attackers. This is due to the fact that dynamic attackers can trivially observe a value for k. While a dynamic scenario is not always possible, another attack vector could be to offload 64-bit integer factorization to custom tools (assuming an attacker manages to locate the key encodings, which in itself is a non-trivial task given our MBAs and point functions). Thus, our key encodings can be considered to be our weakest component. However, our design assumes that an attacker can retrieve a value for k, but we try to make this as hard as possible. The syntactic simplification experiments show that knowledge of a key k is beneficial but not sufficient to simplify any handler.

Table 3: Statistics for backward slicing and forward taint analysis (TA), averaged over 7,000 handlers. Unmarked instruction can be removed as irrelevant. *(Unmark. = not tainted / not sliced; Dyn. = dynamic attacker)*

		Byte-level TA		Bit-level TA		Slicing	
		Static	Dyn.	Static	Dyn.	Static	Dyn.
LOKI	IR paths	1,950	199	1,451	168	1,656	179
	Unmark.	17.49%	17.50%	17.61%	17.62%	5.49%	7.57%
	Time [s]	556	58	710	78	630	67
TIGRESS	IR paths	1		1		1	
	Unmark.	44.70%		44.70%		22.35%	
	Time [s]	1.3		1.6		1.4	

Syntactic Simplification. In the following, we evaluate whether syntactic simplification techniques—namely, forward taint analysis, backward slicing, and symbolic execution—succeed in extracting a core semantics associated with a specific key, either by trying to identify instructions not contributing to a function f's output or by symbolically simplifying f. We use LOKIATTACK as a basis and conduct the respective attack in stage 2 for both a static and a dynamic attacker. We assume that an attacker is given a binary containing seven handlers, $f_0(x,y,c,k), \cdots, f_6(x,y,c,k)$, each containing between 3 and 5 core semantics. Further, each handler f_i contains one predefined core semantics from the set $\{x+y, x-y, x \cdot y, x \wedge y, x \vee y, x \oplus y, x \ll y\}$ that an attacker wants to identify via syntactic simplification. As sample set for our experiments, we generate $1,000$ binaries protected by MBAs but without superoperators, amounting to $7,000$ handlers to analyze. For each binary, we use LOKIATTACK to extract all handlers; for each handler, LOKIATTACK provides us with the parameter locations (and values for the dynamic scenario) and all code paths. For each code path (a list of instructions), we then use the respective stage 2 plugin. We apply the following experiments also to 7,000 TIGRESS handlers (with disabled superoperators), respectively.

Experiment 6: Forward Taint Analysis. *For each of the* $7,000$ *handlers, we conduct a forward taint analysis with byte-level and bit-level granularity. The former is based on* TRITON, *while the more precise bit-level taint analysis is implemented on top of* MIASM. *In general, higher precision is expected to produce fewer false positives and result in fewer tainted instructions. Recall that an attacker's goal is to identify all instructions that do not belong to the core semantics associated with a specific key. Using taint analysis,*

an attacker can remove all instructions not depending on x, y, c, or k (in a dynamic scenario: on a concrete *value for k).*

The resulting data is shown in Table 3 (where *unmarked* instructions refer to instructions that are *not* tainted, i. e., instructions that can be removed). The results show two interesting insights: First, the granularity has negligible impact on the results (difference of 0.12%). Second, the number of tainted instructions is almost equal for a static and a dynamic attacker. This is surprising as for LOKI the number of visited paths in the IR's control-flow graph is significantly lower in the dynamic setting. Intuitively, this means a dynamic attacker has better chances of removing more instructions. However, our results show that the sole benefit of a dynamic attacker is spending less time per handler. In numbers, an attacker is always able to only remove about ~18% of a handler's assembly instructions. Manually inspecting the instructions not tainted revealed that these can always be put into two categories: Either they are part of the inlined VM dispatcher that is responsible for loading the next handler (which is independent of the current handler's semantics), or it is an instruction loading a constant value before it interacts with tainted instructions (comparable to Example 3). To summarize, forward taint analysis fails to remove a single instruction that is related to the core semantics or key encodings. For TIGRESS, on the contrary, only one IR path exists, meaning the handlers are short and simplistic in nature. No difference between bit and byte-wise taint analysis exists; overall, an attacker succeeds in removing 45% of instructions—significantly more than for LOKI.

Experiment 7: Backward Slicing. *Besides forward taint analysis, an attacker can use backward slicing to identify all instructions that contribute to a handler's output. We again consider both a static and dynamic attacker trying to reduce each handler to the core semantics associated with a specific k by removing as many unrelated instructions as possible. Our backward slicing approach is based on* MIASM *and operates on the same* 7,000 *handlers as Experiment 6.*

The results are denoted in Table 3, where an *unmarked* instruction refers to an instruction that was not sliced, i. e., it does not contribute to the output. Other than for taint analysis, a dynamic attacker has a slight advantage compared to a static attacker (2.08%), as they slice slightly fewer instructions. While the static attacker marks all instructions but the inlined dispatcher, our manual inspection revealed that dynamic analysis skips some IR paths depending on irrelevant key values. Compared to forward taint analysis, backward slicing marks more instructions, i. e., it removes fewer instructions (~7% vs. ~18%). This is due to the backward-directed nature of the approach, which allows it to slice instructions loading constant values. We conclude that backward slicing is technically more precise than forward taint analysis, but fails to remove instructions belonging to the core semantics or key encodings. For TIGRESS, slicing succeeds to remove signif-

Table 4: Symbolic execution for semantic depth 3 and 5, each averaged over 7,000 handlers. *(Simplified = percentage of handlers simplified)*

		Depth 3		Depth 5	
		Static	Dynamic	Static	Dynamic
LOKI	IR paths	4,960	559	5,450	703
	Simplified	0%	17.93%	0%	14.64%
	Time [s]	–	94	–	168
TIGRESS	IR paths	1		–	
	Simplified	30.61%		–	
	Time [s]	1.4		–	

icantly more instructions, however, less than taint analysis. This is again due to the loading of constant values.

Experiment 8: Symbolic Execution. *Besides removing instructions not contributing to the output, an attacker can use symbolic execution to extract a handler's core semantics. To this end, a symbolic executor uses simplification rules to syntactically simplify the handler's semantics as much as possible. We use the same* 7,000 *handlers as Experiment 6. We analyze each handler independently with* MIASM*'s symbolic execution engine and measure whether it can be simplified to the original core semantics.*

We model both a static and more powerful dynamic attacker. In the latter scenario, the attacker observes a value for k and thus can nullify all core semantics not related to this specific k. Hence, they obtain a much simpler expression containing only the desired core semantics, albeit in syntactically complex form (due to MBAs). Recall that for the 7,000 handlers, the semantic depth of the core semantics is always 3 (e. g., $x + y$). This intentionally weakens resilience, as superoperators with a higher depth naturally increase both semantic and syntactic complexity. To show this, we create another 7,000 handlers (1,000 binaries à 7 core semantics) with a semantic depth of 5 (e. g., $x + y + c$) and repeat this experiment. We cannot create handlers of depth 5 for TIGRESS, as it is not possible to explicitly set the handler's semantic depth.

All results are shown in Table 4. Notably, a static attacker fails to simplify any of LOKI's handlers. Without knowing a value for k, an attacker has to analyze the expression containing *all* key encodings and their associated core semantics. In other words, an attacker fails to nullify irrelevant core semantics. To significantly simplify the handler, a static attacker has to find a valid key first (reducing the problem to Experiment 5). A dynamic attacker, on the other hand, only has to simplify the MBAs as they already identified the core semantics associated with the key. For depth 3, they succeed in removing all MBAs for ~18% of LOKI's handlers, while, for TIGRESS, ~31% of the handlers can be simplified. In other words, an attacker can simplify significantly more handler for TIGRESS than for LOKI. For a more realistic depth of 5— subsequent experiments show ~80% of LOKI's handlers are at least of depth 5—the attacker's success rate is even lower, namely ~15%. This percentage implies that a number of expressions can be simplified regardless of the higher base depth. This may be the case, e. g., when the random combination of

applied MBAs cancels itself out. Still, this demonstrates our synergy effects are indeed helpful to prevent an attacker from symbolically simplifying the core semantics, leaving them with a complex MBA that conceals the actual semantics.

We conclude that our MBAs are successful in thwarting symbolic execution, one of the most powerful deobfuscation attacks. For a more detailed analysis of how a user of LOKI can trade performance against reducing the attacker's success chances even further (to 6.79%), refer to the Technical Report [62].

Experiment 9: Diversity of MBAs. *An attacker tasked with removing such MBAs may investigate whether a diverse number of expressions exists for the same core semantics. If this is not the case, they can manually analyze each MBA and extend the symbolic executor's limited set of simplification rules by rewriting rules to "undo" specific MBAs. To this end, we assume a dynamic attacker that already symbolically simplified the expression as far as possible without any MBA-specific simplification rules. We do this for each handler type (recall that the 7,000 handlers of depth 3 consist of 7 different core semantics à 1,000 handlers) and then analyze how many different, unique MBA expressions exist.*

Our analysis reveals that, in summary, LOKI generates 5,482 unique MBAs for the 7,000 expressions analyzed (78.31%), while TIGRESS creates only 16 (~0.23%) unique MBAs. Thus, an attacker adding 16 rules to their symbolic executor could simplify all core semantics. This difference can be explained by the fact that TIGRESS uses only a few handwritten rules to create MBAs, while LOKI features a generic approach to synthesize MBAs. To further highlight the difference between both approaches, we repeat this experiment for another set of 7,000 handlers—created in the same configuration but with different random seeds—and calculate the intersection of unique MBAs. TIGRESS re-uses exactly the same 16 MBAs, while LOKI re-uses 109 expressions but generates 5,299 new unique MBAs (i. e., 10,781 unique MBAs in total). Creating simplification rules specific to LOKI is a tedious task (given the high number of unique MBAs) that does not pay off when analyzing other obfuscated instances. For a discussion of what an attacker can achieve when they are in possession of *all* available MBA rewriting rules, refer to Section 7. We conclude that LOKI's MBAs are superior to state-of-the-art approaches relying on a small number of hardcoded MBAs, both in terms of resilience and diversity.

State-of-the-art MBA Deobfuscation. A number of approaches for MBA simplification have been proposed, most notably SSPAM [30], ARYBO [36], NEUREDUCE [32], and MBA-BLAST [48]. The deployed techniques range from pattern matching-based simplification over machine learning to algebraic simplification. Regardless of the underlying technique, they all share one major drawback: They expect the MBAs to be available on the source code level in form of a formula, such as "x + y - y", rather than dealing with them

on the binary level. As a consequence, these deobfuscation tools lack support for MBAs using different bit sizes and operations such as zero-extension or sign-extension. Furthermore, they assume that the MBAs are free of constants and more complex arithmetic operations, such as multiplication or left-shifts. In contrast to these limitations and assumptions, LOKI's MBAs not only employ all these operators but also contain constants, thus making a fair, direct evaluation of our MBAs contained in binaries difficult. To avoid these pitfalls, we use LOKI's term rewriter to generate simpler MBAs—called *artificial* MBAs—and emit them as a formula rather than deploying them in the binary. We make the following artificial restrictions: We (a) emit no constants, (b) do not intertwine the MBAs with the key encoding, (c) remove any information (or operation) relating to size casts, and (d) avoid complex, unsupported operations such as multiplications or left-shifts. Instead, the resulting MBAs are a formula containing only the following operations $\{+, -, \wedge, \vee, \oplus\}$. While this significantly weakens LOKI's MBAs, aforementioned state-of-the-art MBA deobfuscation techniques can now process these artificial MBAs, allowing a fair evaluation. For NEURE-DUCE, we use the Gated Recurrent Unit (GRU)-based Long Short-Term Memory (LSTM) model provided by the authors. We further adapt MBA-BLAST to recursively attempt simplification for subexpressions. MBA-BLAST cannot deal with nested arithmetic expressions; only expressions on the root level of the expression's abstract syntax tree may contain arithmetic operators. All subexpressions must consist purely of Boolean operators. As our MBAs are highly nested, we apply the respective tool recursively on each subexpression until it cannot simplify the expression any longer. We considered evaluating ARYBO [36]; however, we noticed it does not terminate for 64-bit expressions within one hour. Further, ARYBO outputs truth tables in form of expressions representing the relations between different bit positions. Its goal is aiding a human analyst rather than automated simplification. Thus, we exclude it from the following experiment. As a baseline, we port our deobfuscation tooling, LOKIATTACK with the SE plugin, to the source level: We first use aggressive compiler optimizations ("-03") to simplify the MBA and then—as a stage 2 plugin—symbolically summarize it using MIASM's symbolic execution engine. This is the same approach as has been used for the previous experiments.

Experiment 10: MBA Formula Deobfuscation. *For each core semantics from the set $\{x + y, x - y, x \wedge y, x \vee y, x \oplus y\}$, we use LOKI to generate 1,000 artificially simplified MBAs on the source code level. We do this for each recursive term rewriting bound from $[1, 30]$ (during normal operation, LOKI's rewriting bound is randomly chosen between $[20, 30]$). In summary, we generate 5,000 MBAs per rewriting bound, i. e., 150,000 obfuscated expressions in total. We then pass each MBA to the deobfuscation tools MBA-BLAST, NEURE-DUCE, SSPAM, and LOKIATTACK and observe how many expressions they can simplify to the ground truth.*
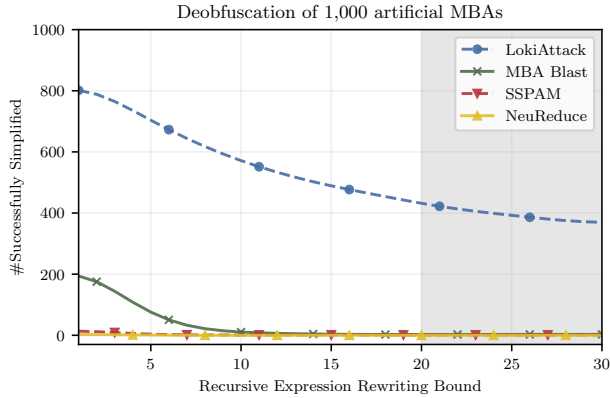
Figure 3: Number of artificial MBAs that have been successfully simplified (averaged over 5 different core semantics). The gray-shaded area marks the recursive rewriting bounds randomly picked by LOKI for our regular MBAs.

Figure 4: The distribution of core semantics with and without superoperators.

The number of simplified expressions, averaged over the five different core semantics, are depicted in Figure 3. As the data shows, our custom deobfuscation tooling, LOKIATTACK, significantly outperforms all state-of-the-art deobfuscation techniques. NEUREDUCE can only simplify a handful of expressions in total. One limitation is that it can only work with inputs up to 100 characters; however, our artificial MBAs with a rewriting bound of 20 contain, on average, 7,960 characters (after removing all whitespaces). We have tried a similar approach as we have employed for MBA-BLAST, however, found it does not improve its accuracy. Studying their dataset used to train the model, we believe that their approach heavily overfits on the training data, a set of simple and short (on average 75 characters without whitespace) MBAs. SSPAM fails to deal with the highly recursive nature of our MBAs, frequently hitting the stack recursion limit. MBA-BLAST performs better and manages to simplify a number of simple MBAs. However, the success rate of all tools decreases with a higher term rewriting bound. LOKI's default is to use a random recursive rewriting bound between 20 and 30, for which all but LOKIATTACK fail to simplify basically any MBA. For example, MBA-BLAST simplifies only 157 of 55,000 MBAs for LOKI's recursive rewriting bounds, [20, 30]. While the success rate of LOKIATTACK may seem high, recall that we artificially weakened these MBAs by excluding a number of operations and removing all constants; Experiment 8 evaluates how LOKIATTACK with the symbolic execution plugin performs on our regular MBAs.

Semantic Attacks. Semantic attacks such as program synthesis exploit the low semantic depth of individual core semantics. We evaluate the impact of our superoperators on these attacks. First, we analyze the average semantic complexity of core semantics with and without superoperators. Then, we perform a high-level experiment to measure the general limits of synthesis-based approaches. Finally, we demonstrate that our superoperators withstand synthesis-based attacks on the binary level. Note that we only consider a dynamic attacker
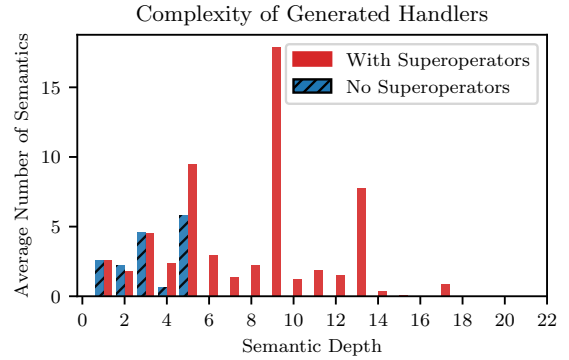
in the following, as knowing a value for k is a prerequisite for any reasonable semantic attack. A static attacker would only learn random behavior, as the key encodings are only valid for a predefined set of keys.

Experiment 11: Complexity of Core Semantics. *To evaluate our superoperators' distribution and their impact on the complexity of core semantics, i.e., their semantic depth, we create* 1,000 *obfuscated binaries without* superoperators *as a baseline for each benchmarking target (cf. Section 6.1) and* 1,000 *binaries with* superoperators. *We compare the two sets on the average number of unique core semantics and their semantic depths. To simplify evaluation, no MBAs are used.*

Without superoperators, each binary on average contains 15.8 core semantics. With superoperators, this number increases to 58.8. Additionally, Figure 4 shows that superoperators have a significantly higher semantic depth, usually ranging from 5 to 13 with a clearly visible peak at depth 9. Compared to obfuscation without superoperators, where only a few core semantics with semantically low complexity are used, superoperators increase the number of unique core semantics and their semantic depth notably. This makes the task of synthesizing semantics more difficult.

Experiment 12: Limits of Program Synthesis. *We evaluate how the success rates of program synthesis relate to semantic complexity. We generate* 10,000 *random expressions for each semantic depth between* 1 *and* 20 *and measure how many of them can be synthesized successfully. Modeling our function* f, *we use* SYNTIA*'s grammar [7] to generate random expressions depending on three variables. Based on the authors' guidance, we set* SYNTIA*'s configuration vector to* (1.5, 50000, 20, 0) *and use it to synthesize each expression.*

Figure 5 shows that simple expressions can be synthesized quite easily; at a semantic depth of 7, only ~50% can be synthesized. For larger semantic depths, it becomes increasingly unlikely to synthesize expressions. Given our results from Experiment 11, we conclude that our superoperators produce core semantics of sufficient depth to impede program synthesis.
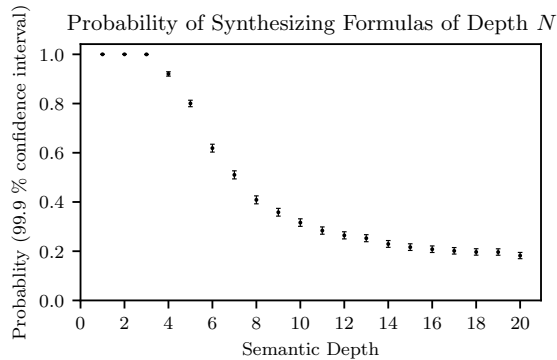
Figure 5: The probability to synthesize a valid candidate for formulas of depth *N*. The error bars are calculated as the 99.9% confidence interval for the true probability.

Experiment 13: Superoperators on Binary Level. *To evaluate the impact of program synthesis, we assume a dynamic attacker has extracted a handler's core semantics (for* LOKI*: associated with a known value of k). They then use* SYNTIA—*configured as in Experiment* 12—*to learn an expression having the same input-output behavior. We create* 400 *obfuscated binaries (without MBAs, with superoperators) for each benchmarking target (cf. Section* 6.1*), randomly pick* 10, 000 *core semantics and measure* SYNTIA*'s success rate.*

Overall, using SYNTIA as a stage 2 plugin on top of LOKI-ATTACK, we managed to synthesize 1, 888 (~19%) of LOKI's expressions and 6, 779 (~68%) of TIGRESS' expressions. On average, it took 157s to synthesize an expression for LOKI and 144s for TIGRESS. The results show that—while both designs employ superoperators—it is crucial how these superoperators are crafted. As outlined in Section 2.1, TIGRESS usually includes independent core semantics, allowing the attacker to split the superoperators into multiple smaller synthesis tasks, each of low semantic complexity. On the other hand, LOKI's design ensures that its superoperators cannot be split into smaller tasks but have high semantic depths. In summary, LOKI is the first obfuscation design showing sufficient protection against program synthesis, an attack vector all state-of-the-art obfuscators fail to account for. Given LOKI's synergy effects and high resilience against syntactic simplification approaches, semantic deobfuscation techniques remain an attacker's last resort. However, even when using program synthesis, arguably the strongest semantic attack, an attacker can only recover less than a fifth of LOKI's core semantics.

7 Discussion

Overhead. Table 2 indicates that the overhead of code obfuscation is generally excessive. However, this cost is accepted in practice because only small, critical parts of the whole program need to be protected (e. g., proprietary algorithms, API accesses, or licensing-related code). As a result, the overhead has to be seen in relation to the whole program. As our case study shows for LIBDVDCSS, using obfuscation only for crit-

ical, well-chosen code parts has no negative impact on the usability of the respective program (here VLC).

MBA Database. Assuming an attacker intends to symbolically simplify MBAs, they may benefit from using a lookup table mapping complex MBAs to simpler expressions. This approach is effective for state-of-the-art obfuscators such as TIGRESS that only use a limited number of hardcoded rewriting rules (cf. Experiment 9). LOKI, in contrast, is the first obfuscator that employs a generic approach to synthesize highly diverse MBAs, resulting in a large number of MBAs (stored in a database for performance reasons). Users of LOKI can keep their MBA database (including the synthesis limit up to which MBAs were synthesized) private. In fact, users could choose arbitrary lower and upper limits as well as completely different grammars to create an MBA database. Without knowing the parameters, a re-creation of the database is not feasible. That said, even in cases where an attacker is in possession of the MBA database, there is no straightforward process to reverse the recursively generated expression (cf. Section 4.4).

Attacker Model. Our evaluation assumes a strong attacker model with significant domain knowledge and access to all kinds of static and dynamic analyses. In practice, an attacker is often weaker. Especially without prior knowledge about the given obfuscation techniques, the usage of additional techniques (e. g., VM bytecode blinding [7] or range dividers [1]) and other countermeasures (e. g., self-modifying code [50] or anti-debugging techniques [33]) complicates analysis.

Human Attacker. Ultimately, code obfuscation schemes are usually broken by human analysts [58]. This is partly because humans excel at recognizing patterns and adapt to the given obfuscation [23]. Collberg et al. [19] define *potency* to denote how *confusing* an obfuscation is for a human analyst. Due to the difficulty of measuring a human's capability with regard to deobfuscation, we restrict our evaluation to automated attacks. We argue that without automated techniques, analysis becomes subjectively harder. Nevertheless, we believe that pattern matching might be the most potent attack on our approach. While we use a fixed structure, we argue that our MBAs remove identifiable patterns. Still, we are not aware of an adequate way of measuring this. However, even if we assume that a human attacker breaks one obfuscated instance, other instances remain hard. This is as our design samples from large search spaces for its critical components, providing significant diversity for MBAs and superoperators. In summary, we expect LOKI to perform reasonably well against human attackers even if this cannot be easily quantified.

8 Related Work

Over the years, a large number of obfuscation techniques were proposed [1, 8, 19, 31, 35, 40, 50, 53, 54, 56, 74, 77, 87]. Many of these techniques are orthogonal to our work and focus on one specific transformation. For an overview over the field

of obfuscation, we refer the interested reader to the overview by Banescu and Pretschner [3]. In the following, we discuss techniques closest to our work.

MBA. Zhou et al. introduced the concept of Mixed Boolean-Arithmetic (MBA) to hide constants and calculations within complex expressions. While conceptually simple, this approach proved effective against many analysis techniques, such as symbolic execution. As a consequence, a number of approaches towards deobfuscating MBAs were proposed, including pattern matching (SSPAM [30]), symbolic simplification using a Boolean expression solver (ARYBO [36]), program synthesis (SYNTIA [7], XYNTIA [51], QSYNTH [25]), machine learning (NEUREDUCE [32]), and algebraic simplification (MBA-BLAST [48]). While those techniques are effective against common MBAs, LOKI's generic approach to synthesize diverse MBAs produces expressions resilient against such attacks (cf. Section 6).

VM Obfuscation. Our prototype implementation LOKI uses a VM-based architecture to showcase our techniques. However, we make no attempt at obfuscating the VM structure itself, which we consider orthogonal to our work. Examples for such work include *virtual code folding*, where the mapping between opcodes and individual handlers is obfuscated to impede static analyses [14, 46, 67, 81]. While they use dynamic keys to determine the next handler, we use keys within our handlers to select a specific core semantics. With regard to deobfuscation, approaches such as VMHUNT [79], VMATTACK [41], and others [42, 64] may succeed in reconstructing LOKI's VM structure (similar to LOKIATTACK). However, they cannot recover individual handler semantics, since they rely on techniques such as symbolic execution and backward slicing, for which LOKI is resilient against by design.

Thwarting Symbolic Execution. With regard to thwarting symbolic execution-based deobfuscation approaches, early work by Sharif et al. [65] already proposed key-based encodings to make path exploration infeasible. Later approaches extend on this work by introducing multi-level opaque predicates (so-called *range dividers*) [1] or artificial loops [54]. LOKI extends these ideas: it does not only make path exploration infeasible, but also prevents symbolic simplification attacks due to its MBAs.

Thwarting Program Synthesis. Program synthesis is one of the most powerful attack vectors [7, 51]. Concurrent work [51] proposes a search-based program synthesis approach outperforming SYNTIA. However, the authors note that merging handlers and increasing a handler's semantic complexity proved effective in thwarting such attacks. This is in line with our evaluation.

9 Conclusion

In this paper, we present and extensively evaluate a set of novel and generic obfuscation techniques that, in combination, succeed to thwart automated deobfuscation attacks.

References

[1] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code Obfuscation against Symbolic Execution Attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

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

[3] Sebastian Banescu and Alexander Pretschner. A Tutorial on Software Obfuscation. *Advances in Computers*, 108:283–353, 2018.

[4] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *ACM Sigplan Notices*, 2006.

[5] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *IEEE Symposium on Security and Privacy*, 2017.

[6] Lucas Barhelemy, Ninon Eyrolles, Guenaël Renault, and Raphaël Roblin. Binary Permutation Polynomial Inversion and Application to Obfuscation Techniques. In *ACM Workshop on Software PROtection (SPRO)*, 2016.

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

[8] Pietro Borrello, Emilio Coppa, and Daniele Cono D'Elia. Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation. In *Conference on Dependable Systems and Networks (DSN)*, 2021.

[9] Jasin Bushnaief. SHA-1. https://gist.github.com/rverton/a44fc8ca67ab9ec32089, 2016.

[10] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-Taint-Analysis: Practical Evasion Techniques against Information Flow-based Malware Defense. Technical report, Secure Systems Lab, Stony Brook University, 2007.

[11] CEA IT Security. Miasm – Reverse Engineering Framework. https://github.com/cea-sec/miasm.

[12] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. Advanced or Not? A Comparative Study of the Use of Anti-Debugging and Anti-VM Techniques in Generic and Targeted Malware. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 323–336, 2016.

[13] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Conference on Dependable Systems and Networks (DSN)*, pages 177–186. IEEE, 2008.

[14] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. DynOpVm: VM-based Software Obfuscation with Dynamic Opcode Mapping. In *International Conference on Applied Cryptography and Network Security*, 2019.

[15] Christian Collberg. Performance vs. Security. https://tigress.wtf/blog.html.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *International Conference on Computer Aided Verification*, 2000.

[17] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[18] Christian Collberg. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/.

[19] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[20] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[21] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-Based Approach. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[22] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1989.

[23] B. Dang, A. Gazet, E. Bachaalany, and S. Josse. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley, 2014.

[24] Sebastian Danicic and Michael R. Laurence. Static Backward Slicing of Non-Deterministic Programs and Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(3):11:1–11:46, 2018.

[25] Robin David, Luigi Coniglioi, and Mariano Ceccato. QSynth – A Program Synthesis based Approach for Binary Code Deobfuscation. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2020.

[26] Denuvo Software Solutions GmbH. Denuvo Anti-Tamper. http://www.denuvo.com.

[27] Michael Eddington. Peach Fuzzer: Discover Unknown Vulnerabilities. https://www.peach.tech/.

[28] Abdelrahman Eid. Reverse Engineering Snapchat (Part I): Obfuscation Techniques. https://hot3eed.github.io/snap_part1_obfuscations.html.

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

[30] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In *ACM Workshop on Software PROtection (SPRO)*, 2016.

[31] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage Binary Code Obfuscation using Improved Virtual Machine. In *International Conference on Information Security*, pages 168–181. Springer, 2011.

[32] Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. NeuReduce: Reducing Mixed Boolean-Arithmetic Expressions by Recurrent Neural Network. In *Conference on Empirical Methods in Natural Language Processing: Findings*, 2020.

[33] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. Software Protection through Anti-Debugging. *IEEE Security & Privacy*, 5(3):82–84, 2007.

[34] Peter Garba and Matteo Favaro. SATURN – Software Deobfuscation Framework Based On LLVM. In *ACM Workshop on Software PROtection (SPRO)*, 2019.

[35] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control Flow based Obfuscation. In *ACM Workshop on Digital Rights Management*. ACM, 2005.

[36] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *GreHack Conference*, 2016.

[37] Sumit Gulwani. Dimensions in Program Synthesis. In *International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010.

[38] Aki Helin. Radamsa: A General-purpose Fuzzer. https://github.com/aoh/radamsa.

[39] Intel Corporation. Pin – A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[40] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In *ACM Workshop on Software PROtection (SPRO)*, 2015.

[41] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. VMAttack: Deobfuscating Virtualization-based Packed Binaries. In *Availability, Reliability and Security (ARES)*, 2017.

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

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[44] Paul Klint. Interpretation Techniques. *Software, Practice and Experience*, 11(9):963–973, 1981.

[45] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2016.

[46] Jae-Yung Lee, Jae Hyuk Suk, and Dong Hoon Lee. VODKA: Virtualization Obfuscation Using Dynamic Key Approach. In *International Workshop on Information Security Applications*, 2018.

[47] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization. In *International Conference on Information and Communications Security*, 2017.

[48] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *USENIX Security Symposium*, 2021.

[49] Weiyun Lu, Bahman Sistany, Amy Felty, and Philip Scott. Towards Formal Verification of Program Obfuscation. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020.

[50] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software Protection through Dynamic Code Mutation. In *International Workshop on Information Security Applications*. Springer, 2005.

[51] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

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

[53] Fukutomo Nakanishi, Giulio De Pasquale, Daniele Ferla, and Lorenzo Cavallaro. Intertwining ROP Gadgets and Opaque Predicates for Robust Obfuscation. *CoRR*, abs/2012.09163, 2020.

[54] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[55] Oreans Technologies. Themida – Advanced Windows Software Protection System. https://www.oreans.com/Themida.php.

[56] Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: An Obfuscation Approach using Probabilistic Control Flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[57] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

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

[59] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[60] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices. In *International Conference on Security and Cryptography (SECRYPT)*, 2013.

[61] Florent Saudel and Jonathan Salwan. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.

[62] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Technical Report: Hardening Code Obfuscation Against Automated Attacks. https://arxiv.org/abs/2106.08913, 2022.

[63] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, 2010.

[64] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy*, 2009.

[65] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

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

[67] Jae Hyuk Suk and Dong Hoon Lee. VCF: Virtual Code Folding to Enhance Virtualization Obfuscation. *IEEE Access*, 8, 2020.

[68] The LLVM Project. The LLVM Compiler Infrastructure. https://llvm.org/.

[69] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT Solvers in Software Security. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[70] Robin Verton. RC4. https://gist.github.com/rverton/a44fc8ca67ab9ec32089, 2015.

[71] VideoLAN. libdvdcss. https://www.videolan.org/developers/libdvdcss.html.

[72] VideoLAN. VLC media player. https://www.videolan.org/.

[73] VMProtect Software. VMProtect Software. https://vmpsoft.com/.

[74] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear Obfuscation to Combat Symbolic Execution. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[75] Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE)*, 1981.

[76] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and Extensible Equality Saturation. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[77] Gregory Wroblewski. General Method of Program Code Obfuscation. In *International Conference on Software Engineering Research and Practice (SERP)*, 2002.

[78] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. Boosting SMT Solver Performance on Mixed-bitwise-arithmetic Expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[79] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partially-virtualized Binary Code Simplification. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[80] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution. In *Conference on Dependable Systems and Networks (DSN)*, 2018.

[81] Chao Xue, Zhanyong Tang, Guixin Ye, Guanghui Li, Xiaoqing Gong, Wei Wang, Dingyi Fang, and Zheng Wang. Exploiting Code Diversity to Enhance Code Virtualization Protection. In *International Conference on Parallel and Distributed Systems*, 2018.

[82] Babak Yadegari and Saumya Debray. Bit-level Taint Analysis. In *IEEE Conference on Source Code Analysis and Manipulation*, 2014.

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

[84] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, 2015.

[85] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[86] Michał Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[87] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop on Information Security Applications*, 2007.