Assessing the Effectiveness of the Tigress Obfuscator Against MOPSA and BinaryNinja

Nicolò Altamura
University of Verona, Italy

Enrico Bragastini
University of Verona, Italy

Marco Campion
INRIA & ENS Paris, France

Mila Dalla Preda
University of Verona, Italy

ABSTRACT

We present an empirical evaluation of the Tigress obfuscator, focusing on its ability to degrade the precision of static analyses performed by two state-of-the-art tools: mopsa, a source-level static analyzer, and BinaryNinja a binary-level decompiler and analysis platform. By applying a variety of lightweight yet diverse obfuscation strategies—such as control flow flattening, opaque predicates, and data encoding—we systematically assess how these transformations affect the analyzability of C programs. Our findings highlight the scenarios in which obfuscation successfully confuses analysis tools and those where it fails to do so.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; • **Theory of computation → Program analysis**.

KEYWORDS

Software Protection, Static Analysis, Code Obfuscation

ACM Reference Format:
Nicolò Altamura, Enrico Bragastini, Marco Campion, and Mila Dalla Preda. 2025. Assessing the Effectiveness of the Tigress Obfuscator Against MOPSA and BinaryNinja. In *Proceedings of the 2025 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks (CheckMATE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3733817.3762702

1 INTRODUCTION

During the past thirty years, a wide range of software protection techniques, such as code obfuscation, tamper-resistance, tamper-detection, and anti-analysis techniques, have been proposed to deter malicious reverse engineering and unauthorized software manipulation [1, 6, 18]. These software protection solutions are used by software developers to protect the intellectual property of their code and prevent code tampering in the MATE (Man-At-The-End) scenario, where the attackers operate with unrestricted access to the code and to its execution environment. In the MATE scenario, attackers have white box access to the code and to its execution, and they can resort to any possible automatic tool (e.g., code analyzers, debuggers, disassemblers, dynamic instrumentation frameworks) to analyze and modify the code under attack. It is well known that absolute prevention of MATE attacks is infeasible and that, given enough time, effort, and determination, a competent programmer can always reverse engineer any application. For this reason, existing software protection techniques are designed to degrade the effectiveness of automated analysis tools [17] commonly employed by attackers, thereby delaying successful exploitation and reducing the overall benefit gained by the attacker.

In this paper, we take the perspective of software protection where attackers use reverse engineering and static analysis tools to understand the inner workings of programs in order to tamper with the code to their own advantage, while code obfuscation is used as a defense technique to obstruct reverse engineering. In the MATE scenario, the goal of software protection techniques, particularly code obfuscation, is to hinder program understanding. However, this objective is difficult to formally define, given the wide variety of tools and strategies that an attacker might employ. For this reason, assessing and measuring the efficiency of software protection solutions is challenging and often confusing [20].

Therefore, in this work we choose to constrain the attacker to specific static analysis tools, allowing us to systematically measure the confusion introduced by simple code obfuscation strategies. While this approach does not address the general problem of evaluating the effectiveness of software protection, it offers valuable insights into how code obfuscation impacts the results of static analysis.

More specifically, we are interested in measuring the effects of basic and well-known obfuscating transformations in degrading the results of static analysis, when the analysis is performed on the source code and on a numerical abstract domain specified in the abstract interpretation framework [8] and when the analysis consists in decompiling obfuscated binaries. We use the Tigress[1] obfuscation tool, a well-documented academic tool for source-to-source obfuscation of C programs, to apply a set of lightweight obfuscation techniques such as data encoding, control flow flattening, opaque predicates and we extended Tigress to include obfuscations based on opaque constants [27]. Given that Tigress operates at the source code level and produces valid C code, we selected analysis tools capable of handling C programs directly or via their compiled binaries. Specifically, we use mopsa [14], a static analyzer that works on C source code, and BinaryNinja [24], a binary analysis platform, to evaluate how such obfuscations impact the effectiveness of numerical program analysis and decompilation. We conducted experiments to compare the resilience of source-level and binary-level analysis techniques against common code obfuscation strategies. With these experiments we want to understand *how* and *where* even relatively simple obfuscating transformations can affect analysis precision.

Our Contribution. To assess the impact of code obfuscation on static analysis, we designed an experimental setup based on two datasets of C programs. The first dataset, manually constructed,

[1] www.tigress.wtf

CheckMATE '25, October 13–17, 2025, Taipei, Taiwan

ACM ISBN 979-8-4007-1906-6/2025/10
https://doi.org/10.1145/3733817.3762702

includes 17 small programs, each intended to isolate a specific C language construct or simple combinations thereof—such as multiple assignments, conditional branches, and loops. The second dataset, named `wrapped-intervals`, consists of 34 programs adapted from [9]. We analyzed both datasets using MOPSA static analyzer, employing both the *interval* and *convex polyhedra* abstract domains. These initial analyses serve as a baseline for evaluating the precision degradation caused by various obfuscating transformations. We then applied several code obfuscation techniques to the programs and re-executed the MOPSA analyses, observing how the analysis results differed from those obtained on the original, unobfuscated versions of the programs. This allowed us to assess the effectiveness of the obfuscation techniques in degrading the analyzer's precision. Additionally, we compiled the original and obfuscated programs and then decompiled them using BinaryNinja to verify whether the decompilation process could handle obfuscated binaries. When decompilation succeeds, we run MOPSA on the decompiled code. This step was intended to understand whether the compilation by `gcc` and the decompilation by BinaryNinja [24] were able to somehow deobfuscate the code, thus improving the outcome of static analysis.

In the remainder of the paper, we describe the tools used in our experiments and present a detailed discussion of the obtained results.

Related Work. The evaluation of software obfuscation techniques remains a challenging area due to the inherent complexity of measuring protection strength in the presence of MATE attackers [19–21]. De Sutter et al. [21] provide a comprehensive surveys to date, analyzing 571 papers in software protection research. They observe that even if dynamic analyses are known to be stronger when dealing with obfuscated code [18], they seem to complement static analyses rather than replace them. For this reason we believe that evaluating the effects of code obfuscation on static analysis is still interesting. In the abstract interpretation framework, specific distance formalisms were introduced (e.g., quasi-metrics [3] and premetrics [4]) in order to formalize meaningful metrics for measuring the imprecision of a static analyzer.

We are aware of other obfuscation tools that are freely available, such as Obfuscator-LLVM [12], but they apply transformations at the intermediate code level (using LLVM IR). To successfully evaluate the effect of the obfuscation on the MOPSA analyzer we needed an obfuscator that works on C code.

2 STATIC ANALYZERS

In our experiments we considered two static analyzers: MOPSA which works directly on C code, and BinaryNinja a binary analysis platform.

2.1 MOPSA

MOPSA [14] is a source-level static analyzer that supports multiple programming languages, including `C` and `Python`. Its primary objective is to detect bugs at compile time in order to prevent failures or vulnerabilities at runtime. MOPSA is built on the principles of abstract interpretation [8], a sound framework for static analysis that over-approximates program behaviors. It can identify a wide range of issues, including common security vulnerabilities (e.g., buffer overflows, division by zero), invalid assertions, and violations of C language semantics (e.g., misuse of standard library functions).

To reason about numerical properties in programs, MOPSA integrates abstract domains from the APRON library [10], which provides a suite of numerical domains capable of handling both integer and floating-point values. In our experiments, we focus on two such domains: the *interval* domain and the *convex polyhedra* domain. The interval domain approximates the possible values of each program variable at a given program point with a lower and upper bound, and is formally defined as

$$Int = \{[a, b] \mid a, b \in \mathbb{R} \cup \{-\infty, +\infty\}, a \leq b\} \cup \{\bot\},$$

where $\bot$ denotes the empty set, representing an unreachable program state. Thus, at each program point we have n intervals, one for each variable in the program. The convex polyhedra domain, on the other hand, allows for a more precise representation of variable relationships by capturing linear inequalities between variables. It models sets of values using constraints of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$,, where x_i are program variables and a_i, b are constant coefficients. Formally, the polyhedra abstract domain is defined as

$$\mathcal{P} = \left\{P \subseteq \mathbb{R}^n \mid P = \{x \in \mathbb{R}^n \mid Ax \leq b\}, A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m\right\} \cup \{\bot\}.$$

For a more detailed description of these abstract domains we refer to [13]. When analyzing a program, MOPSA interprets numerical computations over the selected abstract domain, providing a sound over-approximation of the set of possible values each variable may take. Depending on the chosen domain, this over-approximation can be expressed either as intervals or as systems of linear inequalities, trading off between precision and computational cost. Intervals are lightweight and efficient, making them suitable for quickly detecting basic range violations, while polyhedra provide higher precision by capturing linear relationships among variables at the cost of increased computational overhead. The comparison of analysis results under these two domains allows us to observe how obfuscation affects precision across different levels of abstraction.

2.2 BinaryNinja

BinaryNinja is a framework for executable file analysis and reverse engineering, primarily designed for decompilation targeting major desktop and embedded architectures [24]. Starting from the executable file and extracting its disassembled instructions, BinaryNinja constructs three levels of Intermediate Languages (IL): low, medium and high. The *Low Level IL* translates each disassembled instruction into an intermediate form that retains detailed information about registers, flags, and memory locations. This level includes analyses such as stack content tracking, tail call resolution, and initial constant propagation. The *Medium Level IL* introduces further abstraction by performing arithmetic expression folding, upgrading stack memory locations to variables, and recovering function parameters. The *High Level IL* provides a representation that closely resembles high-level source code. It includes type inference, control flow restructuring, dead code elimination, and variable merging. To extend the decompiler's capabilities, the developers introduced a transformation from the *High Level IL* to a *Pseudo C* representation that can be translated into C source code compliant with the C99 standard, and is therefore potentially recompilable.

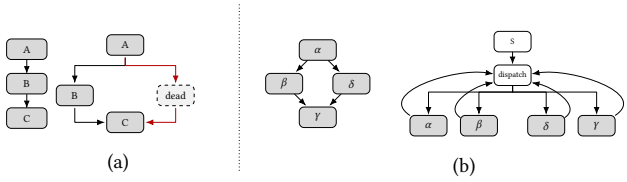

Figure 1: Opaque Predicates (a) and Flattening (b)

3 CODE OBFUSCATION WITH TIGRESS

Tigress is a source-level obfuscator that targets C source code and aims to protect the code in the MATE context [23]. It is widely considered the *de facto* state-of-the-art academic obfuscator as it is well documented and stable. Tigress allows to protect code by employing a series of transformations that are applied directly to the source code. The obfuscating techniques implemented in Tigress include obfuscations applied to data, control flow, function, and the ones based on virtualization. We have used the free version of Tigress [23]. In the following we briefly discuss the main obfuscation techniques that we employed in the evaluation.

3.1 Data Obfuscation

These transformations obfuscate the values held by variables at runtime, by modifying the encoding of data or by replacing integer arithmetic with more complex ones. Tigress distinguishes between obfuscations that target constant data, called *Encode Data*, and the ones that target arithmetic expressions, called *Encode Arithmetic*.

The *encode data* obfuscation technique relies on the use of non-standard representations to encode integer variables. The core idea is to prevent the recovery of constant values in the obfuscated code by using two key procedures: `encode`, which transforms the constant into a hidden representation, and `decode`, which restores the original value when necessary. The encoded values are maintained in a homomorphic form, allowing standard algebraic operations to be performed directly on the encoded data without requiring prior decoding. To preserve compatibility with unprotected code regions, Tigress inserts `decode` operations only when strictly necessary, such as when the encoded value is passed as a function argument or returned from a function. This selective decoding ensures both correctness and minimal exposure of the original constants.

Given a binary operation across the algebra domain (addition, subtraction, multiplication, division) or logic domain (and, or, xor, not), the *encode arithmetic* technique aims to rewrite the operation into a more complex expression. Tigress uses several rewriting rules extracted from [11], as reported in the documentation. The rewriting rules are based on *Mixed Boolean Arithmetic (MBA)*. MBA technique enables the transformation of simple expressions into semantically equivalent, yet syntactically more complex forms by combining boolean and arithmetic operators [27] such as: $x + y = 2 \cdot (x \vee y) - (x \oplus y)$.

3.2 Control Flow Obfuscation

Control flow transformations are designed to hinder program comprehension and analysis by making it more difficult to reconstruct the original control flow of the program. The control flow of any function represents key information to understand the semantics of a program: highlighting the connections across basic blocks lead to understand in which ways a function can react to its arguments. Tigress provides a variety of such transformations. Some of these generate the final code and control flow dynamically at run-time (e.g., `Virtualize`, `Jit`, and `JitDynamic`), while others obfuscate the existing control flow statically by introducing misleading constructs—such as spurious branches (`AddOpaque`), encoded branch targets (`EncodeBranches`), or control flow flattening (`Flatten`).

In our experiments, we focus exclusively on the latter category of control flow transformations. This choice is motivated by our attacker model, which assumes a static analysis perspective. Static analyzers are inherently limited when dealing with dynamically generated code, as the code to be analyzed is not available at compile time and is instead constructed during execution. Concentrating on these transformations allows us to assess software protection techniques in scenarios where the obfuscation overhead plays a critical role in the choice of transformation—for instance, in resource-constrained devices. In contrast, Just-In-Time transformations performed at runtime introduce considerable overhead due to the need for dynamically emitted instructions. In particular, we use `AddOpaque` and `Flatten`, while we omit to use `EncodeBranches` as it inserts raw assembly code into the C code to compute the target of branches, thus preventing the analysis with MOPSA.

The *control flow flattening* technique [25] is a well-known obfuscation that transforms the program control flow into a finite state machine. This transformation introduces two main constructs: a dispatcher and a set of states. Each state corresponds to a basic block from the original control flow graph, while the dispatcher is responsible for determining, based on the current state, the next block to execute (see Figure 1(b)). The effectiveness of this technique in hindering program analysis depends on how the dispatcher is implemented and how the state transitions are encoded.

An *opaque predicate* is a Boolean expression whose outcome is known at obfuscation time but is deliberately constructed to be difficult for an attacker to evaluate statically. First introduced in [7], opaque predicates are commonly employed in software protection to inject misleading or faulty information into the unreachable (dead) branch, such as incorrect variable assignments, spurious data, or code snippets that closely resemble legitimate logic (see Figure 1(a)). The purpose of these additions is to mislead static analyzers, which must conservatively consider both branches in the absence of precise predicate evaluation. Importantly, because the dead branch is never executed, the injected instructions do not affect the program's runtime behavior, thereby preserving correctness while complicating analysis.

3.3 Opaque constants

Opaque constants were introduced in [15, 27] as a foundational primitive for constructing obfuscation transformations capable of defeating static analysis tools. As the name suggests, an opaque constant refers to a code construct that loads a constant value into a register or variable in such a way that its actual constant value cannot be determined through static analysis. Opaque constants can be used to enhance the effectiveness of both control flow and data obfuscation by introducing uncertainty in value propagation and

usage. Although opaque constants are not natively supported in Tigress, their integration was straightforward due to the framework's flexible design. Several methods have been introduced to generate such opaque constant based either on XOR [26], RNC coding [28], 3-SAT [15], k-clique [22] or Mixed Boolean Arithmetic [27].

4 EXPERIMENTAL RESULTS

In this section, we present our experimental setup, including the datasets used and the execution parameters adopted for running Tigress, mopsa, and BinaryNinja[2]. Table 1 reports the results of mopsa analysis on the original and obfuscated code, while Table 2 reports the results of mopsa analysis on the programs decompiled by BinaryNinja starting from the obfuscated binaries.

Datasets. We run our experiments on two dataset of C programs:

- `simple-programs` contains 17 C programs, each one focused on a single construct of the C language or combinations of basic constructs (such as multiple assignments, if-then-else programs, while loops, for loops). Every program in the dataset shares the same structure: a function `int my_fun` that implements the core semantics of the program and returns an integer. Within programs, only integer variables have been used, without any complex structure and no particular input is required.
- `wrapped-intervals` includes 34 C programs taken from [9] and manually adapted to conform to the program structure of the `simple-programs` dataset in order to use the same test automation. Every program is defined through `int foo` that implements the main program functionality. The programs include while and for cycles with different level of nesting, if-then-else constructs with an uninitialized predicate, complex arithmetic operations and bit-wise operations. Within the source code, some macros are defined and we have expanded them before the analysis.

E1*:* Mopsa on the original datasets. mopsa is executed on all C programs in the two datasets using both interval and polyhedra analyses. These analyses are configured in mopsa through the files `c/cell-itv.json` for intervals and `c/cell-rel-itv.json` for polyhedra. As previously mentioned, each program in our datasets produces a single integer output, which we observe at the end of the analysis with mopsa. We retrieve this output by inserting the instruction `_mopsa_print(res)` before the `return` statement in each program. For all the programs in the two dataset the results of interval and polyhedra analysis coincide, meaning that the refined domain of polyhedra is not able to improve the precision of the analysis on the considered programs. For this reason in the tables we report the results only once. In the second column, named **E1: original**, of Table 1 we report the results of mopsa when running the polyhedra analysis. For instance, the first line indicates that for program `t01`, the mopsa analysis using the polyhedra domain yields the single value 100 for the output variable. This result is expressed in interval form as [100, 100], since when dealing with a single variable, the polyhedra representation reduces to an interval. Note that even though a single variable in the polyhedra domain is represented as an interval, the analysis may still produce a more precise interval than standard interval analysis, as polyhedra offers a more accurate approximation of computations by relating program variables with each others. This is not the case for the interval domain, explaining why it is also categorized as a non-relational domain. When considering program `t03` we have that the value of the output variable is approximated by the interval [10000, 32766], meaning that the mopsa analyzer is not able to predict the exact value of the output variable but it outputs a set of potential values, the ones in the returned interval. Of course, the bigger the interval is, the less precise the analysis result is. Regarding program `t08` we write $\top$ to denote the fact that the mopsa analyzer does not return any information on the possible values of the output variable that can assume any possible integer value in the range corresponding to its type (e.g., [-2147483648, 2147483647] for signed integers in 32-bit, [-128, 127] for integers in 8-bit). When the analysis returns `Fail`, it indicates that the mopsa analyzer encountered an issue during the analysis and was unable to produce a result. When running mopsa it is possible to specify the loop-unrolling option `-loop-unrolling=N` that unrolls exactly `N` iterations of the loop. This option is necessary to obtain more precise results in presence of loops. Without loop unrolling or when the chosen `N` parameter is smaller than the number of iterations actually performed by the program, mopsa returns $\top$, meaning that the analysis is not able to capture the while invariant. In our experiments, we set `N` to 10000. We observe that, even with loop unrolling, there are programs on which mopsa is imprecise or fails even in absence of obfuscation. We have manually verified that the programs on which mopsa returns $\top$ are not due to low value of the loop-unrolling parameter N but to the presence of data dependencies on input values (that are statically unknown).

These results, generated using the original code and reported in this second column, provide a baseline for comparison when analyzing the outcomes of the analyzers on the obfuscated code.

E1-BN*:* Mopsa and BinaryNinja on the original datasets. We compile the programs in the dataset using `gcc` and then decompile them with BinaryNinja. After running mopsa on the decompiled code there is no improvement in the analysis results. This suggests that the compilation and subsequent decompilation does not enhance the analyzability of the programs.

In Table 1 we report the results of the analysis on the original and obfuscated programs by reporting the interval of the possible values of the output variable. When we have loss of precision with respect to the results of the analysis on the original program (reported in column **E1**), we report in brackets in red the number of spurious elements introduced in the interval. This measures the amount of imprecision introduced by the obfuscation with respect to the considered analysis. When the analysis of the obfuscated code returns $\top$ we denote with *Max* the fact that the obfuscation has added the maximal amount of confusion to the analysis.

Obfuscation settings. The obfuscating transformations offered by Tigress make random selections among various options, such as the rewriting rules for data obfuscation, the numbering of basic blocks in control flow flattening, the choice of opaque predicates, and the code inserted into dead branches. However, by specifying a fixed seed in Tigress these choices can be made deterministic, thereby ensuring the reproducibility of the experiments. Thus, for

[2]We use mopsa 1.1 and BinaryNinja Personal Edition 5.0.7245.

file	E1: original	E2: data_obf	E3: flat	E4 a: opaque_list_array	E4 b: opaque_input	E4 c: opaque_env	E5: opaque_const
for_loop	[70, 70]	[70, 70]	[70, 70]	[70, 70]	[10, 70] (+60)	[70, 70]	⊤ (*Max*)
for_loop_1	[62, 62]	[62, 62]	[62, 62]	[62, 62]	⊤ (*Max*)	[62, 62]	⊤ (*Max*)
for_loop_2	[5, 5]	[5, 5]	[5, 5]	[5, 5]	⊤ (*Max*)	[5, 5]	[0, 10] (+10)
for_loop_3	[480, 480]	[480, 480]	[480, 480]	[480, 480]	[0, 480] (+480)	[480, 480]	[480, 480]
for_loop_4	[312, 312]	[312, 312]	[312, 312]	[312, 312]	[2, 312] (+310)	[312, 312]	[312, 312]
if_then_else_complex	[10, 10]	[10, 10]	[10, 10]	[10, 10]	[10, 10]	[10, 10]	[10, 94] (+84)
if_then_else_level_1	[-2, -2]	[-2, -2]	[-2, -2]	[-2, -2]	[-2, -2]	[-2, -2]	[-2, 2] (+4)
if_then_else_level_2	[3, 3]	[3, 3]	[3, 3]	[3, 3]	⊤ (*Max*)	[3, 3]	[3, 6] (+3)
if_then_else_simple	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[3, 4] (+1)
simple_assign_input_more_var	[22, 22]	[22, 22]	[22, 22]	[22, 22]	[21, 22] (+1)	[22, 22]	[-1073741806, 1073741841] (+2147483647)
simple_assign_input_one_var	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[-715827882, 715827882] (+1431655764)	[4, 4]	[-715827882, 715827882] (+1431655764)
simple_assign_more_var	[22, 22]	[22, 22]	[22, 22]	[22, 22]	[21, 22] (+1)	[22, 22]	⊤ (*Max*)
simple_assign_one_var	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[-715827882, 715827882] (+1431655764)	[4, 4]	[4, 5] (+1)
while_cycle	[15, 15]	[15, 15]	[15, 15]	[15, 15]	[3, 17] (+14)	[15, 15]	[15, 15]
while_cycle_1	[12, 12]	[12, 12]	[12, 12]	[12, 12]	⊤ (*Max*)	[12, 12]	⊤ (*Max*)
while_cycle_2	[27, 27]	[27, 27]	[27, 27]	[27, 27]	[2, 27] (+25)	[27, 27]	[0, 27] (+27)
while_cycle_3	[27, 27]	[27, 27]	[27, 27]	[27, 27]	[27, 27]	[27, 27]	[7, 27] (+20)
t01	[100, 100]	[100, 100]	[100, 100]	[100, 100]	[100, 100]	[100, 100]	[0, 100] (+100)
t02	[100, 100]	[100, 100]	[100, 100]	[100, 100]	[100, 127] (+27)	[100, 100]	[0, 100] (+100)
t03	[10000, 32766]	⊤ (*Max*)	⊤ (*Max*)	FAIL	[10000, 2147483647] (+2147450881)	[10000, 2147483647] (+2147460881)	⊤ (*Max*)
t04	[100, 127]	⊤ (*Max*)	⊤ (*Max*)	[100, 127]	[100, 127]	[100, 127]	[-2147483648, 127] (+2147483748)
t05	[100, 100]	[100, 100]	[100, 100]	[100, 100]	[100, 127] (+27)	[100, 100]	[0, 100] (+100)
t06	[10, 10]	[10, 10]	[10, 10]	[10, 10]	[10, 15] (+5)	[10, 10]	⊤ (*Max*)
t07	[100, 100]	[100, 100]	[100, 100]	[100, 100]	FAIL	[100, 100]	[5, 100] (+95)
t08	⊤	⊤	⊤	⊤	⊤	⊤	⊤
t09	[10, 30]	⊤ (*Max*)	⊤ (*Max*)	[10, 30]	[10, 30]	[10, 30]	[0, 31] (+11)
t10	FAIL	FAIL	⊤	FAIL	FAIL	FAIL	⊤
t11	[21, 21]	[21, 21]	[21, 21]	[21, 21]	[21, 21]	[21, 21]	⊤ (*Max*)
t12	[112, 112]	[112, 112]	[112, 112]	[112, 112]	[112, 112]	[112, 112]	[112, 114] (+2)
t13	[1, 1]	[1, 1]	[1, 1]	[1, 1]	[1, 1]	[1, 1]	[1, 7] (+6)
t14	[30, 30]	[30, 30]	[30, 30]	[30, 30]	[30, 30]	[30, 30]	⊤ (*Max*)
t15	[1000, 1000]	[1000, 1000]	[1000, 1000]	[1000, 1000]	[1000, 1000]	[1000, 1000]	[1000, 1002] (+2)
t16	[102, 102]	[102, 102]	[102, 102]	[102, 102]	[102, 102]	[102, 102]	[102, 104] (+2)
t17	[2001, 2001]	[2001, 2001]	[2001, 2001]	[2001, 2001]	[2001, 2001]	[2001, 2001]	[2, 2001] (+1999)
t18	[12, 12]	[12, 12]	[12, 12]	[12, 12]	[12, 12]	[12, 12]	[12, 14] (+2)
t19	[202, 202]	[202, 202]	[202, 202]	[202, 202]	[202, 202]	[202, 202]	⊤ (*Max*)
t20	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[4, 4]	[0, 4] (+4)
t21	⊤	⊤	⊤	⊤	⊤	⊤	⊤
t22	[7, 7]	[7, 7]	[7, 7]	[7, 7]	⊤ (*Max*)	[7, 7]	⊤ (*Max*)
t23	[0, 0]	⊤ (*Max*)	⊤ (*Max*)	[0, 0]	[0, 0]	[0, 0]	⊤ (*Max*)
t24	[-1073741825, -1]	⊤ (*Max*)	⊤ (*Max*)	[-1073741825, -1]	[-1073741825, -1]	[-1073741825, -1]	[-1073741825, 7] (+8)
t31	⊤	⊤	⊤	⊤	⊤	⊤	⊤
t32	⊤	⊤	⊤	⊤	⊤	⊤	⊤
t33	[-128, -128]	[-128, -128]	[-128, -128]	[-128, -128]	[-128, -128]	[-128, -128]	[-128, -126] (+2)
t50	[0, 2147483647]	⊤ (*Max*)	⊤ (*Max*)	[0, 2147483647]	[0, 2147483647]	[0, 2147483647]	[-10, 2147483647] (+10)
t51	[3, 98304]	⊤ (*Max*)	⊤ (*Max*)	[3, 98304]	[3, 98304]	[3, 98304]	[3, 98304]
t52	[0, 0]	⊤ (*Max*)	⊤ (*Max*)	[0, 0]	[0, 0]	[0, 0]	[-2147483648, 0] (+2147483648)
t53	[-8, 7]	⊤ (*Max*)	[-8, 7]	[-8, 7]	[-8, 7]	[-8, 7]	[-8, 7]
t54	[-16, 15]	⊤ (*Max*)	[-16, 15]	[-16, 15]	[-16, 15]	[-16, 15]	[-16, 15]
t60	[0, 10]	⊤ (*Max*)	⊤ (*Max*)	[0, 10]	[0, 10]	[0, 10]	[0, 10]
t62	⊤	⊤	⊤	⊤	⊤	⊤	⊤

Table 1: MOPSA **on `simple-programs` and `wrapped-intervals`**

file	E1-BN: original	E2-BN: data_obf	E3-BN: flat	E4-BN: opaque_list_array	E4-BN: opaque_input	E4-BN: opaque_env	E5-BN: opaque_const
for_loop	[70, 70]	[46, 46]	FAIL	[70, 70]	[10, 70] (+60)	[70, 70]	⊤ (*Max*)
for_loop_3	[480, 480]	[480, 480]	FAIL	[480, 480]	[0, 480] (+480)	[480, 480]	[480, 480]
if_then_else_level_1	[-2, -2]	[-2, -2]	FAIL	[-2, 2] (+4)	[-2, 2] (+4)	[-2, -2]	[-2, -2]
if_then_else_simple	[4, 4]	[8, 8]	FAIL	[4, 4]	[4, 4]	[4, 4]	[3, 4] (+1)
simple_assign_input_more_var	[22, 22]	[22, 22]	[22, 22]	[21, 22] (+1)	[22, 22]	[22, 22]	[-1073741806, 1073741841] (+2147483647)
simple_assign_input_one_var	[4, 4]	[4, 4]	FAIL	[4, 4]	[4, 4]	[4, 4]	[-715827882, 715827882] (+1431655764)
simple_assign_one_var	[4, 4]	[4, 4]	[4, 4]	[4, 4]	FAIL	[4, 4]	[4, 5] (+1)
while_cycle	[15, 15]	[20, 20]	FAIL	[15, 15]	[3, 17] (+14)	[15, 15]	[15, 15]
while_cycle_2	[27, 27]	[2, 2]	FAIL	[27, 27]	[2, 27] (+25)	[27, 27]	[27, 27]
t04	[100, 127]	⊤ (*Max*)	FAIL	[100, 127]	[100, 127]	[100, 127]	[-2147483648, 127] (+2147483748)
t05	[100, 100]	[13825, 13825]	FAIL	[100, 100]	[100, 127] (+27)	[100, 100]	⊤ (*Max*)
t07	[100, 100]	[100, 100]	FAIL	[100, 100]	[100, 100]	[100, 2147483647] (+2147483547)	[5, 100] (+95)
t09	[10, 30]	[0, 0]	FAIL	[10, 30]	[10, 30]	[10, 30]	[0, 31] (+11)
t11	[21, 21]	[0, 0]	FAIL	[21, 21]	[21, 21]	[21, 21]	⊤ (*Max*)
t12	[112, 112]	[112, 112]	FAIL	[112, 112]	[112, 112]	[112, 112]	[112, 114] (+2)
t17	[2001, 2001]	⊤ (*Max*)	FAIL	⊤ (*Max*)	[2001, 2001]	[2001, 2001]	[2, 2001] (+1999)
t18	[12, 12]	[12, 12]	FAIL	[12, 12]	[12, 12]	[12, 12]	[12, 14] (+2)
t19	[202, 202]	[0, 0]	FAIL	[202, 202]	[202, 202]	[202, 202]	⊤ (*Max*)
t20	[4, 4]	⊤ (*Max*)	FAIL	[4, 4]	[4, 4]	[4, 4]	[0, 4] (+4)
t22	[7, 7]	[7, 7]	FAIL	[7, 7]	[7, 7]	[7, 7]	⊤ (*Max*)
t33	[-128, -128]	[-7680, -7680]	FAIL	[-128, -128]	[-128, -128]	[-128, -128]	[-128, -126] (+2)
t50	[0, 2147483647]	⊤ (*Max*)	FAIL	[0, 2147483647]	[0, 2147483647]	[0, 2147483647]	[-10, 2147483647] (+10)
t51	[3, 98304]	⊤ (*Max*)	FAIL	[3, 98304]	[3, 98304]	[3, 98304]	[3, 98304]
t52	[0, 0]	⊤ (*Max*)	FAIL	[0, 0]	[0, 0]	[0, 0]	[-2147483648, 0] (+2147483648)
t54	[-16, 15]	[-128, 127] (+224)	FAIL	[-16, 15]	[-16, 16] (+1)	[-16, 16] (+1)	[-16, 16] (+1)

Table 2: BINARYNINJA **and** MOPSA **on `simple-programs` and `wrapped-intervals`**

each dataset we define a file `seeds.json` that associates to each program and obfuscation strategy a given seed. We polish the C programs returned by TIGRESS in order to remove redundant type definitions and function declarations. This polished version is the one that we fed to MOPSA.

E2*:* Mopsa and data obfuscation. We use both `EncodeData` and `EncodeArithmetic` data obfuscation. For encoding constant values, the Tigress documentation currently identifies `poly1` as the only fully supported transformation within the `EncodeData` category that ensures correctness of the obfuscated code. This encoding approach relies on the tautological property that, over the modular ring $\mathbb{Z}/(2^n)$, the composition of a linear permutation polynomial and its inverse yields the identity function: $p(q(x)) = x$, whenever $p = q^{-1}$ and $p \circ q = \text{id} \mod 2^n$. While computing the inverse of a general polynomial over a modular ring can be costly, the inverse of first-degree permutation polynomials can be efficiently computed by carefully selecting the coefficients [27]. The Tigress commands are: `Transform = EncodeData LocalVariables` where `poly1` is chosen by default and `LocalVariables` specifies which local variables have to be targeted, and `Transform = EncodeArithmetic` that targets any operation.

In Table 1, column **E2: data_obf** presents the results of the mopsa analysis on programs obfuscated using `EncodeArithmetic` and `EncodeData`. We observe that the analysis with mopsa either returns the same results obtained on the original code, meaning that no confusion has been added by data obfuscation, or goes to $\top$. Upon manually inspecting these $\top$ results (e.g., program `t17`), it turns out that the loss of precision is due to mopsa's inability to accurately evaluate loop guards that have been altered by the data obfuscation. This confirms that in order to be effective in degrading the precision of static analysis it is convenient to focus on the guards of the conditional statements and loops.

E2-BN*:* Mopsa and BinaryNinja with data obfuscation. We compile the programs obtained trough data obfuscation with `gcc` and provide this executable files as input to BinaryNinja in order to decompile them into C code. Table 2 reports the results of running mopsa on these decompiled obfuscated programs. For space reason, in Table 2 we omit to report the programs for which the results coincides with the ones of Table 1.

The results in column **E2-BN: data_obf** of Table 2 show that BinaryNinja does not improve the results of the analysis in presence of data obfuscation, while for programs: `if_then_else_simple`, `while_cycle`, `while_cycle_2`, `t09`, `t11`, `t19` and `t33`, mopsa returns an unsound result. In these cases the problem is related to the compiler optimization applied to the loop guards that leverages variable underflow. Since mopsa is not able to correctly track the underflow of a variable the analysis results to be unsound. This effectively thwart the analysis of the compiled/decompiled code under mopsa.

E3*:* Mopsa with flattening. Tigress allows us to customize the type of dispatcher used in the control flow flattening obfuscation for orchestrating the jumps across the basic blocks of code. The `flatten` transformation in Tigress has the following options:

- `switch`: every block is translated as a case in a switch statement, and the switch is inserted within an infinite loop;
- `goto`: every block ends with a goto to the next state;
- `indirect`: every block ends with a jump to an indirect goto built using a jump table;
- `call`: every block is a new function that can be called to jump across the various states;
- `concurrent`: similar to call, every block has its own thread: all threads work randomly, but only one is the original block that is doing real work;

In our experiments, we observed that mopsa fails to analyze flattened programs when the dispatcher is implemented using either `goto` or `indirect` jumps, as it raises an error in the presence of `goto` constructs. The `call` option, which splits the obfuscated program into multiple functions, is also not suitable since mopsa does not support inter-procedural analysis. Additionally, the `concurrent` option is not applicable because our analysis does not involve multithreading. For these reasons, we use the `switch`-based dispatcher in our experiments.

Column **E3: flat** of Table 1 reports the result of mopsa on the programs obfuscated with flattening. The results are precise when they coincides with the ones of column **E1: original** and in these cases precision achieved thanks to the use of loop unrolling and the accurate state tracking of the mopsa analyzer. Manual inspection of the imprecise results, indicated by $\top$, reveals that the issue stems from uninitialized guards present in the original programs. While mopsa can analyze these guards in isolation, their combination with control flow flattening causes the analysis to degrade to $\top$. This occurs because mopsa must handle two separate control flow paths, one for each branch of the uninitialized guard, via loop unrolling. When the analysis of one path completes, mopsa attempts to merge it with the results from the other path. Since it cannot determine whether the analysis of the remaining path will terminate, it conservatively returns $\top$. See for example program `t23`, where mopsa returns $[0, 0]$ while the result of the analysis goes to $\top$ when flattening is applied to `t23`.

Additionally, the effectiveness of mopsa analysis is directly impacted by the fixed number of loop unrolling iterations. An adversary applying control flow flattening could exploit this limit to either slow down the analysis or intentionally exceed the iteration bound, ultimately forcing mopsa to return $\top$.

E3-BN*:* Mopsa and BinaryNinja with flattening. We compile with `gcc` the programs previously obfuscated with `Flatten`. When these executable files are passed to BinaryNinja for decompilation, it fails to reconstruct the original C source code. This is because in the compiled flattened code the control flow is controlled by means of a jump table and determining the boundaries of jump tables in binary analysis is challenging [5]. Indeed, BinaryNinja cannot recover all the basic blocks introduced by flattening and generates partial code containing jumps to undefined jump tables. Thus, flattening the source code and then compiling it results in binaries that can no longer be decompiled by BinaryNinja. Exceptions are given by programs `simple_assign_input_more_var` and `simple_assign_more_var` where BinaryNinja is able to successfully reconstruct the high level C code, since given the simplicity of these samples the compiler does not recur to jump tables.

E4*:* Mopsa with opaque predicates. Tigress offers several methods for constructing opaque predicates. The process begins with the `InitOpaque` transformation, which generates a set of invariants that serve as the basis for the opaque predicates, and then uses the `AddOpaque` transformation to insert the specific opaque predicate.

These invariants are stored using a customizable data structure. The supported options for this structure include:

- `list`: use a linked list to store the opaque values
- `array`: use arrays to store the opaque values
- `input`: use any input that is passed into the program to make the opaque dependent on it (the input has to be specified)
- `env`: use entropy for generating the opaque values

In our experiments, we found that the specific configuration of opaque predicates has a notable influence on the precision of static analysis results. When using `list` or `array`, MOPSA is able to precisely evaluate the opaque predicates, namely to recognize the live branch (see column **E4 a** of Table 1). This is because the predicates constructed with these options depend on the initial values of the lists or arrays, which are explicitly initialized in the code. As a result, MOPSA can successfully propagate these values during analysis, allowing it to recognize and handle the opaque predicates accurately. So in this case no imprecision is added to the results of the analysis. When using the `env` option, Tigress generates opaque predicates based on a random value from the environment, which is then multiplied by 0. In this case, MOPSA is able to recognize that the result of the expression is always 0 and can successfully resolve the opaque predicate. With input-dependent opaque predicates, MOPSA cannot determine the exact value of the predicate, as it depends on external input. Consequently, it treats both branches as potentially executable, which leads to a degradation in analysis precision. For this reason, in Table 1 we report the results of the experiments where the opaque predicates are constructed using the `input` option. Observe that in this last case the obfuscated program has an extra-input with respect to the original program.

Of course, in this case, the amount of confusion added by the opaque predicates strongly depends upon the code that is inserted in the dead branch (the one never executed). Tigress provides different options for the code to be inserted in the dead branch:

- `call`: adds a dead branch where a random function is called;
- `bug`: adds a dead branch where a buggy statement is inserted, similar to the live one really executed;
- `true`: adds an always true guard to the live branch, no dead branch is specified;
- `junk`: adds a dead branch where inline data is inserted;
- `fake`: adds a dead branch where a random non existing function is called;
- `question`: makes the predicate more complex and adds the exact same real statement as a dead branch

In our experiments we used the `bug` and `call` option as these are the ones that are suitable for the analysis with MOPSA. The specific instructions added in the dead branch relay on random choices inside Tigress. For this reason there are cases where the analysis remains precise since instructions equivalent to `nop` are inserted in the dead branch. This explains the different amount of imprecision added to the analysis when inserting opaque predicates as reported in column **E4 b** of Table 1. For example, in programs `simple_assign_input_more_var`, `while_cycle` and `t02`, the content of the dead branch has been generated via the `bug` option, and we have a loss of precision in the results of the analysis.

E4-BN*:* Mopsa and BinaryNinja with opaque predicates. We compile with `gcc` the programs obfuscated with opaque predicates. When passing the executable files to BinaryNinja to reconstruct the C code, we need to decide whether or not to include the initial values stored in the data section or to leave them un-initialized. In the first case, the decompiled C code has opaque predicates that MOPSA is able to precisely analyze, meaning that it precisely identifies the live branch. When the initial values are not initialized MOPSA cannot evaluate the opaque predicates and sees both branches as possible thus degrading the results of the analysis. In this case, the noise introduced by the obfuscation strongly depends on the code present in the dead branch. Since Tigress randomly chooses what to insert in the never executed branch, we can have different levels of imprecision in the results of the analysis. See for example the following table where we compare the results of MOPSA on some programs when the initial values are initialized and when they are not:

	original	Opaque env (data init)	Opaque env (data not init)
`t12`	[112, 112]	[112, 112]	[102, 112]
`t18`	[12, 12]	[12, 12]	[2, 12]
`t20`	[4, 4]	[4, 4]	[4, 4]
`t22`	[7, 7]	[7, 7]	[0, 31]

For the programs `t12`, `t18`, `t22` we can see noise in the analysis when data are not initialized, while in program `t20` the results are still precise since the code of the dead branch contains only a `break` instruction that has no impact to the analysis.

In our experiments we choose to include the initial values when decompiling with BinaryNinja. For this reason the results of column **E4-BN** in Table 2 are the same as the ones of column **E4** in Table 1.

E5*:* Mopsa with opaque constants. In experiment **E4**, we observed that confusing the MOPSA analyzer requires relying on data that cannot be determined statically—such as values dependent on runtime inputs or randomness. In these cases, MOPSA fails to recognize invariant numerical properties. For instance, we manually inserted the predicate `if(2*random_number % 2 == 0)`, which is always true, but MOPSA could not detect this due to its dependence on a random value. This means that the insertion of any tautology that depends on input or random values is able to foil MOPSA. Following this intuition we decided to recur to opaque constants in the construction of opaque predicates.

In particular, in our experiments we recur to permutational polynomials [16] of the form $p(x) = \sum_{k=0}^{m} a_k x^k$ that permutes the elements in the ring $\mathbb{Z}/(2^n)$ to build opaque constants [2, 27]. By generating both a permutational polynomial and its inverse we can leverage on the fact that $p(p^{-1}(x)) = x$ in $\mathbb{Z}/(2^n)$ to encode the constant value x to hide. To enhance the complexity of the expression $p(p^{-1}(x))$ we add an expression E that always evaluates to 0. We manually obfuscate E with Mixed Boolean Arithmetic (MBA), by using a set of equivalent expressions derived from linear algebra [27], or mined through program synthesis [17]. This leads to expressions of the form $p(p^{-1}(x)+E)$. In order to prevent MOPSA from recognizing that E always evaluates to zero, we recur to MBA and we make E dependent on non-deterministic inputs read from the stack.

For example, we can use $p(x) = 208 \cdot x^2 + 145 \cdot x + 149$, $p^{-1}(x) = 48 \cdot x^2 + 145 \cdot x + 235$, and expression $E = 10 \cdot (a - b + 2 \cdot (\neg a \wedge b) - (a \oplus b))$ that always evaluates to 0, to build an opaque constant that encodes the value 25 for any a, b in $\mathbb{Z}/(2^8)$:

$$89 + (142 + 10 \cdot a + 246 \cdot b + 20 \cdot \neg(a \vee \neg b) + 246 \cdot (a \oplus b)) \cdot (160 + 224 \cdot a + 32 \cdot b + 192 \cdot \neg(a \vee \neg b) + 32 \cdot (a \oplus b)) + 170 \cdot a + 86 \cdot b + 84 \cdot \neg(a \vee \neg b) + 86 \cdot (a \oplus b)$$

We extended Tigress in order to integrate these opaque constants in the `InitOpaque` and `AddOpaque` transformations. To do so we need to provide Tigress with a routine that sets up the opaque constant `opaque_custom_initialize`, and a routine where we specify the known constant value `N` of the predicate `opaque_custom_VALUE_N`.

By combining opaque constants and opaque predicates the mopsa analyzer is not able to identify the live branch and has to consider both branches as possible. The results of the analysis on the programs obfuscated with the opaque predicates made with opaque constant are reported in column **E5: opaque_const** of Table 1. As discussed earlier, the amount of confusion added strictly depends upon the code inserted in the dead branch.

Observe that, opaque constants could be used for representing the state value associated to every basic block in the control flow flattening obfuscation. Tigress unfortunately does not provide an easy customization of the `Flatten` transformation. We have manually implemented this transformation that integrates opaque constants and the flattening obfuscation for program `t04` and `t18`. In these cases mopsa cannot compute the next basic block to be executed. This imprecision accumulates at every iteration of loop that executes the dispatcher causing the analysis to return $\top$.

E5-BN*:* Mopsa and BinaryNinja with opaque constants. As usual we compile the obfuscated code with `gcc` and pass it to BinaryNinja that is able to reconstruct the source C code. When applying mopsa to the decompiled C code we obtain the same results of **E5**. As for the previous experiment, the result of the analysis is impacted by the content of the dead branch that, e.g, for `t52` and `t04` effectively adds lot of points in the set of possible values without being $\top$. In some other cases, e.g. for `if_then_else_level_1`, `for_loop_3`, the content in the dead branch do not have any meaningful impact on the analysis for the presence of instructions that have no effect on the final value.

5 CONCLUSION

In this paper we have conducted an empirical evaluation of the effectiveness of the Tigress obfuscator in degrading the results of the static source code analyzer mopsa and of the binary analyzer BinaryNinja.

In general, the analyses performed by BinaryNinja in order to decompile the binaries obtained by the compilation of the obfuscated code does not remove the effects of the obfuscation. This is witnessed by the fact that the results of the analysis in the experiments **E2** and **E2-BN**, **E4** and **E4-BN**, and **E5** and **E5-BN** are almost always the same. When they differ is because the optimization of the compiler introduces constructs that mopsa is not able to analyze precisely, and this is not related to obfuscation. Recall that in experiment **E4-BN** we assume to initialize the global values.

As regarding the experiments with control flow flattening, the fact that **E3-BN** almost always fails we observed that the problem is due to the presence of jump table in the compiled code. This means that using control flow flattening on the source code, produces executables that BinaryNinja is no longer able to decompile. Table 3 summarizes the effects of Tigress in the decompilation with BinaryNinja.

	Data_Obf	Flatten	Opaque_Pred
BinaryNinja	✓	×	✓

Table 3: Impact of Tigress on decompilation with BinaryNinja

As regarding the analysis with mopsa we observe that, probably due to the simplicity of our dataset and to the fact that we observe the value of the single output variables, we were not able to appreciate the difference between the interval and polyhedra analysis. Probably, more extensive experiments need to be performed to highlight this point. The fact that the experiments **E1** and **E1-BN** return the same results means that given the compiled code BinaryNinja reconstructs C source code that is precisely analyzable by mopsa, proving that in the absence of obfuscation no confusion in introduced by the compilation/decompilation process.

Regarding the experiments **E2** with data obfuscation we conclude that, in order to be effective in degrading the precision of static analysis, it is convenient to focus on the guards of the conditional statements and loops. Indeed, in our experiments confusion is added whenever the expression in the guards depend upon inputs and random values that mopsa cannot analyze precisely. In the cases where mopsa is not hindered by data encoding, this is mainly because mopsa, similarly to many static analyzers, can reason more effectively within a local context. In particular, the considered data obfuscation was less effective since both the encoding and decoding functions were locally defined within the same function. This suggests that more resilient obfuscation techniques should rely on information that extends beyond the local context, thereby making the analysis significantly more challenging.

Regarding the control flow flattening, in experiment **E3** we have observed that the confusion in mopsa is caused by the combination of flattening and the presence of uninitialized guard in the original code. This suggests that combining opaque predicates and flattening may foil the mopsa analysis, as opaque predicates somehow resemble uninitialized guards. In experiment **E3-BN**, the flattening obfuscation completely defeats BinaryNinja, as recovering jump tables in the presence of undefined targets is a well-known hard problem in binary analysis. More generally, an effective obfuscation strategy must reduce the analysis to a problem that is already recognized as difficult in static analysis.

The experiments with opaque predicate insertion show that mopsa can identify the live branch when the inserted predicates can be statically resolved, while when they depend on inputs or random values mopsa has to consider also the dead branch. Of course the confusion added to the analysis strongly depends on the code inserted in the dead branch. Leveraging this observation we have proposed to combine opaque constants and opaque predicates in order to degrade the results of static analysis. This considerations apply to static analyzers in general.

Overall, our study confirms that even simple obfuscation techniques can impact the results of static analysis when carefully targeted.

As future work, we plan to extend our evaluation to multiple decompilers, such as, for example, those available through dogbolt.org, to determine whether our findings generalize beyond BinaryNinja. A systematic comparison would involve first assessing the decompilers themselves, then analyzing how individual transformations affect their performance. We also aim to consider additional static analyzers, obfuscators, and transformation techniques to further strengthen the generalizability of our results.

ACKNOWLEDGMENTS

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

REFERENCES

[1] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. 2019. Chapter Eight - A Taxonomy of Software Integrity Protection Techniques. *Adv. Comput.* 112 (2019), 413–486. https://doi.org/10.1016/BS.ADCOM.2017.12.007

[2] Lucas Barthelemy, Ninon Eyrolles, Guénaël Renault, and Raphaël Roblin. 2016. Binary Permutation Polynomial Inversion and Application to Obfuscation Techniques. In *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016, Vienna, Austria, October 24-28, 2016*, Brecht Wyseur and Bjorn De Sutter (Eds.). ACM, 51–59. https://doi.org/10.1145/2995306.2995310

[3] Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2022. Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498721

[4] Marco Campion, Caterina Urban, Mila Dalla Preda, and Roberto Giacobazzi. 2023. A Formal Framework to Measure the Incompleteness of Abstract Interpretations. In *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14284)*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer, 114–138. https://doi.org/10.1007/978-3-031-44245-2_7

[5] Cristina Cifuentes and Mike Van Emmerik. 1999. Recovery of Jump Table Case Statements from Binary Code. In *7th International Workshop on Program Comprehension (IWPC '99), May 5-7, 1999 - Pittsburgh, PA, USA*. IEEE Computer Society, 192–199. https://doi.org/10.1109/WPC.1999.777758

[6] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison Wesley Professional.

[7] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 184–196.

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[9] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2014. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *ACM Trans. Program. Lang. Syst.* 37, 1 (2014), 1:1–1:35. https://doi.org/10.1145/2651360

[10] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

[11] Henry S. Warren Jr. 2013. *Hacker's Delight, Second Edition.* Pearson Education. http://www.hackersdelight.org/

[12] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, Brecht Wyseur (Ed.). IEEE, 3–9. https://doi.org/10.1109/SPRO.2015.10

[13] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372. https://doi.org/10.1561/2500000034

[14] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2023. Mopsa-C: Modular Domains and Relational Abstract Interpretation for C Programs (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 565–570. https://doi.org/10.1007/978-3-031-30820-8_37

[15] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*. IEEE Computer Society, 421–430. https://doi.org/10.1109/ACSAC.2007.21

[16] Ronald L Rivest. 2001. Permutation polynomials modulo 2w. *Finite fields and their applications* 7, 2 (2001), 287–292. https://doi.org/10.1006/ffta.2000.0282

[17] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 3055–3073. https://www.usenix.org/conference/usenixsecurity22/presentation/schloegel

[18] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1 (2016), 4:1–4:37. https://doi.org/10.1145/2886012

[19] Bjorn De Sutter. 2025. A New Framework of Software Obfuscation Evaluation Criteria. arXiv:2502.14093 [cs.SE] https://arxiv.org/abs/2502.14093

[20] Bjorn De Sutter, Christian S. Collberg, Mila Dalla Preda, and Brecht Wyseur. 2019. Software Protection Decision Support and Evaluation Methodologies (Dagstuhl Seminar 19331). *Dagstuhl Reports* 9, 8 (2019), 1–25. https://doi.org/10.4230/DagRep.9.8.1

[21] Bjorn De Sutter, Sebastian Schrittwieser, Bart Coppens, and Patrick Kochberger. 2025. Evaluation Methodologies in Software Protection Research. *ACM Comput. Surv.* 57, 4 (2025), 86:1–86:41. https://doi.org/10.1145/3702314

[22] Roberto Tiella and Mariano Ceccato. 2017. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, Martin Pinzger, Gabriele Bavota, and Andrian Marcus (Eds.). IEEE Computer Society, 182–192. https://doi.org/10.1109/SANER.2017.7884620

[23] Tigress. [n. d.]. The Tigress C Obfuscator — tigress.wtf. https://tigress.wtf.

[24] Vector35. [n. d.]. Binary Ninja — binary.ninja. https://binary.ninja.

[25] Chenxi Wang. 2001. *A security architecture for survivability mechanisms.* Ph. D. Dissertation. University of Virginia.

[26] Carolina Zarate, Simson L Garfinkel, Aubin Heffernan, Kyle Gorak, and Scott Horras. 2014. *A survey of xor as a digital obfuscation technique in a corpus of real data.* Technical Report. Monterey, California. Naval Postgraduate School. https://doi.org/10.21236/ada592678

[27] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4867)*, Sehun Kim, Moti Yung, and Hyung-Woo Lee (Eds.). Springer, 61–75. https://doi.org/10.1007/978-3-540-77535-5_5

[28] William Zhu and Clark Thomborson. 2005. A provable scheme for homomorphic obfuscation in software security. In *The IASTED International Conference on Communication, Network and Information Security, CNIS*, Vol. 5. 208–212. https://doi.org/10.1007/11734628_18